Université Paris Cité
École Doctorale 386 - Sciences Mathématiques de Paris Centre
Institut de Recherche en Informatique Fondamentale

Typing Dynamic Languages with Set-Theoretic Types

The Case of Elixir

Thèse de Doctorat en Informatique

Présentée par

Guillaume Duboc

Soutenue publiquement le 19/01/2026

Dirigée par Giuseppe Castagna, Directeur de Recherche, Université Paris Cité
et José Valim, Chief Adoption Officer, Dashbit

Jury

Directeur de thèse	Giuseppe Castagna	Directeur de Recherche	Université Paris Cité
Co-encadrant, membre invité	José Valim	Chief Adoption Officer	Dashbit
Rapporteur	Didier Rémy	Directeur de Recherche	INRIA
Rapporteur	Éric Tanter	Professor Titular	Université du Chili
Examinatrice	Annette Bieniusa	Professeure	RPTU Kaiserslautern-Landau
Examinatrice	Mariangiola Dezani	Professeure émérite	Université de Turin
Examinateur	Ben Greenman	Assistant Professor	University of Utah

This thesis is written in English.

A French abstract is provided below for French-speaking readers.

⋆ ⋆ ⋆

Abstract

Titre : Typage des langages dynamiques par types ensemblistes : le cas d'Elixir

Mots-clés : Systèmes de types · Langages de programmation · Typage graduel · Types ensemblistes · Sous-typage sémantique · Elixir · Inférence de types · Typage dynamique · Typage statique · Implémentation

L'ajout de types statiques à un langage dynamique nécessite de trouver un équilibre entre expressivité, sûreté du typage et rétrocompatibilité - un arbitrage qui conduit souvent à des compromis limitant l'adoption pratique des systèmes de types. Elixir, langage dynamique fonctionnel construit sur la VM Erlang et doté d'un puissant mécanisme de filtrage par motifs, illustre parfaitement ces défis. Cette thèse présente un système de types graduel pour Elixir, fondé sur le concept fondamental de types ensemblistes définis sémantiquement : les types y dénotent des ensembles de valeurs, avec unions, intersections et négations interprétées de façon naturelle.

L'hypothèse centrale de ce travail est que les types ensemblistes sont particulièrement adaptés pour capturer le style de programmation des langages dynamiques. Les types union reflètent le caractère fortement branchant des programmes dynamiques et la diversité des entrées qu'ils traitent. Les types intersection jouent un rôle crucial pour exprimer les flux de données et gérer la surcharge de fonctions – traits courants des langages dynamiques – en liant des types d'entrée spécifiques à leurs types de sortie correspondants. Les types négation permettent une analyse précise du filtrage par motifs « first-match », en excluant explicitement les valeurs capturées par les branches précédentes. Parce qu'ils sont à la fois expressifs et faciles à enseigner, les types ensemblistes constituent un levier pragmatique pour mieux comprendre, utiliser et typer les langages dynamiques.

Un enjeu crucial est d'articuler typage graduel et sous-typage sémantique. Nous représentons les types graduels comme des intervalles, bornés inférieurement par le type statique précis des valeurs connues et supérieurement par le type dynamique de toutes les valeurs possibles, ce qui permet de suivre finement l'information de type aux frontières entre code typé et non typé.

La première partie dresse une présentation formelle du système. Nous définissons une discipline de typage graduel avec effacement de types s'appuyant sur les vérifications déjà présentes dans la VM Erlang, maximisant la sûreté du typage sans altérer la sémantique des programmes ni affecter leur performance. Nous présentons également une analyse typée approfondie du filtrage par motifs d'Elixir. Nous étendons le sous-typage sémantique pour prendre en charge les structures de données du langage – notamment les tuples et les fonctions à arité multiple – et exposons une méthodologie pour ajouter de nouveaux constructeurs de types tout en préservant la décidabilité du sous-typage. Pour éviter la propagation incontrôlée des types dynamiques, nous introduisons des types fonctionnels dits « forts », un aspect crucial pour maintenir la sûreté du typage dans un système graduellement typé.

La seconde partie traite de l'implémentation dans le compilateur Elixir, en comblant l'écart entre l'élégance conceptuelle des types ensemblistes et leur réalisation opérationnelle. Bien que ces concepts soient intuitifs et faciles à expliquer aux développeurs, le travail technique nécessaire pour opérationnaliser cette approche sémantique est délicat : décider le sous-typage sémantique (déterminer si un ensemble de valeurs est contenu dans un autre lorsqu'il est défini par des unions, intersections et négations arbitraires) est complexe d'un point de vue algorithmique et exigeant en calcul. Nous présentons des algorithmes et des structures de données efficaces pour les opérations sur les types, évaluons les performances de compilation et documentons des choix de conception qui minimisent le surcoût tout en conservant l'expressivité. Nous détaillons les représentations : types de base, types littéraux, types graduels, puis types structurels (tuples, maps, fonctions). Les types graduels sont représentés comme des paires de types statiques, un format simple à implémenter et fidèle aux fondations théoriques. L'ensemble forme un guide pour intégrer des types graduels ensemblistes dans des langages dynamiques.

Abstract

Title: Typing Dynamic Languages with Set-Theoretic Types: The Case of Elixir

Keywords: Type Systems · Programming Languages · Gradual Typing · Set-Theoretic Types · Semantic Subtyping · Elixir · Type Inference · Dynamic Typing · Static Typing · Implementation

Adding static types to dynamic languages requires balancing expressiveness, safety, and backward compatibility–trade-offs that often force compromises limiting the practical adoption of type systems. Elixir, a functional dynamic language built on the Erlang VM with pattern matching, guards, and first-match semantics, exemplifies these challenges. This thesis presents a gradual type system for Elixir grounded in the foundational concept of semantically defined set-theoretic types, where types denote sets of values and support union, intersection, and negation operations with their natural set-theoretic interpretations.

The central hypothesis of this work is that set-theoretic types are particularly well-suited for capturing the programming patterns of dynamic languages. Union types naturally reflect the highly branching nature of dynamic programs and the diverse range of inputs they handle. Intersection types play a crucial role in expressing data flow and handling function overloading–common traits of dynamic languages–by linking specific input types to their corresponding output types. Negation types enable precise analysis of pattern matching with first-match semantics by explicitly excluding values captured by preceding branches. This combination of expressiveness and conceptual simplicity, along with the ease of explaining set-theoretic concepts to programmers, highlights the potential of set-theoretic types to improve how dynamic languages are understood, used, and typed.

A key challenge is articulating gradual typing with semantic subtyping. We represent gradual types as intervals bounded below by the precise static type of all known values and above by the dynamic type of all possible values, enabling fine-grained tracking of type information at the boundaries between typed and untyped code. This representation integrates naturally with set-theoretic types.

The first part provides a formal presentation of the system. We define a safe-erasure gradual typing discipline that exploits runtime checks already present in the Erlang VM, maximizing type safety without altering program semantics or introducing performance penalties. We also present an extensive typed analysis of the pattern matching construct present in Elixir. We extend semantic subtyping to handle Elixir's data structures–including tuples and multi-arity functions–and expose a methodology for adding new type constructors while preserving decidable subtyping. To prevent the unchecked spread of dynamic types, we introduce strong function refinements, a crucial aspect for maintaining type safety in a gradually typed system.

The second part addresses the practical implementation within the Elixir compiler, bridging the gap between the conceptual elegance of set-theoretic types and their operational realization. While these concepts are intuitive and easy to explain to developers,

the technical work required to operationalize this semantic approach is delicate: deciding semantic subtyping–determining whether one set of values is contained in another defined by arbitrary unions, intersections, and negations–is algorithmically complex and computationally demanding. We present efficient algorithms and data structures for type operations, evaluate compilation performance, and document design choices that minimize overhead while preserving expressiveness. We detail the representations: base types, literal types, gradual types, then structural types (tuples, maps, functions). Gradual types are represented as pairs of static types, a format that is simple to implement and faithful to the theoretical foundations. This part constitutes a comprehensive guide for language implementers seeking to integrate gradual set-theoretic types into dynamic languages.

Résumé étendu

Ce résumé en français offre une vue d'ensemble de la thèse aux lecteurs francophones.

The rest of this thesis is written in English.

Présentation de la première partie

La première partie de cette thèse pose les fondements théoriques et techniques du système de types d'Elixir. Elle se déploie en six chapitres qui, ensemble, construisent un système de typage graduel à la fois expressif et pratique, capable de s'adapter aux idiomes d'un langage dynamique tout en offrant des garanties statiques précieuses.

Chapitre 2 : Introduction technique Le voyage commence par une présentation des défis que pose le typage d'Elixir. Le système que nous développons repose sur le cadre du *sous-typage sémantique*, enrichi de types ensemblistes (unions, intersections, négations), et doit relever plusieurs défis techniques absents des systèmes existants : le typage des *fonctions multi-arité*, la *propagation du type dynamique* nécessaire à un typage graduel à effacement sûr, l'introduction des *fonctions fortes* qui exploitent les vérifications dynamiques du programmeur, et une *analyse des gardes* qui traduit la richesse du pattern matching d'Elixir en informations de types précises. À travers des exemples concrets, comme la fonction `negate` qui retourne un entier pour un argument entier et un booléen pour un argument booléen, nous illustrons comment ces techniques coopèrent pour produire des types intersections précis et permettre un typage fin même en présence d'incertitude.

Chapitre 3 : Fondements du sous-typage sémantique Ce chapitre présente les assises mathématiques de notre système. L'idée centrale du sous-typage sémantique est d'interpréter les types comme des *ensembles de valeurs* : un type t_1 est sous-type de t_2 si et seulement si l'ensemble des valeurs de t_1 est inclus dans celui de t_2. Cette vision ensembliste permet d'intégrer naturellement les connecteurs booléens (union $\vee$, intersection $\wedge$, négation $\neg$) dans la syntaxe des types, avec une sémantique qui découle directement des opérations sur les ensembles.

Nous rappelons que décider le sous-typage se ramène à décider la *vacuité* d'un type, et que tout type peut être mis sous *forme normale disjonctive*, permettant des algorithmes de décision modulaires pour chaque famille de types (constantes, produits, flèches). Ce chapitre introduit également les *types graduels* et leurs relations spécifiques : la *précision* (qui compare le degré d'incertitude de deux types) et le *sous-typage cohérent* (qui autorise les opérations pouvant réussir pour certaines matérialisations du type dynamique). Un résultat remarquable est le *théorème de représentation* : tout type graduel τ est équivalent à $\tau^{\Downarrow} \vee (? \wedge \tau^{\Uparrow})$, où $\tau^{\Downarrow}$ et $\tau^{\Uparrow}$ sont des types statiques obtenus en remplaçant les occurrences de ? par $\mathbb{0}$ ou $\mathbb{1}$ selon leur variance. Cette représentation permet de réutiliser directement toute l'infrastructure du sous-typage statique pour les types graduels.

Chapitre 4 : Typage statique de Core Elixir Nous définissons ici *Core Elixir*, un calcul minimal capturant les traits essentiels d'Elixir : constantes, variables, abstractions annotées par des *interfaces* (ensembles de types flèches dont l'intersection déclare le type de la fonction), applications, tuples, projections, expressions `case`, et opérateurs arithmétiques. La sémantique opérationnelle distingue explicitement les erreurs d'exécution (ω) levées par la machine virtuelle BEAM, ce qui permet d'énoncer des théorèmes de sûreté précis.

Les règles de typage statique exploitent pleinement les types ensemblistes : les interfaces permettent de typer les fonctions par des intersections de flèches ; les projections acceptent des indices de type union (le résultat est alors l'union des types correspondants) ; le typage des `case` vérifie l'*exhaustivité* (tout cas est couvert) et signale la *redondance* (branches inatteignables). Les théorèmes de *progress* et *preservation* établissent que les programmes bien typés sans règles ω ne produisent jamais d'erreur d'exécution.

Une contribution technique majeure de ce chapitre est l'extension du sous-typage sémantique aux *fonctions multi-arité*. Contrairement à l'encodage usuel par tuples (où $(\mathbb{0}, \texttt{int}) \to \texttt{int}$ serait équivalent à $\mathbb{0} \to \texttt{int}$), nous introduisons une interprétation native utilisant l'élément $\mho$ pour distinguer les domaines vides selon chaque argument. Le théorème 4.4.5 caractérise l'inclusion entre intersections de flèches multi-arité et donne lieu à un algorithme de décision (fonction Φ_n) correct et complet.

Chapitre 5 : Typage graduel Le type dynamique ? entre en scène. L'enjeu est de permettre l'interaction fluide entre code typé et code non typé, sans modifier le comportement à l'exécution, une approche dite à *effacement sûr*, contrairement aux systèmes graduels qui insèrent des proxies. Le défi est que ? tend à « contaminer » les types : appliquer une fonction $\texttt{int} \to \texttt{int}$ à un argument ? ne peut statiquement garantir un résultat entier.

Notre solution repose sur les *fonctions fortes* (strong arrows), notées $(t \to s)^{\star}$. Une fonction est forte si, appliquée à un argument *hors* de son domaine, elle soit échoue explicitement (sur une vérification dynamique), soit retourne néanmoins une valeur dans son codomaine, soit diverge. Cette propriété est vérifiée par un système de typage *faible* auxiliaire qui modélise les vérifications effectuées par la BEAM et par les gardes du programmeur. Ainsi, si une fonction est annotée

$\{(\mathbb{1},$ `int`$) \to$ `int`$\}$ et que son corps commence par un test `is_integer(elem(x,1))`, elle est forte : même appliquée à un argument dynamique, elle retournera un entier (ou échouera).

Le chapitre établit la *soundness graduelle* : un programme typé par le système graduel soit diverge, soit retourne une valeur dont la *forme* correspond au type déduit (par exemple, une valeur de type `int` $\wedge$? est bien un entier), soit échoue sur une vérification dynamique.

Chapitre 6 : Analyse des gardes Les gardes et le pattern matching sont au cœur d'Elixir. Ce chapitre développe une analyse qui traduit les gardes, c'est-à-dire les combinaisons de tests de types (`is_integer`), d'égalités, de comparaisons et de sélections, en informations de types exploitables par le système.

Pour chaque garde, l'analyse calcule deux types : le type *sûrement accepté* (toute valeur de ce type satisfait la garde) et le type *possiblement accepté* (toute valeur satisfaisant la garde appartient à ce type). Quand ces deux types coïncident, l'approximation est exacte ; sinon, un drapeau booléen indique l'imprécision. Par exemple, la garde `is_boolean(elem(x,0)) or elem(x,0) == elem(x,1)` produit $\{$`bool`, ..$\}$ comme type sûrement accepté (tout tuple commençant par un booléen satisfait la garde) et $\{$`bool`, ..$\} \vee \{\mathbb{1}, \mathbb{1}$, ..$\}$ comme type possiblement accepté.

Ces types alimentent les règles de typage des `case` : pour chaque branche, le système calcule le type des valeurs qui l'atteignent en soustrayant les types sûrement capturés par les branches précédentes. Cette analyse fine permet de détecter les branches redondantes, de vérifier l'exhaustivité, et d'inférer automatiquement des types intersections précis pour les fonctions multi-clauses non annotées. Par exemple, la fonction `negate` définie par deux clauses avec gardes `is_integer(x)` et `is_boolean(x)` se voit inférer le type (`int` $\to$ `int`) $\wedge$ (`bool` $\to$ `bool`) plutôt que le type moins précis (`int` $\vee$ `bool`) $\to$ (`int` $\vee$ `bool`).

Chapitre 7 : Discussion et travaux connexes Ce chapitre situe notre contribution dans le paysage des systèmes de types pour langages dynamiques. Parmi les travaux les plus proches, Etylizer (2023) adapte le sous-typage sémantique à Erlang, tandis que nous l'*étendons* avec des fonctionnalités spécifiques à Elixir (notamment le côté graduel) ; eqWAlizer (Meta/WhatsApp) partage nos fondements graduels mais n'exploite ni la propagation dynamique ni les fonctions fortes ; Dialyzer garantit l'absence de faux positifs là où nous garantissons la *soundness* (absence de faux négatifs).

Nous positionnons également notre approche parmi les trois disciplines de typage graduel identifiées par Greenman et al. (2023) : *erasure* (les types n'affectent pas l'exécution), *transient* (insertion de vérifications de forme), et *natural* (proxies complets). Notre système appartient à la discipline *erasure*, la plus populaire en industrie, mais contrairement à TypeScript, Flow ou Hack, nous préservons la *soundness* sans modifier le runtime. Les fonctions fortes jouent le rôle des vérifications transientes : au lieu d'insérer des checks, nous détectons statiquement leur présence dans les gardes du programmeur. Cette stratégie d'*effacement sûr* permet aux développeurs Elixir d'adopter les types progressivement, sans sacrifier compatibilité, performance ou sémantique du langage.

Le chapitre esquisse enfin les directions futures : typage des constructions de concurrence (`receive`), interfaces de processus, et à plus long terme, types comportementaux pour le modèle acteur d'Elixir.

Présentation de la seconde partie

La première partie de cette thèse a posé les fondements théoriques du système de types d'Elixir : sous-typage sémantique, typage graduel, analyse des gardes. Cette seconde partie entreprend le voyage inverse, de la théorie vers le compilateur. Nous y montrons comment transformer des règles déclaratives en algorithmes concrets, comment représenter efficacement les types ensemblistes, et comment intégrer le système dans un langage de production avec plus d'un million d'utilisateurs.

Chapitre 8 : Une API pour les types ensemblistes Le chapitre ouvre sur une observation : les types ensemblistes offrent un paradoxe stimulant. D'un côté, leur construction repose sur une machinerie théorique complexe (interprétation sémantique, formes normales disjonctives, procédures de décision récursives). De l'autre, leur utilisation est d'une simplicité désarmante : penser en ensembles, composer avec $\vee$, $\wedge$, $\setminus$, et s'appuyer sur des intuitions familières.

Nous défendons l'idée que cette complexité peut être encapsulée derrière une API stable et minimaliste. Ce dont un vérificateur de types a besoin est remarquablement restreint : (i) des constructeurs pour les connecteurs ensemblistes et les types de base, (ii) des opérateurs pour les types structurels (domaine, application, projection), et (iii) quelques requêtes fondamentales telles que sous-typage, équivalence, vacuité qui peuvent se composer, par exemple pour déterminer que l'intersection de deux types est vide. Avec ces briques, les règles de typage se réduisent à des appels de constructeurs et destructeurs. Cette vision « theory-first » permet au programmeur de raisonner avec des mots simples (« and », « or », « not ») tout en bénéficiant de la puissance d'un système complet.

Chapitre 9 : Architecture modulaire de représentation Ce chapitre introduit le `Descr`, la structure de données centrale utilisée par les systèmes de sous-typage sémantique. L'idée fondatrice est de représenter chaque type comme une union de composants *disjoints* (entiers, atomes, tuples, fonctions), chacun stocké dans un champ séparé. Cette modularité est la clé : puisque les composants ne se chevauchent jamais, les opérations ensemblistes (union, intersection, différence) se calculent *champ par champ*, puis se combinent. Ajouter un nouveau type de base revient simplement à ajouter un champ et définir ses opérations locales.

Cette architecture s'étend naturellement aux types graduels. Grâce au théorème de représentation (Partie I), tout type graduel τ s'écrit $\tau^{\Downarrow} \vee (? \wedge \tau^{\Uparrow})$. Nous représentons donc τ par une paire de `Descr` statiques, l'un pour la borne inférieure, l'autre pour la supérieure. L'invariant $\tau^{\Downarrow} \leq \tau^{\Uparrow}$ garantit que les opérations composante par composante préservent la validité, permettant de réutiliser directement l'infrastructure statique.

Chapitre 10 : Représentation des types de base et graduels Nous détaillons ici les représentations concrètes. Les six types de base indivisibles (entiers, flottants, binaires, pids, ports, références) sont encodés par un simple masque de 6 bits : l'union devient un OR bit-à-bit, l'intersection un AND, la différence un AND avec complément, le tout en temps constant.

Pour les domaines infinis comme les atomes, où l'on souhaite exprimer « tous les atomes sauf `:foo` et `:bar` », nous adoptons une représentation finie/cofinie : une paire $\langle mode, S\rangle$ où S est un ensemble fini et *mode* indique si S représente les éléments inclus ou exclus. Les opérations ensemblistes se traduisent alors en manipulations d'ensembles finis, avec des formules closes pour chaque combinaison de modes.

Le chapitre formalise également la représentation par paires pour les types graduels, prouve que les opérations préservent l'invariant de validité, et discute une représentation alternative (paires disjointes) que nous n'avons finalement pas retenue en raison de son coût de normalisation.

Chapitre 11 : Types structurels et arbres de décision Les types structurels (fonctions, tuples, listes, maps) sont le cœur de l'expressivité d'Elixir. Leur représentation repose sur les *diagrammes de décision binaires* (BDD), une structure classique pour les formules propositionnelles dont l'usage pour les types ensemblistes a été proposé par Frisch (2004). Chaque type structurel est une formule booléenne sur des littéraux atomiques (des flèches $(t_1,\ldots,t_n) \to t$ pour les fonctions, des tuples $\{\, t_1,\ldots,t_n \,\}$ pour les tuples, etc.). Les BDD offrent des opérations polynomiales pour union, intersection, différence et négation, un avantage décisif sur les formes normales disjonctives.

Cependant, les BDD peuvent exploser en taille lors d'unions consécutives. (2004) introduit les *diagrammes de décision ternaires* (TDD) qui ajoutent une branche médiane pour encoder les unions de manière paresseuse. L'union devient alors linéaire : au lieu de distribuer, on attache simplement le second opérande à la branche médiane. L'intersection et la différence peuvent nécessiter l'expansion de ces unions différées, mais le gain global est substantiel dans les cas typiques.

Le chapitre détaille ensuite les spécialisations pour Elixir : fonctions multi-arité représentées par des TDD cofinies (partitionnées par arité), tuples ouverts et fermés, listes propres et impropres avec leurs algorithmes de vacuité, et maps hybrides combinant champs nommés et domaines typés.

Chapitre 12 : Analyse comparative des représentations Ce chapitre documente les alternatives que nous avons explorées avant de converger vers les TDD. Les *formes normales disjonctives* (DNF) sont efficaces pour union et intersection (temps polynomial), mais la différence est exponentielle, un problème rédhibitoire lorsque l'analyse de pattern matching multiplie les soustractions de types.

Les *DNF compressées* pré-calculent l'intersection des littéraux positifs, éliminant les clauses vides et réduisant la redondance. Cette approche fonctionne pour les maps et tuples (dont l'intersection est calculable directement), mais pas pour les fonctions.

Les *formes union* pour les tuples éliminent entièrement les négations grâce au théorème de décomposition, mais souffrent elles aussi d'une croissance quadratique lors d'intersections répétées.

L'analyse conclut que les TDD représentent le meilleur compromis : polynomiales pour toutes les opérations, avec une union particulièrement efficace grâce à la paresse. Les tables comparatives (PT1, PT2) documentent l'évolution des performances entre v1.18 (DNF) et v1.19 (TDD), montrant notamment comment la correction des formules d'intersection dans les TDD est importante pour sécuriser les gains de perfomance obtenus grâce à la paresse.

Chapitre 13 : Du calcul formel au compilateur Ce chapitre fait le pont entre Core Elixir (notre calcul idéalisé) et Featherweight Elixir (un sous-ensemble strict du langage réel). FW-Elixir étend Core Elixir avec la négation dans les gardes, une fonctionnalité absente du calcul formel mais courante en pratique. Nous montrons comment *compiler* les gardes FW-Elixir vers Core Elixir en poussant les négations vers les feuilles via les lois de De Morgan, et en réencodant les relations absentes (`>=` devient `<` ∨ `==`).

Le chapitre détaille ensuite l'implémentation des règles d'application de fonctions, tant statiques que graduelles. Dans le cas graduel, la notion de *compatibilité* remplace le sous-typage strict : une application est compatible si elle peut réussir pour une certaine matérialisation du type dynamique. Cette vérification exploite le domaine graduel et les bornes extrémales pour éviter les faux positifs tout en préservant la flexibilité.

Chapitre 14 : Évaluation et déploiement Le système de types est intégré dans Elixir depuis la version 1.17 (juin 2024). La v1.17 couvrait atomes et maps ; la v1.18 ajoutait l'inférence de types pour les patterns et les retours de fonctions ; la v1.19 actuelle couvre tous les constructs du langage, incluant tuples, listes, fonctions multi-arité et *protocols* (l'équivalent Elixir des type classes).

Nous évaluons le système sur cinq bases de code majeures : Remote (plus d'un million de lignes), Livebook, Credo, Phoenix et Hex. Sur v1.18, le type-checking représentait 1,6 % à 9,3 % du temps de compilation total. Sur v1.19, malgré l'ajout des types fonctions et de l'inférence, qui rajoute énormément de vérifications de types supplémentaires, les performances se sont maintenues grâce aux optimisations des TDD.

Le retour des développeurs est encourageant : le système détecte du code mort et des erreurs de type sans requérir d'annotations syntaxiques. Des bugs ont été découverts dans des projets majeurs comme Phoenix (utilisé par plus de 14 600 sites), Postgrex et Flame, des erreurs qui avaient survécu aux tests et à la production pendant des années. Comme le résume José Valim : « Elixir is, officially, a gradually typed language. »

Cette seconde partie illustre qu'un système de types fondé sur une théorie robuste peut être implémenté efficacement et déployé avec parcimonie dans un langage (dynamique) de production, sans perturber les pratiques établies. Les types ensemblistes deviennent progressivement un outil quotidien pour les développeurs Elixir, malgré leur complexité théorique et grâce à leur simplicité d'usage.

Contents

List of Figures

NOTATION

Notation conventions

$\overline{x}$	Sequence notation: $x_1, \ldots, x_n$ (overbar denotes finite sequences).
$[i..j]$	Integer interval $\{i, i+1, \ldots, j\}$ (inclusive).

Chapter 3

$\mathscr{T}_{\textbf{static}}$	Set of regular static types.
$\mathbb{0}$	Bottom type, denoting no values.
$\mathbb{1}$	Top type, containing all well-typed values.
b	Base type such as `integer()`, `boolean()`, `atom()`, ...
`int, bool, atom`	Common base types: all integers, booleans, atoms.
`tuple`	Type of all tuples.
`pair`	Type of all pairs (2-tuples).
`base`	Union of all base types.
`arrow`	Type of all arrow types ($= \mathbb{0} \to \mathbb{1}$).
t	*Static* (non-gradual) types; elements of $\mathscr{T}_{\text{static}}$.
$t_1 \vee t_2$	**Union type** (`or`): values in at least one of t_1 or t_2.
$t_1 \wedge t_2$	**Intersection type** (`and`): values in both t_1 and t_2.
$\neg t$	**Negation type** (`not`): values that are *not* of type t.
$\mathscr{D}$	Semantic domain of all values.
d	Arbitrary element of $\mathscr{D}$.
$\mho$	Distinguished undefined input for function domain semantics.
Ω	Runtime error or stuck computation.
$\mathscr{D}_\Omega$	Auxiliary domain for membership induction ($= \mathscr{D} \cup \{\Omega\}$).
$\mathscr{D}_\mho$	Extended domain with undefined input ($= \mathscr{D} \cup \{\mho\}$).
δ	Arbitrary element of $\mathscr{D}_\Omega$.
ι	Arbitrary element of $\mathscr{D}_\mho$.
$d : t$	Predicate: d interprets a value of type t.
$[\![t]\!]$	Set-theoretic interpretation of t: $\{d \in \mathscr{D} \mid d : t\}$.

$t_1 \leq t_2$	Semantic subtyping (Definition 3.1.7): $[\![t_1]\!] \subseteq [\![t_2]\!]$.
$\mathbf{?}$	The gradual type `dynamic()`; denotes an unknown type known only at run time.
$\mathscr{T}_{\mathbf{gradual}}$	set of gradual types with $?$.
τ	*Gradual* types; elements of $\mathscr{T}_{\text{gradual}}$.
$\tau_1 \leq \tau_2$	Subtyping: $\tau_1{}^{\Downarrow} \leq \tau_2{}^{\Downarrow}$ and $\tau_1{}^{\Uparrow} \leq \tau_2{}^{\Uparrow}$.
$\tau_1 \simeq \tau_2$	Type equivalence: $\tau_1 \leq \tau_2$ and $\tau_2 \leq \tau_1$.
$\tau_1 \preccurlyeq \tau_2$	Precision: τ_1 materializes into τ_2 by refining some $?$ occurrences.
$\tau_1 \mathrel{\widetilde{\leqq}} \tau_2$	Consistent subtyping: some materializations satisfy $\tau'_1 \leq \tau'_2$.
$\tau^{\Downarrow}$	Minimal static materialization (guaranteed runtime values) of τ.
$\tau^{\Uparrow}$	Maximal static materialization (possible runtime values) of τ.
$\mathtt{dom}(t)$	Domain operator: argument types that function type t accepts.
$t \circ s$	Application result operator: type resulting from applying function t to argument s.
$\pi_i\, t$	Projection operator: i-th component type of tuple type t.
Chapter 4	
$e[v/x]$	Capture-free substitution: expression e with x replaced by v.
$\mathtt{fv}(e)$	Free variables of expression e.
$\mathtt{dom}(\Gamma)$	Domain of type environment Γ (set of variables).
$\mathscr{E}$	Evaluation context (expression with a hole).
$\mathbb{I}$	Finite interface, set of arrow types $t_i \to t'_i$.
$\lambda^{\mathbb{I}} x.e$	Lambda abstraction annotated by interface $\mathbb{I}$.
$e(\overline{e})$	Multi-arity function application.
$\{\overline{e}\}$	Tuple construction.
$\pi_{e_1}\, e_2$	Tuple projection; e_1 yields index applied to tuple e_2.
$\mathtt{case}\ e\,(\rho_i \to e_i)_{i\in I}$	Case expression for pattern matching.
$\{\overline{t}\,, ..\}$	Open-tuple type: tuples prefix-typed by $\overline{t}$, arbitrary suffix.
$(t_1, \dots, t_n) \to t$	Multi-arity (n) function type.
$t \circ \overline{s}$	n-ary application result operator.
$\mathtt{fun}$	Type of all functions.
$\mathtt{fun}_n$	Type of all functions of arity n.
ω_p	Failure state labelled p for a specific runtime error.
ρ	Test type allowed in type-case guards.
$\Gamma \vdash_{\mathsf{s}} e : t$	Static typing judgement.
$\hookrightarrow$	Expression reduction relation.
Chapter 5	
$(t_1 \to t_2)^{\star}$	Strong arrow; if it returns, result is in t_2, else fails/diverges.

$\Gamma \vdash_{\mathbf{g}} e : t$	Gradual typing judgement.
$\Gamma \vdash_{\mathbf{w}} e \mathrel{\overset{\circ}{\circ}} t$	Weak typing judgement respecting BEAM semantics.
$(t \Rightarrow)$	Strict-domain constructor: inputs $\notin t$ trigger immediate runtime failure.
$(\Rightarrow s)$	Strict-codomain constructor: every non-error output lies in s.
Chapter 6	
p	Pattern (constants, variables, tuples).
g	Guard (boolean expressions in pattern matching).
a	Guard atom (simplified expressions usable in guards).
$p \;\texttt{when}\; g$	Pattern-guard pair combining pattern p with guard g (syntactic sugar for pg).
v/p	Environment from matching value v with pattern p, or `fail` on mismatch.
$v/(pg)$	Match value v against pattern-guard pair pg; yields env or `fail`.
$\texttt{fail}$	Matching failure token.
$a \mathbin{\texttt{?}} \rho$	Type test in guards: atom a is of test type ρ.
$g_1 \;\texttt{and}\; g_2$	Guard conjunction (boolean `and`).
$g_1 \;\texttt{or}\; g_2$	Guard disjunction (boolean `or`).
$<_{\textbf{term}}$	Total order on Elixir values.
$\texttt{size}\, a$	Guard atom computing tuple size.
$\pi_a\, a'$	Guard atom projecting component a' from tuple atom a.
$\hookrightarrow$	Guard reduction relation.
$\texttt{true}, \texttt{false}$	Boolean values in guard evaluation.
$\texttt{tuple}^{i}$	Tuples of i elements of any type.
$\texttt{tuple}^{>i}$	Tuples with at least $i+1$ elements.
$\int pg \wr$	Potentially accepted type of pattern-guard pair pg.
$\wr pg \int$	Surely accepted type of pattern-guard pair pg.
${}^{t}/_{p}$	Environment mapping pattern variables when scrutinee has type t.
$\wr p \int_{\Gamma}$	Type accepted by pattern p under environment Γ.
$\Gamma[x \mathrel{\hat{=}} t]$	Γ with x refined to $\Gamma(x) \wedge t$.
$\Gamma[x \mathrel{\hat{=}} t]_p$	Refinement above, propagated via dependencies of pattern p.
$\mathfrak{b}, \mathfrak{c}$	Precision flags: true if the guard is precise, false if it is imprecise.
$(\Delta, \mathfrak{b})$	Denotes non-error/success conditions for a guard, with precision flag $\mathfrak{b}$.
ω	Denotes an always-erroring guard.
$\mathscr{R}$	Guard result: list of pairs $\{(\Phi, \mathfrak{b})\,;\,(\Delta, \mathfrak{c})\}$, or ω, denoting non-error/success conditions.
$\mathscr{F}$	Failure result: special $\mathscr{R}$ denoting an always-erroring or always-failing guard.
$\Gamma; p \vdash g \mapsto \mathscr{R}$	Guard analysis judgment under pattern p.
$\Gamma \vdash g \mapsto \mathscr{R}$	Guard analysis judgment (pattern implicit).

$\Gamma; t \vdash \overline{pg} \rightsquigarrow \overline{(s, \mathfrak{b})}$	Pattern-guard sequence analysis with input type t.

Chapter 9

`Descr`	Record of atomic-type flags composing a static set-theoretic type.
$[\, t_1 \,\mathbf{,}\, t_2 \,]$	Ordered pair encoding gradual type's static and dynamic bounds.
$\langle t_1, t_2 \rangle$	Disjoint pair storing static part and extra dynamic-only part.
$\mathscr{B}$	Binary decision diagram representing boolean combinations of arrow literals.
$\mathscr{T}$	Ternary decision diagram for lazy union handling.

Chapter 11

$n \,?\, C \,\mathbf{:}\, D$	BDD node: test n, then branch C (true) or D (false).
$n \,?\, C \,\mathbf{:}\, U \,\mathbf{:}\, D$	TDD node: test n, then branch C (constrained), U (uncertain/union), or D (dual).
Φ_k	TDD representing atomic function types of arity k.
{`:open`, `elements`}, {`:closed`, `elements`}	Tuple representation: open (allows extra elements) or closed (exact elements).

Chapter 12

DNF	Disjunctive Normal Form: union of conjunctions of positive and negative atomic types.
$\mathscr{M}$	DNF notation: $\bigvee_{i=1}^{n}(L_i, N_i)$ where (L_i, N_i) are clause pairs.
(L, N)	DNF clause: L intersected positive atomic types, N intersected negative atomic types.
Union form	Representation as union of atomic types (negations eliminated).

Chapter 13

`E`	FW-Elixir expressions.
$\mathbf{T}_{\mathsf{G}}$	Guard compilation function: $\mathscr{G}_{\text{Elixir}} \to \mathscr{G}_{\text{Core}}$.
$\mathbf{N}_{\mathsf{G}}$	Guard negation compilation function (pushes negation into guards).

Chapter 1
Introduction

Elisir di sì perfetta,
di sì rara qualità,
ne sapessi la ricetta,
conoscessi chi ti fa!

Donizetti - *L'elisir d'amore*

1.1 Typing Elixir: the case for set-theoretic types

Elixir (2012) is a dynamic, functional language designed for building scalable and maintainable applications. It runs on the Erlang Virtual Machine called BEAM (2024), a platform renowned for low-latency, distributed, and fault-tolerant computing. Elixir combines Ruby-inspired syntax with Erlang's battle-tested concurrency model, offering a productive development experience while inheriting the robust actor-based architecture where lightweight processes communicate via message passing.

Elixir has gained significant adoption across web applications, embedded systems, data processing, and distributed systems, with notable users including Discord and PepsiCo. This success stems from the underlying BEAM platform, developed by Ericsson in the 1980s as a remarkable achievement in concurrent, distributed, and fault-tolerant computing. Elixir extends this foundation with higher-level constructs–such as protocols for polymorphism, a powerful macro system for metaprogramming and creating domain-specific languages, an enhanced standard library and set of frameworks, while maintaining the efficient, resilient runtime characteristics of BEAM.

The Erlang VM (BEAM)

The BEAM is documented in Erlang's original design and rationale (2003) and is also practically explained in later Erlang/OTP literature (2009). It provides four key capabilities that make it exceptional for building robust, concurrent systems:

- **Actor-based Concurrency:** The BEAM employs an actor model where lightweight processes have isolated memory and communicate via asynchronous message passing. This enables highly concurrent applications. Each process has its own garbage-collected heap, minimizing pause times and ensuring fault isolation.
- **Fault Tolerance:** Embracing a "let it crash" philosophy, BEAM applications are structured as supervised process hierarchies. If a process encounters an error, it terminates cleanly and its supervisor automatically restarts it. This prevents cascading failures and contributes to system resilience.
- **Transparent Distribution:** BEAM has built-in support for distribution, allowing processes on different nodes to communicate seamlessly. This facilitates horizontal scaling and distributed computing without additional complexity for the developer.
- **Hot Code Swapping:** The platform supports updating code in a running system without downtime, which is crucial for high-availability applications that require on-the-fly upgrades.

These features make BEAM an ideal platform for domains like web services, telecommunications, and embedded systems that require both reliability and concurrency.

Note

These same features also pose challenges for static typing. For example, hot code swapping allows any module to be replaced at runtime, bypassing compile-time type safety guarantees. While this specific scenario is beyond the scope of our work (hot code swapping is rarely used in practice), there remains substantial value in performing static type checks during development.

Elixir's Design and Features

Building on Erlang and BEAM, Elixir adds its own syntax, tools, and conventions focused on developer productivity:

- **Approachable Syntax & Metaprogramming:** Elixir's syntax is inspired by Ruby, making it familiar and approachable. It includes a powerful macro system that operates on the Abstract Syntax Tree (AST) at compile time, enabling developers to manipulate the language's syntax to implement their own language constructs.
- **Functional Programming Paradigm:** Elixir emphasizes immutability and pure functions, simplifying concurrent programming by eliminating many common sources of race conditions.
- **Pattern Matching:** Pattern matching is pervasive in Elixir (in function definitions, case expressions, etc.), allowing concise and declarative handling of different data shapes. It

not only leads to expressive code but also provides valuable structural information that a type system can leverage for exhaustiveness checking and type inference.

- **Protocols (ad-hoc polymorphism).** Elixir protocols are the analogue of *interfaces* (Java, C#, Go), *traits* (Rust, Scala), *protocols* (Swift/Obj-C), and *type classes* (Haskell): they declare a required API that unrelated types may implement, with method selection by the dynamic type of the argument (single-dispatch). For example, `String.Chars` requires `to_string/1` (cf. Go's `fmt.Stringer`, Rust's `Display`); `Enumerable` abstracts iteration (cf. Java `Iterable`, Rust `IntoIterator`, Swift `Sequence`). The type system checks conformance and rejects calls on non-implementers.
- **Ecosystem and Tooling:** Elixir boasts a vibrant ecosystem and strong tooling. The Mix build tool manages project compilation, testing, and dependencies; Hex provides package management. The community has produced many libraries and frameworks (notably the Phoenix web framework). Any type system must be introduced in a way that works harmoniously with these existing tools and practices.

The interested reader will find a more detailed overview of these different aspects in Appendix A.

Elixir Importance

Elixir today has an active community of approximately 60,000 professional developers, and a large body of open-source Elixir code is available on platforms like GitHub. Its growing adoption and the scale of code in production underscore the importance of reliability and maintainability. As projects grow, developers increasingly desire stronger guarantees from the language. This context makes the case for adding a robust static type system to Elixir both significant and urgent.

The Challenge of Typing Elixir

Like Erlang, Elixir is dynamically typed: types of values are checked only at runtime. This offers flexibility and rapid prototyping but can lead to runtime errors that a static type system might catch at compile time. As Elixir applications grow in size and complexity, the lack of compile-time type checking becomes a more pressing concern for reliability, maintainability, and refactoring safety.

Developing a static type system for Erlang (and by extension Elixir) has been an open research problem for nearly two decades. Early efforts, such as Marlow and Wadler's work (1997) on typing a subset of Erlang, explored subtyping constraints but suffered from slow type inference and overly complex inferred types, preventing adoption. Lindahl and Sagonas (2006) introduced Dialyzer, a static analysis tool for Erlang based on success typings. Dialyzer (and its Elixir integration via Typespec annotations) can detect certain discrepancies between code and spec annotations, and importantly, it guarantees no false positives. However, this no-false-positive approach comes at the cost of weak guarantees: Dialyzer may miss many bugs (reporting possible issues only when it's certain a mistake exists). Additionally, Dialyzer can be slow on large codebases and often produces cryptic warnings, limiting its usefulness and adoption in the Elixir community.

Several subsequent attempts have tried to bring static typing to the BEAM ecosystem. For example, Valliappan and Hughes (2018) developed an experimental Hindley-Milner style type system for Erlang, but it struggled to naturally express the union and intersection types that frequently arise from Erlang's idioms (like pattern matching with multiple clauses). More recent projects include alternative languages or compilers targeting BEAM, such as Gleam (2019) (a statically typed language for BEAM), and compilers like Hamler (2021) and Caramel (2022) which bring Haskell/OCaml-like typing to BEAM. While these show there is strong interest in static analysis for Erlang/Elixir, they typically do not preserve Elixir's advanced features (like macros and protocols) nor its established ecosystem.

Pattern matching is another central feature of Erlang/Elixir that poses both a challenge and an opportunity for typing. A static type system could provide exhaustiveness checking for pattern matches (warning if some cases are not handled) and unreachable code detection, improving code robustness and design. The Elixir team has expressed the goal of eventually making type information a first-class part of the language–beyond mere annotations for documentation–enabling new idioms and more expressive, safe code for those who opt in.

Since Elixir's semantics are largely a superset of Erlang's (Elixir adds features like protocols and macros on top of Erlang), any successful type system for Elixir would likely apply to Erlang and other dynamically typed BEAM languages as well. The challenge, therefore, lies in introducing static typing without sacrificing the flexibility and productivity that dynamic Elixir developers enjoy. Our approach to this challenge involves two key ideas:

Set-Theoretic Types: Types that are defined by the set of values they contain, supporting rich type constructions like unions, intersections, and negations. These allow precise modeling of complex data domains and function behaviors typical in dynamic languages.

Gradual Typing: A *gradual* type system allows mixing static and dynamic typing, so developers can add types incrementally. Untyped parts of the program can interoperate with typed parts safely, and over time more code can be shifted into the typed regime. This is critical for adoption, as it enables a smooth transition from untyped to fully typed code.

From a theoretical perspective, our work builds on the concept of **semantic** (set-theoretic) types. In this approach, each type is defined as the set of values that belong to that type. Consequently, type operators correspond to set operations: a union type t_1 `or` t_2 denotes the union of the sets of values of t_1 and t_2, an intersection type t_1 `and` t_2 denotes the set of values common to both t_1 and t_2, and a negation type `not` t denotes the set of all values not in t. This interpretation aligns well with the patterns in dynamic languages (2023). For instance, **union types** naturally describe values that may come from multiple alternative types (a very common scenario in dynamically-typed code where functions handle a variety of inputs). **Negation types** allow a precise description of values excluded by certain conditions (e.g., in a series of pattern-matching clauses with a default case, a negation type can represent "all values not handled by previous clauses"). **Intersection types** can capture function overloading and flow-sensitive behaviors by tying specific input types to specific output types. An appealing aspect of this approach is that these set-theoretic type constructs can often be explained to programmers in intuitive terms (sets of values), which may ease the learning curve. We believe that combining set-theoretic types with gradual typing offers a promising path to bringing static guarantees to Elixir while preserving its dynamic spirit.

A Gradual Set-Theoretic Type System for Elixir

We present a gradual type system for Elixir based on the framework, introduced in Castagna and Frisch (2005) and Frisch et al. (2008), of **semantic subtyping** (also known as set-theoretic types). This framework, originally developed for the ℂDuce (2003) programming language, introduced in Benzaken et al. (2003), supports types defined by set operations – including unions, intersections, and negations – which obey the familiar commutativity, associativity, and distributivity laws of set theory. Our type system also features parametric polymorphism with local type inference: functions can be explicitly annotated with type variables, but when they are called, the compiler infers how to instantiate those variables, eliminating the need for verbose type applications at call sites.

Moreover, the system includes a precise treatment of Elixir's **patterns and guards**, by using a **type narrowing** mechanism. As a program branches on a condition (for example, a pattern match or an `is_integer` guard), the type of the tested expression (and related variables) is refined in each branch based on the outcome of the test. This lets us precisely type pattern-matching code and guards (the boolean conditions in clauses) by understanding how they filter down the set of possible values.

Compared to the earlier semantic subtyping work for CDuce, our Elixir type system contributes several new elements and adaptations to address Elixir-specific challenges:

- **Multi-arity function types:** We extend the semantic subtyping framework to account for Elixir/Erlang's notion of function arity. Unlike CDuce (where every function is unary, taking a single argument that could be a tuple), Elixir functions have a fixed number of parameters. We introduce a special function type syntax that explicitly reflects arity, e.g. $(t_1, \ldots, t_n) \to t$, and adapt the subtyping rules accordingly.
- **Guard analysis:** We develop a typing technique to analyze guards (the boolean expressions in pattern matches and after the when keyword in function definitions). Our system interprets guard conditions as type constraints, refining static types within each branch of a conditional. This enables **exhaustiveness** and **redundancy** checks for pattern-match clauses.
- **Records and dictionaries (maps):** We provide a unified type discipline for Elixir's **maps**, covering both their use as fixed-shape records (with a predetermined set of keys) and as open dictionaries (with arbitrary keys of a certain type). This includes tracking which keys are required to exist in a map and the types of their values, versus keys that may or may not be present.
- **Dynamic type integration:** We incorporate a *dynamic* type noted `dynamic()` to represent parts of the code that remain untyped. This is in line with gradual typing principles: the dynamic type allows untyped Elixir code to interoperate with typed code with runtime

checks, and we ensure that gradually adding types does not break program behaviour (often called the gradual guarantee).

- **Strongly-typed function arrows:** We introduce a concept we call **strong arrows**, a gradual typing technique that accounts for runtime type checks (for example, guards or pattern matches inside function bodies) in the static function type. Strong arrows allow the type system to be sound (no false assurances about the absence of type errors) while also being precise, without needing to alter the compiled code. In the terminology of Garcia et al. (2021), this achieves *safe erasure* in gradual typing: we do not change program semantics to accommodate types, yet we retain strong type guarantees.

In summary, our system is a combination of several existing ideas (set-theoretic types, polymorphism with local inference, type narrowing, gradual typing) and novel techniques (guard analysis, strong arrows, and an enhanced map type system). We believe that no single "silver bullet" technique is sufficient for typing a language as feature-rich and dynamic as Elixir. Instead, it is the *combination* of these techniques that allows our type system to model Elixir idioms precisely. By addressing the various facets of the language, we aim to deliver a type system that is expressive enough to be useful and palatable to Elixir developers, increasing the chances of community acceptance and adoption. (Notably, because Elixir's semantics are a superset of Erlang's, our system effectively types Erlang as well, covering all Erlang idioms in addition to Elixir's extensions.[1])

1.1.1 An Overview of our Type System

Static typing appears to be the biggest unmet need voiced by the Elixir community in recent years. Today, Elixir provides Typespecs (2023) for optional type annotations – developers can write specifications (using a syntax in comments or @spec attributes) indicating the expected types of function arguments and return values, and define named types (type aliases). However, these specifications are not enforced by the Elixir compiler. Instead, developers rely on Dialyzer to perform a separate analysis pass to check for certain inconsistencies between code and specs.

While Typespecs and Dialyzer are valuable (especially as documentation), they fall short of the full guarantees and developer experience offered by a true static type system integrated into the language. Dialyzer's **success typing** approach, which avoids false positives, means that many true type errors (situations where one function passes a value to another that will cause a runtime error) might go undetected. Based on community feedback and our own experience, many developers would prefer a stricter system–even if it *occasionally produces false positives*–as long as it catches more actual bugs and offers clearer feedback.
In essence, Elixir developers are asking for types to serve two main purposes:

- **Documentation and Readability:** Type annotations can greatly aid in documenting code behaviour. A function's type signature succinctly conveys what kinds of inputs it expects

[1]Semantically, Elixir is a superset of Erlang with the addition of protocols and macros. Indeed, the Elixir compiler works by producing Erlang Abstract Format (AST). Thus, by typing all Elixir idioms, we also cover all Erlang idioms.

and what it returns, which helps both the original author and others who read the code later. (Elixir already benefits here from Typespecs; a built-in type system would make these guarantees stronger and always up-to-date with the code, since types would be checked as part of compilation.)

- **Interface Contracts and Refactoring Safety:** Perhaps the most compelling benefit of static types is to enforce contracts between different parts of a codebase. If function `caller(arg)` calls another function `callee(arg)`, a static type checker can guarantee that as the code evolves, `caller` will only ever pass arguments of the right type to `callee`, and that `caller` correctly handles the return type that `callee` produces. This is especially valuable in large projects or frameworks, where many functions interact; it helps catch integration errors early and makes large-scale refactoring much safer.

To illustrate the second point, consider a simple example in Elixir:

```elixir
$ integer() -> integer()
def negate(x) when is_integer(x), do: -x
```

The `negate` function receives an `integer()` and returns an `integer()`[2]. Type specifications are prefixed by `$` and each specification applies to the definition it precedes. With our custom negation in hand, we can implement a custom subtraction:

```elixir
$ (integer(), integer()) -> integer()
def subtract(x, y) when is_integer(x) and is_integer(y) do
  x + negate(y)
end
```

This would all work and typecheck as expected, as we are only working with integers.

Now, imagine in the future someone decides to make `negate` polymorphic (here, *ad hoc* polymorphic), by including an additional clause so it also negates booleans:

```elixir
$ (integer() or boolean()) -> (integer() or boolean())
def negate(x) when is_integer(x), do: -x
def negate(x) when is_boolean(x), do: not x
```

The specification at issue uses `integer() or boolean()` stating that both the argument and the result are either an integer or a boolean. This is a union type which has become common place in many programming languages.

However, the type specified for `negate` is not precise enough for the type system to deduce that when `negate` is applied to an integer the result is also an integer.

```
Type warning:
  |  def subtract(x, y) when is_integer(x) and is_integer(y) do
  |    x + negate(y)
```

[2]We follow Erlang convention that basic types are suffixed by "( )", for instance, `string()`.

```
        ^ the operator + expects integer(), integer() as arguments,
          but the second argument can be integer() or boolean()
```

A type system with union types only would not be enough to capture many of Elixir idioms, and it would probably lead to too many false positives. Therefore, in order to evolve contracts over time, we need more expressive types. In particular, to solve this issue we need an *intersection type*, which specifies that `negate` has both type

`integer()->integer()`

(i.e., it is a function that maps integers to integers) *and* type `boolean()->boolean()` (i.e., it is a function that maps booleans to booleans). This type is more precise than the previous one and is written as:

```
$ (integer() -> integer()) and (boolean() -> boolean())
```

With this type, the type checker can infer that applying `negate` to an integer will return an integer. Therefore, in the definition of `subtract`, the application `negate(b)` has type `integer()`, and the function `subtract` is well-typed.

1.1.2 Set-Theoretic Types and Subtyping Relation

Unions, intersections, and (see later on) negations are called *set-theoretic* types, insofar as they can be thought of in terms of sets: if we think of a type as the set of all values of that type (e.g., `integers()` as the set of all integer constants, `boolean()` as the set containing just `true` and `false`, ...), then the union of two types is the set that contains the union of their values (e.g., a value of type `integer() or boolean()` is either an integer value or a boolean value), the intersection of two types is the set that contains the values that are in both types (e.g., a value in the intersection `(integer()->integer()) and (boolean()->integer())` is a function that both maps integers to integers and maps booleans to booleans), and, finally, the negation of a type is its complement, that is, it contains all the (well-typed) values that are not in the type (e.g., a value in `not integer()` is any value that is not an integer).

Notice that an intersection of arrows does not necessarily correspond to multiple definitions of a function. For instance, the following definition is well-typed:

```
$ (integer() -> integer()) and (boolean() -> boolean())
def negate_alt(x), do: (if is_integer(x), do: -x, else: not x)
```

We have seen that we can specify two different types for `negate`, that is:

(1) `(integer() or boolean()) -> (integer() or boolean())`
(2) `(integer() -> integer()) and (boolean() -> boolean())`

and we said that the latter type is “more precise” than the former. Formally, we state that the latter is a *subtype* of the former, meaning that every value of the latter is also a value of the former. In the case of the two types above, the subtyping relation is also *strict*: every function that maps integers to integers and booleans to booleans, is also a function that maps an integer or a boolean to an integer or a boolean, but not vice versa. For example, the constant function `fn x -> 42 end` maps both integers and booleans to integers and thus to `integer() or boolean()`; as such it is a function of the type in (1). However, it does not map booleans to booleans. Therefore, it is not in the intersection type in (2). When two types are one subtype of each other they are said to be *equivalent*, since they denote the same set of values (e.g., `(integer()->integer()) and (boolean()->integer())` is equivalent to

`(integer or boolean())->integer()` .

The type of `negate` or `negate_alt` can also be expressed without intersections, by using parametric bounded quantification[3], but this is seldom the case. For instance, Elixir provides a negation operator named `!`, which is defined for all values. The values `nil` and `false` return `true`, while all other values return `false`. With set-theoretic types, we can give to this operator the following intersection type:

```
18 $ (false or nil -> true) and (not (false or nil) -> false)
```

This type introduces two further ingredients of our type syntax: singleton types and negation types. Namely, the atoms `true`, `false`, and `nil`[4], are also types, called singleton types, because they contain only the constant/atom of the same name. The connective `not` denotes the negation of a type, that is, the type that contains all the well-typed values that are not in the negated type, whence the interpretation of the functional type above.[5]

The advantage of interpreting types as the set of their values is that types satisfy the distributivity and commutativity laws of their set-theoretic counterparts.

For instance, a well-known property of products is that unions of products with a same projection factorize, that is, `{s1,t} or {s2,t}` is equivalent to `{s1 or s2, t}` (Elixir uses curly brackets for products). This is reflected by the behaviour of our type-checker that accepts the following definitions:

```
19 $ t() = {integer() or string(), boolean()}
20 $ s() = {integer(), boolean()} or {string(), boolean()}
21 $ ((t() -> t()), s()) -> s()
22 def apply(f,x) do: f.(x)
```

The first two lines define the types `t()` and `s()` while lines 21-22 define a function whose typing demonstrates that the type-checker considers `t()` and `s()` to be equivalent. This is because it allows an expression of type `t()` to be used where an expression of type `s()` is expected (i.e., `f` which expects an argument of type `t()` is given an argument `x`, which is of type `s()`) and an expression of type `s()` where an expression of type `t()` is expected (i.e., the type specification

[3]Precisely as `$ a -> a when a: integer() or boolean()`: see Section 1.1.3.

[4]In Elixir, atoms are user-defined constants obtained by prefixing an identifier by colon, as in `:ok`, `:error`, and so on. The atoms `true`, `false`, and `nil` are supported without colon for convenience.

[5]The precedence of `and` and `or` is higher than type constructors (arrows, tuples, records, lists), and the negation `not` has the highest precedence of them all.

declares that `apply` returns a result of type `s()`, but the body returns `f(x)` which is of type `t()`). In contrast, languages that use a syntactic definition of subtyping, such as Typed Racket, Flow, or TypeScript, accept the application `f(x)` but reject the typing of `apply`: they cannot deduce that `t()` is a subtype of `s()`.

Finally, we adopt the Typespec conventions wherein `term()` represents the top type (i.e., the type of *all* values) and `none()` denotes the *empty* type, that is, the type that has no value and which is equivalent to `not term()` (likewise, `term()` is equivalent to `not none()`).

1.1.3 Applying Set-Theoretic Types to Elixir

The existing set-theoretic types literature enables our type system to represent several Elixir idioms. We outline some examples in this section.

Nullability Elixir supports null values via the `nil` atom. Thanks to union and singleton types, an `integer()` argument of a function may become nullable by specifying it as `integer() or nil`.

Parametric Polymorphism with Local Type Inference Set-theoretic type-systems feature parametric polymorphism with local type inference: expressions (in particular functions) can be given types containing type variables, but to use them it is not necessary to specify how to instantiate these variables, since the system deduces it–see Castagna et al. (2014) and Castagna et al. (2015).

In our implementation, type variables are identifiers that are quantified by using a postfix `when` in which variables come with their upper bound.[6] Type variables are distinguishable from basic types, since they are *not* suffixed by "`()`". We feature only first order polymorphism (in the terminology of Pierce/TAPL and Harper/PFPL: rank-1, prenex "let-polymorphism"), so `when` can only occur outside a type (never inside it).

The `map` and `reduce` operations over lists are good examples of need for polymorphic types, since most of the functions working with collections (known as "enumerables" in Elixir) cannot be sensibly typed without them. For instance, we have

```
$ ([a], (a -> b)) -> [b] when a: term(), b: term()
def map([h | t], fun), do: [fun.(h) | map(t, fun)]
def map([], _fun), do: []
```

```
$ ([a], b, (a, b -> b)) -> b when a: term(), b: term()
def reduce([h | t], acc, fun), do: reduce(t, fun.(h, acc), fun)
def reduce([], acc, _fun), do: acc
```

[6]We did not specify lower bounds since they are not frequently used and they can be encoded by union types, e.g., $\forall(s \leq \alpha).\alpha \to \alpha \overset{def}{=} \forall(\alpha).(s \vee \alpha \to s \vee \alpha)$; upper bounds can be encoded, too, this time by intersections (e.g., $\forall(s \leq \alpha \leq t).\alpha \to \alpha \overset{def}{=} \forall(\alpha).((s \vee \alpha) \wedge t) \to (s \vee \alpha) \wedge t)$), but their frequency justifies the introduction of specific syntax. The use of postfix `when` for variable quantification is borrowed from Typespec.

meaning that for all types `a` and `b` (i.e., for all `a` and `b` subtypes of `term()`):

- `map` is a binary function that takes a list of elements of type `a` (notation `[a]`), a function from `a` to `b` and returns a list of elements of type `b`;
- `reduce` is a ternary function that takes a list of `a` elements, an initial value of type `b`, a binary function that maps `a`'s and `b`'s into `b`'s, and returns a `b` result.

Local type inference infers that for `map([1, 4], fn x -> negate(x) end)` both type variables must be instantiated by `integer()`, deducing the type `[integer()]` for it.

Intersection can also be used to define the type specification of `reduce` for the case of empty lists (in which case the third argument can be of any type):

```
29 $ (([a] and not [], b, (a, b -> b)) -> b) and
30   (([], b, term()) -> b) when a: term(), b: term()
```

Polymorphic types make inference more precise for other functions. For instance, if we add a default case to the `negate` example (lines 8-9) we obtain the code

```
31 def negate(x) when is_integer(x), do: -x
32 def negate(x) when is_boolean(x), do: not x
33 def negate(x), do: x
```

for which we can deduce, or at least check, the type (notice the use of bounded quantification in line 36)

```
34 $ (integer() -> integer()) and
35   (true -> false) and (false -> true) and
36   (a -> a) when a: not(integer() or boolean())
```

and thus deduce for some function such as

```
37 def foo(x) when is_atom(x), do: negate(x)
```

the type `atom() -> atom()`, since an atom is neither an integer nor a Boolean.

It is possible to define polymorphic types with type parameters. For instance, we can define the type `tree(a)`, the type of nested lists whose elements are of type `a`, as

```
38 $ type tree(a) = (a and not list()) or [tree(a)]
```

and then use it to type the polymorphic function `flatten` that flattens a `tree(a)` returning a list of `a` elements:

```
39 $ tree(a) -> [a] when a: term()
40 def flatten([]), do: []
41 def flatten([x | xs]), do: flatten(x) ++ flatten(xs)
42 def flatten(x), do: [x]
```

The function above is well-typed. The three clauses of its definition are exhaustive (the last one captures all the arguments not captured by the first two). The first clause returns an empty list (thus a value of type `[a]`). The second clause captures any argument that is a non-empty list of `tree(a)` elements (since these are the only lists the function can be applied to), therefore our system deduces that `x` is of type `tree(a)` and `xs` is of type `[tree(a)]`; since `[tree(a)]` is a subtype of `tree(a)` (the latter being defined as the union of the former with another type), then both subsequent applications of `flatten` are well typed; therefore, both return results of type `[a]` whose concatenation (noted `++`) yields against a result of type `[a]`. Finally, the inputs of type `tree(a)` that are not captured by the first two clauses, and are thus processed by the last clause, are just the arguments of type `a` that are not lists (from all inputs of type `tree(a)`, the first clause removes the empty lists, while the second clause removes all the non-empty lists of `tree(a)` elements); therefore the result of this clause is of type `[a and not list()]` and, by subtyping, of type `[a]`, too.

When `flatten` is applied, then the local type inference analyzes the argument, in order to determine the instantiation of the type of the function. If the argument is not a list, then `a` is instantiated to the type of the argument. If it is a list, then `a` is instantiated to the *union of the types of all the non-list elements of this nested list.* For instance, the type statically deduced for the application

```
flatten [3, "r", [4, [true, 5]], ["quo", [[false], "stop"]]]
```

is `[integer() or boolean() or binary()]` (where `binary()` is the type for strings).

Protocols Elixir supports a kind of polymorphism akin to Haskell's typeclasses, via *protocols*. A protocol defines a set of operations that can be implemented for any type. For example, the `String.Chars` protocol requires the implementation of the `to_string` function. This function can convert any data type to a human representation as long as an implementation of the `String.Chars` protocol (viz., of `to_string`) has been defined for that data type. The *union* of all types that implement `String.Chars` is automatically filled in by the Elixir compiler and denoted by `String.Chars.t()`. In the absence of set-theoretic types this union would be approximated by `term()`.

Protocols can combine with parametric polymorphism to define more expressive types, such as collections. In Elixir, lists, sets, and ranges are all said to implement the `Enumerable` protocol which can be represented by the type `Enumerable.t(a)`, that is, the enumerables whose elements are of type `a` (i.e., `a`-lists, `a`-sets, and `a`-ranges). So, for instance, the type of a generic `map` function that processes the elements of an enumerable will be:

```
$ Enumerable.t(a), (a -> b) -> Enumerable.t(b) when a: term, b: term
```

while the function that sums all elements of an enumerable of integer elements will have type `Enumerable.t(integer()) -> integer()`.

The power behind Elixir protocols is that they allow library authors to express requirements in their APIs that are decoupled from the implementation of those requirements. For example, one strength of Elixir is in developing web applications. Web applications often have to encode Elixir data structures into different formats, such as JSON, CSV, XML, etc. To decouple the data types from the encoding logic, the author of a web framework defines a protocol, such as `JSON.Encoder`, and states it can encode any data structure, as long as it implements the `JSON.Encoder` protocol.

With set-theoretic types, library authors can now combine protocols to build additional requirements. One business may require that all of their public data must be available in several different data formats. They can encapsulate this requirement by defining an intersection of existing protocols:

```
$ type export() = JSON.Encoder.t() and CSV.Encoder.t() and XML.Encoder.t()
```

A system with looser requirements may use a union instead of intersection: the data type must implement at least one of the formats in the union, in order to be accepted by the system.

1.1.4 Extending Semantic Subtyping for Elixir

All features presented so far adapt to Elixir what is already possible in the type system of ℂDuce, defined via the set-theoretic interpretation of types of semantic subtyping–see Frisch et al. (2008). There are however several key specific characteristics of Elixir that require the semantic subtyping framework to be modified, improved, and/or extended.

1.1.4 a) Tuple Types

Elixir has fixed-length tuples written with braces, e.g., `{:ok, 42}`. We mirror this at the type level with the product type syntax $\{t_1, \ldots, t_n\}$ which denotes the set of n-tuples whose k-th component has type t_k. This is the standard Cartesian product: a tuple belongs to $\{t_1, \ldots, t_n\}$ iff each component belongs to the corresponding t_k. In particular, if any t_k is `none()`, then $\{t_1, \ldots, t_n\}$ is itself `none()` (a product with the empty set is empty), while `{term(),...,term()}` is the type of all n-tuples. We keep braces `{...}` for tuple types (to align with Elixir values) and reserve parentheses `(...)` for the function-argument notation introduced next.

1.1.4 b) Function Arity

The arity of functions plays an important role in Elixir. While it is possible to test the arity of a function using the expression `is_function` (e.g., `is_function(foo, 2)` tests whether `foo` is a binary function), it is not possible in semantic subtyping to express the type of exactly all functions with a specific arity.[7] This is because, in ℂDuce, all functions are unary, with a function that takes two arguments being considered a unary function that expects a pair. Although it

[7]In our system, to be able to express arity tests in terms of types is crucial for the precise typing of guards and, thus, of functions and pattern matching: cf. Section 1.1.4 c).

is possible to define a type for all functions as `none() -> term()`,[8] it is not possible to give a type specifically for, say, binary functions using `{none(),none()} -> term()`, for the simple reason that a product of the empty set is equivalent to the empty set, and thus the latter type is equivalent to the former. To address this issue, we introduce a special syntax for function types, written as `(`t_1`,`$\cdots$`,`t_n`) -> t` which outlines the arity of the functions and that we already used in the previous examples. This allows the type of all binary functions to be written as `(none(),none()) -> term()`.

However, this requires a non-trivial modification in the set-theoretic interpretation of function spaces: after defining the interpretation of multi-arity functions, subtyping is reframed as a set-containment problem. Solving this problem then produces the decision algorithm for subtyping (see Section 4.4.1 for further details).

1.1.4 c) Guards and Pattern Matching

Another characteristic of Elixir that is not captured by the current research on semantic subtyping is the extensive use of guards both in function definitions and pattern matching.

In the previous examples, we have explicitly declared the type signature of all functions we defined, such as:

```
$ integer() -> integer()
def negate(x) when is_integer(x), do: -x
```

However, our type system is capable to infer the types of functions as the above even in the absence of their type declaration, by considering guards as explicit type annotations for the respective parameters. This not only applies to simple type tests of the parameters as it is the case of `negate/1` in which the guard tests whether the parameter `x` is an integer, but also to more complex tests. For example, for

```
def get_age(person) when is_integer(person.age), do: person.age
```

our system deduces from the guard that `person` must be a record with at least the field `age` defined, and containing a value of type `integer()`, that is, an expression of type `%{..., age: integer()}`. This is a record type: records in Elixir are prefixed by `%` to distinguish them from tuples; the three dots indicate that the record type is open, that is, it types records where other fields may be defined (cf. Section 1.1.5). Since the dot in `person.age` denotes field selection, then our system deduces for the function `get_age` the type `%{..., age: integer()} -> integer()`. One novelty of our system is that it can precisely express (most) guards in terms of types, in the sense that the set of values that satisfy a guard is the set of values that belong to a given type: for instance, the set of all values that

[8]The top type of functions of arity one is *not* `term()->term()`. In our system, every function of this type can be safely applied to any argument of type `term()`, that is, every well-typed argument. But of course not every function satisfies this property: only the total ones.

satisfy the guard `is_integer(person.age)`) coincides with the set of values that have type `%{..., age: integer()}`. Previous systems with semantic subtyping and set-theoretic types did not account for guards, which is the reason why we had to develop a specific analysis technique for them (see Chapter 6 for more details).

Note that, in the absence of such guards, it is the task of the programmer to explicitly provide the type of the whole function by preceding its definition by a type specification. It is also possible to elide parts of the return type of non-recursive functions by using the underscore symbol "_", as in

```
$ integer() -> _
def negate(x) when is_integer(x), do: -x
```

or

```
$ (integer() -> _) and (boolean() -> _)
def negate(x) when is_integer(x), do: -x
def negate(x) when is_boolean(x), do: not x
```

leaving to the type system the task of deducing the best possible types to replace for each occurrence of the underscore.

Exhaustivity Checking Type analysis makes it possible to check whether clauses of a function definition, or patterns in a case expression, are *exhaustive*, that is, if they match every possible input value. For instance, consider the following code:

```
$ type result() =
    %{output: :ok, socket: socket()} or
    %{output: :error, message: :timeout or {:delay, integer()}}

$ result() -> string()
def handle(r) when r.output == :ok, do: "Msg received"
def handle(r) when r.message == :timeout, do: "Timeout"
```

We define the type `result()` as the union of two *record types*: the first maps the atom `:output` to the (atom) singleton type `:ok` and the atom `:socket` to the type `socket()`; the second maps `:output` to `:error` and maps `:message` to a union type formed by an atom and a tuple. Next consider the definition of `handle`: values of type

`%{output: error, message: {:delay, integer()}}`

are going to escape every pattern used by `handle`, triggering a type warning:

```
Type warning:
  | def handle(r) do
        ^^^^^^^^^
      this function definition is not exhaustive.
      there is no implementation for values of type:
        %{output: :error, message: {:delay, integer()}}
```

Note that the type checker is able to compute the exact type whose implementation is missing, which enables fast refactoring since, as the type of `result()` or the implementation of `handle` are modified, the type checker will issue precise new warnings to point out the places where code changes are required.

Redundancy Checking Similarly, it is possible to find useless branches—branches that cannot ever match. For instance, if we add a clause to the previous example:

```
$ result() -> string()
def handle(r) when r.output == :ok, do: "Msg received"
def handle(r) when r.message == :timeout, do: "Timeout"
def handle({:ok, msg}), do: msg
```

then since the specified input type is `result()` (which is a subtype of maps), the third branch will never match (its pattern matches only pairs) and can be deleted.

This will remove useless code, detect unused function definitions, or reveal more complex problems as these hints can indicate areas where the programmer's expectations and the actual logic of the program do not match.

Narrowing Narrowing is the typing technique that consists in taking into account the result of a (type-related) test to refine (i.e., to narrow) the type of variables in the different branches of the test. In Section 1.1.2 we have already presented a simple example in which narrowing is used, namely, in the function `negate_alt` (code in line 17) the type-checker uses the test to narrow the type of `x`, which is `(integer() or boolean())`, to `integer()` in the "do" branch and to `boolean()` in the "else" branch. This is a simple application of narrowing, where the narrowing is performed on the type of a variable whose type is directly tested. However, our system is also able to narrow the type of the variables that occur in the expression tested by a "case" or a "if", even if this expression is not a single variable (some exceptions apply though: see future works). Here is a more complete example where we test the field selection on a variable

```
$ result() -> _
def handle(r) when r.output == :ok, do: {:accepted, r.socket}
def handle(r) when is_atom(r.message), do: r.message
def handle(r), do: {:retry, elem(r.message, 1)}
```

In the example, the type of `r` initially is `result()`. This type is *narrowed* in the first branch to `%{output: :ok, socket: socket()}`, in the second branch to

`%{output: :error, message: :timeout}`

and in the last branch to the remaining

`%{output: :error, message: :delay, integer()}` .

This precision is shown by the fact that `handle` type-checks the following type specification too:

```
75 $ (%{output: :ok, socket: socket()} -> {:accept, socket()}) and
76   (%{output: :error, message: :timeout} -> :timeout) and
77   (%{output: :error, message: {:delay, integer}} -> {:retry, integer})
```

As a matter of fact, deducing the type of the parameters of a function by examining its guards is just yet another application of narrowing where the function parameters are initially given the type `term()` and narrowed by the types deduced for the guards.

Conservative Approximations When analyzing patterns with guards, it may not always be possible to determine the precise type of the captured values. In such cases, we use both lower and upper approximations to ensure that narrowing and exhaustivity/redundancy checking still work. As an example, consider the following simplistic function:

```
78 def foo(x) when map_size(x) == 2, do: Map.to_list(x)
```

We are unable to express by a type the exact domain of this function, which is the set of "all maps of size 2". However, when the guard succeeds, it is clear that `x` is a *map*, and this assumption is enough to deduce by narrowing that the body of the function is well-typed. Using the type of all maps to approximate the set of all maps of size 2 is an over-approximation. We call such a type the *potentially accepted type* of the pattern/guard since it contains all the values that *may* match it. Conversely, consider the following example:

```
79 def bar(x) when (is_map(x) and map_size(x)==2) or is_list(x), do: to_string(x)
80 def bar(x) when length(x) == 2, do: x
```

The first clause matches both the maps of size 2 (but no other map) and any lists. Although we cannot characterize by a type all the values matched by the first clause, we do know that all lists are captured by it and, therefore, the second clause is redundant (`length` being defined only for lists). The type of all lists is an under-approximation (i.e., a subset) of the set of all values that satisfy the guard in the first clause. We refer to this under-approximation as the *surely accepted type* of the pattern/guard since it contains *only* values that *do* match it. Our system makes a distinction between guards that require approximation and those that do not, as further described in Chapter 6.

Complex Guards The analysis of guards is more sophisticated than it appears. First of all, guards are examined left to right by incrementally generating environments during their analysis. An example is the guard in line 79: if we compare it with line 78 we see that we added an `is_map(x)` test. Without it the guard in line 79 would be equivalent to the one in line 78, since when `x` is a list, then `map_size(x) == 2` fails (rather than return `false`), and so does the whole guard: the `is_list(x)` would never be evaluated. To account for this, our analysis examines `is_list(x)` only if the preceding clause may not fail, which is always the case here—though, it can return `false`— and `map_size(x)` is examined only in the environments in which `is_map(x)` succeeds.

Another stumbling block is that the analysis may need to generate for a single guard different type environments under which the continuation of the program is checked, as the following definition shows:

```
81 $ (term(),term()) -> {integer(),term()} or {term(),boolean()} or nil
82 def baz(x, y) when is_boolean(x) or is_integer(y), do: {y,x}
83 def baz(_, _), do: nil
```

The definition above type-checks, but this is possible only because the analysis of the guard `is_boolean(x) or is_integer(y)` in line 82 generates two distinct environments (i.e., one where `y` has type `integer()` and `x` type `term()`, and a second one where their types are inverted) which are both used to deduce *two* types for `{y,x}` which are then united in the result. By the same technique, in the absence of a type specification our system deduces for the first clause of `baz` in line 82 the type

```
84 ((term(),integer()) -> {integer(),term()}) and
85 ((boolean(),term()) -> {term(),boolean()})
```

and the analysis of the code defined in line 83 adds to this intersection the following arrow: `(not(boolean()),not(integer())) -> nil`.

Finally, our type system can analyze arbitrarily nested Boolean combinations of guards which are type tests of complex selection primitives, as the following definition of a parametric guard `is_data(d)` shows:

```
86 defguard is_data(d) when is_tuple(d) and tuple_size(d) == 2 and
87     (elem(d, 0) == :is_an_int and is_integer(elem(d, 1)) or
88      elem(d, 0) == :is_a_bool and is_boolean(elem(d, 1)))
```

Our system deduces that the guard `is_data(d)` succeeds if and only if `d` is of type `data()` defined as follows:

```
89 $ type data() = {:is_an_int, integer()} or {:is_a_bool, boolean()}
```

It is possible to systematize this approach so that whenever we define a type `t()` without using function types and other complex constructs (for instance, dictionary keys of maps, as we see next), we can automatically generate the guard `is_t()` whose accepted type is exactly `t()`.

1.1.5 Records and Dictionaries

In Elixir, maps are a key-value data structure that serves as the primary means of storing data. There are two distinct use cases for maps: as records, where a fixed set of keys is defined, and as dictionaries, where keys are not known in advance and can be dynamically generated. A map type should unify both, allowing the type-checker to sensibly choose when it needs to ensure that some expected keys are present while enforcing type specifications for queried values.

Maps as Records When map are used as records, Elixir provides the *map*.*key* syntax, where :*key* is an `atom()`. If the map returned by the expression *map* does not contain that key at runtime, an error is raised. In Section 1.1.4 c), we saw the following definition:

```
$ %{..., age: integer()} -> integer()
def get_age(person) when is_integer(person.age), do: person.age
```

In more precise terms, the above type is equivalent to:

```
$ %{required(:age)=>integer(),optional(term())=>term()} -> integer()
```

Each “key type” in a map type is either required or optional. Singleton keys are assumed to be required, unless otherwise noted[9]. The triple dot notation means the type also types record values that define more keys than those specified in the type and corresponds to `optional(term()) => term()`. We refer to those as open maps.

A program similar to the above but with optional keys

```
$ %{optional(:age) => integer()} -> _
def get_age(person), do: person.age
```

raises a type error pointing out the possibly undefined keys:

```
Type warning:
 | def get_age(person), do: person.age
                            ^^^^^^^^^^

    key :age may be undefined in type: %{optional(:age) => integer()}
```

Hence, typing gets rid of all `KeyError` exceptions for every use of *map*.*key*, by restricting the use of dot-selection to maps that are known to have the required keys.

Maps as Dictionaries When working with maps as dictionaries, we use the *m*[*e*] syntax to access fields, where *m* and *e* are expressions returning a map and a key, respectively. In this notation, the field may not exist, in which case `nil` is returned. For the given function

```
$ %{optional(:age) => integer()} -> _
def get_age(person), do: person[:age]
```

[9]The compiler will reject `required(term())`, as that would require a map with an infinite amount of keys. However, because a finite type such as `boolean()` can be either required or optional, we require all non-singleton types to be accordingly tagged to avoid ambiguity.

the system infers `integer() or nil` as return type. If the type-checker can infer that the keys are present, then it will omit the `nil` in the return type. For instance, the following function is well typed since both fields are required and, thus, cannot return `nil`:

```
$ %{foo: integer(), bar: integer()} -> integer()
def add(m), do: m[:foo] + m[:bar]
```

The type-checker distinguishes when a key *is* present, *may be* present, or *is not* present. To summarize, for the following function (where `%{}` is the empty map),

```
$ %{foo: integer()}, %{optional(:foo) => integer()}, %{} -> _
def f(m1, m2, m3), do: {m1[:foo], m2[:foo], m3[:foo]}
```

the return type`{integer(), integer() or nil, nil}` is inferred.

The `fetch!` Operation We look at the `Map.fetch!(map, key)` function[10], which expects an arbitrary key to exist in the map, raising a `KeyError` otherwise. A developer may use `fetch!` to denote that a given key was explicitly added in the past and it must exist at this given point. While `map.key` is ill-typed if the key *may be* undefined, `fetch!` is ill-typed only if the key *is always* undefined. So the following program is rejected

```
$ %{not_age: integer()} -> _
def get_age(m), do: fetch!(m, :age)
```

because the field `:age` cannot appear in `m`, but the following one is accepted

```
$ map() -> term()
def get_age(m), do: fetch!(m, :age)
```

because key `:age` may be present in `m`, since `map()` represents any possible map.

Dictionary Keys We have shown how to interact with maps as records and dictionaries where the keys were restricted to singleton types. But e`[`e'`]` (and other map operations) allows for generic use of maps as dictionaries, where the key is the result of an arbitrary expression e'. To model this behaviour, map types can specify the type of values expected by querying over certain fixed key domains, e.g.,

```
$ integer() -> %{optional(integer()) => integer()}
def square_map(i), do: %{i => i ** 2}
```

[10]In Elixir, ending a function name with `!` is a convention that implies the function may raise an exception for valid domains. For example, `File.read!("log.txt")` will raise if the file does not exist, compared to `File.read("log.txt")` which would instead return `:error`.

The map `%{optional(integer()) => integer()}` associates integers to integers, but since `integer()` represents an infinite set of values, all fields cannot be required. Hence, it must be annotated as `optional`.

Key domains cannot overlap. To ensure this property in our system it is possible to use as key domains only a specific set of basic types of Elixir Typespec: `integer()`, `float()`, `atom()`, `tuple()`, `map()`, `list()`, `function()`, `pid()`, `port()`, and `reference()`. However, our type system allows the programmer to define map types that mix dictionary and record fields, that is, types where fields are declared both for singleton keys and for key domains. In that case it is possible to specify in the same type both some singleton keys and the domain keys of these singleton keys, the former taking precedence over the latter. This means that the type

```
$type t() = %{ foo: atom(),
               optional(:bar) =>  atom(),
               optional(atom()) => integer() }
```

represents maps with a mandatory field `:foo` and an optional one `:bar` both set to atoms, and where any other key is an atom associated to an integer.

If e is an expression of the type `t()` above, then the type of the selection e`[`e'`]` will be computed according to the type of the expression e': if e' has type `:foo`, then the selection has type `atom()`; if e' has type `:foo or :bar`, then the selection will be typed by `atom() or nil`, since `:bar` may be absent; if the type of e' intersects `atom()` but it is not a subtype of `:foo or :bar`, then the selection will be typed by

`atom() or integer() or nil` .

If e' has type `not atom()` then the selection will have type `nil` and will issue a warning. If the `fetch!` operation is used in the examples above instead, then the system will deduce the same types minus `nil`, except for the case for e' of type `not atom()`, which will be considered ill-typed. To summarize, if `m` is of type `t()`, then: `m.foo` has type `atom()`; `m.bar` is not well-typed; `m[:bar]` has type `atom() or nil`; `m[:other]` has type `integer() or nil`; `m[m.foo]` has type `atom() or integer() or nil`; `m[42+1]` has type `nil` and will raise a warning.

Finally, `t()` is a type that mixes the characteristics of records (i.e., the first two field declarations) and dictionaries (i.e., the last field declaration). This kind of mix is not uncommon in languages such as Lua*u* or TypeScript, where dictionaries sometimes include a mandatory field of some type (e.g., an integer field `size`). This is less common in Elixir and rather confined to Elixir's "structures" which are maps with a mandatory field `:__struct__` of type `atom()` and any other atom field optional, as in struct `%{__struct__: atom(), optional(atom()) => term()}`. A detailed comparison of our record types and those of Flow, TypeScript, or Lua*u* can be found in Castagna (2023b, Section 5).

Deletions and Updates It is possible to specify missing keys in a map type by adding an optional field that points to `none()` (i.e., the key must be absent since if it were present, then it should be associated to a value of the empty type `none()`, which does not exist). This makes it possible to type a `delete` operation as follows:[11]

```
$ map() -> %{..., optional(:foo) => none()}
def delete_foo(map), do: Map.delete(map, :foo)
```

Similarly to map access, there are different ways to replace a field in a map. The syntax `%{map | key => value}` requires that the key is present and otherwise raises a `KeyError` exception. The presence of such a key can be statically ensured by our system.

1.1.6 Gradual Typing and Strong Arrows

There is an important base of existing code for Elixir. If we want to migrate this code to a typed setting, the ability to blend statically typed and dynamically typed code is crucial. Some code that is working fine may not pass type-checking, therefore a gradual migration approach is preferred to converting the entire codebase to comply with static typing at once. This is the goal of gradual typing as introduced by Siek and Taha (2006). For that, we introduce the type `dynamic()`, which essentially puts the type-checker in dynamic typing mode. In practice, the programmer can think of `dynamic()` as a type that can become at run-time (technically, that *materializes* into: see Castagna et al. (2019)) *any other type*: an expression of type `dynamic()` can be used wherever any other type is expected, and an expression of any type can be used where a `dynamic()` type is expected since, in both cases, `dynamic()` may become at run-time that type.[12] The simplest use case is to declare that a parameter of a function has type `dynamic()`:

```
$ dynamic() -> _
def foo1(x), do: ⋯
```

meaning that in the body of `foo1` the parameter `x` can be given any type—possibly a different type for each occurrence of `x`—and that `foo1` can be applied to arguments of any type. The type `dynamic()` is a new basic type, that can occur in other type expressions. This can be used to constrain the types arguments can have. For instance

```
$ (dynamic() -> dynamic()) -> _
def foo2(f), do: ⋯
```

[11]The type of `delete_foo` is not very useful in practice. It will be useful when combined with "row polymorphism" which permits to check the following type: `%{ a } -> %{optional(:foo) => none(), a} when a: fields()` see future work.

[12] Oversimplifying, one can consider `dynamic()` to be both a supertype and subtype of every other type (while `term()`, which is often confused with `dynamic()` is only the former) with a caveat, subsumption does not apply to `dynamic()` since we cannot consider an expression of a type different from `dynamic()` to be of type `dynamic()`: the application of `dynamic()->dynamic()` to an integer is well-typed because the arrow type materializes in `integer()->dynamic()` and *not* because `integer()` materializes into `dynamic()`.

requires the argument of `foo2` to have a function type. This means that in the body of `foo2` the parameter `f` can be applied to arguments of any type and its result can be used in any possible context, but a use of `f` other than as a function—e.g., `f + 42`—will be rejected. Likewise, an application of `foo2` to an argument not having a functional type—e.g., `foo2({7,42})`—will be statically rejected, as well.

Gradual Typing Guarantees Using `dynamic()` does not mean that type-checking becomes useless. Even in the presence of `dynamic()` type annotations, our type system guarantees that if an expression is given type, say, `integer()`, then it will either diverge, or return an integer value, or fail on a run-time type-check verification. This safety guarantee characterizes the approach known as *sound gradual typing* (Siek and Taha (2006)). This approach was developed for set-theoretic types in Lanvin's PhD thesis (2021) whose results we use here to define subtyping and precision relations using the subtyping relation on non-gradual types (i.e., types in which `dynamic()` does not occur).

There is however a fundamental difference between our approach and the one of sound gradual typing: the latter uses the gradual type annotations present in the source code to *insert* into the compiled code the run-time type-checks necessary to ensure the above type safety guarantee. Instead, one of our requirements (cf. Chapter 14) is that the addition of types *must not* modify the compilation of Elixir. Therefore, we have to design our gradual type system so that it ensures the type safety guarantee by taking into account both the dynamic type-checks performed by the Erlang VM *and those inserted by the programmer*. The goal, of course, is to deduce for every well-typed expression a type which is gradual as little as possible, the best deduction being that of a non-gradual type (the less gradual the type, the more the errors captured at compile time). To that end we introduce the notion of *strong function types*.

Strong Function Types We have seen that our system can deduce the type of a function also in the absence of an explicit annotation when there are guards on the parameters. This means that we can differentiate, on a type level, those two definitions of the identity of type `integer() -> integer()`:

```
$ integer() -> integer()
def id_weak(x), do: x
```

```
$ integer() -> integer()
def id_strong(x) when is_integer(x), do: x
```

This makes sense since, from a runtime perspective, the two definitions above differ as the latter checks that its argument is of type `integer()`, while the former does not. Therefore, if these functions are applied to an argument that *may not be an integer* (e.g., of type `dynamic()`), then we can only be certain that the resulting output is an integer for the function `id_strong`. This allows us to type the following function:

```
$ dynamic() -> {dynamic(), integer()}
def foo3(x), do: {id_weak(x), id_strong(x)}
```

which is accepted by our system since it deduces that when `id_weak` is applied to an argument of an unknown `dynamic()` type, then it cannot give to the result a type more precise than `dynamic()`, while for the same application with `id_strong` it deduces that whenever the application returns a result, this result will be of type integer.

We refer to the function `id_strong` as having a "strong" function type, since it guarantees that when applied to an argument that is *not within its domain*, it will either (i) return a result within its codomain, or (ii) fail on a dynamic type check—performed by the Erlang VM or inserted by the programmer—, or (iii) diverge.[13] Likewise, the function `id_weak` has a "weak" function type, since it may return (and actually, does return) a result not of type `integer()` when applied to an argument that is not of type `integer()`.

Propagation of `dynamic()` `dynamic()` is propagated by the type system through calls of functions such as `id_weak`, but it is stopped when it goes through a function with a strong arrow type, such as `id_strong`: the type `dynamic()` does not appear in the type of the application of `id_strong`. Now, the goal of using `dynamic()` in a program is to instruct the type system to be lenient. In order to maximize this leniency, we propose (probably, as a type-checking option) to use an intersection to propagate `dynamic()` also for functions with a strong arrow type, in a way that increases the permissiveness of the type system without hindering its safety properties. However, this permissiveness comes at the expense of the static detection of some type errors.

In practice, this comes down to deduce for `foo3` a type smaller than the one given in line 122. If we omit the type annotation and, like for `foo3`, there is no explicit guard, then our system assumes that the parameter `x` has type `dynamic()` and deduces for `foo3` the type `dynamic() -> {dynamic(), (integer() and dynamic())}` which is a subtype of the type in line 122 (a classic sound gradual typing approach such as those by Siek and Taha (2006) or Castagna et al. (2019) and Lanvin (2021) would have deduced for this function the return type `{integer(), integer()}`, but also modified its standard compilation by inserting two run-time integer type-checks, one for each occurrence of `x` in the body of `foo3`).

The reason why the intersection occurring in second projection of the result improves the typability of existing code can be generally understood by considering, for some arbitrary type t`()`, the type t`() and dynamic()`. An expression of type t`() and dynamic()` can be used not only in all contexts where an expression of type t`()` is expected, but also in all contexts where a *strict* subtype of t`()` is expected (in which case the use of `dynamic()` will be further propagated). This is useful especially for (strong) functions whose codomain is a union type. For instance, consider again the function `negate` as defined in lines 8–9. This is a strong function whose codomain is `integer() or boolean()`. If this code is coming from some existing base—i.e., without any annotation—then the system deduces that this function takes a `dynamic()` input and returns a result of type

`(integer() or boolean()) and dynamic()`:

thanks to the intersection with `dynamic()` in the result type, it is then possible to use the result

[13] If the argument is in the domain, then only (i) and (iii) are possible.

of `negate` not only where an expression of type `integer() or boolean()` is expected, but also where just an `integer()` or just a `boolean()` is expected, then propagating the dynamic type. Concretely, in this dynamic setting, the function `subtract` as defined in lines 4–6 would still be well typed since the type of `negate(b)` in line 5 could materialize into `integer()` and, in the absence of an explicit annotation, by the propagation of `dynamic()` the type deduced for the result of `subtract` would be `integer() and dynamic()` which thus in turn could be passed to a function expecting a subtype of `integer` (e.g., a division function expecting an input of type `integer() and not 0`, which denotes all integers but 0). Of course, this comes at the expenses of an earlier error detection, since a gradually typed version of `subtract` in which `negate` could be applied to a `boolean()` would fail: this is an error that would be instead statically detected in the absence of the propagation of `dynamic()`. Finally, notice that if we had explicitly defined the type of `negate` to be `dynamic() -> integer() or boolean()` (as we did above in line 122 for `foo3`), then the result of `negate(b)` in line 5 would have been typed as `integer() or boolean()` thus precluding the materialization and the consequent typing of `subtract`.

We extended the semantic subtyping framework with strong function types, which are inhabited by functions satisfying the property described above (see 5.3). Strong function types are the key feature that allows the type-system to take into account the run-time typechecks, either performed by the BEAM or inserted by the programmer. Built-in operations, such as field selection, tuple projections, etc, are, by implementation, strong: the virtual machine dynamically checks that, say, if the field `a` of a value is selected, then the value is a record and the field `a` is defined in it. Our theory extends this kind of checks to user-defined operations—i.e., functions definitions—by analyzing their bodies to check that all the necessary dynamic checks are performed. The functions for which this holds have a strong type, and the system can safely deduce that when they are applied to an argument that *may* be not in their domain (e.g., an argument of type `dynamic()`), then the application will return a value in their codomain (rather than a result of type `dynamic()`), and as explained above, to maximize typability of existing code this codomain is intersected with `dynamic()`.

All this is currently transparent to the programmer since strong types are only used internally by the type checker to deduce the type of functions such as `foo3`. A possible extension of our system would be to allow the programmer to specify whether higher-order parameters require a strong type or not.

1.2 Outline

The type system we envision for Elixir, as introduced in this chapter, combines multiple advanced typing features to address the language's dynamic idioms. The remainder of this thesis is organized in two parts, covering theory and practice, to show how we realize this vision.

To guide readers with different interests, each chapter is labeled as [R] (primarily research/theory oriented), [I] (primarily implementation/practical oriented), or [R/I] (mixed content bridging theory and practice).

Part I – Theoretical Foundations

Chapter 2 – Technical Overview [R/I] A guided tour of the system's key ideas (multi-arity functions, dynamic propagation, strong functions, guard analysis, and inference) with motivating Elixir examples that preview the formal developments.

Chapter 3 – Semantic Subtyping Background [R] Foundations for static and gradual set-theoretic types: set interpretation, disjunctive normal forms, deciding subtyping by checking emptiness, and the decision procedures for products and functions. Introduces core operators (projection, domain, application) and shows how gradual relations and operators lift from the static setting.

Chapter 4 – Static Typing for Core Elixir [R] Defines the calculus, its operational semantics and failure behaviour, and the static typing rules with a type safety theorem. Extends the semantic subtyping framework to tuples and multi-arity functions.

Chapter 5 – Gradual Typing without Changing Runtime [R] Explains the integration of the dynamic type '?' within the type system, and details our erasure gradual typing strategy; introduces strong function types, the auxiliary judgment used to infer them, and dynamic propagation. Shows how gradual rules interleave with the static ones, and extends the semantic subtyping framework to handle strong functions.

Chapter 6 – Typing Guards and Pattern Matching [R] Formalizes patterns and guards and their operational behaviour; presents guard analysis with "sure" and "possible" accepted types and an exactness marker. Improves case typing and supports inferring multi-clause interfaces.

Chapter 7 – Discussion [R] Positions our approach among Erlang/Elixir tools and industry systems (TypeScript, Flow, Hack, Sorbet, Luau), contrasts gradual disciplines, and outlines future directions (e.g., message passing, module system).

Part II – Pragmatic Implementation

Chapter 8 – Gradual Set-Theoretic Types as an API [R/I] Specifies a small and stable type-engine interface: constructors for base and structural types, set operations (union, intersection, difference), a normalization strategy, and core queries (subtyping, equivalence, emptiness, disjointness).

Chapter 9 – Bootstrapping Gradual Set-Theoretic Types [R/I] Devises how to implement the type system in practice: introduces a modular descriptor design (one field per component type), and outlines how to compute set operations field-by-field and adopt the approach incrementally in a compiler.

Chapter 10 – Type Representation Details [I] Implements base and literal domains (as finite sets or their complements), and gradual types as two bound types with a simple invariant; provides a concrete Elixir encoding for gradual types.

Chapter 11 – Implementing Structural Types [I] Realizes Elixir types: functions, tuples, lists, and maps, using boolean decision trees and a lazy variant; treats multi-arity functions and open/closed structures within the descriptor design.

Chapter 12 – Structural Type Alternatives: A Comparative Analysis [I] Reviews and contrasts alternative representations of structural types: disjunctive normal forms, and union-based forms; explains performance and precision trade-offs and why decision trees were adopted.

Chapter 13 – Typing Elixir [R/I] Bridges the formal calculus and a simplified concrete subset of the language; compiles guards; implements static and gradual application.

Chapter 14 – Evaluation [I] Reports integration status in Elixir (v1.17-v1.19), end-to-end performance on large codebases, and bugs uncovered in major projects; discusses rollout and tooling impact.

Reading guidance

- Readers primarily interested in the theoretical foundations should focus on the [R] chapters in Part I (Chapters 3–6) and the theory discussion in Chapter 7. Chapter 2 offers a high-level tour bridging to practice.
- Readers more interested in practical implementation and usage may jump to Part II. Chapters 8–14 cover the API, implementation details, and evaluation; refer back to [R] chapters as needed.

1.3 Related publications

Portions of the work reported in this thesis have appeared in prior publications:

Design of the Type System: The core design principles for adding a type system to Elixir were first published in **Castagna, G., Duboc, G., and Valim, J. (2023)** – "The Design Principles of the Elixir Type System", in *The Art, Science, and Engineering of Programming*, 8 (2). This paper provides an overview of the motivations and initial design choices for our approach.

Technical Development: The detailed theory of our type system, including the guard analysis and gradual typing strategy, is presented in **Castagna, G., and Duboc, G. (2024)** – "Guard Analysis and Safe Erasure Gradual Typing: A Type System for Elixir", *arXiv preprint* arXiv:2408.14345. This preprint corresponds to the latter half of Part I of this thesis, focusing on the novel theoretical contributions.

(Earlier work by the author, not reproduced in this thesis, includes the following:)

> **Castagna, G., Duboc, G., Lanvin, V., and Siek, J. G. (2019)**. "A Space-Efficient Call-by-Value Virtual Machine for Gradual Set-Theoretic Types." In *Proceedings of the 31st Symposium on Implementation and Application of Functional Languages (IFL 2019), pp. 1-12.*
> – *This work was an exploration of runtime support for a gradually typed language with set-theoretic types, not directly tied to Elixir but providing some groundwork for understanding performance considerations.*
>
> **Eisenberg, R. A., Duboc, G., Weirich, S., and Lee, D. (2021)**. "An Existential Crisis Resolved: Type Inference for First-Class Existential Types." *Proceedings of the ACM on Programming Languages, 5(ICFP), Article 64, 29 pages. DOI: 10.1145/3473569.*
> – *This work deals with type inference in Haskell for first-class existentials. While not related to Elixir, it contributed to the author's background in type system research. Both works resolve their issues by strength: Eisenberg et al. introduce "strong existentials" that support lazy projection of encapsulated data, while our approach employs "strong functions" that act as type-level proxies.*

1.4 Project Context and Industrial Collaboration

CIFRE industrial PhD: This thesis work was conducted as a CIFRE industrial doctorate, a French program that enables conducting PhD research in collaboration with an industry partner. The arrangement provided a unique opportunity to develop the type system not just in theory but also in the context of real-world Elixir development. The scientific objectives were twofold: (i) to establish a formal, set-theoretic foundation for a gradual type system for Elixir (the academic research component), and (ii) to implement and integrate that type system into the official Elixir compiler[14] (the industrial/application component).

Industrial partners and funding: The project received financial and logistical support from several companies that rely on Elixir in production and are invested in its long-term evolution. In particular, Fresha, Remote, Supabase, and Starfish Alliance sponsored this research, recognizing that a static type system could greatly benefit the robustness of their Elixir codebases. The funding from these companies, combined with the CIFRE grant, supported the entirety of the PhD work.

Supervision: The work was co-advised by José Valim, the creator and lead maintainer of Elixir. His guidance ensured that the research remained aligned with the goals and idioms of the Elixir language and its community. By working closely with José and other core team members, we

[14]Implementation repository: `https://github.com/elixir-lang/elixir`; Documentation for the emerging type system: `https://hexdocs.pm/elixir/main/gradual-set-theoretic-types.html`.

aimed to design the type system in a way that feels “native” to Elixir and addresses real developer needs. Regular updates and discussions were held with the Elixir team throughout the project, which helped in steering the design and prioritizing features.

Community Engagement:

- **“Bringing Types to Elixir.”** *ElixirConf EU 2023* – Talk by Guillaume Duboc. YouTube video. *This talk introduced for the first time the vision of a set-theoretic type system for Elixir, explaining to a general audience why and how types could be added to the language gradually.*
- **“Gradualise your language with set-theoretic types.”** *Lambda Days 2024* – Talk by Guillaume Duboc. YouTube video. *This presentation was more theory-focused, given at a functional programming conference, covering the underlying theory of set-theoretic types and gradual typing, and how they apply to Elixir.*
- **“Set-theoretic types: the theory, the practice.”** *ElixirConf US 2024.* Talk by Guillaume Duboc. YouTube video. *This talk (planned after much of the implementation was in place) bridges the gap between theory and practice, showing examples of typed Elixir code and discussing the developer experience.*

Through these engagements, we received valuable feedback from both academia and industry practitioners, which has influenced the work presented in this thesis.

Part I

Theoretical Foundations

CHAPTER 2

TECHNICAL OVERVIEW

“I present the advantages and, I dare say, the beauty of programming in a language with set-theoretic types, that is, types that include union, intersection, and negation type connectives.”

Giuseppe Castagna, *Programming with Union, Intersection, and Negation Types* (2023)

The Elixir type system described by Castagna et al. (2024a) is a gradual polymorphic type system based on the polymorphic type system of ℂDuce (Castagna et al., 2014; Castagna et al., 2015). In this part of the thesis, we describe the technical additions that are missing in the ℂDuce type system to type Elixir programs, presenting them one by one. These are the techniques of:

- typing (and subtyping) for *multi-arity functions* (§ 4.4);
- *propagation of the* `dynamic()` *type* necessary for safe-erasure gradual typing (§ 5.1);
- typing (and subtyping) for *strong functions* (§ 5.3);
- *guard analysis* (§ 6.3);
- *type inference for anonymous functions* (§ 6.5);

In this introductory chapter, we present them one after the other by giving some small examples that should help the reader understand the technical developments described in the next sections.

Soundness

The type system we present satisfies the following soundness property:

> *If an expression is of type t, then it either diverges, or produces a value of type t, or fails on a dynamic check either of the virtual machine or inserted by the programmer.*

whose formal statement is given in Theorem 5.2.15. The system is gradual since the type syntax includes a `dynamic()` type used to type expressions whose type is unknown at compile time.[1] The soundness guarantee above is typical of the so-called *sound gradual typing* approaches. These approaches ensure soundness by using typing derivations to insert some suitable dynamic checks at compile time. Our system, instead, does not modify Elixir standard compilation: types are not used for compilation and are erased after type-checking. Our system is, thus, a *type-sound (i.e., safe) erasure gradual typing system*, the first we are aware of. In particular, the compiler does not insert any dynamic check in the code apart from those explicitly written by the programmer. Therefore, our system must ensure soundness by considering only the checks written by the programmer or performed by the Beam machine.

Writing a sound gradual type system for Elixir is easy: since every Elixir computation that does not diverge or error returns a value (no stuck terms, thanks to the BEAM), then a system that types every expression by `dynamic()` is trivially sound. . . but hardly useful. Therefore, we need a system that must fulfill two opposite requirements:

1. it must use `dynamic()` as little as possible so as to be useful, and
2. it must use `dynamic()` enough so as not to hinder the versatility of gradual typing

The first requirement is satisfied by the typing of strong functions, and the second requirement is fulfilled by the propagation of `dynamic()`. We demonstrate both of these aspects through the design choices and challenges we address in our type system. We begin by examining how our static typing foundation handles Elixir's distinctive features, before exploring how gradual typing builds upon it.

Chapter 4 : Static Typing for Elixir

To achieve precise typing (fulfilling our first requirement of minimizing `dynamic()`), our system must handle Elixir's unique language features that distinguish it from languages with existing semantic subtyping systems. We illustrate this with the challenge of multi-arity functions.

Multi-arity Functions (Section 4.4.1) Elixir, as Erlang, defines functions with a fixed arity, which is part of the name of every function. For example, recall function `subtract/2` (l.4 in the general introduction):

```
$ (integer(), integer()) -> integer()
def subtract(x, y) when is_integer(x) and is_integer(y) do
```

[1]We recall that in Elixir, type identifiers end by `()`, e.g., `integer()`, `boolean()`, `none()`, `dynamic()`, etc.

```
  x + negate(y)
end
```

This arity is reflected in its type, where its domain consists of two comma-separated types. The ℂDuce type system can handle unary functions and simulates n-ary functions as unary functions on n-tuples. But this is not sufficient in Elixir: first, applying a function to two arguments or to a pair involves different syntaxes, e.g., `subtract(42,42)` and `test({42,4})`[2].

Second, a programmer can explicitly test whether a function f has arity n using `is_function(`f`,`n`)`. Consequently, we need a type system in which it is possible to express the type of all functions of a given arity. For instance, we may want to give a type to:

```
def curry(f) when is_function(f, 2) do
  fn a -> (fn b -> f.(a, b) end) end
end

def curry(f) when is_function(f, 3) do
  fn a -> (fn b -> (fn c -> f.(a, b, c) end) end) end
end
```

but in current systems with semantic subtyping, we can only express the type of *all* functions, that is, `none() -> term()`.[3] Simulating, say, binary functions with functions on pairs does not work since `{none(), none()} -> term()` is not the type of all binary functions: since the product with the empty set gives the empty set, this type is equivalent to `none() -> term()`, the type of *all* functions: since the product with the empty set gives the empty set, this type is equivalent to `none() -> term()`, the type of *all* functions. This is the reason why we introduced the syntax `(t`$_1$`,...,t`$_n$`) -> t` which outlines the importance of the functions. Now the type of all binary functions can be written as `(none(), none()) -> term()`, and we can declare for the function `curry` the following type (though the type variables or even a gradual type would be more useful than this type).

```
$ (((none(), none()) -> term()) -> none() -> none() -> term()) and
  (((none(), none(), none()) -> term()) ->
                                none() -> none() -> none() -> term())
```

All this requires nontrivial modifications both in the interpretation of types and in the algorithm that decides subtyping. These modifications are described in Section 4.4.1.

[2]When the callee is a *value* (an anonymous function stored in a variable), Elixir uses dot-application: if `f` has arity n, then `f.(`x_1`,...,`x_n`)` applies `f` to n arguments (whereas `f.({x,y})` is a *single*-argument call passing the 2-tuple `{x,y}`). In contrast, named functions are called `g(a,b)` and module-qualified functions `M.g(a,b)`.

[3]A value is of type s `->` t iff it is a function that when applied to an argument of type s, it returns only results of type t; thus, every function vacuously satisfies the constraint `none() -> term()`, which only requires its values to be functions, as there is no value of type `none()`.

Remark

In Elixir, it is possible to add default arguments to a function head, such as `def add(a, b \\ 42), do: a + b`. This allows calling `add(1)` to obtain 43, or calling `add(1,3)` to obtain 4. However, this does not mean that the function has both arities 1 and 2. Rather, this syntax actually defines two separate functions: `add/1` and `add/2`, each with their own arity. The disambiguation is made at the call site, which always specifies the number of arguments. Thus, there are no functions with overloaded arities, and we do not need to type them.

Chapter 5 : Gradual Typing

Strong Functions (Section 5.3) Consider the definition in Elixir of a function `second` that selects the second projection of its argument (`elem(e,n)` selects the (n+1)-th projection of the tuple *e*):

```
def second(x), do: elem(x,1)
```

If the argument of the function is not a tuple with at least two elements, then the BEAM raises a runtime exception. Let us write a signature for `second`:

```
$ dynamic() -> dynamic()
def second(x), do: elem(x,1)
```

This is one of the simplest types we can declare for `second`, since it essentially states that `second` is a function, and nothing more: it expects an argument of an unknown type and returns a result of an unknown type. We can give `second` a type slightly more precise than (i.e., a subtype of) the type above, that is:

```
$ {dynamic(), dynamic(), ..} -> dynamic()
```

which states that the argument of a function of this type must be a tuple with *at least* two elements of unknown type (the trailing "`..`" indicates that the tuple may have further elements). With this declaration, the application of `second` to an argument not of this type will be statically rejected, thus statically avoiding the runtime raise by the BEAM. We can also give the function a nongradual type—i.e., a type in which `dynamic()` does not occur: we call them *static types*—, as:

```
$ {term(), integer()} -> integer()
```

This type declaration states that `second` is a function that takes a pair whose second element is an integer and returns an integer. If this is the type declared for `second`, then the type deduced for the application `second({true,42})` is, as expected, `integer()`. If `dyn` is an expression of type `dynamic()`, then the type deduced for `second(dyn)` will be `dynamic()`: if `dyn` evaluates into a tuple with at least two elements, then the application will return a value that can be of any type, thus we cannot deduce for it a type more precise than `dynamic()`. This differs from current sound gradual typing approaches, which would deduce `integer()` for this application, but also insert a runtime check that verifies that the result is indeed an integer. However, this is not the way an Elixir programmer would have written this function. If the intention of the programmer is that `second` had the type `{term(), integer()} -> integer()`, then the programmer would rather write it as follows:[4]

```
$ {term(), integer()} -> integer()
def second_strong(x) when is_integer(elem(x,1)), do: elem(x,1)
```

This is defensive programming. The programmer inserts a *guard* (introduced by the keyword `when`) that checks that the argument is a tuple whose second element is an integer (the analysis of this kind of complex guards is the subject of Section 6). Thanks to this check (which compensates for the one inserted at compile time by other sound gradual typing approaches), we can safely deduce that `second_strong(dyn)` has type `integer()`. The above is called *strong function*, because the programmer inserted a dynamic check that ensures that even if the function is applied to an argument not in its domain, it will always return a result in its co-domain, i.e., `integer()`—or fail. This allows the system to deduce for `second_strong(dyn)` the type `integer()` instead of `dynamic()`, thus fulfilling our first requirement.

A function can be strong not only because it was defensively programmed, but also thanks to the checks performed at runtime by the BEAM, as for:

```
$ {term(), integer()} -> integer()
def inc_second(x), do: elem(x,1) + 1
```

which is also strong because the BEAM dynamically checks that both arguments of an addition are of type `integer()`. Therefore, also in this case, we know that if the function returns a value, then this is an integer. Thus, we can safely deduce the type `integer()` for `inc_second(dyn)` and, thus, for instance, that the addition `inc_second(dyn) + second_strong(dyn)` is well-typed.

To determine whether a function is strong, we define in Section 5.1 an auxiliary type system that checks whether the function, when applied to arguments *not* in its domain, returns results in its codomain or fails. This will allow us to infer explicit *strong function types*, that is, types of functions that are strong for all arguments.

Propagation of `dynamic()` (Section 5.1) In fact, for both the above applications, `inc_second(dyn)` and `second_strong(dyn)`, our system deduces a type better than (i.e., a subtype of) `integer()`: it deduces `integer() and dynamic()`. This is an *intersection type*, meaning that its expressions have both type `integer()` and type `dynamic()`. What the system does is to propagate the type `dynamic()` of the argument `dyn` of the applications into the result. This is meant to preserve the versatility of the gradual typing that originated the application, thus fulfilling our second requirement: expressions of this type can be used wherever an integer

[4] Elixir's type system inherits parametric polymorphism from ℂDuce. So, a more precise type for `second` would use a type variable *a* which in Elixir is quantified postfixedly by a `when` clause: `{term(), a} -> a when a: term()`. We do not consider polymorphism in the technical development as it is orthogonal to the features we study.

is expected, but also wherever any strict subtype of `integer()` (e.g., natural numbers) is. To see the advantages of this propagation, consider the following example that figured in the general introduction:

```
def negate(x) when is_integer(x), do: -x
def negate(x) when is_boolean(x), do: not x
```

The definition of `negate` is given by multiple clauses tested in the order in which they appear. When `negate` is applied, the runtime first checks whether the argument is an integer, and if so, it executes the body of the first clause, returning the opposite of `x`; otherwise, it checks whether it is a Boolean, and if so returns its negation; in any other case the application fails. Multi-clause definitions, thus, are equivalent to (type-)case expressions (and indeed, in Elixir they are compiled as such). The same static checks of *redundancy* and *exhaustiveness* that are standard for case expressions apply here, too. For instance, if we declare `negate` to be of type `integer() -> integer()`, then the type system warns that the second clause of `negate` is redundant; if we declare the type `term() -> term()` instead, then the function is not well-typed since the clauses are not exhaustive. To type the function above without any warning, we can use a union type, denoted by `or`:

```
$ integer() or boolean -> integer() or boolean()
```

which states that `negate` can be applied to either an integer or a Boolean argument and returns either an integer or a Boolean result. Next, we consider `subtract`

```
$ dynamic(), dynamic() -> integer()
def subtract(a, b), do: a + negate(b)
```

and see whether it type-checks. The type declaration states that `subtract` is a function that, when applied to two arguments of unknown type, returns an integer (or it diverges, or fails). Since the parameter `b` is declared of type `dynamic()`, then the system deduces that `negate(b)` is of type `(integer() or boolean) and dynamic()` (the `dynamic()` in the type of `b` is propagated into the type of the result). To fulfill local requirements, the static type system can assume `dynamic()` to become any type at run-time: following the terminology by Castagna et al. (2019), we say that `dynamic()` can *materialize* into any other type. In the case at issue, addition expects integer arguments. Therefore, the function body is well typed only if we can deduce `integer()` for `negate(b)`. This is possible since the type of this expression is `(integer() or boolean()) and dynamic()` and the system can materialize the `dynamic()` in there to `integer()` thus deriving (a type equivalent to) `integer()`.

Notice the key role played in this deduction by the propagation of `dynamic()`: had the system deduced for `negate(b)` just the type `integer() or boolean()`, then the body would have been rejected by the type system since additions expect `integer()`, and not `integer() or boolean()`.

A similar problem would happen had we declared `subtract` to be of type

```
152 $ integer(), integer() -> integer()
```

In that case, the type `integer() or boolean() -> integer() or boolean()` is not good enough for `negate`: since we assume `b` to be of type `integer()`, then the type deduced for `negate(b)` is again `(integer() or boolean())` which is not accepted for additions. The solution is to give `negate` a better type by using the intersection type

```
153 $ (integer() -> integer()) and (boolean() -> boolean())
```

which is a subtype of the previous type in line 149, and states that `negation` is a function that returns an integer when applied to an integer and a Boolean when applied to a Boolean. This type allows the type system to deduce the type `integer()` for `negate(b)` whenever `b` is an integer. This example shows why it is important to specify (or infer) precise intersection types for functions. The inference system we present in 6.5 will infer for an untyped definition of `negate` the intersection of arrows in line 153 rather than the less precise arrow with unions of line 149.

Finally, we want to signal that the new typing of `negate` in line 153 does not modify the propagation of `dynamic()`: the type deduced for `negate(dyn)` is still `(integer() or boolean) and dynamic()`.

Chapter 6 : Guard Analysis

Until now, the guards employed in our examples primarily involve straightforward type checks on function parameters (e.g., `is_integer(y)`, `is_boolean(x)`). The system we investigate for safe-erasure gradual typing in Section 5.1 exclusively focuses on these kinds of tests. There is a single exception in our examples with a more intricate guard, specifically `is_integer(elem(x,1))` used in line 144. In Elixir, guards can encompass complex conditions, utilizing equality and order relations, selection operations, and Boolean operators. To illustrate, consider the following (albeit artificial) definition:

```
154 def test(x) when is_integer(elem(x,1)) or elem(x,0) == :int, do: elem(x,1)
155 def test(x) when is_boolean(elem(x,0)) or elem(x,0) == elem(x,1), do: elem(x,0)
```

The first clause of the `test` definition executes when the argument is a tuple where either the second element is an integer *or* the first element is the atom `:int`. The second clause requires its argument to be a tuple in which the first element is either equal to the second element or is a Boolean.

To type this kind of definitions, the system needs to conduct an analysis characterizing the set of values for which a guard succeeds. Section 6 presents an analysis that characterizes this set in terms of types. In some cases, it is possible to precisely represent this set with just one type. For example, the set of values satisfying the guard `is_integer(elem(x,1))` in line 144 corresponds exactly to the values of type`{term(), integer(), ..}`. Likewise, the arguments that satisfy the guard of the first clause of `test` in line 154 are precisely those of the union type `{term(), integer(), ..} or {:int, term(), ..}`, where `:int` denotes the singleton type for the value `:int`.[5] However, such a precision is not always achievable, as demonstrated by the guard in the second clause of `test` (line 155). Since it is impossible to characterize by a type all and only the tuples where the first two elements are equal, we have to approximate this set. To represent the set of values that satisfy such guards, we use two types—an underapproximation and an overapproximation—referred to as

- the *surely accepted type* (since it contains only values for which the guard succeeds)
- the *possibly accepted type* (since it contains all the values that have a chance to satisfy the guard)[6]

For the guard in line 155, the surely accepted type is `{boolean(), ..}` since all tuples whose first element is a Boolean satisfy the guard; the possibly accepted type, instead, is `{term(), term(), ..} or {boolean()}` since the only values that may satisfy the guard are those with at least two elements or those with just one element of type `boolean()`. When the possibly accepted type and the surely accepted type coincide, they provide a precise characterization of the guard, as demonstrated in the two previous examples of guards (lines 144 and 154).

The type system uses these types to type case-expressions and multi-clause function definitions. In particular, to type a clause, the system computes all the values that are *possibly* accepted by its guard, minus all those that are *surely* accepted by a previous clause, and use this set of values to type the clause's body. For example, when declaring `test` to be of type `{term(), term(), ..} -> term()`, the system deduces that the argument of the first clause has type `{term(), integer(), ..} or {:int, term(), ..}`. For the second clause, the system subtracts the type above from the possibly accepted type of the second clause's guard (intersected with the input type, i.e., `{term(), term(), ..}`), yielding for `x` the type `{not(:int), not(integer()), ..}`, that is, all the tuples with at least two elements where the first is not `:int` (`not` *t* denotes a *negation type*, which types all the values that are not of type *t*) and the second is not an integer.

If the difference computed for some clause is empty, then the clause is redundant and a warning is issued. This happens, for instance, for the second clause of `test`, if we declare for the function `test` the type `{:int, term(), ..} -> term()`: all arguments will be captured by the first clause.

If the domain of the function (or, for case expressions, the type of the matched expression) is contained in the union of the *surely* accepted types of all the clauses, then the definition is exhaustive. For instance, this is the case if we declare for `test` the type `{term(), boolean()} -> term()`. If, instead, it is contained only in the union of the *possibly* accepted types, then the definition *may* not be exhaustive, and a warning is emitted as for declaring `{term(), term(), ..} or {boolean()} -> term()`. In all the other cases, the

[5]We use `{:int,term(), ..}` rather than `{:int, ..}`, since the absence of a second element would make the guard fail.

[6]Formally, the *surely accepted type* is the largest type contained in all types containing only values that satisfy the guard, and the *possibly accepted type* is the smallest type containing all types that contain only values that satisfy the guard.

definition is considered ill-typed, as for a declared type `tuple() -> term()` (where `tuple()` is the type of all tuples), since a tuple with a single element that is not a Boolean is an argument in the domain that cannot be handled by any clause.

Finally, the guard analysis of 6 produces for each guard a result that is more refined than just the possibly and surely accepted types for the guards. For each guard, the analysis partitions these types into smaller types that will then be used by the inference of 6.5 to produce a typing for non-annotated functions. For instance, for the untyped version of `test` given in lines 154–155 the analysis will produce four different input types that the inference of 6.5 will use to deduce the following intersection type for `test`:

```
156 $ ({term(), integer(), ..} -> integer()) and
157   ({:int, term(), ..} -> term()) and
158   ({boolean()} or {boolean(), not(integer()), ..} -> boolean()) and
159   ({not(boolean() or :int), not(integer()), ..} -> not(boolean() or :int)
```

Splitting the domain of `test` as in the code above is not so difficult since its guards use the connective `or` and, as we show, to compute the split, the system in Section 6 normalizes guards into Boolean disjunctions. Notice, however, that the analysis must take into account the order in which the guards are written. If in line 155 we use the guard `elem(x,0) == elem(x,1) or is_boolean(elem(x,0))`, that is, we swap the order of the operands of the guard, then the arrow type in line 158 is no longer correct, since the application `test({true})` would fail and, therefore, the type `{boolean()}` must not be included in the domain of the arrow in line 158.

Inference (Section 6.5) In a couple of examples we highlighted our system's ability to deduce the function type even in the absence of explicit type declarations. For instance, we said that our type system can infer for `negate` (lines 147–148) the intersection type in line 153, and for `test` (lines 154–155) the type in lines 156–159. This kind of inference is different from that performed for parametric polymorphism by languages of the ML family. Instead, it leverages the guard analysis of Section 6 to derive the type of guarded functions: it simply considers the guards of the different clauses of a function definition as implicit type declarations for the function parameters, and use them for type inference.

This kind of inference is used when explicit type declarations are omitted. This is particularly valuable for anonymous functions of which we saw a couple of examples in the definition of `curry` (lines 128–134) where the body of the two clauses consists of anonymous functions. We aim to avoid imposing an obligation on programmers to explicitly annotate anonymous functions as in:

```
$ list(integer()) -> list(integer())
def bump(lst), do: List.map(fn x when is_integer(x) -> x + 1 end, lst)
```

Here, the guard already provides the necessary information, making explicit annotations superfluous. Additionally, we view the use of an untyped or anonymous function as an implicit application of gradual typing. We have seen in §2, that whenever gradual typing was explicitly introduced by an annotation, the system propagated `dynamic()` in all intermediate results so as to preserve the versatility of the initial gradual typing. We do the same here and propagate the (implicit use of) `dynamic()` in the results of the anonymous/untyped functions by intersecting their inferred type with an extra arrow of the form t `-> dynamic()`, where t is the domain inferred for the function. For example, the type inferred for `negate` will be the type in line 153 intersected with the type `integer() or boolean() -> dynamic()`, while the intersection in lines 156–159 inferred for `test` will have an extra arrow `{term(), term(), ..} or {boolean()} -> dynamic()`. Likewise, the type `(integer() -> integer()) and (integer() -> dynamic())` will be given to the anonymous function in the body of `bump` (line 161); this type is equivalent to the simpler type `integer() -> integer() and dynamic()`. All these concepts are formalized in 6.5.

CHAPTER 3

FOUNDATIONS OF SET-THEORETIC TYPES

"The simplest way to avoid [the circularity between types, typing, and subtyping] is to break it, and the development we did so far clearly suggests where to break it."

Alain Frisch, *A Gentle Introduction to Semantic Subtyping* (2005).

Our work in typing Elixir builds upon decades of development of a proper foundation for set-theoretic types that would be expressive enough to capture the essence of a language like Elixir: that is, the *semantic subtyping framework*. Our starting point is effectively the calculus presented in Lanvin (2021): a lambda calculus with gradual set-theoretic types descending from the initial design of ℂDuce (Frisch, 2004) and extended with gradual types in Castagna et al. (2019).

A brief history of CDuce

ℂDuce, as a language, was designed to support the manipulation of XML documents, and semantic subtyping was developed as a theory to enhance the usual type syntax and constructors (functions, pairs, records) with set connectives (union, intersection, difference), and to make it possible to consider types as sets of values, and typing as denoting the "set of values produced by a program".

The initial design of ℂDuce–already with recursive types–was monomorphic. The addition of polymorphism (Castagna and Xu, 2011; Castagna et al., 2014; Castagna et al., 2015) to the framework came later, allowing types to be parameterized by type variables, with a technique of local type inference that automatically instantiates the type variables whenever a polymorphic function is applied.

Another important extension to the framework was the addition of gradual types by Castagna et al. (2019), which brought the "`dynamic`" type into the set-theoretic framework and thus opened the door to the development of a gradual set-theoretic system suitable for typing dynamic languages.

This framework is rich in expressiveness and allows powerful reasoning on types. Its development over the years makes it suitable for industry adoption: a notable use case so far is in game development, where it is used to type the Roblox *Lua*u (2022) scripting language.

Ongoing work. ℂDuce, both a formal and a programming language has been continuously developed. Its first denotational semantics was given by Lanvin (Lanvin, 2021), and is currently being extended with new features; for example, the polymorphic record types (Castagna and Peyrot, 2025a) are being implemented actively.

Chapter Roadmap

- **Section 3.1 (Static)**: Type algebra, interpretation, and subtyping-as-emptiness.
- **Section 3.2 (Gradual)**: Gradual semantics, precision, and consistent subtyping.
- **Section 3.3 (Implementation)**: We give some details on the implementation of a set-theoretic type system; we defer the representation of types to Part II and discuss algorithmic typing rules via type operators.

In this chapter, we recall the basic concepts and foundations of semantic subtyping, which will be used and extended throughout the thesis to fit the needs of Elixir. We first present (§3.1) the type algebra of static set-theoretic types, their interpretation, and the *decidable* type relations (subtyping, equivalence) that it induces. We then present (§3.2) gradual set-theoretic types and their own interpretation and relations (with the addition of precision and consistent subtyping).

Scope and limitations In this background, we present a monomorphic gradual version of set-theoretic types. The polymorphic version of semantic subtyping can be found in Castagna et al. (2014), Castagna et al. (2015), and Lanvin (2021), and consists of an extension of the interpretation of types to account for type variables. This limitation is orthogonal to our work and does not affect the applicability of our framework to Elixir typing. Polymorphism would require tracking type variables and environments in the interpretation and adapting decomposition rules to quantified structure, but does not alter the core algebra we exploit for Elixir's first-class features (unions, intersections, negation, tuples, and arrows).

3.1 Static Set-Theoretic Types

Throughout this chapter, we are not interested in the implementation of a type system for a given language; instead, we are concerned with the definition of an expressive type algebra that could then be fitted to a given language (in our case, Elixir).

3.1.1 Type algebra

We define a monomorphic nongradual (thus, static) type language, inspired by Lanvin (2021):

Definition 3.1.1 (Types)**.** *Let b range over a (finite) set of base types $\mathcal{T}_{base}$ and c range over a (infinite) set of constants $\mathscr{C}$ (e.g. atoms a and integers i). The set of static types $\mathcal{T}_{static}$ is defined coinductively (allowing infinite unfoldings) by the following grammar:*

$$\textit{Static Types} \quad t ::= c \mid b \mid t \times t \mid t \to t \mid t \vee t \mid \neg t \mid \mathbb{0}$$

but constrain them to ensure finite representability:

- ***(Regularity)** Any type term must have a finite number of distinct sub-terms.*
- ***(Contractivity)** Every infinite branch in the term structure must contain infinitely many occurrences of the $\times$ or $\to$ type constructors.*

Regularity ensures that the types, despite being defined in a coinductive way, are finitely representable. *Contractivity* prevents ill-formed types such as $t = t \vee t$ (which lacks information) or $t = \neg t$ (which cannot denote set).

The **bottom type**—or empty type—is denoted $\mathbb{0}$, and is a type that contains no values. It can be used to represent expressions that do not evaluate to any value (such as exceptions), or diverging functions. The **top type** is the type that contains all values, denoted $\mathbb{1}$. It is defined as $\mathbb{1} = \neg\mathbb{0}$, and can be thought of as the union of all the base types, and all the structural types (tuples and functions). This top type is not a placeholder: giving type $\mathbb{1}$ to an expression is quite restrictive, as it tells the type checker that any function applied to it must be a total function (valid on all inputs). It is the *type of all values* and, similar to Python's `object`, it is some sort of universal type that contains all other types and must usually be refined before use.

Remark (*On the meaning of 'top type'*)

It can denote, for example, in TypeScript, a type used as a placeholder for any type; informally, an expression of the top type can be used anywhere. In practice, TypeScript has two such top-like notions: `unknown`, a *safe* supertype of all values (you must refine it before use), and `any`, an *unsafe* escape hatch that is both supertype and subtype of everything and bypasses checking. The $\mathbb{1}$ in our system is just the former.

Our set-theoretic types include constants c (such as atoms and integers) which are singleton types representing exactly one value– for example, the atom `:ok` is a type that contains only the value `:ok`. The atom `:ok` can be given infinitely many types: `:ok`, `atom`, but also $\mathbb{1}$.

We also conveniently define **intersection**, as $t_1 \wedge t_2 \overset{\text{def}}{=} \neg(\neg t_1 \vee \neg t_2)$ and the type **difference**, $t_1 \setminus t_2 \overset{\text{def}}{=} t_1 \wedge \neg t_2$. This technique keeps the type algebra short and concise, but it does not mean that the union $\vee$ is more important than the others. It would be possible to set $\wedge$ as a primitive, and define $\vee$ as $\neg(\neg t_1 \wedge \neg t_2)$.

Design choice. In semantic subtyping, there is no specific constraint on the degree to which constants are exposed as singletons (also called *literal types*). Thus, it is a design choice to decide which constants are exposed as singletons. In TypeScript, for example, `string`, `number`, and `boolean` are all exposed as literal types (TypeScript Team, 2025). In the Elixir compiler, we arbitrarily chose to initially expose to the programmer *only* a singleton *atoms* (not integers but including booleans). The goal is to avoid value-check antipatterns (e.g., sprinkling guards like x>=0 everywhere to satisfy functions which expect positive integers) while still reaping the benefits of symbolic tags in pattern matching.

3.1.2 Semantics of static set-theoretic types

The static (nongradual) set-theoretic types are interpreted as sets of elements of a domain $\mathscr{D}$. Elements of the domain represent values of the language. We have constants and pairs; functions are represented as finite relations.

Definition 3.1.2 (Set-theoretic domain)**.** *The interpretation domain $\mathscr{D}$ for the static types of Definition 3.1.1 is the set of semantic values d produced inductively by the grammar:*

$$\begin{aligned} d &::= c \mid (d,d) \mid \{(\iota,\delta),\dots,(\iota,\delta)\} \\ \iota &::= d \mid \mho \\ \delta &::= d \mid \Omega \end{aligned}$$

where c ranges over the set $\mathscr{C}$ of constants, $\mho$ is a distinguished element representing undefined function inputs, and Ω represents runtime errors or stuck computations. We require that $\Omega,\mho \notin \mathscr{D}$, $\Omega \neq \mho$ and define $\mathscr{D}_\Omega = \mathscr{D} \cup \{\Omega\}$ and $\mathscr{D}_\mho = \mathscr{D} \cup \{\mho\}$.

Pairs (d,d) are used in the interpretation of product types, while the sets of pairs $\{(\iota,\delta),\dots,(\iota,\delta)\}$ are used for the interpretation of functions. The special element Ω in the codomain of functions represents a possible type error or a stuck term. The special element $\mho$, introduced in Lanvin (2021), is a distinguished "undefined" input that plays a purely semantic role in distinguishing function types with empty domain. It is a subtle but but necessary technical addition that we now explain in detail.

Ω is necessary to represent partial functions. In semantic subtyping as defined by Frisch (2004), the interpretation of a function type $t_1 \to t_2$ is defined as the set of all finite graphs from $\mathscr{D}$ to $\mathscr{D}_\Omega$ (that is, elements of $\mathscr{P}_f(\mathscr{D} \times \mathscr{D}_\Omega)$) that map elements of $[\![t_1]\!]$ to elements of $[\![t_2]\!]$. Formally,

$$[\![t_1 \to t_2]\!] = \left\{R \in \mathscr{P}_f(\mathscr{D} \times \mathscr{D}_\Omega) \mid \forall (x,y) \in R, \text{ if } x \in [\![t_1]\!] \text{ then } y \in [\![t_2]\!]\right\}. \qquad ([\![\to]\!])$$

The purpose of the presence of Ω in the codomain is to allow representing the graph of functions that are not well-defined on all inputs (using pairs of the form (d,Ω) for some $d \in \mathscr{D}$).

The empty domain problem. Notice that, in ($[\![\to]\!]$), if $[\![t_1]\!] = \emptyset$, this condition is vacuously satisfied, creating a situation where $[\![\mathbb{O} \to t]\!]$ becomes identical for all codomains t.

Solution via $\mho$. To break this approximation, we introduce a distinguished element $\mho \notin V$, which serves to float the defined codomain of the function, even when there exists no domain elements. We extend the interpretation domain to $[\![t_1]\!] \cup \{\mho\}$, and modify the interpretation clause as follows:

$$R \in [\![t_1 \to t_2]\!] \quad \text{iff} \quad \forall (x,y) \in R, \text{ if } x \in [\![t_1]\!] \cup \{\mho\} \text{ then } y \in [\![t_2]\!].$$

This forces each function to specify the behaviour at $\mho$, even when its real domain is empty. Consequently,

$$[\![\mathbb{0} \to t]\!] = \mathscr{P}_f\left((\mathscr{D} \times \mathscr{D}_\Omega) \cup \{(\mho, y) \mid y \in [\![t]\!]\}\right)$$

which is clearly covariant in t (so, for example, $[\![\mathbb{0} \to \texttt{int}]\!] \subsetneq [\![\mathbb{0} \to \mathbb{1}]\!]$).

Summary. The element $\mho$ acts as a universal probe that ensures arrow types remain *semantically distinguishable* even when their ordinary domain is empty. Its presence will be essential deciding subtyping for multi-arity functions (Theorem 4.4.5) and when integrating gradual functions (Chapter 5).

Definition 3.1.3 (Set-theoretic interpretation of static types)**.** *We introduce a binary predicate $\delta : t$ that captures when an element $\delta \in \mathscr{D}_\Omega$ belongs to the interpretation of type $t \in \mathscr{T}_{static}$. Let function $\mathscr{C}(\cdot) : \mathscr{T}_{base} \to \mathscr{P}(\mathscr{C})$ be the function that (expectedly, e.g. $\mathscr{C}(\texttt{int}) = \mathbb{Z}$) maps each base type to the set of constants that belong to it. This predicate is defined by induction on the pair (δ, t) ordered lexicographically:*

$$\begin{array}{rcll} c : b & = & c \in \mathscr{C}(b) & \\ (d_1, d_2) : t_1 \times t_2 & = & (d_1 : t_1) \ \texttt{and} \ (d_2 : t_2) & \\ \{(\iota_i, \delta_i) \mid i \in I\} : t_1 \to t_2 & = & \forall i \in I.\ (\iota_i = \mho \ \texttt{or} \ \iota_i : t_1) \implies \delta_i : t_2 & \\ d : t_1 \vee t_2 & = & (d : t_1) \ \texttt{or} \ (d : t_2) & \\ d : \neg t & = & \texttt{not} \ (d : t) & \\ \delta : t & = & \texttt{false} & \textit{otherwise} \end{array}$$

This allows us to define the set-theoretic interpretation *of types $[\![.]\!] : \mathscr{T}_{static} \to \mathscr{P}(\mathscr{D})$ by setting $[\![t]\!] \overset{def}{=} \{d \in \mathscr{D} \mid d : t\}$.*

Remark 3.1.4. *The point of set-theoretic types is to allow us to compose types using set intuitions, with type connectives such as $\vee$, $\wedge$, $\neg$ as set operations ("the type $t_1 \vee t_2$ contains values that are in t_1 or t_2"). All the set-theoretic properties expected from our types happen as a corollary to the interpretation defined above:*

1. *Since $\mathbb{0}$ matches no predicate (but the last false one), we immediately have $[\![\mathbb{0}]\!] = \varnothing$. Since not $(d : \mathbb{0})$ is always true, we have $[\![\mathbb{1}]\!] = \mathscr{D}$.*
2. *From the predicate for union, it is immediate that $[\![t_1 \vee t_2]\!] = [\![t_1]\!] \cup [\![t_2]\!]$. Also, for $t_1, t_2 \in \mathscr{T}_{static}$, we have (through De Morgan's laws):*

$$[\![\neg(\neg t_1 \vee \neg t_2)]\!] = \mathscr{D} \setminus (\mathscr{D} \setminus [\![t_1]\!] \cup \mathscr{D} \setminus [\![t_2]\!]) = [\![t_1]\!] \cap [\![t_2]\!]$$

3. *For any type t, we have $[\![\neg t]\!] = \mathscr{D} \setminus [\![t]\!]$.*

Assumption 3.1.5. *We add the **assumption** that base types are all two-by-two disjoint $\mathscr{T}_{base}$, that is, for every $b_1, b_2 \in \mathscr{T}_{base}$, we have $\mathscr{C}(b_1) \cap \mathscr{C}(b_2) = \varnothing$. This poses no problem: base types are arbitrary labels on sets of constants, and if* `int` $\in \mathscr{T}_{base}$ *and* `float` $\in \mathscr{T}_{base}$, *then we can define, say,* `number` *(the type of numerical values in Elixir), as* `int` $\vee$ `float`.

3.1.2 a) Interpretation of product types

The interpretation of pairs is the Cartesian product: it is the set of pairs whose components are in each component type. Formally, $[\![t_1 \times t_2]\!] = [\![t_1]\!] \times [\![t_2]\!]$. Since $[\![\mathbb{0}]\!] = \varnothing$, the interpretation of $\mathbb{0} \times$ `int` is $\varnothing \times [\![\texttt{int}]\!] = \varnothing$, making $\mathbb{0} \times t$ equivalent to $\mathbb{0}$ for any type t (and similarly for $t \times \mathbb{0}$).

The type of all pairs is `pair` $= \mathbb{1} \times \mathbb{1}$.

3.1.2 b) Interpretation of arrow types

The semantics of an arrow type $t_1 \to t_2$ is the set of finite relations R that satisfy: for every input-output pair $(\iota, \delta) \in R$, if $\iota \in [\![t_1]\!] \cup \{\mho\}$ then $\delta \in [\![t_2]\!]$.

This definition allows functions to have arbitrary behaviour outside their declared domain. For example, consider the Elixir function:

```
def negate(x) when is_integer(x), do: -x
def negate(x) when is_boolean(x), do: not x
```

can be assigned type (`int` $\to$ `int`) because, on integer inputs, it returns an integer.

The interpretation $[\![\texttt{int} \to \texttt{int}]\!]$ contains all the relations that:

- Map integers to integers: $\{(n,m) \mid n,m \in \mathbb{Z}\}$
- Have arbitrary behaviour on non-integers: $\{(x,\delta) \mid x \notin \mathbb{Z}$ or $x = \mho, \delta \in \mathscr{D}_\Omega\}$

Observation 3.1.6. *This permissive interpretation enables nonempty arrow intersections. For example,* (`int` $\to$ `int`) $\wedge$ (`bool` $\to$ `bool`) *is nonempty because it contains relations that behave correctly on both integers and booleans, such as:* $\{(3,-3), (\texttt{true}, \texttt{false})\}$.

This design choice may seem counterintuitive, as it allows giving to functions a function type whose domain is partial. But it is essential for semantic subtyping since it allows intersection types to be meaningful and enables precise type reasoning while maintaining decidability.

Remark (*The meaning of function negations*)

The function negations appear in the theory in the form of $(t_i \to s_i) \setminus (t'_i \to s'_i)$. They are not useful to programmers, but their existence is required to decide emptiness for function types. However, their meaning (which is derived from the set interpretation) can be explained.

Consider the example of (`int` $\vee$ `float`) $\to$ (`int` $\vee$ `float`) and *not* `int` $\to$ `int`.

If a function has this type, then it returns (`int` $\vee$ `float`) on all (`int` $\vee$ `float`) values, BUT it does not return an integer for every integer value.

So there should be at least one integer on which it returns both (`int` $\vee$ `float`) but not `int`, that is, it returns a float. This property is too specific to be used for typing in our system.

The type of all functions is `arrow` $= \mathbb{0} \to \mathbb{1}$. This is a better candidate than $\mathbb{1} \to \mathbb{1}$, which represents all *total* functions (i.e. they are all defined on all inputs). With the interpretation of Definition 3.1.3, we can see that since $[\![\mathbb{0}]\!] = \varnothing$, every relation $R \in \mathscr{P}_f(\mathscr{D}_{\mho} \times \mathscr{D}_{\Omega})$ is in $[\![\mathbb{0} \to \mathbb{1}]\!]$, thus $[\![\mathbb{0} \to \mathbb{1}]\!] = \mathscr{P}_f(\mathscr{D}_{\mho} \times \mathscr{D}_{\Omega})$ which confirms it to be the type of all functions (as every function interpretation will be included in it) [1].

3.1.3 Subtyping as set inclusion

Having defined the interpretation of types, we define the subtyping relation and show how to decide it. The role of subtyping is to dictate whether it is *statically sound* to pass a value to a context, knowing the type of the value and the type expected by the context. Since we interpret types as sets of values, we define subtyping by set containment: If a context can accept values from a set S, then it can accept values from any smaller set $S' \subseteq S$.

Definition 3.1.7 (Subtyping)**.** *For all* $t_1, t_2 \in \mathscr{T}_{static}$, *we define* $(t_1 \leq t_2) \Leftrightarrow ([\![t_1]\!] \subseteq [\![t_2]\!])$ *and* $(t_1 \simeq t_2) \Leftrightarrow (t_1 \leq t_2) \wedge (t_2 \leq t_1)$.

Proposition 3.1.8 (Subtyping $\Leftrightarrow$ emptiness)**.** *For all* $t_1, t_2 \in \mathscr{T}_{static}$,

$$t_1 \leq t_2 \iff t_1 \wedge \neg t_2 \leq \mathbb{0} \iff [\![t_1 \wedge \neg t_2]\!] = \varnothing.$$

Proof. Immediate from Definition 3.1.7 and $[\![t_1 \wedge \neg t_2]\!] = [\![t_1]\!] \cap (\mathscr{D} \setminus [\![t_2]\!]) = \varnothing \iff [\![t_1]\!] \subseteq [\![t_2]\!]$. □

The proposition tells us we can reduce ANY subtyping question to an emptiness question. Instead of asking "is every value in t_1 also in t_2?", we ask "are there any values in t_1 that are NOT in t_2?" This reduction is computationally powerful because it allows us to use the decidability of emptiness to decide subtyping.

3.1.4 Types as disjunctive normal forms (DNFs)

A **major** property of set-theoretic types is that they can always be decomposed (that is, they are equivalent) to a disjunctive normal form (DNF), where the atoms of the DNF are constants, base types, products, and arrows.

Definition 3.1.9 (DNF)**.**
- *A* literal *is either an atom a (where a ranges over $c, b, (t \times t'), (t \to t')$) or its negation $\neg a$.*
- *A* clause *is a finite conjunction of literals*

[1] The interested reader can find more computations of interpretations in Remark 13.2.1.

- *A* DNF *is a finite disjunction of clauses, of the form:*

$$\bigvee_{i\in I}\Big(\bigwedge_{a\in P_i} a \wedge \bigwedge_{a'\in N_i} \neg a'\Big)$$

where a, a' range over all atomic types, which are types of the form $c, b, (t \times t'), (t \to t')$, and P_i, N_i are finite sets of literals.

We adopt the standard convention: $\mathbb{0} \equiv \bigvee_{\emptyset} -$ (empty disjunction). We also know $\mathbb{1}$ to be a union of literals, as $\mathbb{1} = \bigvee_{b\in\mathcal{T}_{\text{base}}} b \vee (\mathbb{1} \times \mathbb{1}) \vee (\mathbb{0} \to \mathbb{1})$.

Observation 3.1.10. *The DNF representation is crucial for the theory* ***and*** *for the implementation of set-theoretic types, because every algorithm dealing with types is defined on their DNF form. A first example of that is the algorithm for deciding emptiness of types, and we later show that type operators (such as computing the domain of a function, or its application to values of a given type) are also defined on their DNF form.*

If the syntactic form of a type is given by Definition 3.1.3, then the DNF form can be considered to be its semantic form. There is no issue in keeping types in their syntactic form, because the DNF form both always exists and can be directly computed, as seen in the following theorem.

Theorem 3.1.11. *There exists a function $\mathcal{U}^+$ that maps types t to their DNF form, such that for all $t \in \mathcal{T}_{static}$,*

$$t \simeq \mathcal{U}^+(t) = \bigvee_{i\in I}\Big(\bigwedge_{a\in P_i} a \wedge \bigwedge_{a'\in N_i} \neg a'\Big)$$

where a, a' range over all atomic types of the form c, b, $(t \times t')$, and $(t \to t')$.

Proof. Lanvin (2021) gives a straightforward proof, defining functions $\mathcal{U}^+$ and $\mathcal{U}^-$ by induction on the structure of the type t.

Definitions (structural).

$$
\begin{array}{rcl@{\qquad}rcl}
\mathcal{U}^+(\mathbb{0}) &=& \mathbb{0} & \mathcal{U}^-(\mathbb{0}) &=& \mathbb{1} \\
\mathcal{U}^+(b) &=& b & \mathcal{U}^-(b) &=& \neg b \\
\mathcal{U}^+(c) &=& c & \mathcal{U}^-(c) &=& \neg c \\
\mathcal{U}^+(t_1 \to t_2) &=& t_1 \to t_2 & \mathcal{U}^-(t_1 \to t_2) &=& \neg(t_1 \to t_2) \\
\mathcal{U}^+(t_1 \times t_2) &=& t_1 \times t_2 & \mathcal{U}^-(t_1 \times t_2) &=& \neg(t_1 \times t_2) \\
\mathcal{U}^+(\neg t) &=& \mathcal{U}^-(t) & \mathcal{U}^-(\neg t) &=& \mathcal{U}^+(t) \\
\mathcal{U}^+(t_1 \vee t_2) &=& \mathcal{U}^+(t_1) \ \vee \ \mathcal{U}^+(t_2) & \mathcal{U}^-(t_1 \vee t_2) &=& \bigvee_{i\in I, j\in J} \Big(\bigwedge_{a\in P_i\cup P_j} a \wedge \bigwedge_{a'\in N_i\cup N_j} \neg a' \Big)
\end{array}
$$

$$
\text{where } \mathcal{U}^-(t_1) = \bigvee_{i\in I} \Big(\bigwedge_{a\in P_i} a \wedge \bigwedge_{a'\in N_i} \neg a' \Big)
$$

$$
\text{and} \quad \mathcal{U}^-(t_2) = \bigvee_{j\in J} \Big(\bigwedge_{a\in P_j} a \wedge \bigwedge_{a'\in N_j} \neg a' \Big)
$$

The proof proceeds by mutual induction on the structure of t, showing that $\mathcal{U}^+(t) \simeq t$ and $\mathcal{U}^-(t) \simeq \neg t$ and that $\mathcal{U}^+(t)$ and $\mathcal{U}^-(t)$ are in DNF. $\square$

Decomposing types into components This decomposition goes further: let $\texttt{base} \overset{def}{=} \bigvee_{b\in\mathcal{T}_{\text{base}}} b$. Since $\mathbb{1} = \texttt{base} \vee \texttt{pair} \vee \texttt{arrow}$, we can write $t = \mathbb{1} \wedge t = (\texttt{base} \wedge t) \vee (\texttt{pair} \wedge t) \vee (\texttt{arrow} \wedge t)$ where every disjunct is disjoint by disjointness of base types and pair/arrow (which is obvious from the domain definition). When applying the formula above to filter the atoms a in the expression of the DNF above, this means we can write each part of any type t into three DNFs: one for base types, one for products, and one for arrows.

If we especially consider the three disjoint sub-types $(\texttt{base} \wedge t)$, $t \wedge \texttt{pair}$ and $t \wedge \texttt{arrow}$, we can write each of them in a disjunctive normal form (DNF):

Theorem 3.1.12 (Type decomposition)**.** *For all $t \in \mathcal{T}_{static}$, the type t can be decomposed into three disjoint components, each expressible in disjunctive normal form:*

$$
\begin{array}{rr}
\textit{(Base types)} & t \wedge \texttt{base} \simeq \bigvee_{i\in I_b} \bigwedge_{\ell\in P_i} \ell \wedge \bigwedge_{\ell'\in N_i} \neg \ell' \ \wedge \ \texttt{base} \\
\textit{(Products)} & t \wedge \texttt{pair} \simeq \bigvee_{i\in I_\times} \bigwedge_{(t,t')\in P_i} t \times t' \wedge \bigwedge_{(s,s')\in N_i} \neg(s \times s') \ \wedge \ \texttt{pair} \\
\textit{(Arrows)} & t \wedge \texttt{arrow} \simeq \bigvee_{i\in I_\to} \bigwedge_{(t,s)\in P_i} (t \to s) \wedge \bigwedge_{(t',s')\in N_i} \neg(t' \to s') \ \wedge \ \texttt{arrow}
\end{array}
$$

where, in the base type component, ℓ, ℓ' range ***only over constants*** *c* ***and base types*** *b).*

Note

The above decomposition is really important and establishes the basis of any implementation of semantic set-theoretic types. Each element of the union can be implemented as its own disjoint component type, which will allow us to decide emptiness of each component type independently, but also to compute type operators intersection, union, difference independently for each component type. Consider indeed the fact that given $t_1, t_2 \in \mathscr{T}_{\text{static}}$, and given an operator $\odot \in \{\wedge, \vee, \smallsetminus\}$, we have:

$$t_1 \odot t_2 = \bigvee_{b \in \mathscr{T}_{\text{base}}} (b \wedge t_1) \odot (b \wedge t_2) \\ \bigvee (\texttt{pair} \wedge t_1) \odot (\texttt{pair} \wedge t_2) \\ \bigvee (\texttt{arrow} \wedge t_1) \odot (\texttt{arrow} \wedge t_2)$$

which shows that an implementation of set-theoretic types can be made modular on each component type that is chosen (which in practice, will not only consist of products and arrows–see for instance the addition in Chapter 4 of multi-arity functions).

Theorem 3.1.20 shows that subtyping is decidable if and only if emptiness is decidable. With the above decomposition, what is left is to show that emptiness of DNFs of base types, products and arrows is decidable.

3.1.5 Deciding emptiness on DNFs

Our approach has the originality of operating on the DNF representation rather than syntactic forms, enabling a unified decision procedure that handles all type components without ad-hoc case analysis.

3.1.5 a) Emptiness of base types

For this simple case, consider a DNF whose atoms ℓ, ℓ' are only constants c and base types b.

$$\bigvee_{i \in I} \bigwedge_{\ell \in P_i} \ell \wedge \bigwedge_{\ell' \in N_i} \neg \ell'$$

For a fixed clause $i \in I$, examine the positive part P_i:

- If P_i contains two distinct constants $c \neq c'$, the intersection is empty.
- If P_i contains two distinct base types $b \neq b'$, the intersection is empty (Assumption 3.1.5).
- If P_i contains a constant c and a base type b with $c \notin b$, the intersection is empty.
- If P_i contains a base type b and a constant $c \in b$, the intersection collapses to the constant c.

Hence $\bigwedge_{\ell \in P_i} \ell$ normalizes to a single atomic type ℓ_i, which is either a constant c_i or a base type b_i (otherwise the clause is empty).

Now examine the negative part N_i:

- Any negated atom disjoint from ℓ_i is redundant and can be removed.
- If N_i contains a base type that is larger than ℓ_i, the clause is empty.
- If ℓ_i is a constant and that constant also occurs in N_i, the clause is empty.

Therefore, the only non-empty clauses are of the form c or $b \wedge \bigwedge_{j=1}^{k} \neg c_j$ where each $c_j \in b$; equivalently, $b \wedge \neg\big(\bigvee_{j=1}^{k} c_j\big)$.

3.1.5 b) Emptiness of products

The core insight underlying this decision procedure is the systematic elimination of top-level differences to isolate positive product literals, whose emptiness can then be checked component-wise. To illustrate this idea, consider the type $(\mathbb{1} \times \mathbb{1}) \setminus (\texttt{int} \times \texttt{int})$, representing all pairs except integer pairs. This type is non-empty, since it is equivalent to the union of (trivially) non-empty product types $(\texttt{int} \times \neg\texttt{int}) \cup (\neg\texttt{int} \times \mathbb{1})$.

Formally, the central problem is to decide the emptiness of the product component:

$$\bigvee_{i \in I} \bigwedge_{(t,t') \in P_i} t \times t' \wedge \bigwedge_{(s,s') \in N_i} \neg(s \times s') \wedge \texttt{pair} \simeq \mathbb{0} \qquad (\times_1)$$

where P_i and N_i denote the positive and negative literals of the DNF representation.

Through our set-theoretic interpretation, which preserves structure under $\cup$, $\cap$, and $\times$ (interpreted as Cartesian product), this reduces to the equivalent set-theoretic decision problem:

$$\bigcup_{i \in I} \Big(\bigcap_{(t,t') \in P_i} [\![t]\!] \times [\![t']\!] \cap \bigcap_{(s,s') \in N_i} \overline{[\![s]\!] \times [\![s']\!]}^{[\![\texttt{pair}]\!]} \Big) = \varnothing \qquad (\times_2)$$

Note that this DNF lives in the space of `pair` (which was present in ($\times_1$) as an intersection), which justifies why products are complemented in the universe $[\![\texttt{pair}]\!]$.

Observation 3.1.13. ***Fundamental insight.*** *The transformation from ($\times_1$) to ($\times_2$) consists in translating a type-theoretic emptiness problem into a set-theoretic containment problem. The decidability of the latter, which we prove now, provides definitive answers to the former. This principle extends to all structural types in semantic subtyping. This makes the choice of interpretation for each type a critical design decision that determines the tractability of subtyping.*

We solve this problem by formulating a general solution to the underlying set-theoretic containment problem. Given index sets $i \in I$ and corresponding pairs P_i, N_i of domain subsets, we must decide:

$$\bigcup_{i \in I} \Big(\bigcap_{(X,Y) \in P_i} (X \times Y) \cap \bigcap_{(X',Y') \in N_i} (\mathscr{D} \times \mathscr{D}) \setminus (X' \times Y') \Big) = \varnothing \qquad (\times_3)$$

where we have replaced type expressions with their corresponding domain subsets.

This formulation admits significant simplification through fundamental properties of Cartesian products and set operations. The intersection distributivity property $\bigcap_{(X,Y) \in P_i} (X \times Y) = (\bigcap_{(X,\_) \in P_i} X) \times (\bigcap_{(\_,Y) \in P_i} Y)$ allows us to factor positive constraints. Similarly, we can rewrite negative constraints by rewriting intersections of complements as differences against unions. Finally, since a union is empty if and only if all its elements are empty, this yields the simplified problem:

$$\forall i \in I. \qquad X_i \times Y_i \setminus \left(\bigcup_{(X',Y')\in N_i} X' \times Y' \right) = \varnothing \tag{$\times_4$}$$

The emptiness of such unions reduces to component-wise emptiness checking, since $X \times Y = \varnothing$ if and only if $X = \varnothing$ or $Y = \varnothing$. Following our earlier example, we decompose these differences using the following lemma:

Lemma 3.1.14. *For all X, Y subsets of $\mathscr{D}$, and N a set of pairs of subsets of $\mathscr{D}$, we have*

$$(X \times Y) \setminus (\bigcup_{(X',Y')\in N} X' \times Y') = \bigcup_{N' \subseteq N} \left(X \setminus (\bigcup_{(X',Y')\in N'} X') \right) \times \left(Y \setminus (\bigcup_{(X',Y')\in N\setminus N'} Y') \right)$$

Proof. Let

$$L = (X \times Y) \setminus \Big(\bigcup_{(X',Y')\in N} X' \times Y' \Big) \quad \text{and} \quad R = \bigcup_{N' \subseteq N} \Big(X \setminus \bigcup_{(X',Y')\in N'} X' \Big) \times \Big(Y \setminus \bigcup_{(X',Y')\in N\setminus N'} Y' \Big).$$

We prove $L = R$ by mutual inclusion.

($L \subseteq R$). Pick $(x, y) \in L$. Then $x \in X$, $y \in Y$, and for all $(X', Y') \in N$ we have

$$(x, y) \notin X' \times Y' \iff x \notin X' \text{ or } y \notin Y'.$$

Define $$N_x = \{(X', Y') \in N \mid x \notin X'\}$$

By construction, $x \notin \bigcup_{(X',Y')\in N_x} X'$, hence $x \in X \setminus \bigcup_{(X',Y')\in N_x} X'$. Moreover, if $(X', Y') \in N \setminus N_x$ then $x \in X'$, so the disjunction above forces $y \notin Y'$. Hence $y \notin \bigcup_{(X',Y')\in N\setminus N_x} Y'$, i.e. $y \in Y \setminus \bigcup_{(X',Y')\in N\setminus N_x} Y'$. Therefore (x, y) belongs to the term of R indexed by $N' = N_x$, so $(x, y) \in R$.

($R \subseteq L$). Conversely, take $(x, y) \in R$. Then there exists $N' \subseteq N$ such that

$$x \in X \setminus \bigcup_{(X',Y')\in N'} X' \quad \text{and} \quad y \in Y \setminus \bigcup_{(X',Y')\in N\setminus N'} Y'.$$

Hence $(x, y) \in X \times Y$. Let $(X', Y') \in N$ be arbitrary. If $(X', Y') \in N'$ then $x \notin X'$; if $(X', Y') \notin N'$ then $y \notin Y'$. In either case, $(x, y) \notin X' \times Y'$, and thus $(x, y) \notin \bigcup_{(X',Y')\in N} X' \times Y'$. Therefore $(x, y) \in L$. □

Interpreting this lemma in our type-theoretic context with $X = [\![t]\!]$, $Y = [\![s]\!]$, and N as a collection of type interpretation pairs (t', s'), the decomposition establishes that emptiness holds precisely when every subset $N' \subseteq N$ of negative constraints forces either the left component t or the right component s to be subsumed by the corresponding union of negative constraints. This characterization provides the foundation for our emptiness decision algorithm.

The lemma immediately suggests an algorithmic approach: it reduces the emptiness problem for product differences to emptiness checking of their constituent components. This reduction enables us to transform the original product emptiness problem into a union emptiness problem, yielding our main theorem:

Theorem 3.1.15 (Emptiness of products)**.**

$$\bigvee_{i\in I}\Big(\bigwedge_{(t,t')\in P_i} t\times t' \wedge \bigwedge_{(s,s')\in N_i} \neg(s\times s')\Big) \leq \mathbb{0}$$
$$\Leftrightarrow \forall i\in I.\ \forall N'\subseteq N_i.\ \Big(\bigwedge_{(t,t')\in P_i} t \leq \bigvee_{(s,s')\in N'} s \ \vee \bigwedge_{(t,t')\in P_i} t' \leq \bigvee_{(s,s')\in N_i\setminus N'} s'\Big)$$

Proof. The result follows directly from Lemma 3.1.14 and the preceding reduction of the problem to its ($\times_4$) form. □

Under the regularity axiom, the set of distinct product atoms reachable from the input is finite; hence the set of obligations generated by Theorem 3.1.15 is finite. The emptiness procedure is a saturation algorithm over this finite set, adding consequences until a fixpoint. Using memoization ensures each obligation is processed at most once, so the algorithm terminates.

3.1.5 c) Emptiness of arrows

With product emptiness decided, we turn to arrow types, which are slightly more complex. Unfortunately, the product decomposition lemma (Lemma 3.1.14) does not directly extend to arrows, because intersections of arrow types cannot be simplified as easily as intersections of products. Intuitively, this reflects the fact that arrow types must preserve the precise dependency between inputs and outputs, preventing naive merging of function types.

Nevertheless, we follow the same high-level strategy: reduce the type-theoretic emptiness question to a set-containment problem on the interpretation domain, then solve it using specialized lemmas. The development closely follows Frisch (2004).

The central problem is to decide the emptiness of:

$$\bigvee_{i\in I}\Big(\bigwedge_{(t,s)\in P_i} (t\to s) \wedge \bigwedge_{(t',s')\in N_i} \neg(t'\to s')\Big) \simeq \mathbb{0} \tag{$\to_1$}$$

which is equivalent to:

$$\bigvee_{i\in I}\Big(\bigwedge_{(t,s)\in P_i} [\![t\to s]\!] \wedge \bigwedge_{(t',s')\in N_i} \overline{[\![t'\to s']\!]}^{[\![\mathtt{arrow}]\!]}\Big) = \varnothing \tag{$\to_2$}$$

Now, if product was defined as the Cartesian product, and made it easy to write $[\![t\times s]\!] = [\![t]\!]\times[\![s]\!]$, what kind of set is the interpretation of an arrow? It is composed of all finite relations of pairs of elements (x,y) for $x\in[\![t]\!]\cup\{\mho\}$ and $y\in[\![s]\!]$, or elements (d,Ω) for $d\in\mathscr{D}\setminus[\![t]\!]$.

Remark 3.1.16. *The element $\mho$ in the domain of arrow interpretations was not present in the original proofs of Frisch (2004), as it was introduced later in Lanvin (2021) to support* *gradual typing. While $\mho$ does not affect the decidability results for simple arrows presented in this chapter, it becomes essential in Chapters 4 and 5 when we extend to multi-arity functions and strong arrow types. For completeness, we already include $\mho$ in all arrow interpretations throughout this chapter.*

We can express this set as:

Proposition 3.1.17. *For all t, s types, we have:*

$$[\![t \to s]\!] = \mathscr{P}_f\left(\overline{([\![t]\!] \cup \{\mho\}) \times \overline{[\![s]\!]}^{\mathscr{D}_\Omega}}^{\mathscr{D}_\mho \times \mathscr{D}_\Omega}\right) \tag{1}$$

Proof. Unfold the definition of $[\![t \to s]\!]$: a value of arrow type is a *finite* relation $R = \{(\iota_i, \omega_i) \mid i \in I\}$ such that $\forall i \in I. (\iota_i = \mho \text{ or } \iota_i \in [\![t]\!]) \implies \omega_i \in [\![s]\!]$. This is equivalent to saying that every pair in R is either

- of the form (x, y) with $x \in [\![t]\!] \cup \{\mho\}$ and $y \in [\![s]\!]$; or
- of the form (x, Ω) with $x \in \mathscr{D} \setminus ([\![t]\!])$.

Hence R is an arbitrary *finite* subset of

$$\big(([\![t]\!] \cup \{\mho\}) \times [\![s]\!]\big) \cup \big((\mathscr{D} \setminus [\![t]\!]) \times \mathscr{D}_\Omega\big). \tag{$\star$}$$

We use the following general identity, which holds for all sets A_1, A_2, B_1, B_2:

$$(A_1 \times A_2) \setminus (B_1 \times B_2) = [(A_1 \cap B_1) \times (A_2 \smallsetminus B_2)] \cup [(A_1 \smallsetminus B_1) \times A_2]$$

to expand the complement of the right-hand side of (1), which gives:

$$\begin{aligned}(\mathscr{D}_\mho \times \mathscr{D}_\Omega) \setminus ([\![t]\!] \cup \{\mho\}) \times (\mathscr{D}_\Omega \setminus [\![s]\!]) &= [([\![t]\!] \cup \{\mho\}) \times (\mathscr{D}_\Omega \setminus (\mathscr{D}_\Omega \setminus [\![s]\!]))] \cup [(\mathscr{D}_\mho \setminus ([\![t]\!] \cup \{\mho\})) \times \mathscr{D}_\Omega] \\ &= [([\![t]\!] \cup \{\mho\}) \times [\![s]\!]] \cup [(\mathscr{D} \setminus [\![t]\!]) \times \mathscr{D}_\Omega]\end{aligned}$$

which is exactly the set in ($\star$). □

Let us define then, for all sets X, Y, writing $X^\mho$ for $X \cup \{\mho\}$, the set $X \to Y$ as:

$$X \to Y \overset{\text{def}}{=} \mathscr{P}_f\left(\overline{X^\mho \times \overline{Y}^{\mathscr{D}_\Omega}}^{\mathscr{D}_\mho \times \mathscr{D}_\Omega}\right)$$

We can then rewrite the problem ($\to_2$) into a pure set form:

$$\bigcup_{i \in I}\Big(\Big(\bigcap_{(X,Y) \in P_i} X \to Y\Big) \setminus \big(\bigcup_{(X',Y') \in N_i} X' \to Y'\big)\Big) = \varnothing \tag{$\to_3$}$$

where we note that, unfortunately, unlike for products, there is no simple way to simplify the intersection of positive arrows. This is intuitive, as we know that if a pair is both a pair of floats and a pair of numbers, we know it's a pair of integers. But if a function is both a function from floats to floats and a function from numbers to numbers, we do not want to merge those into a single function type, because we require the precise dependency between the arguments and the result to be preserved.

Expanding the definition of $X \to Y$, we get:

$$\bigcup_{i\in I}\left(\bigcap_{(X,Y)\in P_i}\mathscr{P}_f\left(\overline{X^{\mho}\times\overline{Y}^{\mathscr{D}_\Omega}}^{\mathscr{D}_\mho\times\mathscr{D}_\Omega}\right)\right)\setminus\left(\bigcup_{(X',Y')\in N_i}\mathscr{P}_f\left(\overline{X'^{\mho}\times\overline{Y'}^{\mathscr{D}_\Omega}}^{\mathscr{D}_\mho\times\mathscr{D}_\Omega}\right)\right)=\varnothing \tag{$\to_4$}$$

Now, since a union is empty if and only if all its elements are empty, and for all sets A, B, we have $A \setminus B = \varnothing \iff A \subseteq B$, this problem is equivalent to:

$$\forall i \in I.\quad \bigcap_{(X,Y)\in P_i}\mathscr{P}_f\left(\overline{X^{\mho}\times\overline{Y}^{\mathscr{D}_\Omega}}^{\mathscr{D}_\mho\times\mathscr{D}_\Omega}\right)\subseteq\bigcup_{(X',Y')\in N_i}\mathscr{P}_f\left(\overline{X'^{\mho}\times\overline{Y'}^{\mathscr{D}_\Omega}}^{\mathscr{D}_\mho\times\mathscr{D}_\Omega}\right) \tag{$\to_5$}$$

Note here that the problem above is about the inclusion of an intersection of parts of sets in a union of parts of sets. This problem can be further decomposed by this general lemma on sets:

Lemma 3.1.18 (Powerset-intersection criterion)**.** *Let* $(A_i)_{i\in I}$ *and* $(B_i)_{j\in J}$ *be abstract sets.*

$$\bigcap_{i\in I}\mathscr{P}_f(A_i)\subseteq\bigcup_{j\in J}\mathscr{P}_f(B_j)\iff\exists j\in J.\ \bigcap_{i\in I}A_i\subseteq B_j.$$

Proof. Two basic facts:

$$\text{(i)}\quad \bigcap_{i\in I}\mathscr{P}_f(A_i)=\mathscr{P}_f\left(\bigcap_{i\in I}A_i\right)\quad\text{and}\quad\text{(ii}'\text{)}\quad\mathscr{P}_f(S)\subseteq\bigcup_{j\in J}\mathscr{P}_f(B_j)\iff\exists j\in J.\ S\subseteq B_j$$

For (i): $X\in\bigcap_i\mathscr{P}_f(A_i)$ iff $(\forall i)\ X\subseteq A_i$ iff $X\subseteq\bigcap_i A_i$ iff $X\in\mathscr{P}_f(\bigcap_i A_i)$.
For (ii′): "$\Rightarrow$" since $S\in\mathscr{P}_f(S)$, inclusion gives $S\in\bigcup_j\mathscr{P}_f(B_j)$, hence $S\subseteq B_j$ for some j. "$\Leftarrow$" if $S\subseteq B_{j_0}$, monotonicity yields $\mathscr{P}_f(S)\subseteq\mathscr{P}_f(B_{j_0})\subseteq\bigcup_j\mathscr{P}_f(B_j)$.
Now combine (i) and (ii′):

$$\bigcap_i\mathscr{P}_f(A_i)\subseteq\bigcup_j\mathscr{P}_f(B_j)\iff\mathscr{P}_f\left(\bigcap_i A_i\right)\subseteq\bigcup_j\mathscr{P}_f(B_j)\iff\exists j.\ \bigcap_i A_i\subseteq B_j\qquad\square$$

Using Lemma 3.1.18, we can rewrite the problem ($\to_5$) as:

$$\forall i\in I.\ \exists(X',Y')\in N_i.\ \bigcap_{(X,Y)\in P_i}\overline{X^{\mho}\times\overline{Y}^{\mathscr{D}_\Omega}}^{\mathscr{D}_\mho\times\mathscr{D}_\Omega}\subseteq\overline{X'^{\mho}\times\overline{Y'}^{\mathscr{D}_\Omega}}^{\mathscr{D}_\mho\times\mathscr{D}_\Omega} \tag{$\to_6$}$$

By applying complementation (which reverses inclusions), and De Morgan's law to rewrite the intersection of complemented sets as a union of these sets, we transform the problem ($\to_6$) into:

$$\forall i\in I.\ \exists(X',Y')\in N_i.\ \bigcup_{(X,Y)\in P_i}X^{\mho}\times\overline{Y}^{\mathscr{D}_\Omega}\supseteq X'^{\mho}\times\overline{Y'}^{\mathscr{D}_\Omega} \tag{3.1.1}$$

or, written otherwise:

$$\forall i \in I.\ \exists (X', Y') \in N_i.\ (X'^{\mho} \times \overline{Y'}^{\mathscr{D}_\Omega}) \setminus \left(\bigcup_{(X,Y)\in P_i} X^{\mho} \times \overline{Y}^{\mathscr{D}_\Omega} \right) = \varnothing \tag{$\to_7$}$$

in which we recognize a problem that we solved for products in Section 3.1.5, equation ($\times_4$).

Indeed, following the decomposition of Lemma 3.1.14, we can decompose the problem ($\to_7$) into:

$$\forall i \in I.\ \exists (X', Y') \in N_i.\ \forall P' \subseteq P_i.\ \left(X'^{\mho} \setminus \bigcup_{(X,Y)\in P'} X^{\mho} \right) \times \left(\overline{Y'}^{\mathscr{D}_\Omega} \setminus \bigcup_{(X,Y)\in P_i \setminus P'} \overline{Y}^{\mathscr{D}_\Omega} \right) = \varnothing \tag{$\to_8$}$$

The final step before concluding, is to notice that, for the special case $P' = P_i$, equation ($\to_8$) produces the constraint:

$$(X'^{\mho} \setminus \bigcup_{(X,Y)\in P_i} X^{\mho}) \times (\overline{Y'}^{\mathscr{D}_\Omega} \setminus \bigcup_{(X,Y)\in \varnothing} \overline{Y}^{\mathscr{D}_\Omega}) = \varnothing$$

which, since by definition the union on an empty set is $\varnothing$, means that the right component $(\overline{Y'}^{\mathscr{D}_\Omega}) = \mathscr{D}_\Omega \setminus Y'$ is never empty (Y' does not contain Ω, as $[\![t]\!] \subseteq \mathscr{D}$). Thus, the left component is necessarily empty, which corresponds to the constraint $X' \subseteq \bigcup_{(X,Y)\in P_i} X$ (noting that $\mho$ plays no essential role in this final step, as said in Remark 3.1.16). This constraint states that the domain of arrow $X' \to Y'$ is included in the union of the domains of the intersection of the arrows in P_i. We have thus obtained our result to compute the emptiness of arrow types which, similar to products, decides the problem if used with memoization (under the regularity axiom):

Theorem 3.1.19 (Emptiness of arrows)**.**

$$\bigvee_{i\in I} \Big(\bigwedge_{(t,s)\in P_i} (t \to s) \ \wedge \bigwedge_{(t',s')\in N_i} \neg(t' \to s') \Big) \leq \mathbb{0}$$
$$\Leftrightarrow \forall i \in I.\ \exists (t', s') \in N_i.\ (t' \leq \bigvee_{(t,s)\in P_i} t) \wedge \forall P' \subsetneq P_i.\ \big(t' \leq \bigvee_{(t,s)\in P'} t \big) \vee \big(\bigvee_{(t,s)\in P_i \setminus P'} s \leq s' \big)$$

Theorem 3.1.20 (Decidability of subtyping)**.** *The relation $\leq$ on $\mathscr{T}_{static}$ is decidable.*

Proof sketch. By Prop. 3.1.8, subtyping reduces to emptiness. We have shown emptiness is decidable in §3.1.5 by decomposing types into disjunctive normal forms and deciding emptiness for each atomic form, in Theorems 3.1.15 and 3.1.19. □

3.2 Gradual Types

3.2.1 Motivation

As established in Chapter 1, we need both set-theoretic types and gradual typing to type Elixir effectively. Set-theoretic types provide the expressive foundation needed to capture Elixir's rich type system (unions, intersections, precise pattern matching), while gradual typing allows seamless integration with Elixir's dynamic nature (untyped code, runtime introspection).

Definition 3.2.1 (Gradual Types)**.** *Bringing back the same sets of types as in Definition 3.1.1, gradual types are defined by allowing the presence of the dynamic type ? in types.*

$$\textit{Gradual Types} \quad \tau \ ::= \ ? \mid c \mid b \mid \tau \times \tau \mid \tau \to \tau \mid \tau \vee \tau \mid \neg\tau \mid \mathbb{0}$$

Lanvin (2021) establishes the foundations for integrating the dynamic type '?′ into set-theoretic types. We now present their construction, and its main results.

3.2.2 Semantics of gradual types

In gradual set-theoretic types, the dynamic type '?′ acts like a type variable that can be instantiated to any type at runtime. A value *always* belongs to a gradual type if it belongs to every possible instantiation of '?′ in that type. Conversely, a value *may* belong to a gradual type if it belongs to at least one instantiation. Lanvin (2021) addresses this through a novel interpretation that uses *tags* to mark whether an element of the interpretation domain $\mathscr{D}$ plays a static or dynamic role. Concretely, their gradual domain is cut into two: elements with a dynamic role $d^?$ and elements with a static role $d^!$. For example, type $? \wedge$ `int` contains all integer elements $3^?$ since it represents the type of values that *may* be integers. Meanwhile, type `int` contains both static (which can surely appear) integers $3^!$ and dynamic (which may possibly appear) integers $3^?$. Using this idea, Lanvin (2021) defines a gradual domain $\mathscr{D}_g$ and a set-theoretic interpretation for gradual types $[\![\cdot]\!]_g : \mathscr{T}_{\text{gradual}} \to \mathscr{D}_g$ (found in Appendix B.1).

This interpretation allows them to explain some interesting properties of gradual types: for instance, that *The interpretations of ? and* $\neg$*? are equal* (thus, $? \simeq \neg ?$), and consist in all values tagged with ?. While this may seem surprising, this behaviour is expected: if the dynamic type ? is seen as *the absence* of type information, then $\neg ?$ provides no information either. Another way to see this is by thinking of ? as a type that stands for any type: if ? stands for any type τ, then it also stands for $\neg\tau$, and thus $\neg ?$ stands for $\neg\neg\tau$, which is equivalent to τ.

Subtyping and precision It also allows Lanvin (2021) to define two relations on gradual types: subtyping and precision.

Definition 3.2.2 (Subtyping on $\mathscr{D}_g$, Lanvin 2021)**.** *We define the* ***subtyping*** *relation* $\leq$ *and the* ***subtyping equivalence*** *relation* $\simeq$ *on* $\mathscr{D}_g$ *as*

$$\tau_1 \leq \tau_2 \overset{def}{\Leftrightarrow} [\![\tau_1]\!]_g \subseteq [\![\tau_2]\!]_g \qquad \textit{and} \qquad \tau_1 \simeq \tau_2 \overset{def}{\Leftrightarrow} (\tau_1 \leq \tau_2) \text{ and } (\tau_2 \leq \tau_1)$$

Subtyping is not enough to type-check gradual programs: it does not allow statically unsound operations to take place, which is crucial for gradual typing. So *precision* is defined: τ_2 *is more precise than* τ_1, noted $\tau_1 \preccurlyeq \tau_2$, if τ_1 is "more dynamic" than τ_2. We can also say that τ_1 *materializes into* τ_2 whenever $\tau_1 \preccurlyeq \tau_2$, and refer to τ_2 as a *materialization* of τ_1.

Definition 3.2.3 (Precision, Lanvin 2021)**.** *The* precision *relation $\preccurlyeq$ in $\mathscr{D}_g$ is defined as*

$$\tau_1 \preccurlyeq \tau_2 \overset{def}{\Leftrightarrow} \forall d \in \mathscr{D}_g, \begin{cases} d^? \in [\![\tau_2]\!]_g & \Longrightarrow d^? \in [\![\tau_1]\!]_g \\ d^! \in [\![\tau_1]\!]_g & \Longrightarrow d^! \in [\![\tau_2]\!]_g \end{cases}$$

These two relations are the *core* way to relate gradual types with each other. For instance, to distinguish ? from $\mathbb{1}$, we can say that $\mathbb{1}$ contains all values, both static and dynamic. In the subtyping lattice, it is the top element: for all types τ, we have $\tau \leq \mathbb{1}$. An expression of type $\mathbb{1}$ is fully typed but too general–it can be any value, and must be refined before use in specific contexts. By constrast, ? represents all *unknown* values. Unlike $\mathbb{1}$, it is *not* the top of the subtyping lattice: in general, $\tau \not\leq ?$ (for $\tau \neq \mathbb{0}$) and $? \not\leq \tau$ (for $\tau \neq \mathbb{1}$). Instead, ? relates to other types through the *precision* relation: $\tau \preccurlyeq ?$ holds for all static types τ, meaning ? is maximally imprecise.

3.2.3 Representation theorem

The direct interpretation of gradual types provides a satisfying way to define subtyping and precision relations on gradual types. However, Lanvin (2021) presents another point of view, which is that gradual types can be represented simply by using a pair of static types. For the interpretation we just described, this corresponds to splitting a gradual type τ between two types: one whose interpretation is $\{d \in [\![\tau]\!] \mid \mathtt{tag}(d) = !\}$ (the static values in τ) and one whose interpretation is $\{d \in [\![\tau]\!] \mid \mathtt{tag}(d) = ?\}$ (the dynamic values in τ). Those are the extremal materializations of gradual types, defined as:

Definition 3.2.4 (Gradual extrema, Lanvin 2021)**.** *For every gradual type $\tau \in \mathscr{T}_{gradual}$, we define the minimal (resp. maximal) materialization of τ, noted $\tau^{\Downarrow}$ (resp. $\tau^{\Uparrow}$), as the static type obtained by replacing every covariant (resp. contravariant) occurrence of ? in τ by $\mathbb{0}$ and every contravariant (resp. covariant) occurrence of ? in τ by $\mathbb{1}$.*

These extrema are related to τ by subtyping.

Property 3.2.5 (Extrema bounds)**.** *For every gradual type τ,* $\quad \tau^{\Downarrow} \leq \tau \leq \tau^{\Uparrow}$

A remarkable and elegant theorem from Lanvin (2021) is then that gradual types can be defined directly in terms of these (static) extrema, as a pair:

Theorem 3.2.6 (Representation of gradual types, Lanvin 2021)**.** *For every gradual type τ, we have*

$$\tau \simeq \tau^{\Downarrow} \vee (? \wedge \tau^{\Uparrow})$$

This result shows that these two types are *bounds* of the gradual type τ. For example, the type `integer() or dynamic(number())` [2], which may appear esoteric to a programmer, denotes a

[2] Formally, `int` $\vee$ `(?` $\wedge$ `number)`: in Elixir, we can write the intersection with dynamic both as if it was a constructor, and also a `dynamic() and number()`

program that outputs values in an unknown set comprised between all integers and all numerical values (integers and floats). A type-checker (and a programmer) can reason on this uncertainty: for the type-checker, it may:

- not allow a list operation to be applied to an expression of this type;
- allow an integer operation to be applied to it, with a *warning*;
- allow a number operation to be applied to it;

Meanwhile, a programmer reading this type will know that its dynamic part from a program fragment that contains uncertainty: perhaps nonannotated modules, or untrusted code (note that we do not promise the reverse: that is, that if a program is typed with a strictly static type, then its execution does not involve any dynamic programs–we will discuss how to possibly recover this property in Chapter 5, Section 5.4.1).

This theorem means that we can always convert a gradual type to this pair, by computing the extrema of the type. For example, $? \times ?$ (the type of pairs whose two elements can be anything at run-time) is equivalent to $? \wedge (\mathbb{1} \times \mathbb{1})$ (the type of values that can be of any subtype of pairs). Likewise, $? \to ?$ (the type of functions whose domain and codomain can be anything) is equivalent to $\mathbb{1} \to \mathbb{0} \vee (? \wedge \mathbb{0} \to \mathbb{1})$, that is, any function type, since $\mathbb{0} \to \mathbb{1}$ and $\mathbb{1} \to \mathbb{0}$ are, respectively, the largest and smallest function types.

3.2.4 Lifting gradual relations from static types

Furthermore, Lanvin (2021) shows that we can also define subtyping and precision using this representation, and that the definition obtained is equivalent to that obtained from the direct interpretation (Defs. 3.2.2, 3.2.3).

Subtyping. Gradual subtyping requires that one type be contained in another *for all possible materializations.* This is equivalent to requiring that both minimal and maximal materializations be ordered (by the static subtyping relation of Definition 3.1.7):

Definition 3.2.7 (Lifted gradual subtyping)**.** *We define the* semantic gradual subtyping *relation* $\leq$ *between gradual types as follows:*

$$\tau_1 \leq \tau_2 \overset{def}{\Leftrightarrow} \begin{cases} \tau_1{}^{\Downarrow} \leq \tau_2{}^{\Downarrow} \\ \tau_1{}^{\Uparrow} \leq \tau_2{}^{\Uparrow} \end{cases}$$

and the semantic gradual equivalence *relation* $\simeq$ *as* $\tau_1 \simeq \tau_2 \Leftrightarrow (\tau_1 \leq \tau_2)$ *and* $(\tau_2 \leq \tau_1)$.

Note

This definition subsumes the definition of subtyping on static types: consider that, for a static type t, $t^{\Downarrow} = t$ and $t^{\Uparrow} = t$. Thus, from now on and in the rest of the thesis (unless in the rare occasions where we are comparing integers) we will use symbol $\leq$ to denote the unique gradual subtyping relation.

Precision. Precision captures when one type is "more dynamic" than another. Intuitively, $\tau_1 \preccurlyeq \tau_2$ means that τ_2 is obtained by refining some occurrences of ? in τ_1. In terms of extrema, this means that the minimal materialization becomes more restrictive, while the maximal materialization becomes more permissive:

Definition 3.2.8 (Gradual Precision)**.** *For all* $\tau_1, \tau_2 \in \mathscr{T}_{gradual}$

$$\tau_1 \preccurlyeq \tau_2 \iff (\tau_1{}^{\Downarrow} \leq \tau_2{}^{\Downarrow}) \text{ and } (\tau_2{}^{\Uparrow} \leq \tau_1{}^{\Uparrow})$$

Consistent subtyping. Finally, we define a new relation, *consistent subtyping*, which asks whether there exists *some* common materialization where subtyping is maintained. This is precisely what is needed for gradual type checking: we allow a function call if there is *some* way to refine the types that makes it safe:

Definition 3.2.9 (Consistent subtyping)**.** *For all* $\tau_1, \tau_2 \in \mathscr{T}_{gradual}$,

$$\tau_1 \lesssim \tau_2 \iff \tau_1{}^{\Downarrow} \leq \tau_2{}^{\Uparrow}$$

For example, $(? \vee \texttt{bool}) \to \texttt{int}$ is a consistent subtype of $\texttt{int} \to ?$ since by materializing both ?'s to `int` we obtain for the former type a type that is a subtype of the type obtained for the latter.

One can take the three equivalences above as the definition of the three relations, and refer to Lanvin (2021) for the proof of their properties (e.g., that $\tau_1 \lesssim \tau_2$ if and only if there exist τ_1' and τ_2' such that $\tau_1 \preccurlyeq \tau_1'$, $\tau_2 \preccurlyeq \tau_2'$ and $\tau_1' \leq \tau_2'$).

Remark (*Implementation message*)

From an implementation point of view, these results convey a strong message: if you have a static type system implemented, it is easy to lift it to a representation of gradual types and to define the gradual type relations.

3.3 Principles of Implementation

There are two distinct aspects to the implementation of a set-theoretic type system:

- The data structure representing types and supporting their set operations. We call it the *type engine* that powers the type system.
- The implementation of the typing rules, via the use of well-typed operators (projection, domain, application, etc.).

The type engine implemented in Elixir (located in the `descr.ex` file of the Elixir types module contributors (2025)) draws inspiration from many sources; first and foremost from the ℂDuce thesis Frisch (2004) and subsequent work Castagna (2016) and Kent (2019). Part II (Chapter 9) provides an in-depth study and implementation guide for such a type engine, covering in detail data structures, algorithms, and optimization techniques.

As for the typing rules, they are implemented using well-typed operators (projection, domain, application, etc.) which appear originally in Frisch (2004). We will detail how to derive these operators, and prove their soundness. Our proofs are extracted from the formalism of Frisch (2004). This exposition serves as a template for us later, when we introduce new operators (for multi-arity functions, for tuples), and when we prove their soundness.

3.3.1 Type operators: extracting information from types

Type operators bridge the gap between the declarative typing rules and their algorithmic implementation. Consider the simple declarative typing rule for projection, and its algorithmic counterpart, which uses the operator $\pi_1(t)$ to compute the first component of a product type t:

$$\frac{\Gamma \vdash e : t_1 \times t_2}{\Gamma \vdash \text{fst}\, e : t_1} \quad \overset{Algorithmic}{\Rightarrow} \quad \frac{\Gamma \vdash e : t}{\Gamma \vdash \text{fst}\, e \,⨾\, \pi_1(t)}$$

The declarative rule assumes that, for expression e, we can find a product supertype $t_1 \times t_2$ (via subtyping) and type its projection with the left component of the product. In practice, though, finding the (smallest, if we want precision) supertype of the product is done by an operator that acts on an arbitrary type t, and, just like emptiness checking, it is defined on the DNF representation of types.

3.3.1 a) Pair projection operator

Given a static type $t \in \mathscr{T}_{\text{static}}$ that is a product ($t \leq \texttt{pair}$), the DNF form $\bigvee_{i\in I} \bigwedge_{(t_1,t_2)\in P_i} (t_1 \times t_2) \wedge \bigwedge_{(t'_1,t'_2)\in N_i} \neg(t'_1 \times t'_2)$ computed by $\mathscr{U}^+(t)$ is not its only possible form. In the case of products, we can also obtain another standard form by doing more computation; here, by *intersecting* the positive products, and *eliminating* the negative ones.

Theorem 3.3.1 (Union form for products)**.** *There exists $\pi : \mathscr{T}_{static} \to \mathscr{P}_f\left(\mathscr{T}^2_{static}\right)$ such that, for any $t \in \mathscr{T}_{static}$ such that $t \leq \texttt{pair}$,*

$$t \simeq \bigvee_{(s_1,s_2)\in\pi(t)} (s_1 \times s_2)$$

Proof. With the DNF form of t given by $\mathscr{U}^+(t) = \bigvee_{i\in I}\Big(\bigwedge_{(t_1,t_2)\in P_i}(t_1 \times t_2) \wedge \bigwedge_{(t'_1,t'_2)\in N_i}\neg(t'_1 \times t'_2)\Big)$, we can start by effectively computing, for each $i \in I$, the intersection of the positive products in P_i, that is $(\bigwedge_{(t_1,t_2)\in P_i} t_1) \times (\bigwedge_{(t_1,t_2)\in P_i} t_2)$. Then, consider the elimination

$$(t_1 \times t_2) \smallsetminus (t'_1 \times t'_2) \simeq \left[(t_1 \wedge t'_1) \times (t_2 \smallsetminus t'_2)\right] \vee \left[(t_1 \smallsetminus t'_1) \times t_2\right]$$

which *produces disjoint products*[a], that we can prune for emptiness by checking if either element is empty. We successively eliminate the negative products in $N_i = \{(s_1^{(1)} \times s_2^{(1)}), \ldots, (s_1^{(n)} \times s_2^{(n)})\}$ by unfolding the difference via De Morgan's laws

$$t_1 \times t_2 \wedge \bigwedge_{(s_1,s_2)\in N_i} \neg(s_1 \times s_2) \simeq \Big(\big((t_1 \times t_2) \smallsetminus (s_1^{(1)} \times s_2^{(1)})\big) \smallsetminus (s_1^{(2)} \times s_2^{(2)}) \cdots\Big) \smallsetminus (s_1^{(n)} \times s_2^{(n)})$$

□

[a]Compared to the formula $(t_1 \times t_2) \setminus (t'_1 \times t'_2) \simeq ((t_1 \setminus t'_1) \times t_2) \vee (t_1 \times (t_2 \setminus t'_2))$ which is also correct, but where both types contain $(t_1 \setminus t'_1) \times (t_2 \setminus t'_2)$.

The elements in $\pi(t)$ are disjoint nonempty products, and can directly be used to compute what the type of the first or second elements of a product is.

Definition 3.3.2 (Projection operator)**.** *Let t be a type such that $t \leq$ `pair`. Let $i \in \{1,2\}$. The projection of t on the i-th component is defined as:*

$$\pi_i(t) \overset{def}{=} \bigvee_{(s_1,s_2)\in\pi(t)} s_i$$

Remark (*Tuple projection in Elixir*)

In Elixir, tuple projection is performed using the `elem/2` function, where `elem(tuple, index)` extracts the element at the given index. This corresponds directly to our projection operator $\pi_i(t)$. Note that, in Elixir, the argument `index` could be any expression, which complicates things.

Using this operator to type a projection is sound because it computes a minimal component of a product type, such that:

Theorem 3.3.3 (Soundness of pair projection)**.** *For all $t \in \mathscr{T}_{static}$ such that $t \leq$ `pair`, for all $t_1, t_2 \in \mathscr{T}_{static}$,*

$$t \leq t_1 \times t_2 \implies \pi_1(t) \leq t_1 \quad \textit{and} \quad \pi_2(t) \leq t_2$$

and in particular, $t \leq \pi_1(t) \times \pi_2(t)$ (for use by the declarative typing rule for projection).

Proof. Let $\pi(t) = \{(s_1^{(k)}, s_2^{(k)})\}_{k\in K}$ so that, by the Union Form Theorem, $t \simeq \bigvee_{k\in K}(s_1^{(k)} \times s_2^{(k)})$ where each $s_1^{(k)} \times s_2^{(k)}$ is non-empty (by construction we prune empty products).
Remember that if $r_1 \times r_2 \leq t_1 \times t_2$ and $r_1 \times r_2 \not\simeq \mathbb{0}$, then $r_1 \leq t_1$ and $r_2 \leq t_2$.
First part. Assume $t \leq t_1 \times t_2$. Then for every $k \in K$ we have $s_1^{(k)} \times s_2^{(k)} \leq t_1 \times t_2$, hence $s_1^{(k)} \leq t_1$ and $s_2^{(k)} \leq t_2$. Taking the union over K yields

$$\bigvee_{k\in K} s_1^{(k)} \leq t_1 \quad \text{and} \quad \bigvee_{k\in K} s_2^{(k)} \leq t_2,$$

i.e., $\pi_1(t) \leq t_1$ and $\pi_2(t) \leq t_2$.
"In particular" inequality. For each k, by monotonicity of $\times$ we have $s_1^{(k)} \times s_2^{(k)} \leq (\bigvee_j s_1^{(j)}) \times (\bigvee_j s_2^{(j)}) = \pi_1(t) \times \pi_2(t)$. Joining over k and using $t \simeq \bigvee_k (s_1^{(k)} \times s_2^{(k)})$ yields $t \leq \pi_1(t) \times \pi_2(t)$. This proves both claims. □

3.3.1 b) Function operators: domain and application

Typing function application in a declarative system with subtyping and in an algorithmic setting follows a similar pattern:

$$\frac{\Gamma \vdash e : t_1 \to t_2 \quad \Gamma \vdash e' : t_1}{\Gamma \vdash e(e') : t_2} \quad \overset{Algorithmic}{\Rightarrow} \quad \frac{\Gamma \vdash e : t \quad \Gamma \vdash e' : t'}{\Gamma \vdash e(e') : t \circ t'} \quad \begin{array}{l} t \leq \mathbb{0} \to \mathbb{1} \\ t' \leq \texttt{dom}(t) \end{array}$$

The declarative rule assumes that e can be found to have a given single arrow type, and for e' to have a type that is a subtype of the domain of the arrow. In the algorithmic rule, we assume two operators, $\texttt{dom}(t)$ and $\circ$, that compute the domain and application result of applying a function type t to an argument type t'.

Domain operator. *Empty disjuncts.* To compute function operators, we must prune empty clauses of the function DNF.

$$\bigwedge_{i \in P} (t_i \to s_i) \wedge \bigwedge_{j \in N} \neg(t_j \to s_j) \not\simeq \mathbb{0} \quad \Leftrightarrow \quad \exists j_0 \in N, \ \bigwedge_{i \in P} (t_i \to s_i) \leq (t_{j_0} \to s_{j_0})$$

This is important because the domain and result operators are defined directly in terms of the positive function literals, ignoring negative ones, as long as they don't make the disjunct empty.

The domain operator then extracts the set of argument types that a function can accept.

Definition 3.3.4 (Domain operator)**.** *For a function type $t \leq \mathbb{0} \to \mathbb{1}$, we define its domain* $\texttt{dom}(t)$ *by collecting only the* positive *arrow antecedents from each* nonempty *clause (indexed by I^+) of the DNF form of t:*

$$\texttt{dom}(t) \overset{def}{=} \bigwedge_{i \in I^+} \bigvee_{s_i \to t_i \in P_i} s_i \quad \textit{where} \quad t \simeq \bigvee_{i \in I^+} \bigwedge_{(s_i, t_i) \in P_i} (s_i \to t_i) \wedge \bigwedge_{(s'_i, t'_i) \in N_i} \neg(s'_i \to t'_i)$$

For each clause, it takes the union of all domain types from positive arrow literals, then intersects these unions across all clauses. This means that, for example, the domain of function type $(\texttt{atom} \to \texttt{atom}) \wedge (\texttt{bool} \to \texttt{bool})$ is $\texttt{atom} \vee \texttt{bool}$ (it can be applied to atoms or booleans), while the domain of function type $(\texttt{atom} \to \texttt{atom}) \vee (\texttt{bool} \to \texttt{bool})$ is $\texttt{atom} \wedge \texttt{bool} \simeq \mathbb{0}$: since the function could be defined either on atoms or on booleans (or both, but it is not statically known), there is no known value that it surely accepts, so the domain is empty. Unless they can cancel out a whole function type (e.g., $(\texttt{atom} \to \texttt{atom}) \wedge \neg(\texttt{atom} \to \mathbb{1}) \simeq \mathbb{0}$), negated arrows constrain specific input/output pairs but don't affect the overall domain. For instance, the type $(\texttt{atom} \to \texttt{atom}) \wedge \neg(\texttt{atom} \to \texttt{:ok})$ specifies a function from atoms to atoms that cannot return `:ok` on all atoms, but its domain is still `atom`.

Note

The domain operator exhibits interesting variance properties: the domain of an intersection of arrows is the union of their domains, while the domain of a union is the intersection of the domains. For instance, $(\texttt{int} \to \texttt{int}) \wedge (\texttt{bool} \to \texttt{bool})$ admits both integers and booleans, whereas $(\texttt{float} \to$

`float`) $\vee$ (`number` $\to$ `number`) safely accepts only `float` $\wedge$ `number` = `float`.
This illustrates why removing empty clauses in the domain operator is important: if we consider the domain of (`int` $\to$ `int`) $\wedge\neg$(`int` $\to$ `int`) $\simeq \mathbb{0}$, its domain is obviously not `int`: it is Any, which is neutral for intersection, validating the behaviour that the domain of a union of functions is the intersection of their domains.

Application result operator. The application result operator computes the type that results from applying a function type to an argument type.

Definition 3.3.5 (Result operator)**.** *Given a function type $t \leq \mathbb{0} \to \mathbb{1}$ and an argument type $s \leq \texttt{dom}(t)$, the result of applying t to s is defined over the nonempty positive clauses of the DNF form of t, indexed by I^+:*

$$t \circ s \overset{def}{=} \bigvee_{i \in I^+} \bigvee_{\substack{Q \subsetneq P_i \\ s \not\leq \bigvee_{s_j \to t_j \in Q} s_j}} \bigwedge_{s_j \to t_j \in P_i \setminus Q} t_j$$

where $t \simeq \bigvee_{i\in I^+}\left(\bigwedge_{(s_i,t_i)\in P_i}(s_i \to t_i) \wedge \bigwedge_{(s'_i,t'_i)\in N_i}\neg(s'_i \to t'_i)\right)$.

The result operator computes all possible outcomes of applying the function type t to the argument type s. The key insight is that we must consider all possible intersections between the argument type and subsets of the function domains.

Quote (*Definition 2.20 (Lanvin, 2021)*)

To understand this calculation, consider a simple intersection of arrows $t \equiv \bigwedge_{p\in P} s_p \to t_p$. If the argument type s intersects the domain of several arrows (i.e., $s \wedge s_{p_1} \wedge \ldots \wedge s_{p_n} \neq \emptyset$ for $p_1, \ldots, p_n \in P$), then the application may return a result in the intersection of all these arrows' codomains, namely $t_{p_1} \wedge \ldots \wedge t_{p_n}$, provided the argument resolves to a value that is indeed in the intersection of the aforementioned domains. However, the argument may also resolve to a value that is outside this intersection. Thus, we must consider all possible intersections between s and subsets of $\{s_p \mid p \in P\}$, and take the union of all possible results. The operator achieves this by excluding subsets Q that entirely contain s, thereby obtaining all subsets $P_i \setminus Q$ of P_i that have non-empty intersection with s.

Example: Function application. Consider applying the function type (`number` $\to$ `number`) $\vee$ (`int` $\to$ `int`) to argument type s = `int`. The uniform disjunctive normal form has two disjuncts: P_1 = {`number` $\to$ `number`} and P_2 = {`int` $\to$ `int`} of size 1 each. Thus, for disjunct 1, only the empty subset $Q = \emptyset$ satisfies the condition, yielding result `number`. For disjunct 2, we again get $Q = \emptyset$, yielding result `int`. The final result is `number` $\vee$ `int` $\simeq$ `number`.

Remark (*Typing function application.*)

These operators implement the static application principle: given $s \leq \texttt{dom}(t)$, we compute $t \circ s$ to obtain the application's type. This principle forms the foundation for implementing function application in type systems based on semantic subtyping.

Soundness of domain and application operators. In order to use these operators, we need the soundness properties:

Theorem 3.3.6 (Soundness of domain)**.** *For all $t \in \mathscr{T}_{static}$ with $t \leq \mathbb{0} \to \mathbb{1}$ and all $s_{\text{arg}}, t_{\text{res}} \in \mathscr{T}_{static}$,*

$$t_{\text{fun}} \leq s_{\text{arg}} \to t_{\text{res}} \quad \Longrightarrow \quad s_{\text{arg}} \leq \texttt{dom}(t_{\text{fun}})$$

Sketch. For $t_{\text{fun}} \simeq \mathbb{0}$, we have $\texttt{dom}(\mathbb{0}) = \mathbb{1}$, which concludes the proof. Otherwise, write the (non-empty) DNF of t_{fun} as $(\bigvee_{i \in I^+} C_i)$ with $C_i = \bigwedge_{(u,v)\in P_i}(u \to v) \wedge \bigwedge_{(u',v')\in N_i} \neg(u' \to v')$. From $t \leq s_{\text{arg}} \to t_{\text{res}}$ we get $C_i \leq s_{\text{arg}} \to t_{\text{res}}$ for each i. By the arrow emptiness characterization Thm. 3.1.19 (applied to the positive part $\bigwedge_{(u,v)\in P_i}(u \to v)$–we can ignore the negatives because *by assumption* the clause is nonempty), the LHS condition forces $s_{\text{arg}} \leq \bigvee_{(u,v)\in P_i} u$. Intersecting these inclusions over all i yields $s_{\text{arg}} \leq \bigwedge_{i \in I^+} \bigvee_{(u,v)\in P_i} u = \texttt{dom}(t)$. □

Theorem 3.3.7 (Soundness and optimality of application)**.** *Let $t \in \mathscr{T}_{static}$ with $t \leq \mathbb{0} \to \mathbb{1}$ and $s \in \mathscr{T}_{static}$ with $s \leq \texttt{dom}(t)$. Then:*

1. ***Soundness.*** $t \leq s \to (t \circ s)$.
2. ***Optimality.*** *For every $t_{\text{res}} \in \mathscr{T}_{static}$, if $t \leq s \to t_{\text{res}}$ then $t \circ s \leq t_{\text{res}}$.*

Sketch. If $t_{\text{fun}} \simeq \mathbb{0}$, we have $t_{\text{fun}} \circ s \simeq \mathbb{0}$ (as a union over an empty set), which concludes. Otherwise, let $t \simeq \bigvee_{i \in I^+} C_i$ be the (nonempty) DNF as above and let $U_i \overset{\text{def}}{=} \bigvee_{(u,v)\in P_i} u$. By $s \leq \texttt{dom}(t)$ we have $s \leq U_i$ for all i.
(1) Soundness. Fix a clause i and abbreviate P_i as a finite set of arrows $\{u_j \to v_j\}_{j \in P_i}$. Define

$$R_i(s) \overset{\text{def}}{=} \bigvee_{\substack{Q \subsetneq P_i \\ s \not\leq \bigvee_{j\in Q} u_j}} \; \bigwedge_{j \in P_i \setminus Q} v_j.$$

Applying the arrow containment characterization (from Thm. 3.1.19) to $\bigwedge_{j\in P_i}(u_j \to v_j)$ with argument s and result $R_i(s)$: i) we do have $s \leq \bigvee_{j \in P_i} u_j$ ii) for any $Q \subsetneq P_i$, it is true that *either* $s \leq \bigvee_{j\in Q} u_j$*, or else* $s \not\leq \bigvee_{j \in Q} u_j$ and by construction $\bigwedge_{j\in P_i\setminus Q} v_j \leq R_i(s)$. Hence $\bigwedge_{j \in P_i}(u_j \to v_j) \leq s \to R_i(s)$. Since adding $\bigwedge N_i$ only refines the left-hand side, $C_i \leq s \to R_i(s)$. Taking the union over i and using the definition $t \circ s \overset{\text{def}}{=} \bigvee_{i\in I^+} R_i(s)$ yields $t \leq s \to (t \circ s)$, concluding.
(2) Optimality. Assume $t \leq s \to t_{\text{res}}$, hence $C_i \leq s \to t_{\text{res}}$ for each i. Apply Th.(3.1.19) to $C_i \leq s \to t_{\text{res}}$ gives: for every $Q \subsetneq P_i$ with $s \not\leq \bigvee_{j\in Q} u_j$ we must have $\bigwedge_{j\in P_i\setminus Q} v_j \leq t_{\text{res}}$. Taking the union over all such Q gives $R_i(s) \leq t_{\text{res}}$, and finally $\bigvee_{i\in I^+} R_i(s) = t \circ s \leq t_{\text{res}}$. □

In particular, this proves that the application operator $\circ$ computes the best (smallest) possible type that can be deduced by the application typing rule.

3.3.2 Gradual Type Operators

Lanvin (2021) claim that any gradual type operator may be lifted from the static ones. In general, the principle is that if F is a **monotone** operator on static types, then the gradual version of F is given by

Proposition 3.3.8 (Gradual Extension)**.**

$$\tilde{F}(\tau) = (F(\tau^{\Downarrow}) \wedge F(\tau^{\Uparrow})) \vee (? \wedge (F(\tau^{\Downarrow}) \vee F(\tau^{\Uparrow})))$$

with a simplification for when F is increasing with respect to ≤ (this is the case for the projection operator):

$$\tilde{F}(\tau) = F(\tau^{\Downarrow}) \vee (? \wedge F(\tau^{\Uparrow}))$$

and if F is decreasing with respect to ≤ (this is the case for the domain operator):

$$\tilde{F}(\tau) = F(\tau^{\Uparrow}) \vee (? \wedge F(\tau^{\Downarrow}))$$

The idea behind this principle is to compute a range: a largest and a smallest type obtainable by this operator. The type $F(\tau^{\Downarrow}) \wedge F(\tau^{\Uparrow})$ is the smaller of $F(\tau^{\Downarrow})$ and $F(\tau^{\Uparrow})$, while the union is the larger of the two.

We will discuss this extension in more detail (and in the concrete case of defining the result operator for gradual types) in Chapter 13, Section 13.2.

Example (*Motivating example: lifting projection and domain*)

Consider the gradual pair type

$$\tau \;=\; ? \wedge (\texttt{int} \times \texttt{bool})$$

Since products are covariant, the extrema are

$$\tau^{\Downarrow} \simeq \mathbb{0} \wedge (\texttt{int} \times \texttt{bool}) \simeq \mathbb{0} \qquad \text{and} \qquad \tau^{\Uparrow} \simeq \mathbb{1} \wedge (\texttt{int} \times \texttt{bool}) \simeq \texttt{int} \times \texttt{bool}$$

Let F be the projection on the first component, $F(t) = \pi_1(t)$ (monotone increasing). By Proposition 3.3.8,

$$\widetilde{F}(\tau) \;=\; F(\tau^{\Downarrow}) \;\vee\; (? \wedge F(\tau^{\Uparrow})) \;\simeq\; \mathbb{0} \;\vee\; (? \wedge \texttt{int}) \;\simeq\; ? \wedge \texttt{int}.$$

Now consider the gradual function type

$$\tau' \;=\; (? \to \texttt{int}) \wedge (\texttt{bool} \to ?).$$

Arrows are contravariant in the domain and covariant in the codomain, hence

$$\tau'^{\Downarrow} \simeq (\mathbb{1} \to \texttt{int}) \wedge (\texttt{bool} \to \mathbb{0}), \qquad \tau'^{\Uparrow} \simeq (\mathbb{0} \to \texttt{int}) \wedge (\texttt{bool} \to \mathbb{1}).$$

Let G be the domain operator $G(t) = \texttt{dom}(t)$ (monotone decreasing). By Proposition 3.3.8,

$$\widetilde{G}(\tau') \;=\; G(\tau'^{\Uparrow}) \;\vee\; (? \wedge G(\tau'^{\Downarrow})) \;\simeq\; \texttt{bool} \;\vee\; (? \wedge \mathbb{1}) \;\simeq\; \texttt{bool} \vee ?.$$

Reading: the function accepts booleans with certainty, and possibly any other input.

Conclusion We have assembled the static and gradual foundations of semantic subtyping, connected them through DNFs and emptiness, and extracted the key operators that power algorithmic typing. These tools will serve as our compass in the remainder of the thesis.

As a bridge to what follows, the next chapter introduces Core Elixir: a small calculus and its static fragment, with operational semantics, declarative and algorithmic typing rules (with progress and preservation), and extends semantic subtyping to tuples and multi-arity functions.

CHAPTER 4

STATIC TYPING FOR CORE ELIXIR

This chapter introduces *Core Elixir*, a calculus that captures the essential features of the Elixir programming language relevant to our type system. While the goal is eventually to type Elixir with a gradual type system, we make the choice of first presenting its static fragment and the technical developments required to make it work. We extend semantic subtyping to handle multi-arity function types, first-class tuples (closed and open variants), and establish soundness for the static type system.

After this, Chapter 5 will raise these foundations to the gradual setting by introducing the dynamic type ?, and will feature a more detailed discussion of the safety features of the type system of Elixir.

Chapter Roadmap

- **Section 4.1 (Calculus)**: introduces the calculus and its semantics (Figs. 4.1–4.2).
- **Section 4.2 (Rules)**: presents the static typing rules (Fig. 4.4) and their design trade-offs.
- **Section 4.3 (Soundness)**: establishes progress and preservation, with a precise statement of what the ω rules permit and forbid.
- **Section 4.4 (Multi-arity Functions)**: extends the metatheory of semantic subtyping to multi-arity function types.
- **Section 4.5 (Tuples)**: develops the semantic interpretation and subtyping for tuples (closed and open).

$$\begin{array}{llll}
\textbf{Expressions} & e & ::= & c \mid x \mid \lambda^{\mathbb{I}}x.e \mid e(e) \mid \{\overline{e}\} \mid \pi_e\, e \mid \texttt{case}\ e\,(\rho_i \to e_i)_{i\in I} \mid e+e \\
\textbf{Test types} & \rho & ::= & b \mid \{\overline{\rho}\} \mid \{\overline{\rho},..\} \\
\textbf{Base types} & b & ::= & \texttt{int} \mid \texttt{bool} \mid \texttt{atom} \mid \texttt{fun} \mid \texttt{tuple} \\
\textbf{Types} & t & ::= & b \mid c \mid t \to t \mid \{\overline{t}\} \mid \{\overline{t},..\} \mid t \vee t \mid \neg\, t \\
\textbf{Interfaces} & \mathbb{I} & ::= & \{t_i \to t'_i\}_{i=1..n}
\end{array}$$

Figure 4.1: Expressions and Types Syntax

4.1 Core Elixir Calculus

4.1.1 Core Syntax

The syntax of Core Elixir is given in Figure 4.1. It is a typed λ-calculus with constants ranged over by c (these include integers, booleans, atom constants, but also tuples of constants, such as {1,{true, :a}}, etc.); variables ranged over by x; λ-abstractions $\lambda^{\mathbb{I}}x.e$ annotated by interfaces (ranged over by $\mathbb{I}$, and which are finite sets of arrows whose intersection declares the type of the λ-abstraction); application $e(e)$; tuples $\{\overline{e}\}$ (we use the overbar to denote sequences, that is, $\overline{e}$ stands for $e_1,...,e_n$); projections $\pi_e\, e$ where the projection index subscripting the π symbol is an expression that should evaluate to an integer; type-case expressions case $e\,(\rho_i \to e_i)_{i\in I}$ where ρ denotes easily checkable types such as base types or tuples of base types; and, for illustrating the typing of BEAM-checked operators, the arithmetic sum $+$.

4.1.2 Operational Semantics

The language has strict weak-reduction semantics defined by the reduction rules in Figure 4.2. The semantics is defined in terms of values v and evaluation contexts $\mathscr{E}$:

$$\begin{array}{lll}
\textbf{Values} & v & ::= c \mid \lambda^{\mathbb{I}}x.e \mid \{\overline{v}\} \\
\textbf{Contexts} & \mathscr{E} & ::= \Box \mid \mathscr{E}(e) \mid v(\mathscr{E}) \mid \{\overline{v},\mathscr{E},\overline{e}\} \mid \pi_{\mathscr{E}}\, e \mid \pi_v \mathscr{E} \\
 & & \mid \texttt{case}\ \mathscr{E}\,(\rho_i \to e_i)_{i\in I} \mid \mathscr{E}+e \mid v+\mathscr{E}
\end{array}$$

Reduction Rules The reduction rules are standard: [APP] is the call-by-value beta-reduction where $e[v/x]$ denotes the capture-free substitution of x with v in e, [PROJ] defines tuple projection, and [MATCH] implements a first-match reduction strategy for type-case expressions: the reductum e_j is the first branch of the type-case whose test ρ_j “matches" the value v. Given a value v and a test type ρ, we denote by $v \in \rho$ (formally defined in Figure 4.3) the fact that v belongs to the set represented by ρ (e.g., $0 \in$ int and $\{0,1\} \in$ tuple), and we write $v \not\in \rho$ if not (e.g., $0 \not\in$ bool). The goal of the context rule is to model the left-to-right evaluation order of (Core) Elixir.

Failure Reductions These correspond to explicit runtime errors raised by the Erlang VM, and they will be used to make the statement of type safety properties more precise, by explicitly identifying which failure states are prevented in a typed program. Failures are denoted as a

$$
\begin{array}{llcll}
[\textsc{App}] & (\lambda^{\mathbb{I}}x.e)(v) & \hookrightarrow & e[v/x] & \\
[\textsc{Proj}] & \pi_i\,\{\,v_0,..,v_n\,\} & \hookrightarrow & v_i & \text{if } i \in [0..n] \\
[\textsc{Match}] & \texttt{case}\ v\left(\rho_i \to e_i\right)_{i\in I} & \hookrightarrow & e_j & \text{if } v \in \rho_j \text{ and } v \notin \bigvee_{i<j}\rho_i \\
[\textsc{Plus}] & v + v' & \hookrightarrow & v'' & \quad\text{if } v, v' \text{ are integers} \\
 & & & & \text{and } v'' = v + v' \\
[\textsc{Context}] & \mathscr{E}[e] & \hookrightarrow & \mathscr{E}[e'] & \text{if } e \hookrightarrow e' \text{ without} \\
 & & & & \text{rule } [\textsc{Context}] \\
 & & & & \\
[\textsc{App}_\omega] & v(v') & \hookrightarrow & \omega_{\textsc{BadFunction}} & \text{if } v \neq \lambda^{\mathbb{I}}x.e \\
[\textsc{Proj}_{\omega,\textsc{range}}] & \pi_v\,\{\,v_0,..,v_n\,\} & \hookrightarrow & \omega_{\textsc{OutOfRange}} & \text{if } v \neq i \text{ for } i = 0..n \\
[\textsc{Proj}_{\omega,\textsc{notTuple}}] & \pi_{v'}\,v & \hookrightarrow & \omega_{\textsc{NotTuple}} & \text{if } v \neq \{\,\overline{v}\,\} \\
[\textsc{Match}_\omega] & \texttt{case}\ v\left(\rho_i \to e_i\right)_{i\in I} & \hookrightarrow & \omega_{\textsc{CaseEscape}} & \text{if } v \notin \bigvee_{i\in I}\rho_i \\
[\textsc{Plus}_\omega] & v + v' & \hookrightarrow & \omega_{\textsc{ArithError}} & \text{if } v \text{ or } v' \text{ not integers} \\
[\textsc{Context}_\omega] & \mathscr{E}[e] & \hookrightarrow & \omega_p & \text{if } e \hookrightarrow \omega_p \text{ without} \\
 & & & & \text{rule } [\textsc{Context}_\omega]
\end{array}
$$

Figure 4.2: Standard and Failure Reductions

labeled symbol ω_p, where the label p informs of the type of exception raised (e.g., $\omega_{\textsc{ArithError}}$ for trying to sum non-integer values). Note that the first error encountered will be the one raised, since rule $\textsc{Context}_\omega$ short-circuits the evaluation.

$\mathbb{B}(c)$ maps constants onto their base types (e.g. integers i onto `int`)

$$
\begin{array}{ll}
\forall c & c \in \mathbb{B}(c) \\
\forall x, e, \mathbb{I} & (\lambda^{\mathbb{I}}x.e) \in \texttt{fun} \\
\forall v_1, .., v_n & \{\,v_1,..,v_n\,\} \in \{\,\rho_1,..,\rho_n\,\} \iff \forall i = 1..n \quad v_i \in \rho_i \\
\forall v & v \in \rho \implies \forall \rho \leq \rho' \quad v \in \rho'
\end{array}
$$

Figure 4.3: Inductive Definition for $v \in \rho$

4.1.3 Types in Core Elixir

The types of Core Elixir are defined in Figure 4.1. Base types include integers, Booleans, the type of all atoms `atom`, the type of all functions `fun`, the type of all tuples `tuple`. We also have open tuple types: $\{\,\overline{t}, ..\,\}$ denotes any tuple starting with a sequence of elements of types $\overline{t}$. Types include set-theoretic connectives: the connectives union $\vee$ and negation $\neg$ are represented in the syntax, while intersection is defined as $t_1 \wedge t_2 = \neg(\neg t_1 \vee \neg t_2)$, and difference is defined as $t_1 \setminus t_2 = t_1 \wedge \neg t_2$. The top type $\mathbb{1}$, the type of all values, is defined as $\mathbb{1} = \texttt{int} \vee \texttt{atom} \vee \texttt{fun} \vee \texttt{tuple}$,[1] while the bottom type $\mathbb{0}$ is defined as $\mathbb{0} = \neg\mathbb{1}$. Note that, since constants are included in types, every value that does not contain λ-abstractions can be typed as its singleton type (containing

[1] In Elixir, `bool` is contained in `atom`, since the truth values are the atoms `:true` and `:false`.

only the value itself). Types are defined coinductively (for type recursion) and, as customary in semantic subtyping, they are contractive (no infinite unions or negations) and regular (a condition necessary for the decidability of the subtyping relation): see, e.g., Frisch et al. (2008) for details.

Remark (*Link with Elixir types*)

This formalization maps directly to Elixir's type syntax. Base types include `integer()`, `atom()`, `fun()`, and `tuple()`. The type of all values $\mathbb{1}$ corresponds to `term()`, while the bottom type $\mathbb{0}$ corresponds to `none()`. Union types are expressed as t_1 `or` t_2, negation as `not` t. Intersection types are expressed as t_1 `and` t_2, and difference types as t_1 `and not` t_2. Note that `boolean()` is simply an alias for the union `true or false`.

Note that type-case expressions do not check all types, but only "test types", ranged over by ρ. These are types that do not contain arrow types, or set-theoretic connectives. Arrow types are excluded because Elixir can only test whether a value is a function or not (and its arity, see Section 4.4.1), but it cannot test whether it has a given functional type, or connectives, as those will be later encoded by patterns in Chapter 6.

4.2 Static Typing Rules

Figure 4.4 presents the rules that define the static type system for Core Elixir. These rules are standard, with constants typed as themselves (rule (cst)), variable introduction (rule (var)), function application (rule (app)), function introduction (rule (λ)), pattern matching (rule (case)), tuple projections (rule (proj)), and arithmetic operations (rule (add)), and a subsumption rule (rule (sub)). We assume a decidable subtyping relation $\leq$ on types, as defined in Chapter 3. In Section 4.4, we extend this relation to multi-arity functions. We now describe the few specificities of the static type system.

Warnings for Potentially Unsafe Tuple Projections. Certain rules are marked with "ω" to indicate that using them triggers a compiler warning about a possible run-time type error. In particular, rules (proj$_\omega$) and (proj$_\omega^{\mathbb{1}}$) allow more liberal tuple projections, but whenever either rule is applied the type checker will issue a warning that the tuple access might result in an "index out of range" exception at run time.

Intersection Types for Functions. Rule (λ) for lambda abstractions supports intersection types in function annotations. Given a function annotated with an interface $\mathbb{I} = \{t_i \to s_i \mid i \in I\}$ the checker demands that the body is well-typed for every arrow in the interface. Concretely, $\Gamma \vdash \lambda^{\mathbb{I}}x.e : \bigwedge_{i\in I}(t_i \to s_i)$ holds only if, for every $(t_i \to s_i) \in \mathbb{I}$, we have $\Gamma, x : t_i \vdash e : s_i$

Union Types for Tuple Indices. Rule (proj) handles tuple projections where the index expression may be a union of integer constants. Thanks to union and singleton integer types, the

$$\text{(cst)}\ \frac{}{\Gamma \vdash_{\mathsf{s}} c : c} \qquad \text{(var)}\ \frac{\Gamma(x) = t}{\Gamma \vdash_{\mathsf{s}} x : t} \qquad \text{(tuple)}\ \frac{\overline{\Gamma \vdash_{\mathsf{s}} e : t}}{\Gamma \vdash_{\mathsf{s}} \{\,\overline{e}\,\} : \{\,\overline{t}\,\}}$$

$$(\lambda)\ \frac{\forall (t_i \to s_i) \in \mathbb{I}\ (\Gamma, x : t_i \vdash_{\mathsf{s}} e : s_i)}{\Gamma \vdash_{\mathsf{s}} \lambda^{\mathbb{I}}(x).e : \bigwedge_i (t_i \to s_i)} \qquad \text{(app)}\ \frac{\Gamma \vdash_{\mathsf{s}} e : t_1 \to t_2 \quad \Gamma \vdash_{\mathsf{s}} e' : t_1}{\Gamma \vdash_{\mathsf{s}} e(e') : t_2}$$

$$\text{(case)}\ \frac{\Gamma \vdash_{\mathsf{s}} e : t \quad \forall i \in I \big(t \wedge \rho_i \setminus (\bigvee_{j<i} \rho_j) \not\leq \mathbb{0} \Rightarrow \Gamma \vdash_{\mathsf{s}} e_i : t'\big)}{\Gamma \vdash_{\mathsf{s}} \mathtt{case}\ e \big(\rho_i \to e_i\big)_{i \in I} : t'}\ t \leq \bigvee_{i \in I} \rho_i$$

$$\text{(proj)}\ \frac{\Gamma \vdash_{\mathsf{s}} e' : \bigvee_{i \in K} i \quad \Gamma \vdash_{\mathsf{s}} e : \{\, t_0 , \ldots , t_n , .. \}}{\Gamma \vdash_{\mathsf{s}} \pi_{e'}\, e : \bigvee_{i \in K} t_i}\ K \subseteq [0, n]$$

$$(\text{proj}_{\omega})\ \frac{\Gamma \vdash_{\mathsf{s}} e' : \mathtt{int} \quad \Gamma \vdash_{\mathsf{s}} e : \{\, t_0 , \ldots , t_n \}}{\Gamma \vdash_{\mathsf{s}} \pi_{e'}\, e : \bigvee_{i \leq n} t_i} \qquad (\text{proj}_{\omega}^{\mathbb{1}})\ \frac{\Gamma \vdash_{\mathsf{s}} e' : \mathtt{int} \quad \Gamma \vdash_{\mathsf{s}} e : \mathtt{tuple}}{\Gamma \vdash_{\mathsf{s}} \pi_{e'}\, e : \mathbb{1}}$$

$$\text{(add)}\ \frac{\Gamma \vdash_{\mathsf{s}} e_1 : \mathtt{int} \quad \Gamma \vdash_{\mathsf{s}} e_2 : \mathtt{int}}{\Gamma \vdash_{\mathsf{s}} e_1 + e_2 : \mathtt{int}} \qquad \text{(sub)}\ \frac{\Gamma \vdash_{\mathsf{s}} e : t_1}{\Gamma \vdash_{\mathsf{s}} e : t_2}\ t_1 \leq t_2$$

Figure 4.4: Declarative static type system

index expression e' in $\pi_{e'}\, e$ can have a finite-union type of specific indices; the result type is then the union of the corresponding tuple field types. For example, if e' has type $0 \vee 2$ (meaning it evaluates to 0 or 2) and e is a 3-tuple of type $\{\, t_0, t_1, t_2 \,\}$, then $\pi_{e'}\, e : t_0 \ \vee \ t_2$

Attainable Case Branches & Exhaustiveness. Rule (case) types only the branches that are attainable. For a branch $\rho_i \to e_i$ being attainable means that the matched expression e must be able to produce a value of type ρ_i that is not captured by a previous branch. In semantic subtyping a type is a set of values, and the type of an expression overapproximates the set of values the expression may produce. Thus, a branch $\rho_i \to e_i$ is attainable if the set of values that may be produced by e (i.e., those in the type t of e), intersected with the values in the test type ρ_i (i.e., the set of values captured by the branch), minus all values captured by previous branches (i.e., all values in ρ_j for $j < i$) is non-empty. The side condition $t \leq \bigvee_{i \in I} \rho_i$ ensures exhaustiveness since it checks that all values that e may produce (i.e., those in t) are contained in the set of all types checked by the expression. For instance, we can encode $\mathtt{if}\, e\, \mathtt{then}\, e_1\, \mathtt{else}\, e_2$ as the type-case on singleton types $\mathtt{case}\, e\, (\mathtt{true} \to e_1, \mathtt{false} \to e_2)$ and exhaustiveness will check that e is of type `bool` (i.e., $\mathtt{true} \vee \mathtt{false}$).

Note

The reader may wonder why the presence of a (statically detected) non-attainable branch does not yield a type error. The reason is that the actual attainability of a branch cannot be decided locally. For instance, to deduce the intersection type $(\mathtt{int} \to \mathtt{int}) \wedge (\mathtt{bool} \to \mathtt{bool})$ for the function $\lambda^{\{\mathtt{int} \to \mathtt{int}, \mathtt{bool} \to \mathtt{bool}\}} x.\mathtt{case}\ x\ (\mathtt{int} \to x{+}1, \mathtt{bool} \to \neg x)$, the system types the case-expression twice:

once under the assumption x:`int`, making the `bool` branch unattainable, and once under the assumption x:`bool` making the `int` branch unattainable. Thus, each branch is attainable at some point, though not at the same time. The property of being statically attainable is, thus, a global property, not expressible in a compositional system. The type-checker will check that every branch of every case is typed at least once, and emit a "unused branch" warning when this condition is not met. In the next chapter, Section 5.2, we assume attainability ("every branch is typed at least once across interfaces") to prove the soundness of the gradual system.

The type system of Figure 4.4 is sufficient to type *non-gradual* Core Elixir programs formed by expressions that use only static types, that is, expressions where the type '?' never appears in the interface of a function.

4.3 Static Type Safety

In this section, we establish the type safety of the static type system.

To simplify the typing rule for pattern matching (and the associated proof of soundness), we assume that the type patterns of each case expression are pairwise disjoint, that is, $\forall(i,j) \in I^2, i \neq j \Rightarrow \rho_i \wedge \rho_j \leq \mathbb{0}$. This is a minor restriction that can be obtained simply by rewriting case expressions, subtracting to each case the union of the previous cases (which is easy to do since test types are a finite union of base types and top types). This definition is not restrictive either, as any non-disjoint case expression can be compiled into a disjoint one by subtracting the union of the previous cases from the current one for each branch.

Lemma 4.3.1 (Permutation)**.** *Let e be an expression, t, t_1, t_2 be types, Γ an environment, and $x, y \notin \mathtt{dom}(\Gamma)$ distinct variables (i.e., $x \neq y$). Then*

$$\Gamma, x\colon t_1, y\colon t_2 \vdash_{\mathsf{s}} e : t \implies \Gamma, y\colon t_2, x\colon t_1 \vdash_{\mathsf{s}} e : t.$$

Proof. Proceed by induction on the derivation of $\Gamma, x : t_1, y : t_2 \vdash_{\mathsf{s}} e : t$, analysing the last typing rule. Every non-base case follows directly from the induction hypothesis applied to its premises. □

Lemma 4.3.2 (Weakening)**.** *Let e be an expression, t, s types, Γ an environment, and $x \notin \mathtt{dom}(\Gamma)$ a variable. If $x \notin \mathtt{fv}(e)$ then*

$$(\Gamma, x\colon s \vdash_{\mathsf{s}} e : t) \implies (\Gamma \vdash_{\mathsf{s}} e : t)$$

Proof. We proceed by induction on the derivation of $\Gamma, x\colon s \vdash_{\mathsf{s}} e : t$ and analyse its last rule.

(cst) Trivial: the type of a constant does not depend on the environment.

(var) $e = y$ with $y \neq x$ (because $x \notin \mathtt{fv}(e)$). Hence $(y : t) \in \Gamma$ and the conclusion follows by rule (var).

(λ) Suppose $e = \lambda^{\mathbb{I}} y. e'$. By inversion $\Gamma, x : s, y : t_i \vdash_{\mathsf{s}} e' : s_i$ for each $(t_i \to s_i) \in \mathbb{I}$. Using

Lemma 4.3.1 we swap x and y, then apply the induction hypothesis to obtain $\Gamma, y : t_i \vdash_s e' : s_i$ for all i. Re-applying rule λ produces the desired judgement.

Structural rules (tuple, app, case, plus, $\leq$) These rules pass the environment unchanged to their premises, so the induction hypothesis applies directly.

□

Lemma 4.3.3 (Substitution)**.** *Let e, e_1 be expressions, t, t_1 types, and $x \notin \texttt{dom}(\Gamma)$.*

$$(\Gamma, x : t_1 \vdash_s e : t) \wedge (\Gamma \vdash_s e_1 : t_1) \Longrightarrow (\Gamma \vdash_s e[e_1/x] : t)$$

Proof. By induction on the size of the derivation tree and case analysis on the last non-(sub) typing rule used to derive $\Gamma, x : t_1 \vdash_s e : t$.

(cst) Immediate since e is a constant and does not depend on x.

(var) $e = y$. There are two cases:

- $y = x$. Then $e[e_1/x] = e_1$ so by assumption $\Gamma, x : t_1 \vdash_s x : t$ and $x \notin \texttt{dom}(\Gamma)$. Necessarily then, we have $t_1 \leq t$. Applying rule (sub) to $\Gamma \vdash_s e_1 : t_1$ concludes.
- $y \neq x$. Then $e[e_1/x] = y$, and the result follows since $\Gamma, x : t_1 \vdash_s y : t$.

(λ) $e = \lambda^{\mathbb{I}} y.e_1$. By inversion, $\Gamma, x : t_1, y : t_i \vdash_s e_i : s_i$ for all $(t_i \to s_i) \in \mathbb{I}$. Rearranging the variables by Permutation 4.3.1, and by induction hypothesis, $\Gamma, y : t_i \vdash_s e_i[e_1/x] : s_i$ for all $i \in I$. This concludes by re-applying the (λ) typing rule.

Structural rules (tuple, app, case, proj, proj$_\omega$, proj$_\omega^{\mathbb{1}}$, plus, $\leq$) maintain the same environment in the conclusion and premises, and involve sub-expressions in the premises. Hence, they are handled in the same way as the (λ) rule by directly applying the induction hypothesis to their premises. □

Lemma 4.3.4 (Progress)**.** *If $\varnothing \vdash_s e : t$ where no ω-marked rules were used, then either e is a value or there exists e' such that $e \hookrightarrow e'$.*

Proof. Our set of reduction rules (see Figure 4.2), including failure reductions, is complete. This means that every expression that is not a variable—thus, *a fortiori*, every closed expression—is either a value, or it can be reduced to another expression (which will be closed, too) or to a failure $\omega \in \{\omega_{\text{CaseEscape}}, \omega_{\text{OutOfRange}}, \omega_{\text{NotTuple}}, \omega_{\text{BadFunction}}, \omega_{\text{ArithError}}\}$.
We will prove that for a well-typed expression in $\vdash_{\text{static}}$, the failure cases are impossible. Let's assume there exists an expression e such that $e \hookrightarrow \omega_p$, where p is one of the failure cases. For each case, since e is well-typed, we consider the last structural typing rule that was used to type e before some eventual subsumption.

1. Case $p = \text{CaseEscape}$: In this case, $e = \texttt{case}\ v\,(\rho_i \to e_i)_{i \in I}$ where $v \notin \bigvee_{i \in I} \rho_i$. By inversion, (case) implies that $\varnothing \vdash_s v : t'$ where $t' \leq \bigvee_{i \in I} \rho_i$. This contradicts our assumption, as v

must belong to $\bigvee_{i\in I}\rho_i$.

2. Case $p = \textsc{NotTuple}$: Here, $e = \pi_{v'}\, v$ where v is not a tuple. By inversion on (proj), we have $v : \{\, t_0, .., t_n\,\textbf{,}\, ..\}$ for some n, which contradicts our assumption.
3. Case $p = \textsc{OutOfRange}$: In this scenario, $e = \pi_i\,\{\, v_0, .., v_n \,\}$ where $i \notin [0..n]$. This is explicitly forbidden by rule (proj), which ensures that the index is in the range of the tuple.
4. Case $p = \textsc{BadFunction}$: In this scenario, $e = v(\, v'\,)$ where v is not a lambda-abstraction (and is, thus, a constant c). By inverting the (app) typing rule, we have $v : t_1 \to t_2$ and $v' : t_1$. This contradicts our assumption: a constant c is typed with the (cst) rule, with type c, and it does not contain functional types so $c \wedge t_1 \to t_2 \simeq \mathbb{0}$ thus subsumption cannot give a constant a functional type.
5. Case $p = \textsc{ArithError}$: Here, $e = v_1 + v_2$ where either v_1 or v_2 is not an integer. Inverting the (+) typing rule gives us $v_1 : \texttt{int}$ and $v_2 : \texttt{int}$. This contradicts our assumption, as both v_1 and v_2 must be integers. □

Remark 4.3.5. *It's worth noting that $p = \textsc{OutOfRange}$ is not prevented by the typing rules (proj$_\omega$) and (proj$_\omega^{\mathbb{1}}$), which allow expressions like $\pi_3\,\{1,2\}$ to be typed as $1 \vee 2$ or $\mathbb{1}$. While these rules seem necessary to avoid burdening programmers with statically proving index bounds, a practical implementation should include a rule that raises a type error for $\pi_e\, e'$ when $e' : \{\, t_0, .., t_n\,\}$ and $e : \neg[0..n]$, that is, when the type system can statically prove that the index will be out of range as in the case with $\pi_3\,\{1,2\}$.*

Lemma 4.3.6 (Preservation)**.** *If $\varnothing \vdash_{\mathsf{s}} e : t$ and $e \hookrightarrow e'$, then $\varnothing \vdash_{\mathsf{s}} e' : t$*

Proof. By induction on the size of the derivation tree and case analysis on the typing rule used to derive $\varnothing \vdash_{\mathsf{s}} e : t$. The reduction hypothesis excludes rules (cst), (var) and (λ). In every case, if $e \hookrightarrow e'$ is a context reduction, then we apply the induction hypothesis to its premises and conclude by re-applying the typing rule. Thus, we only explicitly treat rules for which there is a distinct reduction:

(app) $e = e_1(\, e_2\,)$. By inversion of the typing rule, we have $\varnothing \vdash_{\mathsf{s}} e_1 : t_1 \to t_2$ and $\varnothing \vdash_{\mathsf{s}} e_2 : t_1$. Since the reduction is β-reduction, we have $e_1 = \lambda^{\mathbb{I}} x.e_1'$ and $e' = e_1'[e_2/x]$. By Substitution 4.3.3, we deduce that $\varnothing \vdash_{\mathsf{s}} e_1'[e_2/x] : t_2$.

(case) $e = \texttt{case}\; e'\,(\rho_i \to e_i)_{i\in I}$. By inversion of the typing rule, we have $\varnothing \vdash_{\mathsf{s}} e' : t'$ and $\forall i \in I\; \big((t' \wedge \rho_i) \setminus (\bigvee_{j<i} \rho_j) \not\leq \mathbb{0} \Rightarrow e_i : t\big)$. Due to the (case) reduction, e' is a value of type t'. The exhaustiveness condition on the case typing rule tells us that $t' \leq \bigvee_{i\in I}\rho_i$, so there exists $i_0 \in I$ (the first ρ_i that matches) such that $\texttt{case}\; e'\,(\rho_i \to e_i)_{i\in I} \hookrightarrow e_{i_0}$ and $(t' \wedge \rho_{i_0}) \setminus (\bigvee_{j<i_0} \rho_j) \not\leq \mathbb{0}$, thus $\varnothing \vdash_{\mathsf{s}} e_{i_0} : t$ which concludes.

(proj) $e = \pi_j\,\{\, v_0, .., v_n\,\}$ with $j \in \{0, .., n\}$ and $e' = v_j$. By inversion of the typing rule, we have $v_j : t_j$ and $e : \bigvee_{i\in K} t_i$ for **some** $K \subseteq [0, n]$. Notice then that the premise forces j to be of type $\bigvee_{i\in K} t_i$, meaning that $j \in K$ (since singleton integers are disjoint). Thus, the (proj) rule can also be applied with $\{j\}$ over K, which yields $e : v_j$ and concludes. □

Using the Progress and Preservation lemmas, we can now conclude the standard type safety theorems for the static type system. The first theorem applies to programs that were type-checked without any use of the ω-rules (i.e., without relying on warnings for potential runtime errors). The second theorem handles the case where ω-rules were used, in which case the only possible runtime type error is an out-of-range tuple projection.

Theorem 4.3.7 (Static Type Safety). *If $\varnothing \vdash_s e : t$ has a derivation that does not use any ω-marked rules, then evaluation of e will never produce a runtime type error. In particular, either e diverges (evaluates forever without a value) or e evaluates to some value v with $\varnothing \vdash_s v : t$. (Well-typed programs cannot "go wrong.")*

Theorem 4.3.8 (ω-Type Safety). *If $\varnothing \vdash_s e : t$ is derivable even with use of ω-rules, then either e diverges, or e evaluates to a value v with $\varnothing \vdash_s v : t$, or e terminates abruptly with an "index out of range" failure ($\omega_{\text{OutOfRange}}$). This last scenario can occur only as a result of the (proj$_\omega$)/(proj$_\omega^{\mathbb{1}}$) rules - it is precisely the case that was flagged by a static warning.*

4.4 Extending semantic subtyping: multi-arity functions

When studying Elixir, its expressions and informal types, from the semantic subtyping framework point of view one thing stands out: there are complex types that are not handled by the original system. The most important ones are tuples and maps (the latter was solved in Castagna (2023b) and Castagna and Peyrot (2025a) for polymorphic records), which are used extensively in Elixir. Another one is the multi-arity function type, needed to represent the types of functions with multiple arities. The ℂDuce (2004) system only allows for single-arity function types, which is not sufficient to represent all Elixir functions. In this section we will show how to extend the semantic subtyping framework to handle these three types. For any language designer willing to adopt semantic subtyping, this section is a good starting point to understand how to extend the system to handle new types. We will also show how to use the set-theoretic semantics of the original system to define the semantics of these new types. This is necessary for the theory, to have well-founded types. But it is also important for the implementation, as the algorithms that manipulate types, in particular deciding emptiness, are defined by resolving equation on sets, seen as the sets of domain values. A correct interpretation of some new types will thus eventually produce both a good foundation for proving properties, and after a fair amount of work to turn the equations on sets into algorithms, a good implementation.

We also note that, in some cases, it may just be possible to reuse previous types of the framework in order the decide subtyping for new types. For instance, tuples could be encoded as a pair of pairs and an integer type representing the size of the tuple. In that case, there would be no need to re-define the interpretation of tuples, but the algorithm used will not be native and instead depend on the previously defined algorithm for pairs and integer types.

In this section, in order to demonstrate the technique, we extend the semantic subtyping framework with genuine multi-arity function types. Tuples are developed separately in Section 4.5.

4.4.1 Multi-arity Functions

Function arity plays an important role both in Elixir and in Erlang, as it is used in conjunction with function names to identify functions. This is reflected by the presence among the guards of the test `is_function(f, n)` that tests whether `f` is a function of arity `n`, and whose usage we showed in the definition of the `curry` function in lines 128–134. To properly support multi-arity functions, we extend the syntax of Core Elixir as follows:

$$
\begin{array}{lrcl}
\textbf{Expressions} & e & ::= & \cdots \mid \lambda^{\mathbb{I}}\,\overline{x}.e \mid e(\,\overline{e}\,) \\
\textbf{Types} & t & ::= & \cdots \mid (t_1, \ldots, t_n) \to t \\
\textbf{Test Types} & \rho & ::= & \cdots \mid \mathtt{fun}_n
\end{array}
$$

Abstractions have lists of parameters $\overline{x}$, applications have lists of arguments $\overline{e}$, the domain of an arrow type is a list of types $\overline{t}$, and it is now possible to test whether a value is a function of arity n, by checking the test type $\mathtt{fun}_n$.

In the previous sections we used the subtyping relation introduced in Chapter 3, assuming that it kept working for our calculus. But the tuples and non-unary function types require an extension of the theory, to define and decide a subtyping relation on these new types. In semantic subtyping this is not always straightforward: although the techniques to do so are extensively explained in the literature (e.g., (Frisch et al., 2008; Castagna and Frisch, 2005; Castagna, 2020; Castagna, 2023a)), it may not be obvious how to adapt them to specific situations. There are two ways to do so: either by defining an encoding of your custom types into existing types or by extending the semantic interpretation of types to support them. For instance, it is possible to encode multi-arity function types by using single arity arrows and a tuple of two elements: one element contains the arrow type (with multi-arguments represented as tuple types), and the other contains the arity (more precisely, the singleton type of the integer constant representing the arity). According to this encoding, two arrow types are comparable only if they have the same arity element (see the semantic interpretation in §4.4.2 below). The set of all functions of arity n is encoded as $\{\, n, (\mathbb{0} \to \mathbb{1})\,\}$, and $\{\, \mathtt{int}, (\mathbb{0} \to \mathbb{1})\,\}$ is the type of all functions. When implemented, this encoding (which requires integer singletons) is not very efficient, but it works.

But using an encoding is not always possible: strong arrows (presented in our introduction to the Elixir type system, in Chapter 1) require checking properties that are not within the scope of the existing theory of semantic subtyping. In that case we need to extend the theory of semantic subtyping to include the new types. This consists of two steps:

1. defining the semantic interpretation of the new type;
2. deriving from this interpretation the decomposition rules to check subtyping for the new type.

We describe below these steps first for multi-arity function types [2].

Remark (*Gradual functions require* $\mho$)

There is another reason to go for a proper representation for multi-arity types, which is related to gradual types (see Chapter 5). When mixing with the dynamic type, it breaks the representation of types that contain dynamic as arguments. Consider the function

$$\{?, \texttt{int}\} \to \texttt{int}$$

which in this representation is the type of functions that, given any first argument and an integer, return an integer. The theorem for representing gradual types (Theorem 3.2.6) tells us that this type is equivalent to

$$\{\mathbb{1}, \texttt{int}\} \to \texttt{int} \vee (? \wedge (\{\mathbb{0}, \texttt{int}\} \to \texttt{int}))$$

where we obtain the left term by replacing all covariant (resp. contravariant) occurrences of $?$ by $\mathbb{0}$ (resp. $\mathbb{1}$), and the right term by doing the opposite of that. We notice that this states that such a function can be either a total function on pairs of $\mathbb{1}$ and `int`, or a subtype of all functions that take some value, and an integer, and return an integer. But the latter type, in the original formulation of semantic subtyping, is not representable: it is equivalent to $\mathbb{0} \to \texttt{int}$ since $\{\mathbb{0}, \texttt{int}\}$ is equivalent to $\mathbb{0}$.

4.4.2 Set-theoretic interpretation

The interpretation of types with single arity arrows is given in Definition 3.1.3. Under domain $\mathscr{D}$, denoting $\mathscr{D}_\mho$ as $\mathscr{D} \cup \{\mho\}$, and $\mathscr{D}_\Omega$ as $\mathscr{D} \cup \{\Omega\}$, we have, for all $R \in \mathscr{P}_f(\mathscr{D}_\mho \times \mathscr{D}_\Omega)$,

$$(R : t \to s) \quad \Leftrightarrow \quad \forall (\iota, \delta) \in R,\ ((\iota : t) \text{ or } \iota = \mho) \implies (\delta : s)$$

We extend this interpretation to multi-arity functions: given $n \in [0, 255]$[3], for all $R \in \mathscr{P}_f\left({\mathscr{D}_\mho}^n \times \mathscr{D}_\Omega\right)$,

$$(R : (t_1, .., t_n) \to s) \;\Leftrightarrow\; \forall ((\iota_1, .., \iota_n), \delta) \in R.\ (\forall i = 1..n.\ (\iota_i : t_i) \text{ or } \iota_i = \mho) \implies (\delta : s)$$

This defines the interpretation of multi-arity functions as parts of ${\mathscr{D}_\mho}^n \times \mathscr{D}_\Omega$. So for instance, type $(\mathbb{0}, \texttt{int}) \to \texttt{int}$ contains relations with pairs such as $((\mho, 1), 2)$ (and all pairs $((d_1, d_2), \delta)$ where d_2 is not in the integer domain).

Remark 4.4.1. *As a justification for the use of $\mho$, consider what the natural interpretation for multi-arity functions would be if we did not use it:*

$$(R : (t_1, .., t_n) \to s) \quad \Leftrightarrow \quad \forall ((d_1, .., d_n), \delta) \in R.\ (\forall i \in \{1, ..., n\}.\ d_i : t_i) \implies (\delta : s)$$

We would not be able to represent more precise types such as $(\mathbb{0}, \texttt{int}) \to \texttt{int}$: since there is no element in the interpretation of $\mathbb{0}$, this type is interpreted as $\mathscr{P}_f\left(\mathscr{D}^2 \times \mathscr{D}_\Omega\right)$ which is not the correct type.

[2]We will present the same development for the strong arrow types in Chapter 5 (Section 5.3).

[3]The maximum arity of a function in Elixir is 255.

This representation requires an update to the algorithm that decides set containment, which will then be used to decide subtyping. We start by defining the formal set of elements in the domain of a multi-arity function:

Definition 4.4.2. *Let $X_1,..,X_n,Y$ be subsets of $\mathscr{D}$. We define*

$$(X_1,..,X_n) \to Y = \{R \in \mathscr{P}_f\left({\mathscr{D}_\mho}^n \times \mathscr{D}_\Omega\right) \mid \\ \forall((\iota_1,..,\iota_n),\delta) \in R.\ (\forall i = 1..n.\ (\iota_i \in X_i)\ or\ \iota_i = \mho) \Rightarrow \delta \in Y\}$$

The first step is to show that this new definition provides a form for functions that is similar to the one we had for single-arity functions, as the finite parts of an analytic expression of the domain and codomain sets.

Lemma 4.4.3. *For all $X_1,..,X_n,Y$ subsets of $\mathscr{D}$, writing $X_i^\mho = X_i \cup \{\mho\}$, we have*

$$(X_1,..,X_n) \to Y = \mathscr{P}_f\left(\overline{X_1^\mho \times \cdots \times X_n^\mho \times \overline{Y}^{\mathscr{D}_\Omega}}^{{\mathscr{D}_\mho}^n \times \mathscr{D}_\Omega}\right)$$

Proof. By reciprocal inclusion:

- $(\subset)$. Let $R \in (X_1,..,X_n) \to Y$. Consider any element $((\iota_1,..,\iota_n),\delta) \in R$. Two cases:
  Case 1: There exists at least one i such that $\iota_i \notin X_i$ and $\iota_i \neq \mho$. Then $((\iota_1,..,\iota_n),\delta)$ is not in $X_1^\mho \times \cdots \times X_n^\mho \times \overline{Y}^{\mathscr{D}_\Omega}$, so it belongs to its complement.
  Case 2: For all i, $\iota_i \in X_i$ or $\iota_i = \mho$. By definition of R, this implies $\delta \in Y$, which means $\delta \notin \overline{Y}^{\mathscr{D}_\Omega}$. Therefore, $((\iota_1,..,\iota_n),\delta)$ is not in $X_1^\mho \times \cdots \times X_n^\mho \times \overline{Y}^{\mathscr{D}_\Omega}$, and thus belongs to the complement.

  In both cases, $((\iota_1,..,\iota_n),\delta) \in \overline{X_1^\mho \times \cdots \times X_n^\mho \times \overline{Y}^{\mathscr{D}_\Omega}}^{{\mathscr{D}_\mho}^n \times \mathscr{D}_\Omega}$. Since R is finite, we have $R \in \mathscr{P}_f\left(\overline{X_1^\mho \times \cdots \times X_n^\mho \times \overline{Y}^{\mathscr{D}_\Omega}}^{{\mathscr{D}_\mho}^n \times \mathscr{D}_\Omega}\right)$.

- $(\supset)$. Let $R \in \mathscr{P}_f\left(\overline{X_1^\mho \times \cdots \times X_n^\mho \times \overline{Y}^{\mathscr{D}_\Omega}}^{{\mathscr{D}_\mho}^n \times \mathscr{D}_\Omega}\right)$. Let $((\iota_1,..,\iota_n),\delta) \in R$.

  Assume that $\forall i = 1..n, (\iota_i \in X_i$ or $\iota_i = \mho)$. We need to show that $\delta \in Y$.

  By definition of R, we have $((\iota_1,..,\iota_n),\delta) \in \overline{X_1^\mho \times \cdots \times X_n^\mho \times \overline{Y}^{\mathscr{D}_\Omega}}^{{\mathscr{D}_\mho}^n \times \mathscr{D}_\Omega}$.
  This means $((\iota_1,..,\iota_n),\delta) \notin X_1^\mho \times \cdots \times X_n^\mho \times \overline{Y}^{\mathscr{D}_\Omega}$.
  Given our assumption that $\forall i = 1..n, (\iota_i \in X_i$ or $\iota_i = \mho)$, we know that $(\iota_1,..,\iota_n)$ satisfies the domain constraints. Therefore, the only way for $((\iota_1,..,\iota_n),\delta)$ to not be in $X_1 \times \cdots \times X_n \times \overline{Y}^{\mathscr{D}_\Omega}$ is if $\delta \notin \overline{Y}^{\mathscr{D}_\Omega}$.
  Therefore, $\delta \in Y$, which proves that $R \in (X_1,..,X_n) \to Y$. □

Then, we want to give a formal proof of a result that is a generalization of Lemma 4.6 of Frisch (2004):

Lemma 4.4.4. *Let $X^{(1)},\dots,X^{(n)}$ and $Y_i^{(1)},\dots,Y_i^{(n)}$ be subsets of $\mathscr{D}$ for $i \in P$. Then,*

$$\bigcap_{i\in P}(X^{(1)}\times\ldots\times X^{(n)})\smallsetminus(Y_i^{(1)}\times\ldots\times Y_i^{(n)}) = \bigcup_{\iota:P\to[1,n]}\prod_{k=1}^{n}\Big(X^{(k)}\smallsetminus\bigcup_{\{i\in P\mid\iota(i)=k\}}Y_i^{(k)}\Big)$$

Proof. By reciprocal inclusion:

($\supseteq$) Fix $\iota : P \to [1,n]$ a total function, and let $x=(x_1,..,x_n)$ be in $\prod_{k=1}^n\big(X^{(k)}\smallsetminus\bigcup_{\{i\mid\iota(i)=k\}}Y_i^{(k)}\big)$. Then $x\in X^{(1)}\times\cdots\times X^{(n)}$. For every $i_0\in P$, set $k_0=\iota(i_0)$. Since $x_{k_0}\in X^{(k_0)}\smallsetminus\bigcup_{\{i\mid\iota(i)=k_0\}}Y_i^{(k_0)}$, we have $x_{k_0}\notin Y_{i_0}^{(k_0)}$, hence $x\notin Y_{i_0}^{(1)}\times\cdots\times Y_{i_0}^{(n)}$. Because i_0 was chosen arbitrary, and this property holds for all $i_0\in P$,we have $x\in\bigcap_{i\in P}\big((X^{(1)}\times\cdots\times X^{(n)})\smallsetminus(Y_i^{(1)}\times\cdots\times Y_i^{(n)})\big)$.

($\subseteq$) Let $x=(x_1,..,x_n)$ belong to the left-hand side. Then for every $i\in P$, $x\in(X^{(1)}\times\cdots\times X^{(n)})\smallsetminus(Y_i^{(1)}\times\cdots\times Y_i^{(n)})$, so for every $i\in P$ there exists $k_i\in[1,n]$ with $x_{k_i}\notin Y_i^{(k_i)}$. Define ι_0 to be the function that maps each $i\in P$ to the corresponding k_i, i.e., $\iota_0(i)=k_i$. For every $k\in[1,n]$ we have $x_k\in X^{(k)}$ and for every i with $\iota_0(i)=k$, we have $x_k\notin Y_i^{(k)}$; hence $x_k\in X^{(k)}\smallsetminus\bigcup_{\{i\mid\iota_0(i)=k\}}Y_i^{(k)}$ (the union is $\varnothing$ if no such i). Therefore, $x\in\prod_{k=1}^n\big(X^{(k)}\smallsetminus\bigcup_{\{i\mid\iota_0(i)=k\}}Y_i^{(k)}\big)$, so x belongs to the union of the right-hand side. □

Thus we can prove the multi-arity set-containment theorem:

Theorem 4.4.5 (Multi-arity Set-Containment)**.** *Let $n\in\mathbb{N}$. Given families of subsets of the domain $\mathscr{D}$, $(X_i^{(1)})_{i\in P},..,(X_i^{(n)})_{i\in P}$, $(X_i)_{i\in P}$, $(Y_i^{(1)})_{i\in N},..,(Y_i^{(n)})_{i\in N}$, $(Y_i)_{i\in N}$, then,*

$$\bigcap_{i\in P}\Big(X_i^{(1)},..,X_i^{(n)}\Big)\to X_i\subseteq\bigcup_{j\in N}\Big(Y_j^{(1)},..,Y_j^{(n)}\Big)\to Y_j \quad\Leftrightarrow\quad \exists j_0\in N.$$
$$\forall\iota:P\to[1,n+1].\left(\exists k\in\iota(P),k\le n.\ Y_{i_0}^{(k)}\subseteq\bigcup_{\{i\in P\,|\,\iota(i)=k\}}X_i^{(k)}\right)\ \textbf{or}\left(\bigcap_{\{i\in P\,|\,\iota(i)=n+1\}}X_i\subseteq Y_{j_0}\right)$$

where, by convention, $\bigcap_\varnothing(\cdot)$ is $\mathscr{D}_\mho$ for subsets of $\mathscr{D}_\mho$ (domains) and is $\mathscr{D}_\Omega$ for subsets of $\mathscr{D}_\Omega$ (codomains).

This theorem states that to check whether an intersection of arrows of arity n is contained in a union of arrows of the same arity, we need to find at least one arrow in the union (the "$\exists j_0\in N$" in the right-hand side of the formula) for which such a containment holds. The theorem then reduces this containment problem on arrows, to multiple smaller containment checks on their domain and return types. It thus enables the definition of a recursive algorithm that decides subtyping. Notice that the multiple smaller problems are related by an "**or**" which means that a naive implementation of the algorithm described by Theorem 4.4.5 may have to backtrack and, therefore, unroll all the memoized solutions found in the current run.

Before delving into the proof of this theorem, we illustrate its main idea with a simple corollary: the containment between two binary arrow types.

Corollary 4.4.6 (Application to two binary arrows)**.**

$$(X_1^{(1)}, X_1^{(2)}) \to X_1 \subseteq (Y_1^{(1)}, Y_1^{(2)}) \to Y_1$$

if and only if: $(Y_1^{(1)} \subseteq X_1^{(1)})\ \mathit{AND}\ (Y_1^{(2)} \subseteq X_1^{(2)})\ \mathit{AND}\ (X_1 \subseteq Y_1)$

Proof. Consider the statement of the theorem with these parameters:

- $P = \{1\}$: We have a single function type on the left side of the containment
- $N = \{1\}$: We have a single function type on the right side
- $n = 2$: The functions have arity 2 (taking two arguments)

The theorem's statement provides a necessary and sufficient condition such that:

$$(X_1^{(1)}, X_1^{(2)}) \to X_1 \subseteq (Y_1^{(1)}, Y_1^{(2)}) \to Y_1$$

According to the theorem, this containment holds if and only if:

$$\exists j_0 \in N \text{ such that } \forall \iota : P \to [1, n+1]$$

and one of these conditions is satisfied:

- $\exists k \in \iota(\{1\}), k \leq n$ such that $Y_{j_0}^{(k)} \subseteq \bigcup_{\{i \in P | \iota(i) = k\}} X_i^{(k)}$
- OR $\bigcap_{\{i \in P | \iota(i) = n+1\}} X_i \subseteq Y_{j_0}$

Since $N = 1$ (one function on the right), we have only one choice for j_0, which is 1.
Since $P = 1$ (one function on the left), the functions to check are $\iota : \{1\} \to [1, 3]$. There are three possibilities:

- Case 1: $\iota(1) = 1$
  First condition: $\exists k \in \{1\}$ such that $Y_1^{(k)} \subseteq \bigcup_{\{i \in P | \iota(i) = k\}} X_i^{(k)}$. Since there is only one choice for k, that is $k = 1$, this is equivalent to checking $Y_1^{(1)} \subseteq X_1^{(1)}$.
  Second condition: $\bigcap_{\{i \in P | \iota(i) = 3\}} X_i \subseteq Y_1$. Since no i satisfies $\iota(i) = 3$, this intersection is over an empty set, so this condition is false because the intersection is thus $\mathscr{D}_\Omega$ by convention, and Y_1 does not contain Ω.
  For this case, the whole condition simplifies to: $(Y_1^{(1)} \subseteq X_1^{(1)})$
- Case 2: $\iota(1) = 2$
  First condition: $\exists k \in \{2\}$ such that $Y_1^{(k)} \subseteq \bigcup_{\{i \in P | \iota(i) = k\}} X_i^{(k)}$. As in Case 1, we can only choose $k = 2$ and thus this is equivalent to checking $Y_1^{(2)} \subseteq X_1^{(2)}$
  Second condition: Similar to Case 1, evaluates to false
  For this case, the whole condition simplifies to: $Y_1^{(2)} \subseteq X_1^{(2)}$
- Case 3: $\iota(1) = 3$

First condition: it can never be satisfied since it requires that $\exists k \in \{3\}$ such that $k \leq 2$. So this is false.

Second condition: $\bigcap_{\{i \in P \mid \iota(i)=3\}} X_i \subseteq Y_1$. Since $\iota(1) = 3$, this becomes $X_1 \subseteq Y_1$

For this case, the condition simplifies to: $X_1 \subseteq Y_1$

CONCLUSION For the containment to hold, all three cases must be satisfied: $(Y_1^{(1)} \subseteq X_1^{(1)})$ and $Y_1^{(2)} \subseteq X_1^{(2)})$ $(X_1 \subseteq Y_1)$. $\square$

This confirms the standard contravariant/covariant subtyping rule: parameter types are contravariant (Y subtypes of X), return type is covariant (X_1 subtype of Y_1). As final note on this corollary, notice that the condition on the left-hand side of the **or** is never satisfied for $\{i \in P \mid \iota(i) = k\} = \varnothing$: this is because $Y_{i_0}^{(k)}$ always contains $\mho$ and therefore it cannot be contained in the empty set. It is this very condition that ensures that, contrary to the unary case, even if one of the domains $Y_j^{(i)}$ is (the interpretation of) the empty type, the subtyping relation must still check the containment of the codomains (i.e., the right-hand side of the **or**). Without $\mho$, we would have a weaker third condition for the containment, that is: $(Y_1^{(1)} \subseteq X_1^{(1)})$ AND $(Y_1^{(2)} \subseteq X_1^{(2)})$ AND $((Y_1^{(1)} = \varnothing)$ OR $(Y_1^{(2)} = \varnothing)$ OR $(X_1 \subseteq Y_1))$.

We now prove theorem 4.4.5.

Proof. Using Lemmas 4.4.3 and 3.1.18, and the notation $\iota^{-1}(k) = \{i \in P \mid \iota(i) = k\}$.

$$\bigcap_{i\in P}\left(X_i^{(1)},\dots,X_i^{(n)}\right) \to X_i \subseteq \bigcup_{j\in N}\left(Y_j^{(1)},\dots,Y_j^{(n)}\right) \to Y_j$$

$$\overset{(4.4.3)}{\Leftrightarrow} \bigcap_{i\in P}\mathscr{P}_f\left(\overline{X_i^{(1)\mho} \times \dots \times X_i^{(n)\mho} \times \overline{X_i}^{\mathscr{D}_\Omega}}^{\mathscr{D}_\mho{}^n \times \mathscr{D}_\Omega}\right) \subseteq \bigcup_{j\in N}\mathscr{P}_f\left(\overline{Y_j^{(1)\mho} \times \dots \times Y_j^{(n)\mho} \times \overline{Y_j}^{\mathscr{D}_\Omega}}^{\mathscr{D}_\mho{}^n \times \mathscr{D}_\Omega}\right)$$

$$\overset{(4.8)}{\Leftrightarrow} \exists j_0 \in N.\ \bigcap_{i\in P}\overline{X_i^{(1)\mho} \times \dots \times X_i^{(n)\mho} \times \overline{X_i}^{\mathscr{D}_\Omega}}^{\mathscr{D}_\mho{}^n \times \mathscr{D}_\Omega} \subseteq \overline{Y_{j_0}^{(1)\mho} \times \dots \times Y_{j_0}^{(n)\mho} \times \overline{Y_{j_0}}^{\mathscr{D}_\Omega}}^{\mathscr{D}_\mho{}^n \times \mathscr{D}_\Omega}$$

$$\overset{(4.4.4)}{\Leftrightarrow} \exists j_0 \in N.\ \bigcup_{\iota:P\to[1,n+1]}\left(\prod_{k=1}^{n}\left(\overline{\bigcup_{i\in\iota^{-1}(k)} X_i^{(k)\mho}}^{\mathscr{D}_\mho}\right) \times \overline{\bigcup_{i\in\iota^{-1}(n+1)} \overline{X_i}^{\mathscr{D}_\Omega}}^{\mathscr{D}_\Omega}\right) \subseteq \overline{Y_{j_0}^{(1)\mho} \times \dots \times Y_{j_0}^{(n)\mho} \times \overline{Y_{j_0}}^{\mathscr{D}_\Omega}}^{\mathscr{D}_\mho{}^n \times \mathscr{D}_\Omega}$$

$$\Leftrightarrow \exists j_0 \in N.\ \bigcup_{\iota:P\to[1,n+1]}\left(\prod_{k=1}^{n}\left(Y_{j_0}^{(k)\mho} \cap \bigcap_{i\in\iota^{-1}(k)} \overline{X_i^{(k)\mho}}^{\mathscr{D}_\mho}\right) \times \left(\overline{Y_{j_0}}^{\mathscr{D}_\Omega} \cap \bigcap_{i\in\iota^{-1}(n+1)} X_i\right)\right) = \varnothing$$

The equivalence above labeled 4.4.4 is the application of Lemma 4.4.4 to the set

$$\bigcup_{i\in P}(\overbrace{\mathscr{D}_\mho \times \dots \times \mathscr{D}_\mho}^{i \text{ times}} \times \mathscr{D}_\Omega) \setminus (X_i^{(1)\mho},\dots,X_i^{(n)\mho},\overline{X_i}^{\mathscr{D}_\Omega})$$

while the last equivalence is obtained by bringing the right-hand side set to the left-hand side, applying De Morgan's law on the negated unions, and applying the intersections of products component-wise.

Let a map $\iota : P \to [1, n+1]$, and $k \in [1, n]$.

- <u>How can the set ${Y_{j_0}^{(k)}}^{\mho} \cap \bigcap_{\{i \in P\,;\,\iota(i)=k\}} \overline{{X_i^{(k)}}^{\mho}}^{\mathscr{D}_\mho}$ be empty?</u>
  Immediately, we notice that if $\{i \in P \mid \iota(i) = k\}$ is empty, then the intersection is on an empty set and is $\mathscr{D}_\mho$. Therefore, the intersection cannot be empty since both ${Y_{j_0}^{(k)}}^{\mho}$ and $\mathscr{D}_\mho$ contain $\mho$. So a first necessary condition is that $\{i \in P \mid \iota(i) = k\} \neq \varnothing$.
  Then by applying De Morgan's law we obtain that this intersection is empty iff ${Y_{j_0}^{(k)}}^{\mho} \subseteq \bigcup_{\{i \in P \mid \iota(i)=k\}} {X_i^{(k)}}^{\mho}$. Since $\mho$ is in both sides of the inclusion, we can discard it obtaining: $Y_{j_0}^{(k)} \subseteq \bigcup_{\{i \in P \mid \iota(i)=k\}} X_i^{(k)}$. Finally since in the formula k is the index of the domains, it must be in $[1, n]$. Therefore, the condition that $\{i \in P \mid \iota(i) = k\} \neq \varnothing$ is equivalent to require that $\exists k \in \iota(P) \cap [1, n]$, from which we obtain the first **or** clause of the theorem.
- <u>How can the set $\overline{Y_{j_0}}^{\mathscr{D}_\Omega} \cap \bigcap_{\{i \in P\,;\,\iota(i)=n+1\}} X_i$ be empty?</u>
  First, note that if there is no $i \in P$ such that $\iota(i) = n+1$, then the intersection is on an empty set and is $\mathscr{D}_\Omega$. Hence its intersection with $\overline{Y_{j_0}}^{\mathscr{D}_\Omega}$ is never empty, as it contains ω. If that is not the case, then this intersection is empty iff $\bigcap_{\{i \in P \mid \iota(i)=n+1\}} X_i \subset Y_{j_0}$.
  Finally, note that the convention that an intersection over the empty set over subsets of $\mathscr{D}_\Omega$ (which is the case for codomain sets) is $\mathscr{D}_\Omega$ takes into account the special case mentioned above. □

4.4.3 Subtyping algorithm

Notation (arity and symbols). Fix $n \in \mathbb{N}$. Every arrow $f^{(i)} \in P$ has arity n and is written $f^{(i)} = (s_1^{(i)}, \ldots, s_n^{(i)}) \to s^{(i)}$. We reserve s-letters for the left-hand side (LHS, indexed by $i \in P$) and t-letters for the right-hand side (RHS); the index $k \in [1, n]$ denotes argument positions.

Subtyping algorithm. From the proof of Theorem 4.4.5 we see that the subtyping problem

$$\bigwedge_{i \in P} \left(s_1^{(i)}, \ldots, s_n^{(i)}\right) \to s^{(i)} \leq \bigvee_{j \in N} \left(t_1^{(j)}, \ldots, t_n^{(j)}\right) \to t^{(j)} \tag{4.4.1}$$

is decided by finding a single arrow (let us say it is $j = 1$) on the right hand side such that

$$\bigwedge_{i \in P} \left(s_1^{(i)}, \ldots, s_n^{(i)}\right) \to s^{(1)} \leq \left(t_1^{(1)}, \ldots, t_n^{(1)}\right) \to t^{(1)} \tag{4.4.2}$$

Let us then define a function that for all $n \in \mathbb{N}$ decides (4.4.2). This is expressed by function Φ_n of $n+2$ arguments, the first $n+1$ arguments being triples of a boolean and two types, and the last one a set of arrow types. The function is defined as follows:

$$\begin{aligned}
&\Phi_n((b_1,t_1,s_1),\dots,(b_n,t_n,s_n),(b,t,s),\emptyset) = (\exists i\in[1,n].\ (b_i\ \textbf{and}\ t_i\le s_i))\ \textbf{or}\ (b\ \textbf{and}\ s\le t)\\
&\Phi_n((b_1,t_1,s_1),\dots,(b_n,t_n,s_n),(b,t,s),\{(t'_1,..,t'_n)\to t'\}\cup P) =\\
&\qquad\qquad \Phi_n((b_1,t_1,s_1),..,(b_n,t_n,s_n),(\texttt{true},t,s\wedge t'),P)\ \textbf{and}\\
&\qquad\qquad \forall j\in[1..n].\ \Phi_n((b_1,t_1,s_1),..,(\texttt{true},t_j,s_j\vee t'_j),..,(b_n,t_n,s_n),(b,t,s),P)
\end{aligned}$$

To understand this definition and its parameters:

- t_j tracks the RHS domain component t_j (argument position j).
- s_j accumulates the LHS domains at position j: on a leaf it represents $\bigvee_{i\in\iota^{-1}(j)} s_j^{(i)}$ (encoded via unions).
- (b_j) records whether position j was selected at least once (so the corresponding union is non-empty).
- (b,t,s) tracks the codomain: t is the RHS codomain, while s accumulates the LHS codomains $\bigwedge_{i\in\iota^{-1}(n+1)} s^{(i)}$.

As noted earlier, when $\iota^{-1}(j)=\varnothing$ the domain-side containment $t_j\le\bigvee_{i\in\iota^{-1}(j)} s_j^{(i)}$ cannot hold; the booleans b_j witness that a position has been chosen. For the codomain, the convention that an empty intersection over subsets of $\mathscr{D}_\Omega$ is $\mathscr{D}_\Omega$ explains why $b=\texttt{false}$ makes the codomain clause unsatisfiable.

Theorem 4.4.7. *For all $n\in\mathbb{N}$, for P a set of arrows of arity n,*

$$\bigwedge_{f\in P} f\le(t_1,..,t_n)\to t \Leftrightarrow \Phi_n((\texttt{false},t_1,\mathbb{0}),..,(\texttt{false},t_n,\mathbb{0}),(\texttt{false},t,\mathbb{1}),P)$$

APPLICATION. For $P=\{(s_1,..,s_n)\to s\}$, we have shown previously in the application of Theorem 4.4.5 why this results in subtyping (resp. supertyping) for the codomain s (resp. the argument $s_1,..,s_n$) with the codomain t (resp. the argument $t_1,..,t_n$). Looking at Φ_n and its recursive definition, we can see that the call that consumes the only arrow in P will produce $n+1$ final calls: one for each of the n arguments, and one for the result type. Each of those has exactly one Boolean variable set to `true` with the remaining set to `false`. Each argument call will lead to a condition $(t_i\le s_i)$, and the result call will lead to condition $(s\le t)$. Thus, we have:

$$(s\le t)\ \textbf{and}\ \forall i\in[1,n].\ (t_i\le s_i)$$

Proof. We prove that the function Φ_n correctly implements the set-containment conditions from Theorem 4.4.5. The proof proceeds by induction on the size of P.
First, let us recall the set-containment theorem for multi-arity functions:
For a subtyping problem:

$$\bigwedge_{i\in P}\left(X_i^{(1)},\dots,X_i^{(n)}\right)\to X_i\subseteq\bigvee_{j\in N}\left(Y_j^{(1)},\dots,Y_j^{(n)}\right)\to Y_j$$

This is equivalent to: $\exists j_0\in N$ such that for all mappings $\iota:P\to[1,n+1]$, either:

- $\exists k \in \iota(P) \cap [1,n]$ such that $Y_{j_0}^{(k)} \subset \bigcup_{i\in\iota^{-1}(k)} X_i^{(k)}$
- or $\bigcap_{i\in\iota^{-1}(n+1)} X_i \subset Y_{j_0}$

For our specific case with a single arrow on the right ($N = \{1\}$), the problem simplifies to:

$$\bigwedge_{i\in P} \left(s_i^{(1)}, \ldots, s_i^{(n)}\right) \to s_i \leq (t_1, \ldots, t_n) \to t$$

Which holds if and only if for all mappings $\iota : P \to [1, n+1]$, either:

- $\exists k \in \iota(P) \cap [1,n]$ such that $t_k \leq \bigvee_{i\in\iota^{-1}(k)} s_i^{(k)}$
- or $\bigwedge_{i\in\iota^{-1}(n+1)} s_i \leq t$

Base Case: $|P| = 1$ When P is a single arrow $f = (s_1, \ldots, s_n) \to s$, there are only $n+1$ mappings to consider from P to $[1, n+1]$. Those are the $\iota(f) = k$ for $k \in [1, n+1]$. Let $k \in [1, n+1]$ and $\iota(f) = k$. If $k = n+1$, then there is no $k \in \iota(P) \cap [1,n]$, and $\iota^{-1}(n+1)$ points to f, which produces condition $s \leq t$. If $k \in [1,n]$, then $\iota(P) \cap [1,n] = \{k\}$, and $\iota^{-1}(k)$ points to f, which produces condition $t_k \leq s_k$. So we have: $(s \leq t)$ **and** $\forall i \in [1,n]. (t_i \leq s_i)$.
On the other hand, we have shown in the Application above that the call to function Φ_n returns:

$$(s \leq t) \textbf{ and } \forall i \in [1,n]. (t_i \leq s_i)$$

which allows to conclude that the base case holds.

Inductive Step Assume the theorem holds for a set P. We prove it holds for $P \cup \{f'\}$ where $f' = (t'_1, \ldots, t'_n) \to t'$.
By definition:

$$\begin{aligned}&\Phi_n((b_1,t_1,s_1),\ldots,(b_n,t_n,s_n),(b,t,s),\{f'\}\cup P) = \\ &\quad \Phi_n((b_1,t_1,s_1),\ldots,(b_n,t_n,s_n),(\texttt{true},t,s\wedge t'),P) \textbf{ and} \\ &\quad \forall j \in [1,n]\,.\,\Phi_n((b_1,t_1,s_1),\ldots,(\texttt{true},t_j,s_j \vee t'_j),\ldots,(b_n,t_n,s_n),(b,t,s),P)\end{aligned}$$

This recursive definition corresponds exactly to the set-containment conditions:

1. The first recursive call (with $(\texttt{true}, t, s \wedge t')$) represents mapping f' to position $n+1$, meaning we check the codomain condition: $\bigcap_{i\in\iota^{-1}(n+1)} t_i \subseteq t$. The boolean flag is set to true to indicate this position has been chosen.
2. The remaining n recursive calls (with $(\texttt{true}, t_j, s_j \vee t'_j)$) represent mapping f' to each position $j \in [1,n]$, meaning we check the domain condition for each argument position: $t_j \subseteq \bigvee_{i\in\iota^{-1}(j)} s_i^{(j)}$.

Through each recursive call, we're effectively building all possible mappings $\iota : P \to [1, n+1]$.
For each mapping, we check whether the containment conditions hold.
When we reach the base case ($P = \emptyset$), we check:

- For each j where b_j is true: $t_j \leq s_j$ (domain condition)
- If b is true: $s \leq t$ (codomain condition)

These are precisely the set-containment conditions required for subtyping.

Conclusion By induction, we've shown that for all $n \in \mathbb{N}$ and for any set P of arrows of arity n:

$$\bigwedge_{f \in P} f \leq (t_1, \ldots, t_n) \to t \iff \Phi_n((\texttt{false}, t_1, s_1), \ldots, (\texttt{false}, t_n, s_n), (\texttt{false}, t, s), P)$$

The function Φ_n correctly implements the set-containment conditions required for subtyping of multi-arity functions as specified in Theorem 4.4.5. □

Following Frisch (2004), we realize that arguments s_i and s are accumulators for the arguments and codomain of the arrows in P. We can then define a function Φ'_n with fewer arguments (more precisely, where the first $n+1$ arguments are just pairs of a Boolean and a type, instead of specifying two types), by integrating those into the parameters t_i and t using set operations.

Let us define Φ'_n:

$$\begin{aligned}
&\Phi'_n((b_1, t_1), .., (b_n, t_n), (b, t), \varnothing) = \big(\exists j \in [\![1; n]\!].\ (b_j \textbf{ and } t_j \leq \mathbb{0})\big) \textbf{ or } (b \textbf{ and } t \leq \mathbb{0}) \\
&\Phi'_n((b_1, t_1), .., (b_n, t_n), (b, t), \{(s_1, .., s_n) \to s\} \cup P) = \\
&\qquad (\Phi'_n((b_1, t_1), .., (b_n, t_n), (\texttt{true}, t \wedge s, s), P) \textbf{ and} \\
&\qquad\qquad \forall j \in [1, n].\ \Phi'_n((b_1, t_1), .., (\texttt{true}, t_j \setminus s_j), .., (b_n, t_n), (b, t), P))
\end{aligned}$$

The link between Φ_n and Φ'_n is given by:

$$\Phi_n((b_1, t_1, s_1), .., (b_n, t_n, s_n), (b, t, s), P) = \Phi'_n((b_1, t_1 \setminus s_1), .., (b_n, t_n \setminus s_n), (b, s \setminus t), P)$$

Thus we can use for the subtyping problem:

$$\bigwedge_{f \in P} f \leq (t_1, .., t_n) \to t \iff \Phi'_n((\texttt{false}, t_1), .., (\texttt{false}, t_n), (\texttt{false}, t), P)$$

Theorem 4.4.8. *For all $n \in \mathbb{N}$, for P a set of arrows of arity n,*

$$\bigwedge_{f \in P} f \leq (t_1, .., t_n) \to t \iff \Phi'_n((\texttt{false}, t_1), .., (\texttt{false}, t_n), (\texttt{false}, \neg t), P)$$

Proof. Let us show that, for all n, for all booleans $b_1, \ldots, b_n, b$ and types $t_1, s_1, \ldots, t_n, s_n, t, s$, and all finite P,

$$\Phi_n\big((b_1, t_1, s_1), \ldots, (b_n, t_n, s_n), (b, t, s), P\big) \iff \Phi'_n\big((b_1, t_1 \setminus s_1), \ldots, (b_n, t_n \setminus s_n), (b, s \setminus t), P\big).$$

We proceed by induction on $|P|$.

Base $P = \varnothing$. By the intended semantics of Φ_n, at the leaf we check the witnesses collected by the booleans:

$$\Phi_n(\dots,\varnothing) \;\equiv\; \Big(\exists j \le n.\, b_j \wedge (t_j \le s_j)\Big) \vee \big(b \wedge (s \le t)\big).$$

which is equivalent to

$$\Big(\exists j \le n.\, b_j \wedge (t_j \setminus s_j \le \mathbb{0})\Big) \vee \big(b \wedge (s \setminus t \le \mathbb{0})\big),$$

which is precisely $\Phi'_n\big((b_1,t_1\setminus s_1),\dots,(b_n,t_n\setminus s_n),(b,s\setminus t),\varnothing\big)$.

Step $P = \{(s'_1,\dots,s'_n)\to s'\}\cup P'$. Unfolding Φ_n once yields the usual $n{+}1$ branches that enumerate where the new arrow is mapped:

$$\Phi_n(\dots,P) \;\equiv\; \underbrace{\Phi_n\big(\dots,(b,\,t,\,s\wedge s'),\,P'\big)}_{\text{map to result}} \wedge \bigwedge_{j=1}^{n} \underbrace{\Phi_n\big(\dots,(b_j,\,t_j,\,s_j\vee s'_j),\dots,(b,t,s),\,P'\big)}_{\text{map to argument } j}.$$

Apply the induction hypothesis to each conjunct. For the result branch, the translated parameter becomes

$$(b,\,(s\wedge s')\setminus t) \;=\; (b,\,(s\setminus t)\wedge s') \quad \text{since} \quad (x\wedge y)\setminus z \;=\; (x\setminus z)\wedge y.$$

For an argument j, the translated parameter becomes

$$(b_j,\,t_j\setminus(s_j\vee s'_j)) \;=\; (b_j,\,(t_j\setminus s_j)\setminus s'_j) \quad \text{since} \quad x\setminus(y\vee z) \;=\; (x\setminus y)\setminus z.$$

Moreover, in both cases the boolean for the chosen position is set to `true`, exactly as in the definition of Φ'_n. Therefore, if we denote by $\vec{B}$ the vector of arguments $\vec{B} = (b_1,t_1),\dots,(b_n,t_n)$ and allow mutating one of its arguments (the j-th one) with syntax $\vec{B}[j:(b'_j,t'_j)]$, the IH yields

$$\Phi_n(\dots,P) \iff \Phi'_n\big(\vec{B},(\texttt{true},(s\setminus t)\wedge s'),P'\big) \wedge \bigwedge_{j=1}^{n} \Phi'_n\big(\vec{B}[j:(\texttt{true},(t_j\setminus s_j)\setminus s'_j)],(b,s\setminus t),P'\big)$$

which is exactly the unfolding clause of Φ'_n applied to the translated parameters. $\square$

Function operators for n-arity. We previously defined the domain and application result operators for single-arity functions (3.3.1 b)), and discuss now the n-ary generalization of these operators.

Typing rules. The declarative typing rule for application (arity n) is naturally:

$$(\text{app}_n)\;\frac{\Gamma\vdash_{\mathsf{s}} e:(t_1,\dots,t_n)\to s \qquad \forall i.\,\Gamma\vdash_{\mathsf{s}} e_i : t_i}{\Gamma\vdash_{\mathsf{s}} e(e_1,\dots,e_n):s}$$

For the algorithmic rule, we wrap the types of the arguments into a tuple type and compare it with the domain of the function, also defined as a tuple type.

$$(\mathrm{app}_n^{\mathrm{alg}})\ \frac{\Gamma \vdash e : t \qquad \forall i. \Gamma \vdash_{\mathsf{s}} e_i : t_i}{\Gamma \vdash e(e_1, \ldots, e_n) : t \circ \{ t_1, \ldots, t_n \}} \quad \begin{array}{c} t \leq \overbrace{(\mathbb{0}, \ldots, \mathbb{0})}^{n} \to \mathbb{1} \\ \{ t_1, \ldots, t_n \} \leq \mathtt{dom}(t) \ \forall i. t_i \not\simeq \mathbb{0} \end{array}$$

Indeed, we required the definition of n-arity function types in order to be able to distinguish between types such as $(\mathbb{0}, \mathtt{int}) \to \mathtt{int}$ and $(\mathbb{0}, \mathbb{0}) \to \mathtt{int}$, which would not have been possible with tuples since $\{ \mathbb{0}, \mathtt{int} \} \simeq \mathbb{0} \simeq \{ \mathbb{0}, \mathbb{0} \}$. But we can still use tuple types for the operators. Notice, however, a subtlety: the declarative rule suggests that we can find arguments e_i (whose types we call s_i) are subtypes of *each* of the arguments t_i of the function type found for the function e. This is written $\forall i. s_i \leq t_i$, which is equivalent to $\{ s_1, \ldots, s_n \} \leq \{ t_1, \ldots, t_n \}$ if and only if $\forall i. s_i \not\simeq \mathbb{0}$. Since we do not care much about applying functions to empty arguments, we add this condition to the algorithmic rule.

Write the DNF of an n-ary function type t as

$$t \simeq \bigvee_{i \in I^+} \Big(\bigwedge_{(\overline{u}, v) \in P_i} (u_1, .., u_n) \to v \wedge \bigwedge_{(\overline{u}', v') \in N_i} \neg (u'_1, .., u'_n) \to v' \Big)$$

where I^+ indexes only nonempty clauses.

Definition 4.4.9 (Domain operator (n-ary))**.** *For a function type t as above, the domain is the intersection (over nonempty clauses) of the unions of full input tuples from positive arrows:*

$$\mathtt{dom}(t) \overset{def}{=} \bigwedge_{i \in I^+} \bigvee_{(\overline{u}, v) \in P_i} \{ u_1, .., u_n \}$$

Boundary case: $\mathtt{dom}(\mathbb{0}) = \mathbb{1}$.

Definition 4.4.10 (Application result operator (n-ary))**.** *Given n argument types* $s_1, \ldots, s_n$ *, and such that* $\{ s_1, \ldots, s_n \} \leq \mathtt{dom}(t)$,

$$t \circ \overline{s} \overset{def}{=} \bigvee_{i \in I^+} \bigvee_{\substack{Q \subsetneq P_i \\ \{ s_1, \ldots, s_n \} \not\leq \bigvee_{(\overline{u}, v) \in Q} \{ u_1, \ldots, u_n \}}} \bigwedge_{(\overline{u}, v) \in P_i \setminus Q} v$$

Special case: if there exists an i such that $s_i \simeq \mathbb{0}$*, then* $t \circ (s_1, \ldots, s_n) \simeq \mathbb{0}$*. This makes sense as, then, the argument diverges thus, due to the strict semantics, the application diverges therefore it is typed as* $\mathbb{0}$.

In the same way we recalled for single-arity function operators (Theorem 3.3.6 and Theorem 3.3.7), we prove the soundness of these n-ary operators to justify their use for implementing the n-ary declarative rule for application.

Theorem 4.4.11 (Soundness of domain (n-ary))**.** *For all* $n \geq 1$*, for all static types* $t, s \in \mathscr{T}_{static}$ *and* $t_1, \ldots, t_n \in \mathscr{T}_{static}$*, if* $t \leq (t_1, .., t_n) \to s$ *then* $\{ t_1, .., t_n \} \leq \mathtt{dom}(t)$.

Proof. Write the DNF of t as $t \simeq \bigvee_{i\in I^+}\Big(\bigwedge_{(\overline{u},v)\in P_i}\overline{u}\to v \;\wedge\; \bigwedge_{(\overline{u}',v')\in N_i}\neg(\overline{u}'\to v')\Big)$, where I^+ indexes only nonempty clauses. From $t \le ((t_1,..,t_n)\to s)$ and monotonicity of $\vee$, we have for every $i\in I^+$:

$$\Big(\bigwedge_{(\overline{u},v)\in P_i}\overline{u}\to v \;\wedge \bigwedge_{(\overline{u}',v')\in N_i}\neg(\overline{u}'\to v')\Big) \le (t_1,..,t_n)\to s.$$

Fix such an i and consider only the positive arrows P_i. Apply Theorem 4.4.5 to the subtyping $\bigwedge_{(\overline{u},v)\in P_i}\overline{u}\to v \le \overline{a}\to s$. For any $k\in[1,n]$, choose the mapping that sends every arrow in P_i to position k: then $\iota^{-1}(k) = P_i$ and $\iota^{-1}(n{+}1) = \varnothing$. By the empty-intersection convention for codomains (the second disjunct in the theorem is false when $\iota^{-1}(n{+}1) = \varnothing$), the theorem forces the domain-side condition $t_k \le \bigvee_{(\overline{u},v)\in P_i} u_k$. Since k was arbitrary, we obtain $\forall k.\ t_k \le \bigvee_{(\overline{u},v)\in P_i} u_k$, which implies $\{\,t_1,\dots,t_n\,\} \le \bigvee_{(\overline{u},v)\in P_i}\{\,u_1,\dots,u_n\,\}$ (pointwise characterization of subtyping for closed n-tuples). Because $i\in I^+$ was arbitrary, we conclude $\{\,t_1,\dots,t_n\,\} \le \bigwedge_{i\in I^+}\bigvee_{(\overline{u},v)\in P_i}\{\,u_1,\dots,u_n\,\} = \mathtt{dom}(t)$. □

Theorem 4.4.12 (Soundness and optimality of application (n-ary))**.** *For all $n\ge 1$, for all $t\in\mathscr{T}_{static}$ and all $\overline{s}$ with $\{\,s_1,..,s_n\,\}\le \mathtt{dom}(t)$:*

$$\textit{(i)}\, t\le\overline{s}\to\ t\circ\overline{s} \quad \textit{and} \quad \textit{(ii) for all } r\in\mathscr{T}_{static},\ \textit{if } t\le\overline{s}\to r \textit{ then } t\circ\overline{s}\le r$$

Proof. If $\overline{s}\simeq\mathbb{0}$ the claim is immediate. Otherwise, write $t\simeq\bigvee_{i\in I^+} C_i$ with $C_i = \bigwedge_{(\overline{u},v)\in P_i}\overline{u}\to v \wedge \bigwedge_{(\overline{u}',v')\in N_i}\neg(\overline{u}'\to v')$, and set, for each i,

$$R_i(\overline{s}) \overset{\text{def}}{=} \bigvee_{\substack{Q\subsetneq P_i\\ \{\,s_1,\dots,s_n\,\}\not\le\bigvee_{(\overline{u},v)\in Q}\{\,u_1,\dots,u_n\,\}}} \bigwedge_{(\overline{u},v)\in P_i\setminus Q} v \qquad\text{so that}\qquad t\circ\overline{s} = \bigvee_{i\in I^+} R_i(\overline{s})$$

(i) Soundness. Fix $i\in I^+$. We prove $C_i\le\overline{s}\to R_i(\overline{s})$ using Theorem 4.4.5. Let $\iota: P_i\to[1,n{+}1]$ be arbitrary and set $Q_\iota \overset{\text{def}}{=} \bigcup_{k\le n}\iota^{-1}(k)$ (so $P_i\setminus Q_\iota = \iota^{-1}(n{+}1)$). Consider two cases:

- Case A: $\{\,s_1,\dots,s_n\,\}\not\le\bigvee_{(\overline{u},v)\in Q_\iota}\{\,u_1,\dots,u_n\,\}$. By definition of $R_i(\overline{s})$, the disjunct $\bigwedge_{(\overline{u},v)\in P_i\setminus Q_\iota} v = \bigwedge_{(\overline{u},v)\in\iota^{-1}(n+1)} v$ is contained in $R_i(\overline{s})$, hence the codomain clause $\bigwedge_{(\overline{u},v)\in\iota^{-1}(n+1)} v\le R_i(\overline{s})$ holds.
- Case B: $\{\,s_1,\dots,s_n\,\}\le\bigvee_{(\overline{u},v)\in Q_\iota}\{\,u_1,\dots,u_n\,\}$. We show the domain clause holds: assume, towards a contradiction, that for every $k\in\iota(P_i)\cap[1,n]$, $s_k\not\le\bigvee_{(\overline{u},v)\in\iota^{-1}(k)} u_k$. For each such k, pick $x_k\in[\![s_k\wedge\neg\bigvee_{\iota(i)=k}u_k]\!]$, and build $d\in[\![\{\,s_1,\dots,s_n\,\}]\!]$ with $d_k = x_k$ on those coordinates (arbitrary elsewhere). If $d\in\big[\!\big[\bigvee_{(\overline{u},v)\in Q_\iota}\{\,u_1,\dots,u_n\,\}\big]\!\big]$, there exists $j\in Q_\iota$ with $d\in\big[\!\big[\{\,u_1,\dots,u_n\,\}_j\big]\!\big]$, hence $d_{\iota(j)}\in\big[\!\big[u_{j,\iota(j)}\big]\!\big]$, contradicting the choice of $x_{\iota(j)}$. Thus $d\notin\bigvee_{Q_\iota}\overline{u}$, i.e. $\overline{s}\not\le\bigvee_{Q_\iota}\overline{u}$, contradiction. Hence there exists $k\in\iota(P_i)\cap[1,n]$ with $s_k\le\bigvee_{(\overline{u},v)\in\iota^{-1}(k)}u_k$, and the domain clause holds.

Since the theorem's condition holds for all ι, we obtain $C_i\le\overline{s}\to R_i(\overline{s})$. Joining over i yields $t\le\overline{s}\to\bigvee_i R_i(\overline{s}) = \overline{s}\to\ t\circ\overline{s}$.
(ii) Optimality. Assume $t\le\overline{s}\to r$, hence $C_i\le\overline{s}\to r$ for all i. Fix i and let $Q\subsetneq P_i$ with

$\{\overline{s}\} \not\leq \bigvee_Q \{\overline{u}\} w$. Choose a coordinate k_0 witnessing this failure, i.e. $s_{k_0} \not\leq \bigvee_{(\overline{u},v)\in Q} u_{k_0}$, and define ι_Q by mapping every element of Q to k_0 and every element of $P_i \setminus Q$ to $n+1$. Then the domain clause fails for ι_Q, so by the theorem the codomain clause must hold: $\bigwedge_{(\overline{u},v)\in P_i\setminus Q} v \leq r$. Taking the union over all such Q gives $R_i(\overline{s}) \leq r$; joining over i yields $t \circ \overline{s} \leq r$. □

4.5 Extending semantic subtyping: tuples

Tuples are a fundamental data structure in Elixir, serving critical roles in the language's idioms and implementation. They are pervasively employed in return value conventions (e.g., `{:ok, result}` or `{:error, reason}`) to indicate operation success or failure–a pattern that forms the backbone of Elixir's error handling strategy. Additionally, tuples serve as the underlying representation for Elixir's abstract syntax tree, which follows a LISP-like structure manipulated extensively by the macro system. Prior work by Frisch (2004) employed pairs as primitive constructs, subsequently using them to encode more complex structures such as lists. While we could adopt a similar encoding approach for Elixir tuples, we have instead opted for a direct native interpretation of tuples in our semantic domain.

Interpretation & subtyping To interpret the tuple types of Figure 4.4, we extend the interpretation domain $\mathscr{D}$ with native tuple constructs. For any $n \in \mathbb{N}$:

$$\mathscr{D} \ni d ::= \ldots \mid (d_1, d_2, \ldots, d_n)$$

The set-theoretic interpretation of n-ary tuple types is given by:

$$((d_1, d_2, \ldots, d_n) : \{t_1, t_2, \ldots, t_n\}) \Leftrightarrow \forall i \in [1,n].\,(d_i : t_i)$$
$$((d_1, d_2, \ldots, d_m) : \{t_1, t_2, \ldots, t_n, ..\}) \Leftrightarrow (m \geq n) \wedge \forall i \in [1,n].\,(d_i : t_i)$$

Castagna (2023b, Appendix D) describes in detail the intuition (which consists in successively eliminating the negations by distributing them over the positive tuples) and the formula to decide subtyping for fixed-arity (closed) tuples, showing that

Theorem 4.5.1 (Fixed-arity subtyping criterion)**.** *Let $n \in \mathbb{N}$ be the arity of the tuples, P a set of positive tuples, and N a set of negative tuples. Then,*

$$\bigwedge_{(s_1,\ldots,s_n)\in P} \{s_1, \ldots, s_n\} \wedge \bigwedge_{(t_1,\ldots,t_n)\in N} \neg\{t_1, \ldots, t_n\} \qquad \textit{is empty}$$

$$\textit{if and only if} \qquad \forall h : N \to [1,..,n].\, \exists i \in P. \left(\bigwedge_{(s_1,\ldots,s_n)\in P} s_i \leq \bigvee_{h(t_1,\ldots,t_n)=i} t_i \right)$$

This result comes as a direct corollary of our Lemma 4.4.4, ensuring the use of closed arity tuples in our type system. Now we need to handle open-arity tuples, written $\{t_1, \ldots, t_k, ..\}$, which denote tuples with at least k elements of types $t_1, \ldots, t_k$, possibly followed by any number of further elements of arbitrary type.

A closed positive k-tuple intersected with negative tuple constraints has the following form:

$$\{\, t_1, \dots, t_k \,\} \wedge \bigwedge_{u \in N} \neg u \tag{closed$_1$}$$

where u ranges over either closed tuples $\{\, t^u_1, \dots, t^u_{l_u} \,\}$ or open tuples $\{\, t^u_1, \dots, t^u_{l_u}\, \boldsymbol{,}\, .. \}$. We can immediately discard all closed negatives with $l_u \neq k$ and all open negatives with $l_u > k$, as their negation contains $\overline{t}$. For each remaining open negative with $l_u \le k$, we close it at arity k by padding with $\mathbb{1}$; we write the result as $u^{\downarrow k}$ (e.g. for arity 3, $u = \{\, \texttt{int}\, \boldsymbol{,}\, .. \}$ becomes $u^{\downarrow 3} = \{\, \texttt{int}, \mathbb{1}, \mathbb{1} \,\}$). This transformation preserves the difference with the closed positive k-tuple, hence (closed$_1$) is equivalent to

$$\{\, t_1, \dots, t_k \,\} \wedge \bigwedge_{u \in N} \neg(u^{\downarrow k}) \tag{closed$_2$}$$

which is already decided by the fixed-arity criterion.

Then, we turn to the case when the positive intersection turns to a single open tuple:

$$\{\, t_1, \dots, t_k\, \boldsymbol{,}\, .. \} \wedge \bigwedge_{u \in N} \neg u \tag{open$_1$}$$

This situation can also be reduced to a finite number of fixed-arity checks. Let $L \overset{\text{def}}{=} \max\Big(k,\ \max\{\, l_u \mid u \in N \text{ is open}\}\Big)$. Only arities $m \in \{k, \dots, L\}$ are relevant: beyond L, the relevant negatives and the positive side stabilize (positive coordinates $i > k$ become $\mathbb{1}$, as well as all negative open coordinates by definition of L). The next theorem states this finite check precisely.

Theorem 4.5.2 (Emptiness for one open positive tuple — finite check)**.** *Let $p = \{\, s_1, \dots, s_k\, \boldsymbol{,}\, .. \}$ be a single positive tuple and let N be a finite set of negative tuples (each either $\{\, t^u_1, \dots, t^u_{l_u} \,\}$ or $\{\, t^u_1, \dots, t^u_{l_u}\, \boldsymbol{,}\, .. \}$). For any arity m, let $\tau_m \overset{\text{def}}{=} \overbrace{\{\mathbb{1}, \dots, \mathbb{1}\,\}}^{m}$ and, for any tuple x, write $x^{\downarrow m} \overset{\text{def}}{=} x \wedge \tau_m$ (its closure at arity m).*
Define

$$L \overset{\text{def}}{=} \max\Big(k,\ \max\{\, l_u \mid u \in N \text{ is open}\}\Big)$$

Then the intersection

$$\{\, s_1, \dots, s_k\, \boldsymbol{,}\, .. \} \wedge \bigwedge_{u \in N} \neg u$$

is empty iff both hold:

(A) for every $m \in [k..L-1]$, the fixed-arity test of Theorem 4.5.1 applied to $p^{\downarrow m}$ and $n^{\downarrow m}$ decides emptiness;

(B) the fixed-arity test of Theorem 4.5.1 at arity L says empty, i.e., applied to $p^{\downarrow L}$ together with $u^{\downarrow L}$ for u open (closed negatives ignored).

Proof. *(Only-if).* If the mixed intersection is empty, then for each $m \in [k..L]$ its restriction to arity m is empty; intersecting all tuples with τ_m yields the closed instance $(p^{\downarrow m}; \{n^{\downarrow m}\})$, so the fixed-arity test at m says empty (A). Removing closed negatives gives a relaxation; intersecting with τ_L provides the opens-only instance at arity L, which is empty (B).
(If). Assume (A) and (B). Suppose, for contradiction, there is a witnessing tuple v of length $m \geq k$.

- If $m \in [k..L-1]$, then intersecting with τ_m produces a non-empty fixed-arity instance at arity m witnessed by v, contradicting (A).
- If $m \geq L$. Then the fixed-arity test at arity m is nonempty, witnessed by v. We can consider each open negative constraint that v satisfies: by definition of L, it only has $\mathbb{1}$ on the coordinates beyond L. Thus, truncating v to size L still satisfies all of them. Relaxing the closed negative conditions is fine as well (v keeps satisfying a less strict condition). Thus, from v we obtain a witness that contradicts (B) which tested at arity L with only open negatives.

Therefore the mixed intersection is empty. □

Note that, while checking the emptiness of tuples of arity $m \in [k..(L-1)]$, a special case arises that can short-circuit the rest: if we know that there are no closed negatives of arity m, then we can conclude immediately that the tuple is empty, since those open negatives will cancel out the rest as well (e.g. the emptiness of $\{\texttt{int}, \texttt{int}\} \setminus \{\texttt{int}, ..\} \vee \{\texttt{int}, \texttt{int}, \texttt{int}, ..\}$ gets decided at arity 1; there is no need to check the rest).

Example (*Application.*)

Consider

$$p = \{\texttt{int}, ..\} \qquad (k=1) \quad \text{and} \quad N = \{\{\mathbb{1}\}, \{\texttt{int}, \mathbb{1}, ..\}, \{\mathbb{1}, \texttt{bool}, ..\}, \{\mathbb{1}, \mathbb{1}\}\}.$$

The open negatives have lengths 2, hence $L = \max\big(k, \max\{2,2\}\big) = 2$.
Step 1: check at arity $m = 1$. It is $\{\texttt{int}\} \wedge \neg\{\mathbb{1}\}$, which is empty. Step 2: we continue with the opens-only fixed-arity test at arity $L = 2$. We close all tuples at arity 2, discarding the closed negative $\{\mathbb{1}, \mathbb{1}\}$, to obtain the problem

$$\underbrace{\{\texttt{int}, \mathbb{1}\}}_{p^{\downarrow 2}} \wedge \neg \underbrace{\{\texttt{int}, \mathbb{1}\}}_{\text{first negative}} \wedge \neg \underbrace{\{\mathbb{1}, \texttt{bool}\}}_{\text{second negative}},$$

which is empty since it contains a type and its negation simultaneously: $\{\texttt{int}, \mathbb{1}\} \wedge \neg\{\texttt{int}, \mathbb{1}\}$.

Eliminating tuple negatives: four cases To convert a tuple type into a union of positive tuples (i.e., eliminate negations), we consider each positive–negative pair and apply one of four rules depending on whether each tuple is closed or open. Recall that $p \setminus n \simeq p \wedge \neg n$.

Case 1: Closed/Closed. Consider $\{t_1, \ldots, t_k\} \setminus \{t'_1, \ldots, t'_\ell\}$.

- If $k \neq \ell$: the negative is irrelevant (different arities are disjoint), so the result is just $\{\, t_1, \ldots, t_k \,\}$.
- If $k = \ell$: apply the *coordinatewise decomposition*—the difference is a union where exactly one coordinate differs:

$$\{\, t_1, \ldots, t_k \,\} \smallsetminus \{\, t'_1, \ldots, t'_k \,\} \simeq \bigvee_{j=1}^{k} \{\, t_1 \wedge t'_1, \ldots, t_{j-1} \wedge t'_{j-1},\ t_j \smallsetminus t'_j,\ t_{j+1}, \ldots, t_k \,\} \qquad (\text{elim}_1)$$

Case 2: Closed/Open. Consider $\{\, t_1, \ldots, t_k \,\} \smallsetminus \{\, u_1, \ldots, u_\ell \,,..\}$.

- If $\ell > k$: the negative is irrelevant (requires more elements than the positive has).
- If $\ell \leq k$: close the open negative at arity k by padding with $\mathbb{1}$, then apply Case 1:

$$\{\, t_1, \ldots, t_k \,\} \smallsetminus \{\, u_1, \ldots, u_\ell \,,..\} \simeq \{\, t_1, \ldots, t_k \,\} \smallsetminus \{\, u_1, \ldots, u_\ell, \overbrace{\mathbb{1}, \ldots, \mathbb{1}}^{k-\ell} \,\} \qquad (\text{elim}_2)$$

Case 3: Open/Closed. Consider $\{\, t_1, \ldots, t_k \,,..\} \smallsetminus \{\, u_1, \ldots, u_\ell \,\}$.

- If $\ell < k$: the negative is irrelevant (the positive requires at least k elements, but the negative has fewer).
- If $\ell \geq k$: split by arity. The negative only affects tuples of exactly arity ℓ:

$$\begin{aligned}
\{\, t_1, \ldots, t_k \,,..\} \smallsetminus \{\, u_1, \ldots, u_\ell \,\} \simeq\ & \underbrace{\bigvee_{m=k}^{\ell-1} \{\, t_1, \ldots, t_k, \overbrace{\mathbb{1}, \ldots, \mathbb{1}}^{m-k} \,\}}_{\text{arities } k \text{ to } \ell-1\text{: negative irrelevant}} \\
& \vee \underbrace{\left(\{\, t_1, \ldots, t_k, \overbrace{\mathbb{1}, \ldots, \mathbb{1}}^{\ell-k} \,\} \smallsetminus \{\, u_1, \ldots, u_\ell \,\} \right)}_{\text{arity } \ell\text{: apply Case 1}} \qquad (\text{elim}_3) \\
& \vee \underbrace{\{\, t_1, \ldots, t_k, \overbrace{\mathbb{1}, \ldots, \mathbb{1}}^{\ell+1-k} \,,..\}}_{\text{arities} \geq \ell+1\text{: negative irrelevant}}
\end{aligned}$$

Case 4: Open/Open. Consider $\{\, t_1, \ldots, t_k \,,..\} \smallsetminus \{\, u_1, \ldots, u_\ell \,,..\}$.

Let $L \stackrel{\text{def}}{=} \max(k, \ell)$. For any arity m, define the *arity-m closures*:

$$p_m \stackrel{\text{def}}{=} \{\, t_1, \ldots, t_k, \overbrace{\mathbb{1}, \ldots, \mathbb{1}}^{m-k} \,\} \qquad \text{and} \qquad n_m \stackrel{\text{def}}{=} \{\, u_1, \ldots, u_\ell, \overbrace{\mathbb{1}, \ldots, \mathbb{1}}^{m-\ell} \,\}$$

The key insight is that for all arities $m \geq L$, the difference $p_m \smallsetminus n_m$ has the same structure (only padded with more $\mathbb{1}$s). Thus we can decompose into:

- **Arities below ℓ** (if $k < \ell$): the negative is irrelevant since it requires at least ℓ elements.

$$\bigvee_{m=k}^{\ell-1} p_m$$

- **Arities** $\geq L$: apply the coordinatewise decomposition at arity L, then lift to an open tuple. Define $s_i \overset{\text{def}}{=} \begin{cases} t_i & \text{if } i \leq k \\ \mathbb{1} & \text{if } k < i \leq L \end{cases}$ and $v_i \overset{\text{def}}{=} \begin{cases} u_i & \text{if } i \leq \ell \\ \mathbb{1} & \text{if } \ell < i \leq L \end{cases}$.

$$\{\, p_L \smallsetminus n_L\,,\,..\} \simeq \bigvee_{j=1}^{L} \{\, s_1 \wedge v_1, \ldots, s_{j-1} \wedge v_{j-1},\ s_j \smallsetminus v_j,\ s_{j+1}, \ldots, s_L\,,\,..\}$$

Combining these:

$$\{\, t_1, \ldots, t_k\,,\,..\} \smallsetminus \{\, u_1, \ldots, u_\ell\,,\,..\} \simeq \bigvee_{m=k}^{\ell-1} p_m \vee \{\, p_L \smallsetminus n_L\,,\,..\} \tag{elim$_4$}$$

Theorem 4.5.3 (Decomposition for tuples)**.** *For any $t \in \mathscr{T}_{static}$ with $t \leq$ `tuple`, there exist finite sets $\mathscr{C}$ of closed tuples $\{\, s_1, \ldots, s_n\,\}$ and $\mathscr{O}$ of open tuples $\{\, r_1, \ldots, r_k\,,\,..\}$ such that*

$$t \simeq \bigvee_{\{\,\overline{s}\,\} \in \mathscr{C}} \{\,\overline{s}\,\} \vee \bigvee_{\{\,\overline{r}\,,\,..\} \in \mathscr{O}} \{\,\overline{r}\,,\,..\}.$$

Proof sketch. Put $t \wedge$ `tuple` in DNF. After compressing the positive intersections into single tuples, use the four elimination cases for tuple negations described above (closed/-closed (elim$_1$), closed/open (elim$_2$), open/closed (elim$_3$), open/open (elim$_4$)). □

The existence of such a decomposition makes it easy to define projection for tuples, by collecting the union of the types at a given index (such an operator was defined in Definition 3.3.2), and to prove that it is sound in the same way as Theorem 3.3.3. It also justifies the possibility of representing tuple types as a union of closed/open tuples for implementation, detailed in Chapter 12 Section 12.3.

Conclusion

We defined Core Elixir with a small-step semantics that makes failure states explicit and a static type system based on set-theoretic types and interfaces, supporting simple type tests in pattern matching, open tuples, and multi-arity functions. We proved progress and preservation, delineating programs that use no ω-rules (no runtime type errors) from those that may fail only due to ω-marked operations (e.g., out-of-range tuple projections), and we extended semantic subtyping to genuine multi-arity functions with decision procedures for subtyping and typed application, and closed/open tuples. These foundations align with Elixir practice and prepare the next chapter, where we introduce the dynamic type ? and lift the static system to a gradual setting.

CHAPTER 5

SAFE ERASURE GRADUAL TYPING

"Designing a gradually-typed language means much more than throwing in dynamic checks and making sure that programs do not get stuck."

Ronald Garcia & Éric Tanter,
Gradual Typing as if Types Mattered
(WGT 2020).

There is an ambiguity, when designing a gradual type system, as to what the finality of such a system: is the end-goal to statically type-check all programs (in which case, we might consider a gradual system as an "unfinished" static system, waiting for us to devise the perfect theory and its magical type inference), or is there something more to it?

We do not decide this question here. Instead, we take a pragmatic approach: what is the consequence of trying to dip set-theoretic function types into an existing language (especially an already popular one like Elixir, with non-typed safety mechanisms already in place, such as the use of defensive guards)? Our realization is that of a weakness of the semantics of set-theoretic function types: these describe functions with a defined behaviour on a given domain type, and a completely undefined behaviour outside of it. Usually, in gradual typing, this unknown behaviour is controlled by adding proxies around functions: there is no need to care about what a function does outside its domain if it errors before that. This approach is ill-suited for us, one reasing being that we do not wish to modify the runtime behaviour of the language–we want to take the path of *erasure* (along with Typescript). This issue with modifying the runtime, besides not being appealing to all programmers (such as those who do not wish to use typing in their dynamic language), is that wrapping every function with runtime-checks[1] is a heavy transformation. And if we did go this way, we would realize, in Elixir, that we have just double the BEAM's work: indeed, there are only a certain amount of checks baked into the language. It is already possible for the programmer to enforce some type checks on their inputs/outputs. Our issue then, when designing a type erasure system, is *how to lift this information into the types*. We do not want to perform more *enforcing* of types, but more *inferring* of the behaviour of programs. In particular, we want to be able to infer (and encode) in a type what happens when a function is applied outside of its domain. The notion of *strong function types* is exactly that: it requires an auxiliary system to infer the behaviour of functions, and the properties of this new type can then be used to refine the type of dynamic programs.

The notion of strong function type does not make sense in a system where functions cannot be applied outside of their domain. But as long as the dynamic exists in a system, strong functions (or any type that captures "undefined behaviour") will prove useful in controlling them.

Chapter Roadmap

- **Section 5.1 (Rules)**: introduces the dynamic type $?$, the *safe-erasure gradual typing* discipline, and its typing rules (Figure 5.2) that extend the static system $\vdash_s$ (Figure 4.4). It also explains strong function types and their introduction via an auxiliary weak judgment (Figure 5.3).
- **Section 5.2 (Soundness)**: establishes the metatheory for the gradual and weak judgments, including progress, preservation, and a Gradual Soundness theorem, and clarifies the relation with the static subsystem.
- **Section 5.3 (Strong Semantics)**: gives a set-theoretic interpretation of strong arrows and derives decision procedures for subtyping with strong/weak arrows. See also Subsection 5.3.3 (Specialization), which sketches strict-domain and strict-codomain constructors and relates them to success types and strong arrows.
- **Section 5.4 (Design Limitations)**: examines how strong functions propagate static information across dynamic boundaries, documents limitations, and discusses refinement patterns and failure modes.

5.1 Gradual Typing Rules

Throughout this chapter we use t and τ interchangeably to denote gradual types, defined in Figure 5.1 as the types of Figure 4.1 extended with the dynamic type $?$. In particular, although Chapter 4 wrote static types with t, from here on we do not insist on this distinction. Even more: the typing rules of Chapter 4 carry over unchanged. The type discipline defines three nested type systems of increasing permissiveness. We write judgments $\Gamma \vdash_s e : \tau$ for static typing (presented

[1]As is the case, for instance, of Reticulated Python, Sorbet (by Stripe), Hack (HHVM) (by Facebook) which can enforce parameters and return hints, PHP 7+ with the strict_types directive, Typed Racket, etc.

Base types	b	$::=$	`int` | `bool` | `atom` | `fun` | `tuple`
Types	t	$::=$	$b \mid c \mid ? \mid t \to t \mid \{\overline{t}\} \mid \{\overline{t}, ..\} \mid t \vee t \mid \neg t$

Figure 5.1: Gradual Types Syntax

in Chapter 4) and we now introduce judgements $\Gamma \vdash_{\mathsf{g}} e : \tau$ for gradual typing, and $\Gamma \vdash_{\mathsf{w}} e \,\mathbin{⨟}\, \tau$ for weak typing. The static system guarantees that well-typed programs do not "go wrong" (except in one case discussed below regarding tuple indexing) and predicts the type of the result. The gradual system and the weak system that we now present extend the static rules to more flexibly accommodate the dynamic type ?. The weak system is the most permissive one and captures the run-time type-soundness of the BEAM virtual machine, alas deducing highly imprecise (i.e., dynamic) types.

Crucially, the static typing rules and metatheory from Chapter 4 carry over unchanged. The proofs are parametric in the subtyping relation and do not depend on the particular shape of types. Since gradual subtyping is defined as a lifting of static subtyping (see Section 3.2.4 and Definitions 3.1.7, 3.2.7), the Chapter 4 safety results apply verbatim when instantiated to static types. Indeed, the static system $\vdash_{\mathsf{s}}$ can still type expressions that mention '?' as the subtyping relation carries over to gradual types; for example, applying a function expecting '?' to a '?' argument uses (app). Otherwise the checker switches to the gradual rules in Figure 5.2; the auxiliary judgment for strong arrows (Figure 5.3) supports this mode.

We handle the dynamic type '?' by a technique we dubbed *safe erasure gradual typing*, which implies the use of additional rules (Figure 5.2) and an auxiliary system to infer *strong function types* (Figure 5.3), which are key aspects of our approach.

The use of consistent subtyping ($\lessapprox$, Definition 3.2.9) means that the type-checker has failed to type the expression with the static rules $\vdash_{\mathsf{s}}$, and switched to a *gradual mode* of operation, which is described by the rules in Figure 5.2. Figure 5.2 presents only the rules that are specific to the gradual type system, but for every "static" rule in Figure 4.4 there exists the same rule in the gradual system where each $\vdash_{\mathsf{s}}$ is changed into $\vdash_{\mathsf{g}}$.[2] When operating in gradual mode, the type-checker works differently, since the goal is to check operations like application or projection *could* succeed at runtime.

Remark

For the purpose of simplifying the presentation, we use single-arity function types for this chapter. The n-arity typing rules work similarly to the single-arity ones, and the real difficulty of n-arity functions resides in extending subtyping and computing type operators (which was treated in the previous Chapter 4). Lifting those (subtyping and operators) to the gradual setting is straightforward, as argued in Section 3.2.

Tuple projection Rule ($\text{proj}_?$) checks that types can materialize into correct ones (i.e., `int` for the index, and a tuple type for the tuple), in which case all we can deduce for the projection is the

[2]In practice, there is a unique deduction system, and the distinction between $\vdash_{\mathsf{s}}$ and $\vdash_{\mathsf{g}}$ is only for the sake of the presentation.

$$(\text{app}_?)\ \frac{\Gamma \vdash_{\text{g}} e_1 : t_1 \quad \Gamma \vdash_{\text{g}} e_2 : t_2}{\Gamma \vdash_{\text{g}} e_1(e_2) : ?}\ t_1 \lesseqqgtr (t_2 \to \mathbb{1})$$

$$(\text{app}_\star)\ \frac{\Gamma \vdash_{\text{g}} e_1 : (t_1 \to t)^\star \quad \Gamma \vdash_{\text{g}} e_2 : t_2}{\Gamma \vdash_{\text{g}} e_1(e_2) : ? \wedge t}\ t_2 \lesseqqgtr t_1$$

$$(\text{proj}_?)\ \frac{\Gamma \vdash_{\text{g}} e_1 : t_1 \quad \Gamma \vdash_{\text{g}} e_2 : t_2}{\Gamma \vdash_{\text{g}} \pi_{e_2} e_1 : ?}\ \begin{cases} t_1 \lesseqqgtr \texttt{tuple} \\ t_2 \lesseqqgtr \texttt{int} \end{cases}$$

$$(\text{proj}_\star)\ \frac{\Gamma \vdash_{\text{g}} e : \{ t_0, \ldots, t_n \} \quad \Gamma \vdash_{\text{g}} e' : t}{\Gamma \vdash_{\text{g}} \pi_{e'} e : ? \wedge \bigvee_{i \leq n} t_i}\ t \lesseqqgtr \texttt{int}$$

$$(\text{case}_\star)\ \frac{\Gamma \vdash_{\text{g}} e : t \qquad \forall i \in I\ \big((t \wedge \rho_i) \smallsetminus (\bigvee_{j<i} \rho_j) \not\leq \mathbb{0} \Rightarrow \Gamma \vdash_{\text{g}} e_i : t'\big)}{\Gamma \vdash_{\text{g}} \texttt{case}\ e\big(\rho_i \to e_i\big)_{i \in I} : ? \wedge t'}\ (t \lesseqqgtr \vee_{i \in I} \rho_i)$$

$$(\text{plus}_\star)\ \frac{\Gamma \vdash_{\text{g}} e_1 : t_1 \quad \Gamma \vdash_{\text{g}} e_2 : t_2}{\Gamma \vdash_{\text{g}} e_1 + e_2 : \texttt{int} \wedge ?}\ \begin{cases} t_1 \lesseqqgtr \texttt{int} \\ t_2 \lesseqqgtr \texttt{int} \end{cases}$$

$$(\lambda_\star)\ \frac{\Gamma \vdash_{\text{g}} \lambda^{\mathbb{I}} x.e : t_1 \to t_2 \qquad \Gamma, x : ? \vdash_{\text{w}} e \between t_2 \wedge ?}{\Gamma \vdash_{\text{g}} \lambda^{\mathbb{I}} x.e : (t_1 \to t_2)^\star}$$

Figure 5.2: Gradual Typing Rules (Additional)

type '?'. However, in rule (proj$_\star$), if we know that the expression has type $\{\tau_1, \ldots, \tau_n\}$ and that the index materializes into an integer, then it can be typed with '?' intersected with the union of the tuple contents: $? \wedge \bigvee_{i=1..n} \tau_i$.

Function application Rule (app$_?$) checks whether the (gradual) types of the argument t_2 and of the function t_1 can materialize into two static types such that the materialized argument type is a subtype of the materialized function's domain, i.e. $t_1 \lesseqqgtr t_2 \to \mathbb{1}$. When this condition holds, the application is assigned type '?'. While this type is imprecise—essentially indicating the application may yield results of any type—it represents the only sound deduction in this context, albeit less precise than what rule (app) would provide if applicable. For instance, it is unsound to assign type `int` to the application of the function $(\lambda^{\{\texttt{int} \to \texttt{int}\}}(x).x)$ to a dynamic argument. Although this function returns integers when given integer inputs (justifying type `int` → `int`), it would return a Boolean result if passed a Boolean argument at runtime.

That constitutes a significant limitation of adding '?' to a type system: '?' tends to escape and contaminate the entire system, making type deductions less precise and, thus, less useful. Generally, there is no way to statically determine a concrete return type when a function is applied to a dynamic value, unless runtime type checks are inserted to enforce the function's signature—the approach taken by current sound gradual typing systems. However, this solution is unavailable to us: a core design principle of Castagna et al. (2024a) requires the type system integration to affect only static type-checking without modifying Elixir's runtime behaviour.

Fortunately, Elixir's VM already performs type checks both implicitly through strong operations such as '+', and explicitly via programmer-defined guards. We leverage both mechanisms in the type system by introducing a new notion of functional type: strong arrows, denoted by a $\star$,

$$\textbf{Types} \qquad t ::= \cdots \mid (t \to t)^\star$$

Strong function types A strong arrow denotes functions that work as usual on their domain, but when applied to an argument outside their domain they either *fail on an explicit runtime type check*, or *return a value of their codomain type*, or diverge. In Section 2 (line 144) we gave the example of the function `second_strong`, which in our calculus can be encoded as

$$\lambda^{\{\{\mathbb{1},\texttt{int},..\}\to\texttt{int}\}}(x).\,\texttt{case}\,x\,(\,\{\mathbb{1},\texttt{int}\}\to\pi_1\,x\,)$$

It is a function that returns an integer if its argument is a tuple whose second element is an integer, and otherwise "fails", that is, it reduces to $\omega_{\text{CaseEscape}}$. The notion of strong arrow is not relevant to a standard static type system, but to a gradual type system where uncertainty is both a problem (modules are not annotated, and the type-checker must infer types) and a feature (some programming idioms are inherently dynamic). The purpose of a strong arrow is then to guarantee that a function, when applied to a dynamic argument, will return a value of a specific type, as seen in rule (app$_\star$). This rule is only used when static type-checking fails, and it has to preserve the flexibility of the typing, as other functions would then struggle to type-check a fully static return type. Thus, rules annotated by $\star$ introduce '?' in their conclusion, in the form of an intersection. This property, which was described in §2 of the introduction, is called *dynamic propagation*. Alongside '?', a static type is propagated to be used by the type-checker to detect type incompatibilities. If an argument of type $(? \wedge \texttt{int})$ is used where a Boolean is expected, a static type error will be raised. And if an argument of type $? \wedge (\texttt{int} \vee \texttt{bool})$ is used where a Boolean is expected, the type-checker in gradual mode will allow it, by considering that the argument could become a Boolean at runtime (corresponding to the materialization of ? into `bool`).

Inference of strong function types. The introduction rule for strong arrows ($\lambda_\star$) requires an auxiliary type-checking judgment $\Gamma \vdash_{\text{w}} e ⨾ t$ defined in Figure 5.3. This type system models the type checks performed by the Elixir runtime. Indeed, if $\Gamma \vdash_{\text{w}} e ⨾ t$, then e either diverges, or fails on a runtime error, that we know of, or evaluates to a value of type t. Therefore, this system must accept expressions such as $\texttt{case}\ 42\ (\texttt{bool} \to 5)$ which directly reduces to $\omega_{\text{CaseEscape}}$. This system is similar to the declarative one of Figure 4.4, but with additional "escape hatches" that make strong operations permissible no matter the type of their operands. For instance, since '+' is strong (the Elixir VM checks at runtime that the operands of an addition are both integers), then rule $(+^\circ)$ only asks that its terms are well-typed. If the addition does not fail, then it returns an integer (typed as $\texttt{int} \wedge ?$ for dynamic propagation). Other such operations are tuple projection, pattern-matching, and also function application. Using this system, we infer strong function types with rule ($\lambda_\star$) of the gradual type system $\vdash_{\text{g}}$; if a function $\lambda^{\{t_1\to t_2\}}(x).\,e$ has type $t_1 \to t_2$, then this type is strong if, with x of type ?, the body e can be checked to have type t_2 (actually $t_2 \wedge ?$ for dynamic propagation) using the rules of Figure 5.3. The rules explicitly allow expressions

$$(\text{cst}^\circ)\ \frac{}{\Gamma \vdash_{\mathsf{w}} c : c \wedge ?} \qquad (\text{var}^\circ)\ \frac{\Gamma(x) = t}{\Gamma \vdash_{\mathsf{w}} x : t \wedge ?} \qquad (\text{tuple}^\circ)\ \frac{\forall i = 1..n.\ (\Gamma \vdash_{\mathsf{w}} e_i : t_i)}{\Gamma \vdash_{\mathsf{w}} \{ e_1, .., e_n \} : \{ t_1, .., t_n \}}$$

$$(\lambda^\circ)\ \frac{\forall (t_i \to s_i) \in \mathbb{I}. (\Gamma, x : t_i \vdash_{\mathsf{w}} e : s_i) \quad \Gamma, x : ? \vdash_{\mathsf{w}} e : \mathbb{1}}{\Gamma \vdash_{\mathsf{w}} \lambda^{\mathbb{I}} x.e : \bigwedge_{i \in I} (t_i \to s_i)}$$

$$(\lambda^\circ_\star)\ \frac{\Gamma \vdash_{\mathsf{w}} \lambda^{\mathbb{I}} x.e : t_1 \to t_2 \quad \Gamma, x : ? \vdash_{\mathsf{w}} e : t_2 \wedge ?}{\Gamma \vdash_{\mathsf{w}} \lambda^{\mathbb{I}} x.e : (t_1 \to t_2)^\star} \qquad (\lambda^\circ_?)\ \frac{\Gamma \vdash_{\mathsf{w}} \lambda^{\mathbb{I}} x.e : t}{\Gamma \vdash_{\mathsf{w}} \lambda^{\mathbb{I}} x.e : ?}$$

$$(\text{app}^\circ)\ \frac{\Gamma \vdash_{\mathsf{w}} e_1 : t_1 \to t_2 \quad \Gamma \vdash_{\mathsf{w}} e_2 : t_1}{\Gamma \vdash_{\mathsf{w}} e_1(e_2) : t_2} \qquad (\text{app}^\circ_\star)\ \frac{\Gamma \vdash_{\mathsf{w}} e_1 : (t_1 \to t_2)^\star \quad \Gamma \vdash_{\mathsf{w}} e_2 : \mathbb{1}}{\Gamma \vdash_{\mathsf{w}} e_1(e_2) : t_2 \wedge ?}$$

$$(\text{app}^\circ_?)\ \frac{\Gamma \vdash_{\mathsf{w}} e_1 : \mathbb{1} \quad \Gamma \vdash_{\mathsf{w}} e_2 : \mathbb{1}}{\Gamma \vdash_{\mathsf{w}} e_1(e_2) : ?}$$

$$(\text{proj}^\circ)\ \frac{\Gamma \vdash_{\mathsf{w}} e_2 : \bigvee_{i \in K} i \quad \Gamma \vdash_{\mathsf{w}} e_1 : \{ t_0, \dots, t_n, .. \}}{\Gamma \vdash_{\mathsf{w}} \pi_{e_2} e_1 : \bigvee_{i \in K} t_i}\ K \subseteq [0, n]$$

$$(\text{proj}^\circ_{\mathtt{int}})\ \frac{\Gamma \vdash_{\mathsf{w}} e_2 : \mathbb{1} \quad \Gamma \vdash_{\mathsf{w}} e_1 : \{ t_0, \dots, t_n \}}{\Gamma \vdash_{\mathsf{w}} \pi_{e_2} e_1 : \bigvee_{i \leq n} t_i} \qquad (\text{proj}^\circ_{\mathbb{1}})\ \frac{\Gamma \vdash_{\mathsf{w}} e_2 : \mathbb{1} \quad \Gamma \vdash_{\mathsf{w}} e_1 : \mathbb{1}}{\Gamma \vdash_{\mathsf{w}} \pi_{e_2} e_1 : ?}$$

$$(\text{case}^\circ)\ \frac{\Gamma \vdash_{\mathsf{w}} e : t \quad \forall i \in I. \left(t \wedge \rho_i \not\leq \mathbb{0} \Rightarrow \Gamma \vdash_{\mathsf{w}} e_i : s \right)}{\Gamma \vdash_{\mathsf{w}} \mathtt{case}\ e \left(\rho_i \to e_i \right)_{i \in I} : s} \quad (\text{and}^\circ)\ \frac{\Gamma \vdash_{\mathsf{w}} e : t_1 \quad \Gamma \vdash_{\mathsf{w}} e : t_2}{\Gamma \vdash_{\mathsf{w}} e : t_1 \wedge t_2}$$

$$(\text{add}^\circ)\ \frac{\Gamma \vdash_{\mathsf{w}} e_1 : \mathbb{1} \quad \Gamma \vdash_{\mathsf{w}} e_2 : \mathbb{1}}{\Gamma \vdash_{\mathsf{w}} e_1 + e_2 : \mathtt{int} \wedge ?} \qquad (\text{sub}^\circ)\ \frac{\Gamma \vdash_{\mathsf{w}} e : t_1 \quad t_1 \leq t_2}{\Gamma \vdash_{\mathsf{w}} e : t_2}$$

Figure 5.3: Weak Type System

that are known to fail at compile time. As another example, consider rule (case°) in Figure 5.3, which does not have an exhaustiveness condition because an escaping expression will not return a value but fail at runtime. Note that, in this rule, if no pattern matches, then any type can be chosen for the result and, thus, the rule $(\text{case}^\circ_{\mathbb{0}})$

$$(\text{case}^\circ_{\mathbb{0}})\ \frac{\Gamma \vdash_{\mathsf{w}} e : t \qquad t \wedge \bigvee_i \rho_i \simeq \mathbb{0}}{\Gamma \vdash_{\mathsf{w}} \mathtt{case}\ e \left(\rho_i \to e_i \right)_{i \in I} : \mathbb{0}}$$

which types a case-expression that always fails, is admissible. Which is why the expression `case` 42 (`bool` → 5) we hinted above is well-typed with type $\mathbb{0}$.

To summarize, we have presented in Figures 4.4, 5.2, and 5.3 three declarative systems that work together to model different typing disciplines over programs: Figure 4.4 presents a fully static discipline, where subtyping is used to check compatibility between types, and function type annotations are enforced. This is what is expected of a fully annotated program. Figure 5.2 only comes into play when the previous type-checking fails. It uses a more relaxed relation on types, consistent subtyping, to check programs whose types are gradual (i.e., where '?' occurs in them). Figure 5.3 serves as an auxiliary system to infer strong function types, but its elaboration mirrors the semantics of the BEAM VM: every syntactically correct program typechecks with type $\mathbb{1}$, since it either diverges, returns a value (necessarily of type $\mathbb{1}$), or fails due to BEAM checks. However, some programs have more precise types which are passed around like information to be used later.

5.2 Gradual Soundness

Our calculus is based on the BEAM, which is already 'type-safe' in the sense that it dynamically checks the type of value before doing any operation. This is what produces named errors ω_p in the operational semantics described Figure 4.2. But this is not what type-safe means to the programmer: their need is to forbid such crashes, which is exactly the point of the 'static type safety' $\vdash_s$ result we presented in the previous chapter (Thm. 4.3.7). What if we prevent all named errors, but for one kind: the errors that happen when taking an out of bound element on a tuple. This is still a *safety* result, which we presented in Theorem 4.3.8 (ω-type safety). These two results, as we mentioned earlier, carry over to the gradual world (where '?' can appear in types): if a program can be typed only by the rules of system $\vdash_s$, then it enjoys *static type safety*. This presents a clear prize and motivation for programmers, to annotate their programs with well-written types.

The characterization of typing a program with the gradual system $\vdash_g$ cannot be the same, as allowing *unsafe* operations in the presence of '?' means that all named errors ω_p can be produced by an expression. Thus, its role is to establish type-based reasoning that works over the entire code-base, both static and dynamic fragments. We will achieve this *gradual soundness* result (Thm. 5.2.15) by sending over our typing judgements $\Gamma \vdash_g e : t$ to the weak system $\Gamma \vdash_w e ⨾ t$ which is permissive but has the crucial property that, if an expression is typed $\Gamma \vdash_w e ⨾ t$, then e can diverge, crash on a runtime error ω_p, or return a value v with $\varnothing \vdash v ⨾ t$, meaning that the shape of v is defined by t. For instance, if the type t deduced for a given expression is '`int`' then any value the expression reduces to is necessarily an integer; if it is '?', then the value can be any value; if it is '$? \to ?$', then the value will be a λ-abstraction. Note that, while the first two theorems ensure that well-typed expressions produce only well-typed values of the same type, the third theorem ensures only that any value produced by the expression will satisfy $v ⨾ t$, that is, that it will have the expected shape: because of weak-reduction, a gradually-typed expression can return a λ-abstraction whose body is not well-typed—though, it will be (type) safe in every context. Thus, in particular, if the expression is of type $t_1 \to t_2$, then Theorem 5.2.15 ensures that it can only return values that are λ-abstractions annotated by (a subtype of) of $t_1 \to t_2$.

5.2.1 Progress

There is nothing to prove here: since our reduction semantics accounts for explicit named errors ω_p, every expression is either a value or reduces to an expression or a named error.

Lemma 5.2.1 (Progress)**.** *For all expressions e, either:*

- $\exists v$ *s.t.* $e = v$;
- $\exists e'$ *s.t.* $e \hookrightarrow e'$;
- *or* $\exists p$ *s.t.* $e \hookrightarrow \omega_p$.

Remark (*Blame in Elixir?*)

Classical gradual calculi introduce *explicit casts* to mediate values crossing typed and untyped regions. Each cast is annotated with a *blame label* so that, when a runtime check fails, the semantics can attribute responsibility to the boundary that inserted the cast. Elixir has no such casts: type annotations are static and erased, and execution does not distinguish typed from untyped code. Runtime checks (e.g., in pattern matching, guards, arithmetic) are part of the BEAM's ordinary semantics, and we do not associate them with blame labels. Thus, in our system, failures surface uniformly as ω_p.

5.2.2 Technical lemmas

Before diving into the proofs, we introduce a mild restriction on type annotations that simplifies the reasoning without loss of generality. We consider only interfaces $\mathbb{I} = \{t_i \to s_i \mid i \in I\}$ that satisfy the conditions $\forall (i,j) \in I^2, (t_i \wedge t_j)^{\Uparrow} \leq \mathbb{0}$, and $\forall i \in I, t_i{}^{\Uparrow} \not\leq \mathbb{0}$. In other words, the domains of the arrows in the interface must always be pairwise disjoint, meaning that they do not overlap, and be nonempty. While this restriction might seem limiting, any arbitrary interface can be statically converted into an equivalent one (meaning that the types obtained by intersecting the arrows in the interfaces are equivalent) that adheres to this rule, though this conversion process may lead to a considerable increase in the size of the interface. This conversion is done by using the property that, for every type t_1, t_2, s_1, s_2 we have $(t_1 \to t_2) \wedge (s_1 \to s_2) \simeq (t_1 \setminus s_1 \to t_2) \wedge (s_1 \setminus t_1 \to s_2) \wedge (t_1 \wedge s_1 \to t_2 \wedge s_2)$.

Example (*Overlapping Interfaces*)

Consider, for instance, the following interface that does not satisfy this restriction: $\{\texttt{int} \to \texttt{int}; 5 \to 5; ? \to ?\}$. Through static transformation, we can derive an equivalent valid interface:

$$\{(\texttt{int} \setminus 5) \to \texttt{int}; (\texttt{int} \wedge 5) \to (\texttt{int} \wedge 5); (5 \setminus \texttt{int}) \to 5; (? \setminus (\texttt{int} \vee 5)) \to ?\}$$

which simplifies to:

$$\{(\texttt{int} \setminus 5) \to \texttt{int}; 5 \to 5; (? \setminus (\texttt{int} \vee 5)) \to ?\}$$

This technique extends to interfaces containing multiple overlapping gradual types. As an illustration, the interface $\{? \to \texttt{int}; ? \to 5\}$ can be simplified to $\{? \to \texttt{int} \vee 5\}$. Such transformations preserve the original interface's semantic intent, albeit at the cost of increased complexity in some cases.

Lemma 5.2.2 (Weakening)**.** *If* $\Gamma \vdash_w e ⨾ t$ *then* $\Gamma, x : s \vdash_w e ⨾ t$

Proof. By induction on the size of the derivation tree, and case analysis on the typing rule used to derive $\Gamma \vdash_{\mathrm{w}} e \mathbin{⨾} t$. □

Lemma 5.2.3 (Exchange)**.** *If* $\Gamma_1, x : s, y : t, \Gamma_2 \vdash_w e \mathbin{⨾} u$ *then* $\Gamma_1, y : t, x : s, \Gamma_2 \vdash_w e \mathbin{⨾} u$.

Proof. By induction on the derivation of $\Gamma_1, x : s, y : t, \Gamma_2 \vdash_{\mathrm{w}} e \mathbin{⨾} u$ with case analysis on the last rule. For all rules, premises are obtained by weakening or exchange on subderivations and the IH applies since contexts differ only by the order of two bindings. Notable cases:

(var°) If $e = z$, then z is typed from its binding in the context. In the permuted context, the same binding is present, so the judgment holds. This covers $z = x$, $z = y$, or $z \notin \{x, y\}$.

($\lambda°$), ($\lambda^\circ_\star$), ($\lambda^\circ_?$) Premises extend the context with $x : t_i$ or x:?; apply the IH to each premise after permuting $x : s$ and $y : t$ in the outer context; then reapply the rule.

Structural rules (and°), (sub°) are immediate by IH on premises; application, projection, case, and arithmetic similarly follow by IH on premises. □

Lemma 5.2.4 (Strengthening)**.** *If* $\Gamma, x : s \vdash_w e \mathbin{⨾} t$ *and* $x \notin \mathrm{fv}(e)$*, then* $\Gamma \vdash_w e \mathbin{⨾} t$.

Proof. By induction on the derivation of $\Gamma, x : s \vdash_{\mathrm{w}} e \mathbin{⨾} t$ and case analysis on the last rule. All rules are stable under removing an unused binding; the only delicate case is (var°).

(var°) $e = z$. Since $x \notin \mathrm{fv}(e)$ we have $z \neq x$. The premise $\Gamma, x : s, z : t' \vdash_{\mathrm{w}} z \mathbin{⨾} t$ reduces to $\Gamma, z : t' \vdash_{\mathrm{w}} z \mathbin{⨾} t$ by removing the unused $x : s$; hence the conclusion holds.

Other rules For (and°), (sub°), application, projection, case, arithmetic, and lambda rules, apply the IH to each premise (the added $x : s$ is not used in e nor in bodies where it is not introduced as the bound variable); then reapply the rule. □

In the system for $\vdash_{\mathrm{w}} e \mathbin{⨾} t$, every well-typed closed expression can be typed with ?.

Lemma 5.2.5 (Static typing implies dynamic typing)**.** *If* $\Gamma \vdash_w e \mathbin{⨾} t$ *then* $\Gamma \vdash_w e \mathbin{⨾}$ *?*

Proof. By induction on the size of the derivation tree, and case analysis on $\Gamma \vdash_{\mathrm{w}} e \mathbin{⨾} t$.

(cst°) By subsumption since $c \wedge ? \leq ?$.

(var°) By subsumption.

(and°) By induction hypothesis on either of the premises.

($\lambda°$), ($\lambda^\circ_\star$) Instead of applying those rules, apply rule ($\lambda^\circ_?$).

($\lambda^\circ_?$) Immediate since the conclusion is ?.

(app°), (app$^\circ_\star$) Replace the use of (app) with (app$^\circ_?$).

(app$^\circ_?$) Immediate since the conclusion is ?.

(case$^\circ_?$) By induction hypothesis, all branches can be typed with ?. Re-apply the rule with

$s = ?$.

(add°) Immediate by subsumption since $\texttt{int} \wedge ? \leq ?$.

(sub°) Immediate by induction hypothesis on the single premise. □

Corollary 5.2.6. *If* $\Gamma \vdash_W e : t$, *then* $\Gamma \vdash_W e : t \wedge ?$.

Proof. By application of Lemma 5.2.5 and rule (and°). □

Lemma 5.2.7 (Substitution)**.** *If* $\Gamma, x : s \vdash_W e : t$ *then, for all* e' *such that* $\Gamma \vdash_W e' : s$ *holds, we have* $\Gamma \vdash_W e[e'/x] : t$.

Proof. By induction, and case analysis on $\Gamma, x : s \vdash_W e : t$.

(cst°) Immediate by weakening (5.2.2).

(var°) $e = y$.

- If $y = x$, then by inversion on rule (var°) we have $(t = s \wedge ?)$. Also, $e[e'/x] = e'$ and $\Gamma \vdash_W e' : s$. By Corollary 5.2.6, we have $\Gamma \vdash_W e' : s \wedge ?$, which concludes.
- If $y \neq x$, then $e[e'/x] = y$. By *strengthening* (Lemma 5.2.4, removing the unused assumption $x : s$) from $\Gamma, x : s \vdash_W y : t$ we derive $\Gamma \vdash_W y : t$, hence $\Gamma \vdash_W e[e'/x] : t$.

(and°) Immediate by IH on both premises, and reapplying rule (and°).

(app°) We have: $e = e_1\, e_2$, $\Gamma, x : s \vdash_W e_1 : t_1 \to t_2$, $\Gamma, x : s \vdash_W e_2 : t_1$.
By IH: $\Gamma \vdash_W e_1[e'/x] : t_1 \to t_2$ and $\Gamma \vdash_W e_2[e'/x] : t_1$.
Applying rule (app°) gives: $\Gamma \vdash_W (e_1\, e_2)[e'/x] : t_2$

(app$^\circ_\star$) Same as above, ending with applying rule (app$^\circ_\star$).

(app$^\circ_?$) We have $e = e_1\, e_2$, $\Gamma, x : s \vdash_W e_1 : t_1$, $\Gamma, x : s \vdash_W e_2 : t_2$.
By IH: $\Gamma \vdash_W e_1[e'/x] : t_1$ and $\Gamma \vdash_W e_2[e'/x] : t_2$, where obviously $t_1 \leq \mathbb{1}$ and $t_2 \leq \mathbb{1}$.
Thus, applying (app$^\circ_?$) gives $\Gamma \vdash_W (e_1\, e_2)[e'/x] : ?$.

(λ°) $e = \lambda^{\mathbb{I}} y.e_0$. By inversion, $(\forall (t_i \to s_i) \in \mathbb{I}, \Gamma, x : s, y : t_i \vdash_W e_0 : s_i)$ and $(\Gamma, x : s, y : ? \vdash_W e_0 : \mathbb{1})$.
We use exchange (Lemma 5.2.3) to switch x and y. Then, by IH,

$$\begin{cases} \forall (t_i \to s_i) \in \mathbb{I}, (\Gamma, y : t_i \vdash_W e_0[e'/x] : s_i) \\ \Gamma, y : ? \vdash_W e_0[e'/x] : \mathbb{1} \end{cases}$$

Applying rule (λ°) gives: $\Gamma \vdash_W (\lambda^{\mathbb{I}} y.e_0)[e'/x] : \bigwedge_{i \in I} (t_i \to s_i)$

($\lambda^\circ_\star$) By inversion $\Gamma, x : s \vdash_W \lambda^{\mathbb{I}} y.e : (t_1 \to t_2)^\star$ and $\Gamma, x : s, y : ? \vdash_W e : t_2$.
By exchange (Lemma 5.2.3) on the second premise and IH, $\Gamma, y : ? \vdash_W e[e'/x] : t_2$.
By IH on the first premise, $\Gamma \vdash_W \lambda^{\mathbb{I}} y.e[e'/x] : t_1 \to t_2$.
We can then reapply rule ($\lambda^\circ_\star$) to conclude.

($\lambda^{\circ}_{?}$) Immediate by IH.

(case°) $e = \texttt{case}\, e\,(t_i \to e_i)_i$ is given type $s' \wedge ?$ (the rule uses s, but it is already used in this lemma's statement), and $\Gamma \vdash_{\mathrm{w}} e \mathbin{⁏} t$.
By inversion, for all i, if $t \wedge \rho_i \not\leq \mathbb{0}$, $\Gamma \vdash_{\mathrm{w}} e_i \mathbin{⁏} s'$.
By IH, $\Gamma \vdash_{\mathrm{w}} e[e'/x] \mathbin{⁏} t$ and for all i such that $t \wedge \rho_i \not\leq \mathbb{0}$, we have $\Gamma \vdash_{\mathrm{w}} e_i[e'/x] \mathbin{⁏} s'$.
Applying rule (case°) gives $\Gamma \vdash_{\mathrm{w}} \texttt{case}\, e[e'/x]\,(\rho_i \to e_i[e'/x])_i \mathbin{⁏} s' \wedge ?$.

(add°) Immediate by IH on both premises, and reapplying rule (add°).

(sub°) Immediate by IH and reapplying rule (sub°). □

Definition 5.2.8 (Value type operator)**.** *We define the operator* $\texttt{type}(\cdot)$ *from values to types as follows:*

$$\texttt{type}(c) = c \wedge ?$$
$$\texttt{type}(\lambda^{\mathbb{I}} x.e) = \bigwedge_{(t\to s)\in\mathbb{I}} (t \to s)$$

Lemma 5.2.9 (Value type operator)**.** *If* $\varnothing \vdash_{\mathrm{w}} v \mathbin{⁏} t$*, then* $\varnothing \vdash_{\mathrm{w}} v \mathbin{⁏} \texttt{type}(v)$*.*

Proof. Trivial for a constant c as it is typed with $c \wedge ? \leq ?$. A simple application of subsumption concludes.
For a lambda-abstraction, both rules ($\lambda_{?}$) and ($\lambda_{\star}$) rely on rule (λ) being applied earlier, and (λ) typechecks exactly the interface of a function, which is $\texttt{type}(v)$. □

In the system for $\vdash_{\mathrm{w}} e \mathbin{⁏} t$, every well-typed closed expression can be typed with $?$.

Lemma 5.2.10 (Substitution by value)**.** *If* $\Gamma, x : ? \vdash_{\mathrm{w}} e \mathbin{⁏} t$ *then, for all "⁏-well-typed" value* $\varnothing \vdash_{\mathrm{w}} v \mathbin{⁏} t'$*, we have* $\Gamma \vdash_{\mathrm{w}} e[v/x] \mathbin{⁏} t$*.*

Proof. Substitution by value. By induction on e. We eliminate the cases where the last rule used in the derivation of $\Gamma, x : ? \vdash_{\mathrm{w}} e \mathbin{⁏} t$ is subsumption (sub°) or intersection (and°), since then it is trivial to prove the result by IH:

Rule (sub°) By IH, we have $\Gamma \vdash_{\mathrm{w}} e[v/x] \mathbin{⁏} s$ for some $s \leq t$. We conclude by subsumption.

Rule (and°) We have $t = t_1 \wedge t_2$ and by inversion on (and°) and IH on the premises, we have $\Gamma \vdash_{\mathrm{w}} e[v/x] \mathbin{⁏} s_1$ and $\Gamma \vdash_{\mathrm{w}} e[v/x] \mathbin{⁏} s_2$ for some $s_1 \leq t_1$ and $s_2 \leq t_2$. We conclude by reapplying rule (and°).

Then we can reason by case analysis on the shape of e which is typed by structural rules:

Case $e = c$ Immediate since $e[v/x] = c$.

Case $e = y \neq x$ Immediate since $y \in \Gamma$ and $e[v/x] = y$.

Case $e = x$ Then x is typed by rule (var°) with $x : ?$, hence $(t \simeq ?)$. Since v is well-typed, by

Lemma 5.2.5, we have $\varnothing \vdash_{\mathrm{w}} v \mathbin{⨾} ?$, which concludes by *strengthening* (Lemma 5.2.4).

Case $e = e_1\, e_2$ Consider the typing rule used to type e.

Rule (app°) By inversion, $\Gamma, x : ? \vdash_{\mathrm{w}} e_1 \mathbin{⨾} t_1 \to t_2$ and $\Gamma, x : ? \vdash_{\mathrm{w}} e_2 \mathbin{⨾} t_1$.
By IH, $\Gamma \vdash_{\mathrm{w}} e_1[v/x] \mathbin{⨾} t_1 \to t_2$ and $\Gamma \vdash_{\mathrm{w}} e_2[v/x] \mathbin{⨾} t_1$.
So by (app°), $\Gamma \vdash_{\mathrm{w}} (e_1\, e_2)[v/x] \mathbin{⨾} t_2$.

Rule ($\mathbf{app}^{\circ}_{?}$) Immediate by IH similar to above.

Rule ($\mathbf{app}^{\circ}_{\star}$) Immediate by IH similar to above.

Case $e = \lambda^{\mathbb{I}} y.e_0$ Consider the typing rule used.

Rule (λ°) By inversion, $(\forall (t_i \to s_i) \in \mathbb{I}, \Gamma, x : ?, y : t_i \vdash_{\mathrm{w}} e_0 \mathbin{⨾} s_i)$ and $\Gamma, x : ?, y : ? \vdash_{\mathrm{w}} e_0 \mathbin{⨾} \mathbb{1}$
We use exchange (Lemma 5.2.3) to switch x and y. Then, by IH, $(\forall (t_i \to s_i) \in \mathbb{I}, \Gamma, y : t_i \vdash_{\mathrm{w}} e_0[v/x] \mathbin{⨾} s_i)$ and $\Gamma, y : ? \vdash_{\mathrm{w}} e_0[v/x] \mathbin{⨾} \mathbb{1}$ So we re-apply (λ°) to get $\Gamma \vdash_{\mathrm{w}} \lambda^{\mathbb{I}} y.e_0[v/x] \mathbin{⨾} \bigwedge_{i \in I} (t_i \to s_i)$.

Rule ($\lambda^{\circ}_{?}$) Immediate by IH.

Rule ($\lambda^{\circ}_{\star}$) By inversion $\Gamma, x : ? \vdash_{\mathrm{w}} \lambda^{\mathbb{I}} y.e \mathbin{⨾} t_1 \to t_2$ and $\Gamma, x : ?, y : ? \vdash_{\mathrm{w}} e \mathbin{⨾} t_2$. By exchange (Lemma 5.2.3) on the second premise and IH, $\Gamma, y : ? \vdash_{\mathrm{w}} e[v/x] \mathbin{⨾} t_2$ By IH on the first premise, $\Gamma \vdash_{\mathrm{w}} \lambda^{\mathbb{I}} y.e[v/x] \mathbin{⨾} t_1 \to t_2$. We can then reapply ($\lambda^{\circ}_{\star}$) to conclude.

Case $e = \texttt{case}\, e' (\rho_i \to e_i)_i$ Rule (case°).
By inversion, $\Gamma, x : ? \vdash_{\mathrm{w}} e' \mathbin{⨾} t$ and $\forall i$, if $t \wedge \rho_i \not\leq \mathbb{0}$ then $\Gamma, x : ? \vdash_{\mathrm{w}} e_i \mathbin{⨾} t'$.
By IH, $\Gamma \vdash_{\mathrm{w}} e'[v/x] \mathbin{⨾} t$ and for all i, if $t \wedge \rho_i \not\leq \mathbb{0}$ then $\Gamma \vdash_{\mathrm{w}} e_i[v/x] \mathbin{⨾} t'$.
We can then reapply (case°) to conclude.

Case $e = e_1 + e_2$ Rule (add°).
By inversion, $\Gamma, x : ? \vdash_{\mathrm{w}} e_1 \mathbin{⨾} t_1$ and $\Gamma, x : ? \vdash_{\mathrm{w}} e_2 \mathbin{⨾} t_2$.
By IH, $\Gamma \vdash_{\mathrm{w}} e_1[v/x] \mathbin{⨾} t_1$ and $\Gamma \vdash_{\mathrm{w}} e_2[v/x] \mathbin{⨾} t_2$.
We can then reapply (add°) to conclude. □

5.2.3 Subject reduction and soundness

Lemma 5.2.11. *(Subject Reduction) If* $\Gamma \vdash_{\mathrm{w}} e \mathbin{⨾} t$ *and* $e \hookrightarrow e'$, *then* $\Gamma \vdash_{\mathrm{w}} e' \mathbin{⨾} t$.

Proof of Subject Reduction. By induction on the derivation of $\Gamma \vdash_{\mathrm{w}} e \mathbin{⨾} t$ and case analysis on the reduction rule: If the last rule is subsumption (sub°) or intersection (and°), we can directly apply the IH to the premise and obtain the result.

Reduction $\mathscr{E}$ $e = \mathscr{E}[e_0]$ with $e_0 \hookrightarrow e'_0$ and $\mathscr{E} \neq \square$. Expression e_0 is typed by a subtree of the derivation tree of $\Gamma \vdash_{\mathrm{w}} e \mathbin{⨾} t$. Thus, by IH, its type is preserved after reduction. Hence the type of $\mathscr{E}[e'_0]$ is preserved.

Reduction $[\beta]$ $e = (\lambda^{\mathbb{I}} x.e_1)\, v_2$

Consider the last rule used to type the application.

Rule (app$^\circ$) This case implies type preservation by substitution lemma. Indeed, by inversion we have $\Gamma \vdash_w \lambda^{\mathbb{I}} x.e_1 \mathbin{⨾} t' \to t$. With $\Gamma \vdash_w v_2 \mathbin{⨾} t'$, by substitution lemma, $\Gamma \vdash_w e_1[v_2/x] \mathbin{⨾} t$.

Rule (app$^\circ_?$) We prove that the result of the reduction is "⨾-well-typed". By inversion, $\Gamma, x : ? \vdash_w e_1 \mathbin{⨾} \mathbb{1}$. Since $\Gamma \vdash_w v_2 \mathbin{⨾} t_2$, by Lemma 5.2.10, we have $\Gamma \vdash_w e_1\,[v_2/x] \mathbin{⨾} \mathbb{1}$. We conclude by Lemma 5.2.5 that $\Gamma \vdash_w e_1\,[v_2/x] \mathbin{⨾} ?$.

Rule (app$^\circ_\star$) By inversion, $\Gamma, x : ? \vdash_w e_1 \mathbin{⨾} t \wedge ?$. Since $\Gamma \vdash_w v_2 \mathbin{⨾} t_2$, by lemma 5.2.10, $\Gamma \vdash_w e_1[v_2/x] \mathbin{⨾} t \wedge ?$ which concludes.

Reduction [+] The result is immediately a well-typed integer.

Reduction [case] Rule (case$^\circ$). In this case $t = s \wedge ?$.

By inversion, for the branches, we have for all $i \in I$, if $t \wedge \rho_i \not\leq \mathbb{0}$ then $\Gamma, x : ? \vdash_w e_i \mathbin{⨾} s$.

The case expression reduces to one of those branches. But this *notably* does not suffice to obtain preservation, since those branches that we reduce to have been typed with s, not $s \wedge ?$.

However, Lemma 5.2.6 tells us that all branches typed with s are also typed with $s \wedge ?$. Thus, the reduction preserves the typing. □

To link back to the gradual system, we use the fact that every expression well-typed in the former is well-typed in the latter.

Lemma 5.2.12 (Gradual typing implies strong typing)**.** *If* $\Gamma \vdash_g e : t$ *then* $\Gamma \vdash_w e \mathbin{⨾} t$*.*

Proof. Every rule in the gradual system of Figure 5.2 has a more general counterpart in Figure 5.3. □

We formulate three soundness results that depend on whether the unsafe ω-rules or the gradual rules are used to type expressions.

Theorem 5.2.13 (Static Type Safety)**.** *If $\varnothing \vdash_s e : t$ has a derivation that does not use any ω-marked rules, then evaluation of e will never produce a runtime type error. In particular, either e diverges (evaluates forever without a value) or e evaluates to some value v with $\varnothing \vdash_s v : t$. (Well-typed programs cannot "go wrong.")*

Theorem 5.2.14 (ω-Type Safety)**.** *If $\varnothing \vdash_s e : t$ is derivable even with use of ω-rules, then either e diverges, or e evaluates to a value v with $\varnothing \vdash_s v : t$, or e terminates abruptly with an "index out of range" failure ($\omega_{\text{OutOfRange}}$). This last scenario can occur only as a result of the (proj$_\omega$)/(proj$^{\mathbb{1}}_\omega$) rules - it is precisely the case that was flagged by a static warning.*

Theorem 5.2.15 (Gradual Soundness)**.** *If $\varnothing \vdash_g e : t$ then either e diverges, or e reduces to ω_p for some p, or there is a value v such that $e \hookrightarrow^* v$ and $\varnothing \vdash_w v \mathbin{⨾} t$.*

Proof. Corollary of the subject reduction 5.2.11 and progress 5.2.1 lemmas. □

The first theorem states that, in the absence of gradual typing, if no warning is emitted, then we are in a classic static typing system. If a warning is raised but gradual typing is still not used, then the second theorem states that the only possible runtime failure is the out-of-range selection of a tuple. If gradual typing is used, then Theorem 5.2.15 states that any resulting value will have the shape described by the inferred type.

5.3 Extending semantic subtyping: strong arrows

Integrating the strong arrow types introduced in Section 5.1 into the semantic subtyping framework of of Frisch (2004) requires a similar approach to that used in Chapter 4 for multi-arity function types. We must first define how to interpret strong arrow types as sets within the domain $\mathscr{D}$. We then use this interpretation to decompose the emptiness problem of a disjunctive normal form of arrow literals into simpler, more tractable problems.

For the sake of simplicity, we present the semantic interpretation of unary strong arrows, without the $\mho$ element that distinguishes $\mathbb{0} \to t$ types. The extension to multi-arity functions is straightforward.

Recall that, according to (Frisch, 2004, Definition 4.2), for a domain $\mathscr{D}$ and $X, Y \subseteq \mathscr{D}$ we define

$$X \to Y \overset{\text{def}}{=} \{R \in \mathscr{P}_f(\mathscr{D} \times \mathscr{D}_\Omega) \mid \forall (d, \delta) \in R.\ d \in X \Rightarrow \delta \in Y\}$$

and that by (Frisch, 2004, Lemma 4.7) we have

$$X \to Y = \mathscr{P}_f\left(\overline{X \times \overline{Y}^{\mathscr{D}_\Omega}}^{\mathscr{D} \times \mathscr{D}_\Omega}\right).$$

The interpretation, to be defined, of a strong arrow type $(X \to Y)^\star$ is necessarily a subset of the interpretation of $X \to Y$, insofar as it contains only relations that represent functions in $X \to Y$ that are *strong*. These are functions that for every possible argument (whether in X or not) return either a value in Y or Ω. In other words, the strong arrow type $(X \to Y)^\star$ contains sets of finite sets of pairs (d, δ) where if d is in the domain of the function then δ is in its codomain, but with the additional requirement that if d is *not* in the domain of the function, then δ must be either in Y or be Ω. More compactly, $(X \to Y)^\star = (X \to Y) \cap \mathscr{P}_f(\mathscr{D} \times Y_\Omega)$, where the intersection confirms that it is indeed a subset of the interpretation of the arrow type.

Note

Strong functions have a defined behaviour outside their domain: they necessarily error, diverge, or return a value of their precise codomain type. Thus, a function of type $(\texttt{int} \to \texttt{int})^\star$, when given Booleans, can either error or return *integers*; it cannot also have type $(\texttt{bool} \to \texttt{bool})^\star$ unless it is the always-erroring-diverging function $(\mathbb{1} \to \mathbb{0})$.

We use this interpretation to derive an algorithm for deciding subtyping in the presence of strong arrow types. As the interpretation reveals, strong arrow types are subsets of arrow types. This relationship makes it straightforward to extend the existing framework: in the presence of strong arrows, the disjunctive normal form of a type remains a union of intersections of literals on the same constructors. However, for functional spaces, the literals can now be either arrows or strong arrows. Therefore, deciding subtyping of function spaces requires solving emptiness problems of the following form:

$$\bigwedge_{i\in I}(t_i \to s_i) \wedge \bigwedge_{j\in P}(t_j \to s_j)^\star \wedge \bigwedge_{k\in R}\neg(t_k \to s_k) \wedge \bigwedge_{l\in Q}\neg(t_l \to s_l)^\star \quad \leq \quad \mathbb{0}$$

This containment can be further simplified since $\bigwedge_{j\in P}(t_j \to s_j)^\star \simeq (\bigvee_{j\in P} t_j \to \bigwedge_{j\in P} s_j)^\star$ (see Lemma 5.3.3). Therefore, writing t and s for $\bigvee_{j\in P} t_j$ and $\bigwedge_{j\in P} s_j$, we can rewrite the previous containment as:

$$\bigwedge_{i\in I}(t_i \to s_i) \wedge (t \to s)^\star \wedge \bigwedge_{k\in R}\neg(t_k \to s_k) \wedge \bigwedge_{l\in Q}\neg(t_l \to s_l)^\star \quad \leq \quad \mathbb{0}$$

This holds if and only if one of the following formulas holds (cf. Lemma 5.3.5)

$$\text{either} \qquad \exists j \in R.\ \bigwedge_{i\in I}(t_i \to s_i) \wedge (t \to s)^\star \leq (t_k \to s_k) \tag{5.3.1}$$

$$\text{or} \qquad \exists \ell \in Q.\ \bigwedge_{i\in I}(t_i \to s_i) \wedge (t \to s)^\star \leq (t_\ell \to s_\ell)^\star \tag{5.3.2}$$

Finally, the condition (5.3.1) is solved by applying Lemma 5.3.6:

$$\bigwedge_{i\in I}(t_i \to s_i) \wedge (c \to d)^\star \leq a \to b \iff \begin{cases} a \leq \bigvee_{i\in I} t_i \vee c \\ \forall J \subseteq I.\ \left(a \leq \bigvee_{j\in J} t_j\right) \vee \left(\bigwedge_{j\in I\setminus J} s_j \wedge d \leq b\right) \end{cases}$$

and condition (5.3.2) by applying Lemma 5.3.9:

$$\bigwedge_{i\in I}(t_i \to s_i) \wedge (c \to d)^\star \leq (a \to b)^\star \iff \begin{cases} a \leq \bigvee_{i\in I} t_i \vee c \\ \forall J \subseteq I.\ \left(\bigvee_{j\in J} t_j = \mathbb{1}\right) \vee \left(\bigwedge_{j\in I\setminus J} s_j \wedge d \leq b\right) \end{cases}$$

5.3.1 Set Semantics

Instead of defining the interpretation of strong *arrows* we define the interpretation of strong *sets* of finite relations.

Definition 5.3.1. *Let X be a subset of $\mathscr{P}_f(\mathscr{D} \times \mathscr{D}_\Omega)$.*

- $\mathit{dom}(X) = \{d \in \mathscr{D} \mid \forall R \in X.\ (d, \Omega) \notin R\}$
- $\mathit{cod}(X) = \{d' \in \mathscr{D} \mid (\mathtt{dom}(X) = \varnothing) \vee (\exists R \in X. \exists d \in \mathit{dom}(X).\ (d, d') \in R)\}$
- $X^\star = X \cap \mathscr{P}_f\left(\mathscr{D} \times \left(\mathit{cod}(X) \cup \{\Omega\}\right)\right)$

Lemma 5.3.2. *Let $X, Y \subseteq \mathscr{D}$ with $X \neq \varnothing$. Then,*

$$\mathtt{dom}(X \to Y) = X \qquad \textit{and} \qquad \mathit{cod}(X \to Y) = Y \tag{5.3.3}$$

Proof. Given $X \to Y = \mathscr{P}_f\left(\overline{X \times \overline{Y}^{\mathscr{D}_\Omega}}^{\mathscr{D} \times \mathscr{D}_\Omega}\right)$, set

$$S := \overline{X \times \overline{Y}^{\mathscr{D}_\Omega}}^{\mathscr{D} \times \mathscr{D}_\Omega} = \left(\overline{X}^{\mathscr{D}} \times \mathscr{D}_\Omega\right) \cup \left(X \times Y\right),$$

so $X \to Y = \mathscr{P}_f(S)$ and every $R \in X \to Y$ satisfies $R \subseteq S$.

$\underline{\mathtt{dom}(X \to Y) = X}$ If $d \in X$, then $(d, \Omega) \notin S$ (since $d \notin \overline{X}^{\mathscr{D}}$ and $\Omega \notin Y$), hence $(d, \Omega) \notin R$ for all $R \subseteq S$; thus $d \in \mathtt{dom}(X \to Y)$. Conversely, if $d \notin X$, then $R := \{(d, \Omega)\} \subseteq \overline{X}^{\mathscr{D}} \times \mathscr{D}_\Omega \subseteq S$, so $R \in X \to Y$ and $(d, \Omega) \in R$, whence $d \notin \mathtt{dom}(X \to Y)$.

$\underline{\mathrm{cod}(X \to Y) = Y}$

(⊇) Let $y \in Y$. Since $\mathtt{dom}(X \to Y) = X \neq \varnothing$, we may pick any $x \in \mathtt{dom}(X \to Y)$ and set $R := \{(x, y)\}$. Since $(x, y) \in X \times Y \subseteq S$, we have $R \in X \to Y$, and with $x \in \mathtt{dom}(X \to Y)$ the definition of $\mathrm{cod}(\cdot)$ yields $y \in \mathrm{cod}(X \to Y)$.

(⊆) Let $y \in \mathrm{cod}(X \to Y)$. Then, given $\mathtt{dom}(X \to Y) \neq \varnothing$, there exist $R \in X \to Y$ and $x \in \mathtt{dom}(X \to Y) = X$ such that $(x, y) \in R$. Because $R \subseteq S$ and $x \in X$, the membership $(x, y) \in S = (\overline{X}^{\mathscr{D}} \times \mathscr{D}_\Omega) \cup (X \times Y)$ forces $y \in Y$. Hence $y \in Y$.

Combining both inclusions we obtain $\mathrm{cod}(X \to Y) = Y$ whenever $\mathtt{dom}(X \to Y) \neq \varnothing$ (equivalently, $X \neq \varnothing$). If $X = \varnothing$, then $X \to Y = \mathscr{P}_f(\mathscr{D} \times \mathscr{D}_\Omega)$ and $\mathtt{dom}(X \to Y) = \varnothing$, so by Definition 5.3.1 we get $\mathrm{cod}(X \to Y) = \mathscr{D}$. □

Main subtyping decision problem (set formulation). Fix finite families of base sets $(X_i, Y_i)_{i \in I}$ and $(U_j, V_j)_{j \in J}$, with $X_i, U_j \subseteq \mathscr{D}$ (domains) and $Y_i, V_j \subseteq \mathscr{D}$ (codomains). Write $(X \to Y)$ for ordinary arrows and $(X \to Y)^\star$ for strong arrows. By definition above, $(X \to Y)^\star$ is the set of relations $R \subseteq \mathscr{D} \times (Y_\Omega)$ that are "strong" in the sense that on $d \in X$ one must return in Y, on $d \notin X$ one may return in Y, or be Ω. We build *arrow literals* by allowing negation $\neg$ in front of either kind:

$$\ell ::= (X_i \to Y_i) \mid \neg(X_i \to Y_i) \mid (U_j \to V_j)^\star \mid \neg(U_j \to V_j)^\star.$$

A generic subtyping query $A \subseteq B$ is equivalent to testing the *emptiness* of a finite union of finite intersections of such literals (a DNF over arrow literals):

$$\underbrace{\bigcup_{p=1}^{m} \bigcap_{q=1}^{n_p} \ell_{p,q}}_{\text{union of intersections of (strong/ordinary) arrows and their negations}} \qquad = \varnothing, \quad \text{each } \ell_{p,q} \text{ of the four forms above,}$$

where $\neg$ denotes complement with respect to a fixed ambient universe of arrows. All the lemmas that follow are devoted to rewriting such expressions–collapsing intersections of strong arrows, translating mixed intersections/unions into base-set conditions, and eliminating the remaining finite unions–until the emptiness test reduces to inclusions between the underlying sets X_i, Y_i, U_j, V_j.

First, notice that the finite union of finite intersections above is empty if and only if each of the intersections is empty. Thus, we reduce to solving the generic problem $\bigcap_{q=1}^{n} \ell_q = \varnothing$ which is thus, if we write the literals as:

$$\left(\bigcap_{i\in P} X_i \to Y_i\right) \cap \left(\bigcap_{i\in N} \neg(X_i \to Y_i)\right) \cap \left(\bigcap_{j\in Q} U_j \to V_j{}^\star\right) \cap \left(\bigcap_{j\in R} \neg(U_j \to V_j)^\star\right) = \varnothing \qquad (*_0)$$

Through De Morgan's laws, we know that each intersection of negated arrows is equivalent to the complement of the union of the arrows, thus:

$$\bigcap_{i\in N} \neg\big(X_i \to Y_i\big) = \mathscr{P}_f\left(\mathscr{D} \times \mathscr{D}_\Omega\right) \smallsetminus \Big(\bigcup_{i\in N} X_i \to Y_i\Big)$$

and

$$\bigcap_{j\in R} \neg(U_j \to V_j)^\star = \mathscr{P}_f\left(\mathscr{D} \times \mathscr{D}_\Omega\right) \smallsetminus \Big(\bigcup_{j\in R} (U_j \to V_j)^\star\Big)$$

Using this, we can rewrite equation $(*_0)$ into the following set-inclusion problem:

$$\left(\bigcap_{i\in P} X_i \to Y_i\right) \cap \left(\bigcap_{j\in Q} (U_j \to V_j)^\star\right) \subseteq \left(\bigcup_{i\in N} X_i \to Y_i\right) \cup \left(\bigcup_{j\in R} (U_j \to V_j)^\star\right) \qquad (*_1)$$

Our first step then is to introduce Lemma 5.3.3 as a *normalisation step*: for finite I it collapses $\bigcap_{i\in I} (X_i \to Y_i)^\star$ into a single strong arrow $(\bigcup_{i\in I} X_i \to \bigcap_{i\in I} Y_i)^\star$. This removes an outer intersection on strong arrows and leaves one arrow with a union in the domain and an intersection in the codomain.

Lemma 5.3.3. *Let I be a finite non-empty set, then we have:*

$$\bigcap_{i\in I} (X_i \to Y_i)^\star \quad = \quad \left(\bigcup_{i\in I} X_i \to \bigcap_{i\in I} Y_i\right)^\star$$

Proof. Proving both inclusions.

($\supseteq$) Suppose $R \in (\bigcup_{i\in I} X_i \to \bigcap_{i\in I} Y_i)^\star$. By Lemma 5.3.2 we know that the codomain of $(\bigcup_{i\in I} X_i \to \bigcap_{i\in I} Y_i)$ is $(\bigcap_{i\in I} Y_i)$. Let $(d,\delta) \in R$. Let $i_0 \in I$.

- if $d \in X_{i_0}$, then $d \in \bigcup_{i\in I} X_i$ so $\delta \in \bigcap_{i\in I} Y_i \subseteq Y_{i_0}$ by definition of R.
- if $d \notin X_{i_0}$, then the strongness property of the arrow interpretation R belongs to ensures that $\delta \in (\bigcap_{i\in I} Y_i) \cup \{\Omega\} \subseteq Y_i \cup \{\Omega\}$.

($\subseteq$) Now, suppose $R \in \bigcap_{i\in I} (X_i \to Y_i)^\star$. Let $(d,\delta) \in R$.

- if $d \in \bigcup_{i\in I} (X_i)$. For all $i \in I$, either $d \in X_i$, thus $\delta \in Y_i$, or $d \notin X_i$, thus $\delta \in Y_i \cup \{\Omega\}$. Since there exists at least one j_0 such that $d \in X_{j_0}$, then we know that that $\delta \in Y_{j_0}$. Since Y_{j_0} does not contain Ω, then $\delta \neq \Omega$ and thus we have proven that $\delta \in \bigcap_{i\in I} (Y_i)$.
- if $d \notin \bigcup_{i\in I} (X_i)$, we can apply the same reasoning as in the previous case, except that we cannot deduce that $\delta \neq \Omega$. We thus have $\delta \in \bigcap_{i\in I} (Y_i \cup \{\Omega\})$. □

The application of this lemma to equation $(*_1)$ means that instead of having a finite intersection of strong arrows, the generic problem is reduced to a single strong arrow.

$$\left(\bigcap_{i\in P} X_i \to Y_i\right) \cap (U \to V)^\star \subseteq \left(\bigcup_{i\in N} X_i \to Y_i\right) \cup \left(\bigcup_{j\in R} (U_j \to V_j)^\star\right) \qquad (*_2)$$

where we abbreviate

$$U := \bigcup_{j\in Q} U_j \qquad \text{and} \qquad V := \bigcap_{j\in Q} V_j.$$

The inclusion $(*_2)$ can be reformulated as an inclusion between finite parts of base sets, which will allow us to apply Lemma 5.3.5.

Recall from Section **??** that every arrow type $X \to Y$ equals $\mathscr{P}_f(E)$ for some base set E, and that strong arrows are intersections of two such types: $(X \to Y)^\star = (X \to Y) \cap (\mathbb{1} \to Y)$. We introduce base sets E_i, F_i, G_j, H_j as follows:

- **Positive arrows:** for each $i \in P$, let $X_i \to Y_i = \mathscr{P}_f(E_i)$;
- **The strong arrow:** write $U \to V = \mathscr{P}_f(E_0)$ and $\mathbb{1} \to V = \mathscr{P}_f(E_1)$, so $(U \to V)^\star = \mathscr{P}_f(E_0) \cap \mathscr{P}_f(E_1)$;
- **Negative arrows:** for each $i \in N$, let $X_i \to Y_i = \mathscr{P}_f(F_i)$;
- **Negative strong arrows:** for each $j \in R$, let $(U_j \to V_j)^\star = \mathscr{P}_f(G_j) \cap \mathscr{P}_f(H_j)$.

With this notation, $(*_2)$ becomes an inclusion of finite-parts sets:

$$\bigcap_{i\in P\cup\{0,1\}} \mathscr{P}_f(E_i) \subseteq \bigcup_{i\in N} \mathscr{P}_f(F_i) \cup \bigcup_{j\in R} \left(\mathscr{P}_f(G_j) \cap \mathscr{P}_f(H_j)\right)$$

There, Lemma 5.3.5 converts the global inclusion in $(*_2)$ into an *existential witness* on base sets:

$$\exists i_0 \in N: \bigcap_{i\in P\cup\{0,1\}} E_i \subseteq F_{i_0} \quad \text{or} \quad \exists j_0 \in R: \bigcap_{i\in P\cup\{0,1\}} E_i \subseteq G_{j_0} \cap H_{j_0},$$

or equivalently (by Lemma 3.1.18):

$$\exists i_0 \in N : \bigcap_{i \in P \cup \{0,1\}} \mathscr{P}_f(E_i) \subseteq \mathscr{P}_f\left(F_{i_0}\right) \quad \text{or} \quad \exists j_0 \in R : \bigcap_{i \in P \cup \{0,1\}} \mathscr{P}_f(E_i) \subseteq \mathscr{P}_f\left(G_{j_0}\right) \cap \mathscr{P}_f\left(H_{j_0}\right)$$

so, after these transformations, going back to arrow forms, we obtain

$$\begin{aligned} &\exists i_0 \in N : \left(\bigcap_{i \in P} X_i \to Y_i\right) \cap (U \to V)^\star \subseteq X_{i_0} \to Y_{i_0} \\ \text{or} \quad &\exists j_0 \in R : \left(\bigcap_{i \in P} X_i \to Y_i\right) \cap (U \to V)^\star \subseteq (U_{j_0} \to V_{j_0})^\star \end{aligned} \tag{$*_3$}$$

which means that we are left to decide that the intersection of arrows on the left-hand side is contained in either

i) a single weak arrow (Lemma 5.3.6), or

ii) a single strong arrow $(U_{j_0} \to V_{j_0})^\star$, which is in fact the intersection of the weak arrow $(U_{j_0} \to V_{j_0})$ with the powerset $\mathscr{P}_f\left(\mathscr{D} \times V_{j_0} \cup \{\Omega\}\right)$. Since a set is included in the intersection of two sets if and only if it is included in both of them, the problem reduces to deciding both inclusion in a weak arrow (already done) and a given powerset. The latter is treated in Lemma 5.3.7.

We recall, from Chapter 3, the following lemma:

Lemma 5.3.4 (Powerset-intersection criterion)**.** *Let $(A_i)_{i \in I}$ and $(B_i)_{j \in J}$ be abstract sets.*

$$\bigcap_{i \in I} \mathscr{P}_f(A_i) \subseteq \bigcup_{j \in J} \mathscr{P}_f\left(B_j\right) \iff \exists j \in J.\ \bigcap_{i \in I} A_i \subseteq B_j.$$

Lemma 5.3.5.

$$\begin{aligned} &\bigcap_{i \in I} \mathscr{P}_f(X_i) \subseteq \bigcup_{i \in P} \mathscr{P}_f(Y_i) \cup \bigcup_{i \in Q} \mathscr{P}_f(Z_i) \cap \mathscr{P}_f(W_i) \\ \iff &\left(\exists i_0 \in P. \bigcap_{i \in I} X_i \subseteq Y_{i_0}\right) \vee \left(\exists i_0 \in Q. \bigcap_{i \in I} X_i \subseteq Z_{i_0} \cap W_{i_0}\right) \end{aligned}$$

Proof. We spell out the proof using only the (finite) powerset reading of the operators:

$$U \in \mathscr{P}_f(Y) \iff \left(U \subseteq Y \ \wedge\ U \text{ is finite}\right).$$

Membership in (co)products of families is also expanded as usual:

$$U \in \bigcap_{i \in I} \mathscr{A}_i \iff \forall i \in I,\ U \in \mathscr{A}_i, \qquad U \in \bigcup_{k \in K} \mathscr{B}_k \iff \exists k \in K,\ U \in \mathscr{B}_k.$$

We further use the facts (immediate from the two displays above):

(F1) $U \in \bigcap_{i\in I} \mathscr{P}_f(X_i)$ iff U is finite and $U \subseteq \bigcap_{i\in I} X_i$.

(F2) $U \in \bigcup_{i\in P} \mathscr{P}_f(Y_i)$ iff $\exists i \in P$, $U \subseteq Y_i$ and U is finite.

(F3) $U \in \bigcup_{i\in Q}\big(\mathscr{P}_f(Z_i) \cap \mathscr{P}_f(W_i)\big)$ iff $\exists i \in Q$, $\big(U \subseteq Z_i \,\wedge\, U \subseteq W_i\big)$ (and U is finite).

(⇒) Assume

$$\bigcap_{i\in I} \mathscr{P}_f(X_i) \subseteq \Big(\bigcup_{i\in P} \mathscr{P}_f(Y_i)\Big) \cup \Big(\bigcup_{i\in Q} \mathscr{P}_f(Z_i) \cap \mathscr{P}_f(W_i)\Big). \tag{†}$$

We prove the right-hand disjunction of the statement. We reason by contradiction. Suppose

$$\neg\exists i_0 \in P.\ \bigcap_{i\in I} X_i \subseteq Y_{i_0} \quad\text{and}\quad \neg\exists i_0 \in Q.\ \bigcap_{i\in I} X_i \subseteq Z_{i_0} \cap W_{i_0}. \tag{5.3.4}$$

From the first negation, for each $p \in P$ choose a witness $x_p \in \big(\bigcap_{i\in I} X_i\big) \setminus Y_p$. From the second negation, for each $q \in Q$ choose a witness $t_q \in \big(\bigcap_{i\in I} X_i\big) \setminus (Z_q \cap W_q)$; i.e., for each q we have either $t_q \notin Z_q$ or $t_q \notin W_q$.
Now set the *finite* set

$$U := \{x_p \mid p \in P\} \cup \{t_q \mid q \in Q\}.$$

(Here we use that P and Q are finite index sets of the finite unions on the right-hand side.) By construction $U \subseteq \bigcap_{i\in I} X_i$ and U is finite, hence by (F1)

$$U \in \bigcap_{i\in I} \mathscr{P}_f(X_i).$$

We now show U is *not* in the right-hand side of (†), contradicting (†).

- For any $p \in P$, we have $x_p \in U$ with $x_p \notin Y_p$, hence $U \not\subseteq Y_p$. Therefore, by (F2), $U \notin \mathscr{P}\big(Y_p\big)$ for all $p \in P$, and thus $U \notin \bigcup_{i\in P} \mathscr{P}(Y_i)$.
- For any $q \in Q$, by choice of t_q we have $t_q \notin Z_q$ or $t_q \notin W_q$, hence $U \not\subseteq Z_q$ or $U \not\subseteq W_q$; therefore $U \notin \mathscr{P}_f\big(Z_q\big)$ or $U \notin \mathscr{P}_f\big(W_q\big)$, so $U \notin \mathscr{P}_f\big(Z_q\big) \cap \mathscr{P}_f\big(W_q\big)$. Consequently, by (F3), $U \notin \bigcup_{i\in Q}\big(\mathscr{P}_f(Z_i) \cap \mathscr{P}_f(W_i)\big)$.

Thus U is in the left-hand side of (†) but in neither component of the right-hand side, contradicting (†). The contradiction shows that (5.3.4) is false; hence

$$\Big(\exists i_0 \in P.\ \bigcap_{i\in I} X_i \subseteq Y_{i_0}\Big) \vee \Big(\exists i_0 \in Q.\ \bigcap_{i\in I} X_i \subseteq Z_{i_0} \cap W_{i_0}\Big).$$

(⇐) We prove each disjunct implies the inclusion.
If $\exists i_0 \in P$ with $\bigcap_{i\in I} X_i \subseteq Y_{i_0}$, then by monotonicity of $\mathscr{P}_f(-)$ and (F1)–(F2),

$$\bigcap_{i\in I} \mathscr{P}_f(X_i) = \mathscr{P}\left(\bigcap_{i\in I} X_i\right) \subseteq \mathscr{P}\big(Y_{i_0}\big) \subseteq \bigcup_{i\in P} \mathscr{P}(Y_i),$$

hence the desired inclusion holds.
If $\exists i_0 \in Q$ with $\bigcap_{i\in I} X_i \subseteq Z_{i_0} \cap W_{i_0}$, then by monotonicity of $\mathscr{P}_f(-)$ and (F1) we have

$$\bigcap_{i\in I} \mathscr{P}_f(X_i) \subseteq \mathscr{P}_f\left(\bigcap_{i\in I} X_i\right) \subseteq \mathscr{P}_f\left(Z_{i_0} \cap W_{i_0}\right) = \mathscr{P}_f\left(Z_{i_0}\right) \cap \mathscr{P}_f\left(W_{i_0}\right) \subseteq \bigcup_{i\in Q} \mathscr{P}_f(Z_i) \cap \mathscr{P}_f(W_i)$$

yielding the desired inclusion.
Combining the two directions establishes the equivalence. □

Remark. The proof of the "⇒" direction only uses that the unions on the right are *finite* (so that one can collect finitely many counterexamples into a single finite set U) together with the finite-character readings of $\mathscr{P}((-))$ and $\mathscr{P}_f((-))$; this is precisely the "strongness" exploited in Lemmas 5.3.6 and 5.3.7.

Lemma 5.3.6.

$$\bigcap_{i\in I}(X_i \to Y_i) \cap (U \to V)^\star \subseteq (W \to Z) \Leftrightarrow \begin{cases} W \subseteq \bigcup_{i\in I} X_i \cup U \\ \forall J \subseteq I.(W \subseteq \bigcup_{j\in J} X_j) \vee (\bigcap_{j\in I\setminus J} Y_j \cap V \subseteq Z) \end{cases}$$

Proof. Write, for $A, B \subseteq \mathscr{D}$,

$$A \to B \;=\; \mathscr{P}_f\left(\overline{A \times \overline{B}^{\mathscr{D}_\Omega}}^{\mathscr{D}\times\mathscr{D}_\Omega}\right) \;=\; \mathscr{P}_f\left((\overline{A}^{\mathscr{D}} \times \mathscr{D}_\Omega) \cup (\mathscr{D} \times B)\right).$$

Thus, for every $d \in \mathscr{D}$ and $\delta \in \mathscr{D}_\Omega$,

$$(d,\delta) \in A \to B \text{ iff } \left(d \notin A\right) \text{ or } \left(\delta \in B\right). \tag{$\to$}$$

By Definition 5.3.1, since $\operatorname{cod}(U \to V) = V$ (Lemma 5.3.2),

$$(U \to V)^\star \;=\; (U \to V) \;\cap\; \mathscr{P}_f(\mathscr{D} \times (V \cup \{\Omega\})).$$

Set
$$\mathscr{L} \;:=\; \bigcap_{i\in I}(X_i \to Y_i) \;\cap\; (U \to V)^\star.$$

A useful computation (domain). For any arrow $A \to B$, using ($\to$) we have $(d,\Omega) \in A \to B$ iff $d \notin A$; hence

$$\mathtt{dom}(A \to B) = \{d \mid (d,\Omega) \notin A \to B\} = A.$$

Since $\mathscr{P}_f(-)$ is closed under intersections,

$$\mathtt{dom}(\mathscr{L}) = \{d \mid (d,\Omega) \notin \textstyle\bigcap_{i\in I}(X_i \to Y_i) \cap (U \to V)\} = \left(\bigcup_{i\in I} X_i\right) \cup U. \tag{$*$}$$

(⇒) **Necessity.** Assume $\mathscr{L} \subseteq W \to Z$.
(i) Domain side. Since $W \to Z$ forbids (d, Ω) for all $d \in W$, inclusion $\mathscr{L} \subseteq W \to Z$ forces $W \subseteq \mathtt{dom}(\mathscr{L})$. By $(*)$, this is $W \subseteq \big(\bigcup_{i\in I} X_i\big) \cup U$.
(ii) Codomain side. Fix $J \subseteq I$ and suppose, for contradiction, that $W \not\subseteq \bigcup_{j\in J} X_j$ and $\big(\bigcap_{j\in I\setminus J} Y_j \cap V\big) \not\subseteq Z$. Then pick $w \in W \setminus \big(\bigcup_{j\in J} X_j\big)$ and $z \in \big(\bigcap_{j\in I\setminus J} Y_j \cap V\big) \setminus Z$. Consider the partial function $R = \{(w, z)\}$. By construction and $(\to)$:

- For every $j \in J$, since $w \notin X_j$, we have $\{(w, z)\} \in X_j \to Y_j$.
- For every $i \in I \setminus J$, since $z \in Y_i$, we have $\{(w, z)\} \in X_i \to Y_i$.
- As $z \in V$, $\{(w, z)\} \in U \to V$, and also $(w, z) \in \mathscr{D} \times (V \cup \{\Omega\})$, so $\{(w, z)\} \in (U \to V)^\star$.

Hence $R \in \mathscr{L}$. But $w \in W$ and $z \notin Z$, so $R \notin W \to Z$, contradicting $\mathscr{L} \subseteq W \to Z$. Therefore, for each $J \subseteq I$,

$$\big(W \subseteq \textstyle\bigcup_{j\in J} X_j\big) \text{ or } \big(\textstyle\bigcap_{j\in I\setminus J} Y_j \cap V \subseteq Z\big).$$

(⇐) **Sufficiency.** Assume the two conditions in the statement. Let $R \in \mathscr{L}$ and fix $(w, o) \in R$ with $w \in W$. We prove $(w, o) \in W \to Z$, i.e., $o \neq \Omega$ and $o \in Z$.
First, $o \neq \Omega$: if $o = \Omega$, then $(w, \Omega) \in R \in \mathscr{L} \subseteq \bigcap_{i\in I}(X_i \to Y_i) \cap (U \to V)$ implies $w \notin X_i$ for all $i \in I$ (by $(\to)$) and $w \notin U$, hence $w \notin \big(\bigcup_{i\in I} X_i\big) \cup U$; this contradicts $w \in W \subseteq \big(\bigcup_{i\in I} X_i\big) \cup U$.
Now $o \in \mathscr{D}$. Define $J \subseteq I$ by $J := \{\, i \in I \mid o \notin Y_i \,\}$. Because by construction $(w, o) \in X_i \to Y_i$ for every $i \in I$, $(\to)$ yields $w \notin X_j$ for all $j \in J$, hence $w \notin \bigcup_{j\in J} X_j$. Applying the second condition (with this J), the left disjunct is impossible (since $w \in W$), so the right disjunct must hold:

$$\bigcap_{i\in I\setminus J} Y_i \cap V \subseteq Z.$$

But $(w, o) \in (U \to V)^\star$ forces $o \in V \cup \{\Omega\}$ and we already ruled out $o = \Omega$, hence $o \in V$. Moreover, by the choice of J, $o \in Y_i$ for all $i \in I \setminus J$. Therefore $o \in \big(\bigcap_{i\in I\setminus J} Y_i \cap V\big) \subseteq Z$, proving $(w, o) \in W \to Z$.
Since this holds for every $(w, o) \in R$ with $w \in W$, we have $R \in W \to Z$. As $R \in \mathscr{L}$ was arbitrary, $\mathscr{L} \subseteq W \to Z$.

Combining both directions yields the equivalence. □

Remark (*Proof by set decomposition*)

This lemma can also be proved using a technique similar to Lemma 4.9 from p.73 of Frisch's Thesis (2004), that is, by directly decomposing the intersection of finite parts of set product complements as a union.

Lemma 5.3.7.

$$\begin{aligned}&\bigcap_{i\in I}(X_i \to Y_i) \cap (U \to V)^\star \subseteq \mathscr{P}_f\,(\mathscr{D}\times(Z\cup\{\Omega\}))\\ \iff &\left(\bigcap_{i\in I} Y_i \cap V \subseteq Z\right) \wedge \left(\forall\, J \overset{J\neq\emptyset}{\subseteq} I\,.\left(\mathscr{D}\subseteq\bigcup_{j\in J} X_j\right) \vee \left(\bigcap_{j\in I\setminus J} Y_j \cap V \subseteq Z\right)\right)\end{aligned}$$

Proof. Recall the characterization of arrows (valid for $o \in \mathscr{D}\cup\{\Omega\}$):

$$(d,o) \in (A \to B) \quad\iff\quad (d \notin A)\ \vee\ (o \in B). \tag{5.3.5}$$

By Lemma 5.3.2, $\operatorname{cod}(U \to V) = V$, hence

$$(U \to V)^\star \;=\; (U \to V)\ \cap\ \mathscr{P}_f\,(\mathscr{D}\times(V\cup\{\Omega\}))\,.$$

Set

$$\mathscr{L} \;:=\; \bigcap_{i\in I}(X_i \to Y_i)\ \cap\ (U \to V)^\star.$$

Note that $R \in \mathscr{L}$ iff $R \subseteq \mathscr{D}\times(V\cup\{\Omega\})$ and, for every $(d,o)\in R$,

$$\forall i \in I.\,\big(d \notin X_i\ \vee\ o \in Y_i\big) \quad\text{and}\quad (d \notin U\ \vee\ o \in V),$$

(⇒) Assume $\mathscr{L} \subseteq \mathscr{P}_f\,(\mathscr{D}\times(Z\cup\{\Omega\}))$.
(1) Let $o \in \big(\bigcap_{i\in I} Y_i\big)\cap V$. For any $d \in \mathscr{D}$, by (5.3.5) we have for all i that $\{(d,o)\} \in X_i \to Y_i$ since $o \in Y_i$, and $\{(d,o)\} \in (U \to V)^\star$ since $o \in V$; hence $\{(d,o)\}\in\mathscr{L}$. The inclusion forces $o \in Z\cup\{\Omega\}$; but $o\in\mathscr{D}$, thus $o \in Z$. Therefore $\big(\bigcap_{i\in I} Y_i\big)\cap V \subseteq Z$.
(2) Fix a nonempty $J \subseteq I$. Suppose, towards a contradiction, that $\mathscr{D} \not\subseteq \bigcup_{j\in J} X_j$ and $(\bigcap_{j\in I\setminus J} Y_j)\cap V \not\subseteq Z$. Choose $d \in \mathscr{D}\setminus\bigcup_{j\in J} X_j$ and $o \in \big(\bigcap_{j\in I\setminus J} Y_j\big)\cap V\setminus Z$. Then, using (5.3.5):

$$\forall i \in I\setminus J,\ (d,o)\in X_i \to Y_i \text{ since } o \in Y_i; \qquad \forall j \in J,\ (d,o) \in X_j \to Y_j \text{ since } d \notin X_j;$$

and $(d,o) \in (U \to V)^\star$ because $o \in V$. Hence $(d,o) \in \mathscr{L}$ but $o \notin Z\cup\{\Omega\}$, contradiction. Therefore for every nonempty $J \subseteq I$,

$$\left(\mathscr{D}\subseteq\bigcup_{j\in J} X_j\right) \vee \left(\bigcap_{j\in I\setminus J} Y_j \cap V \subseteq Z\right).$$

(⇐) Assume the two conditions on the right-hand side hold. Let $R \in \mathscr{L}$ and $(d,o)\in R$. We prove $o \in Z\cup\{\Omega\}$.
If $o = \Omega$ there is nothing to show. Otherwise $o \in \mathscr{D}$ and, since $R \in (U \to V)^\star$, we have $o \in V$. If $o \in \bigcap_{i\in I} Y_i$, the first condition yields $o \in Z$. Otherwise set

$$J \;:=\; \{\, i \in I \mid o \notin Y_i\,\}.$$

Then $J \neq \emptyset$, and because $(d, o) \in X_i \to Y_i$ for all i, (5.3.5) gives $d \notin X_j$ for every $j \in J$, i.e. $d \notin \bigcup_{j\in J} X_j$. By the second condition (applied to this J), we must have $\left(\bigcap_{i\in I\setminus J} Y_i\right) \cap V \subseteq Z$ (the other disjunct fails at d). But by definition of J, $o \in \bigcap_{i\in I\setminus J} Y_i$, and we already know $o \in V$; hence $o \in Z$.
Thus $o \in Z \cup \{\Omega\}$ for every $(d, o) \in R$, so $R \subseteq \mathscr{D} \times (Z \cup \{\Omega\})$, i.e. $R \in \mathscr{P}_f(\mathscr{D} \times (Z \cup \{\Omega\}))$. Since $R \in \mathscr{L}$ was arbitrary, $\mathscr{L} \subseteq \mathscr{P}_f(\mathscr{D} \times (Z \cup \{\Omega\}))$.
Combining both directions establishes the equivalence. □

Now that we have established the set-theoretic characterizations of weak and strong arrows, we derive the corresponding subtyping solutions. We first establish those solutions for the case where the right-hand side is either a single weak arrow, or a single strong arrow. As shown previously for interpretation sets, this suffices to solve the subtyping problem entirely.

Theorem 5.3.8.

$$\bigwedge_{i\in I}(t_i \to s_i) \wedge (c \to d)^\star \leq (a \to b) \iff \begin{cases} a \leq \bigvee_{i\in I} t_i \vee c \\ \forall J \subseteq I. \quad \left(a \leq \bigvee_{j\in J} t_j\right) \vee \left(\bigwedge_{j\in I\setminus J} s_j \wedge d \leq b\right) \end{cases}$$

Proof. By application of Lemma 5.3.6. □

Theorem 5.3.9. *If $a \neq \mathbb{0}$, then*

$$\bigwedge_{i\in I}(t_i \to s_i) \wedge (c \to d)^\star \leq (a \to b)^\star \iff \begin{cases} a \leq \bigvee_{i\in I} t_i \vee c \\ \forall J \subseteq I. \left(\bigvee_{j\in J} t_j = \mathbb{1}\right) \vee \left(\bigwedge_{j\in I\setminus J} s_j \wedge d \leq b\right) \end{cases}$$

Proof. Combine Lemmas 5.3.6 and 5.3.7: we take the conjunction of their right-hand side conditions, and we observe that $\bigvee_{j\in J} t_j = \mathbb{1}$ implies $a \leq \bigvee_{j\in J} t_j$. □

Remark 5.3.10. *The side-condition $a \neq \mathbb{0}$ in Theorem 5.3.9 is necessary: when the domain is empty, any strong arrow has codomain $\mathscr{D}$ regardless of the target Z'. Equivalently in the set presentation, with $W = \emptyset$ we must take $Z = \mathscr{D}$ for any Z', so dropping $a \neq \mathbb{0}$ would make the statement false.*

5.3.2 Subtyping Algorithm

We can now derive the solutions to the subtyping problem for mixed weak and strong arrows on types.

Theorem 5.3.11 (Subtyping with weak/strong arrows)**.** *Let I, P, Q, R be finite. Write $t = \bigwedge_{i\in P} u_i$ and $s = \bigvee_{i\in P} w_i$. Then the mixed entailment*

$$\Big(\bigwedge_{i\in I}(t_i \to s_i)\Big) \wedge (t \to s)^\star \;\le\; \Big(\bigvee_{i\in R}(t_i \to s_i)\Big) \vee \Big(\bigvee_{i\in Q}(a_i \to b_i)^\star\Big)$$

holds iff at least one of the following finite witnesses *exists:*

(R-branch) *$\exists\, i_0 \in R$ such that*

$$\Big(\bigwedge_{i\in I}(t_i \to s_i)\Big) \wedge (t \to s)^\star \;\le\; (t_{i_0} \to s_{i_0}),$$

equivalently (by Theorem 5.3.8, with $a = t_{i_0}$, $b = s_{i_0}$, $c = t$, $d = s$):

$$\begin{cases} t_{i_0} \le \bigvee_{i\in I} t_i \vee t, \\ \forall J \subseteq I.\ \Big(t_{i_0} \le \bigvee_{j\in J} t_j\Big) \vee \Big(\bigwedge_{j\in I\setminus J} s_j \wedge s \le s_{i_0}\Big). \end{cases}$$

(Q-branch) *$\exists\, j_0 \in Q$ such that*

$$\Big(\bigwedge_{i\in I}(t_i \to s_i)\Big) \wedge (t \to s)^\star \;\le\; (a_{j_0} \to b_{j_0})^\star,$$

equivalently (by combining Theorem 5.3.9, with $a = a_{j_0} \ne \mathbb{0}$, $b = b_{j_0}$, $c = t$, $d = s$):

$$\begin{cases} a_{j_0} \le \bigvee_{i\in I} t_i \vee t, \\ \forall J \subseteq I.\ \Big(\bigvee_{j\in J} t_j = \mathbb{1}\Big) \vee \Big(\bigwedge_{j\in I\setminus J} s_j \wedge s \le b_{j_0}\Big). \end{cases}$$

Proof. *Apply the splitting Lemma 5.3.5 to reduce the problem to an existential witness on one target (R or Q); then discharge the resulting inclusion by Theorems 5.3.8 (weak target) and 5.3.9 (strong target).* □

Remark (*Q-branches must be seen at least once*)

In the Q-branch, the witness must satisfy $a_{j_0} \ne \mathbb{0}$; otherwise a strong-arrow target collapses to codomain $\mathscr{D}$ independently of b_{j_0}, matching the side-condition in Theorem 5.3.9.

Corollary 5.3.12 (One-strong target)**.** *For $a \neq \mathbb{0}$,*

$$(t \to s) \wedge (c \to d)^{\star} \;\leq\; (a \to b)^{\star} \quad\Longleftrightarrow\quad \begin{cases} a \leq t \vee c, \\ s \wedge d \leq b, \\ (t = \mathbb{1}) \vee (d \leq b). \end{cases}$$

Proof. *Instance of the Q-branch with $I = P = \emptyset$.* □

5.3.3 Specialization of strong arrows: strict domain and codomain

▸ **Theoretical Detour** While the rest of this section developed the theory of strong arrows and a decision algorithm for emptiness that makes them implementable, we now present a theoretical perspective that has not been completely developed but is nevertheless interesting and could be implemented by finding a suitable representation for the new types it describes.

In semantic subtyping the ordinary arrow $t_1 \to t_2$ asserts *only* that a value produced by applying the function to an argument of type t_1 is itself of type t_2. Outside t_1 the function may return anything, including diverging or raising an error. With gradual typing this underspecification allows imprecision to leak from the dynamic type ? and erode static information. We regain precision by enriching the type lattice with two orthogonal constructors:

(i) a *strict-domain* constructor, ensuring failures on inputs outside a designated domain, and
(ii) a *strict-codomain* constructor, guaranteeing that *every* successful result belongs to a designated range, irrespective of the input.

Conjoining either constructor with the usual arrow yields two function types: Dialyzer's *success types* (for the strict-domain constructor) and our *strong* function types (for the strict-codomain constructor).

Formalization. With usual notation for domain $\mathscr{D}$, we recall that a function type is represented by a *finite* graph $R \subseteq \mathscr{D}_{\mho} \times \mathscr{D}_{\Omega}$ such that: $R : t_1 \to t_2 \iff \forall(\iota,\delta) \in R.\ ((\iota : t_1) \vee (\iota = \mho)) \Longrightarrow \delta : t_2$ The surprising part of this definition is that it does not exclude *from the start* the functions that are defined outside of t_1, which is what enables type intersection. However, if we want, on a single function, we could ask that it *does not* represent functions undefined outside its domain. Moreso, we can forget about the single function, and define the *sets of all functions that are undefined outside of a given domain* t, call the type of all these functions $(t \Rightarrow)$, and define its interpretation as:

Definition 5.3.13 (Strict-domain constructor)**.**

$$f : (t \Rightarrow) \iff \forall(\iota,\delta) \in f.\ \iota \notin [\![t]\!] \Longrightarrow \delta = \Omega$$

In a similar way, we can consider the *set of all functions that only output values of type s*, call the type of all these functions $(\Rightarrow s)$, and define its interpretation as:

Definition 5.3.14 (Strict-codomain constructor)**.**

$$f : (\Rightarrow s) \quad\Longleftrightarrow\quad \forall(\iota,\delta) \in f.\, (\delta = \Omega) \vee (\delta : s)$$

Recreating success type and strong arrow. These types can be composed with the ordinary arrow to recreate two useful refinements

$$\underbrace{(t \to s) \wedge (t \Rightarrow)}_{\text{Dialyzer success type}}, \qquad \underbrace{(t \to s) \wedge (\Rightarrow s)}_{\text{strong arrow}}.$$

A *success type* guarantees that *whenever* the function returns, the argument was in t. A *strong* arrow (as seen previously) guarantees that *whenever* the function returns, the result is in s. The formula $(t \to s)^\star \overset{\text{def}}{=} (t \to s) \wedge (\Rightarrow s)$ could act as a new definition for the strong arrow.

5.4 Typed-Untyped interactions in Core Elixir

Our type system supports the integration of untyped code into a statically typed setting in the following way:

Using Strong Functions for Dynamic Arguments. When an untyped (dynamic) expression is used in a statically typed context, we employ strong function types to preserve as much type information as possible. For example, consider a function f with type

$$\textstyle\bigwedge_i (t_i \to s_i) \wedge (t \to s)^\star$$

where $(t \to s)^\star$ denotes a strong arrow type, with $s \leq \bigvee_i s_i$. If f is applied to a dynamically typed argument, our system can still infer the result type to be $? \wedge s$. In other words, even though the argument is dynamic, the result is refined to type s (along with the $?$ marker indicating potential imprecision). This incurs a loss of precision since we approximate the entire codomain of the intersection type but it allows some static type information to carry through dynamic calls.

Limitations of Inferring Strong Functions. A drawback of our approach is that it is easier to infer strong function types for values whose properties can be checked explicitly by the BEAM virtual machine's guards. For instance, simple base types like integers or atoms can be directly checked with built-in guards (such as `is_integer/1` or `is_atom/1`), making strong annotations for those functions straightforward. In contrast, complex types like "list of integers" cannot be fully validated by a single guard (since there is no built-in guard that checks an entire list's contents). As a result, functions operating on such types are hard to prove strong using our system. For example, consider a function that processes a list of integers:

```
# Intended type: list(integer) -> list(integer)
def process_integer_list(lst) do
  Enum.map(lst, fn x when is_integer(x) -> x * 2 end)
end
```

Listing 5.1: Function that cannot be strong due to lack of type checks

Operationally, this function is strong: the `Enum.map` will succeed only if every element of the list is an integer (the anonymous function explicitly guards x with `is_integer(x)`). However, our static analysis cannot easily prove its strongness because doing so would require hard-coding knowledge of `Enum.map`'s behaviour, specifically, that it stops with an error as soon as a non-integer is encountered. This hard-coding, which is possible, would not allow recognizing that the same function, when implemented with recursion, is also operationally strong.

```
# Intended type: list(integer) -> list(integer)
def process_integer_list(lst) do
  case lst do
    [] -> []
    [x | xs] -> [x * 2 | process_integer_list(xs)]
  end
end
```

Listing 5.2: Recursive version of a strong integer map

In general, our type system lacks a way to express "list of integers" as a guardable property, so it cannot infer a strong function type for `process_integer_list/1`. This illustrates a limitation: not every function that is semantically strong (i.e., that will never produce type errors at runtime if called in untyped code) can be assigned a strong function type in the type system.

Despite this limitation, a programmer can sometimes work around it by restructuring functions. For instance, if the goal is to ensure each element of a list is an integer before using it, the programmer can explicitly check elements within the function body. This approach may require extra flexibility, especially when interfacing with untyped libraries, but it enables more functions to be recognized as strong.

Refinement via Dynamic Intersection Types. Even when a function is not strong, our type system can utilize refined dynamic types to propagate static information. For example, suppose we have a function with the type

$$(\texttt{tuple} \to \texttt{tuple}) \land (\texttt{bool} \to \texttt{bool})$$

meaning it accepts either any tuple or any boolean and returns the same kind of value. If we have an expression of type $? \land \texttt{tuple}$ (a value that is dynamically typed but has been checked to be a tuple at runtime), we are allowed to pass it to the function. The result in that case will have type $? \land \texttt{tuple}$. In general, performing a runtime check (such as using a guard like `is_tuple/1`) refines the type of an expression from $?$ to a more specific $? \land t$. Once refined in this way, the static component t of the type can propagate through function calls. Thus, our system ensures that static type information is not lost when dynamic values are partially checked via guards.

According to the framework developed by Greenman et al. (2023), our approach can be viewed as a hybrid of *erasure* and *transient* semantics for gradual typing. Pragmatically, our system resembles that of an **Erasure** semantics: we use static types only at compile time and erase them before runtime, introducing no new runtime type checks of our own. However, we take advantage of the fact that the BEAM virtual machine already performs certain *tag checks* on values at runtime. These tag checks come in two forms: (a) **hardcoded checks in operations** – for example, the + operator in BEAM will only operate on numbers and will raise an error if given, say, an atom; function application itself requires that the callee is actually a function and that the correct arity is provided; and (b) **explicit guards in code** – developers can write guards (boolean expressions in clause heads or case statements) combining basic type tests (`is_integer/1`, `is_atom/1`, etc.), structural checks (like extracting a key from a map or an element from a tuple), and comparisons. These guards are richer than simple shape checks, since they can be arbitrary boolean combinations (`and`, `or`, `not`) of such predicates.

Our proposal is to **combine these existing runtime checks with ordinary function types** to infer strong function types in the static analysis. Intuitively, if the runtime is already ensuring certain type conditions, the compiler can use that knowledge to classify some functions as strong. A strong function type $(t \to s)^\star$ guarantees that when the function is applied to a dynamic input, the output will be of type $(? \wedge s)$. In essence, a strong function acts as a *certified runtime filter*: it promises that although the input might be dynamic, the output respects the target type s (with only a $?$ marker left to indicate potential uncertainty).

Thanks to the rich semantics we adopt for function types (supporting intersection of function types), a function in our system can simultaneously have both *ordinary* and *strong* type components. For example, a single function could have an intersection type $\bigwedge_i (t_i \to s_i)$ for its fully static behaviour, and additionally a strong arrow type $(\bigvee_i t_i \to \bigvee_i s_i)^\star$ to capture its behaviour on dynamic inputs. In practice, this means if the function is applied to a value that matches one of the expected domain types t_i, the type system will ensure a correspondingly refined output type s_i. If instead the function is applied to a value of unknown type (a dynamic), the strong arrow $(\bigvee_i t_i \to \bigvee_i s_i)^\star$ kicks in, and the output is conservatively typed as $? \wedge (\bigvee_i s_i)$. In all cases, the dynamic aspect of the value is tracked so that it is never mistakenly treated as purely static.

However, it is important to note the caveat highlighted by Greenman, Dimoulas, and Felleisen (2023) about transient semantics: not every useful type property in Elixir can be enforced by a guard and, therefore, not every function that is *operationally* strong can be assigned a strong type. For instance, as discussed, there is no general guard to ensure "this list contains only integers," which limits our ability to declare `process_integer_list/1` as strong in the type system. In contrast, certain functions operating on lists can still be proven strong if they explicitly check elements. Consider an alternative approach to the earlier example:

```
$ list(integer) -> integer
def safe_head([x | _]) when is_integer(x), do: x
def safe_head(_), do: :error
```

Listing 5.3: Function that can be strong with explicit element checking

This `safe_head/1` function is easier to recognize as strong. Its specified type is `list(integer) -> integer`, and indeed if it is called with a dynamic argument, it will always return an integer (in the case where the list's head is an integer) or the atom `:error` (if the head is not an integer). From the type system's perspective, during the check for strongness, the guard `is_integer(x)` refines the type of x to `? ∧ int`. Therefore, the type checker can verify that **whenever** `safe_head/1` is applied to a value of type `?` (an unknown list), it will produce a result of type `? ∧ int` (essentially, an integer at runtime, if it returns at all). In this way, `safe_head/1` meets the criteria for a strong function. This example demonstrates that by writing functions with explicit checks (in this case, examining just the head of the list), a programmer can achieve strong function behaviour even for list-processing functions. Of course, this often means returning a safe fallback (like `:error`) for inputs that do not meet the expected condition. Not every scenario allows such convenient checking, but when it is possible, it enables the static type system to enforce type implications across the boundary between static and dynamic code.

Before moving on, we clarify the safety guarantees we aim for in a gradually typed setting. There are essentially two properties of interest:

- **Static Type Safety:** Well-typed terms (under the static type system, denoted $\vdash_s$) do not go wrong. In other words, if a program is well-typed in the purely static sense, it will never raise a runtime type error (it may still diverge).
- **Gradual Soundness:** In the gradual system ($\vdash_g$), expressions behave in a way consistent with their type, even in the presence of dynamic code. The result of applying a strong function can be *trusted* to be in the codomain of the function type. And expressions evaluates to values whose types is shape-consistent with the type of the expression. This serves as the general reasoning principle of the system, rather than the static type system, which is only a special case (fully annotated programs).

The first property is the familiar slogan "(statically) typed code does not go wrong": a program typed under the static system $\vdash_s$ will not raise a runtime type error (it may still diverge). This guarantee applies to code checked only with static rules (that, we recall, can handle gradual types by subtyping). By contrast, the second property does not make such a promise: when dynamic code interacts with a function that is not strongly typed, the system preserves soundness by propagating dynamic uncertainty. Consequently, such interactions can still go wrong at runtime; the failures are attributable to the dynamic parts, not to the statically typed ones.

5.4.1 (Statically) Typed Code Does Go Wrong

Note

Disclaimer. The following remarks are reflexions intended for future work; they are exploratory rather than established results.

At first glance, our gradual system $\vdash_{\mathsf{g}}$,built on the sound gradual operators of Lanvin et al. (2021), appears capable of preserving some form of the *"(statically) typed code does not go wrong"* property. That is, we could expect that in the case that $\Gamma \vdash_{\mathsf{g}} e : t$ and t is a *static* type, then evaluating e will never raise a runtime type error. But while this work for the purely static subsystem ($\vdash_{\mathsf{s}}$) in Chapter4, it does not hold for the gradual system. We believe it is possible to modify the gradual system to recover this property by attaching the dynamic marker ? (to alert the type system) to any value that might cause a type error. If that mechanism works perfectly, a well-typed expression of a static type would carry no hidden dynamism that could lead to a type failure.

While our design correctly propagates the ? marker in many cases, there are scenarios where a well-typed expression e can have a sub-expression typed using dynamic rules, without '?' appearing in the type of e. We categorize our findings into "successes" (cases where our system prevents this) and "failure modes" (cases that where it does not).

5.4.1 a) Successes.

- **Dynamic arguments to a Strong Function:** When a dynamic value is passed as an argument to a strong function, the function's return type is appropriately marked as dynamic, preserving the possibility of failure. For example, consider an application $e = f(\overline{a})$ where f has a strong function type and the arguments $\overline{a}$ are dynamic. According to our typing rules (specifically the rule for dynamic application, often noted as ($\text{app}_{?}$)), the resulting type of e is $? \wedge (cod(f))$ - essentially the output type of f with a ? tag. This means the type system explicitly records that the result may not be fully trusted, thereby keeping the possibility of a failure visible in the type. The dynamic aspect of the inputs is not mistakenly hidden by the strong function; instead, it propagates to the output type.
- **Application with Refined Arguments:** Consider a function f with type $\mathsf{int} \to \mathsf{int}$ applied to an argument x of type $? \wedge \mathsf{int}$. In our type system, the application is analyzed by splitting the argument's type into its dynamic and static components. The computation (informally denoted by "$\circ$" and "$\vee$" for our type operators) would proceed as follows:

$$\big((\mathsf{int} \to \mathsf{int}) \circ (\mathbb{O} \wedge \mathsf{int})\big) \vee \big(? \wedge \big((\mathsf{int} \to \mathsf{int}) \circ (\mathsf{int})\big)\big) = ? \wedge \mathsf{int}$$

  The resulting type for $f(x)$ is $? \wedge \mathsf{int}$, showing that even though x had a static component (int), the presence of the ? marker on x forces the result to carry a ? marker as well. In simpler terms, if there's any uncertainty about the input at runtime, our type system ensures the output is treated with corresponding uncertainty. The dynamic marker is preserved through the function call, so the possibility of a type error (if x turned out not to be an integer at runtime) is never erroneously ruled out by the static type checker.

In those cases, whenever a dynamic value could introduce a runtime type mismatch, the system accounts for that risk by including ? in the result. Unfortunately, this dynamic propagation does not happen in all cases. We discovered two patterns of code where a runtime type error can be masked by a static type.

5.4.1 b) Failure modes.

Annotations that mask dynamism: Static type annotations can sometimes give a false sense of security by hiding dynamic aspects of a computation. For example, suppose we define a function

$$f = \lambda(\mathsf{int} \to \mathbb{1})\, y.\, \{\, 42, e' \,\}$$

where e' is some expression of type ? that will *certainly fail* at runtime. (For instance, imagine $g = \lambda(? \to ?)\, x.\pi_x(x)$, a contrived function that attempts an invalid operation and will throw a runtime error when called, and $e' = g(\texttt{true})$) The function f is annotated to take an int and return a term (a top-like supertype of all terms). Internally, f ignores its argument y and simply constructs a 2-tuple $\{\, 42, e' \,\}$. Now consider the call `f(42)`. According to our type system, this call type-checks successfully: 42 is an integer, and f is supposed to return a value, so the result is given type $\mathbb{1}$. Crucially, the type system sees no ? in the result type – it has been *masked* by the annotation that f returns a term. At runtime, however, evaluating `f(42)` will try to produce the tuple $\{\, 42, e' \,\}$ which entails evaluating e'. Since e' is a dynamic expression that fails (e.g., it might perform an illegal operation), the entire call `f(42)` will **crash with a runtime type error**. This contradicts the expectation that an expression typed as $\mathbb{1}$ (a static type) would not go wrong. The problem here is that the type annotation on f's return allowed a dynamically-typed failing expression to lurk inside a static type. Essentially, the annotation was accepted even though the body of f was less precise (contained more dynamism) than the annotation indicated.

Projection from Union-like Tuples: Another failure mode arises from how we compute types for projections (either from tuples, or maps) on union types. Consider a value e that comes from a pattern match or conditional, and as a result we know its type is the union of two possibilities:

$$\{?, \mathtt{bool}\} \vee \{\mathbb{1}, \mathtt{atom}\}$$

In words, in one scenario e is a tuple holding a dynamic value and a boolean; in another scenario e is a tuple holding a completely unconstrained value ($\mathbb{1}$) and an atom. Now suppose the program selects the index 0 from e (i.e., evaluates `elem(e,0)`. What type should this have? Using our type operator for projection, we compute:

$$\pi_0(t) = \pi_0(t^{\Downarrow}) \vee (? \wedge \pi_0(t^{\Uparrow}))$$

where $t^{\Downarrow}$ and $t^{\Uparrow}$ represent the static and dynamic components of the union type t. In this case, the calculation yields the type $\mathbb{1}$ (the top type) for $\pi_0(t)$. The ? marker gets lost in the join of the two alternatives. Intuitively, because one branch of the union said “the first element could be anything” ($\mathbb{1}$), the type system ends up concluding that $\pi_0(t)$ could be anything – and it drops the ? qualifier in the process. This means a value of type ? has effectively snuck into a position labeled as $\mathbb{1}$ without the ? tag. It is sound to do so, as $? \leq \mathbb{1}$; but it breaks the static safety, as the source of this dynamic value may cause an error, which the static type of the whole expression will have failed to inform us about.

5.4.1 c) Our proposed solution.

Given the above failures, we propose two pragmatic adjustments:

1) Using Annotations as Guides, Not Absolutes: We should restrict how type annotations are applied so that they cannot hide imprecise types. Specifically, when a programmer provides a type annotation for an expression, the compiler should not allow the found return type to be strictly smaller than the provided return type. In the earlier example of function f, this rule would mean that while declaring f's type as $(\texttt{int} \to \mathbb{1})$, the actual type assigned to the function (after seeing its definition) is $(\texttt{int} \to \{42, ?\})$. In other words, the annotation is only used to guide the expected input type (int), but the output is still marked as ? (specifically a tuple of 42 and something dynamic), which is directly found when synthesizing the type of the body of f. With this change, calling `f(42)` would yield a result of type $\{42, ?\}$ rather than a plain `term`, correctly signaling that the result may be coming from dynamic code and thus is not guaranteed safe.

2) Dynamic-Preserving Unions: We adjust the way union types are formed, during type-checking, so that a single dynamic alternative cannot be hidden by other static alternatives. Concretely, when we take the union of types $t_1 \dots t_n$ while typing, for instance, a case expression with n branches, if any one of the t_i is a dynamic type (that is, if $t_i{}^{\Uparrow} \setminus t_i{}^{\Downarrow} \neq \emptyset$, then the entire union is to be 'marked' as dynamic (by intersecting it with ?). In the previous example, if it came from such a union, the type of t would become

$$? \wedge (\{\mathbb{1}, \texttt{bool}\} \vee \{\mathbb{1}, \texttt{atom}\})$$

and now computing $\pi_0(t)$ yields ?. This way, the presence of a dynamic possibility in the union is preserved in the resulting type of the projection. This change prevents dynamic values from being hidden in union types.

Formal rule sketches We sketch two conservative tweaks that make the ideas above precise while preserving metatheory.

1) Annotations as guides (do not widen outputs by the annotation). Given an annotated function $\lambda^{\mathbb{I}} x.e$ with interface $\mathbb{I} = \{ t_i \to s_i \mid i \in I \}$, we type the body under the annotated inputs but keep the *found* codomain:

$$(\lambda_{\text{ann}})\ \frac{\forall (t_i \to s_i) \in \mathbb{I}.\ (\Gamma, x : t_i \vdash_{\mathsf{g}} e : s'_i)}{\Gamma \vdash_{\mathsf{g}} \lambda^{\mathbb{I}} x.e : \bigwedge_{i \in I}(t_i \to s'_i)}$$

Intuitively, we forbid the usual “widen-to-annotation” subsumption step $s'_i \leq s_i$ that would mask dynamism; instead we record s'_i.

Subject reduction (sketch). Reductions of $(\lambda^{\mathbb{I}} x.e)\ v$ step to $e[v/x]$. Since the assigned codomain is exactly the synthesized s'_i for e, preservation reduces to the usual substitution lemma for $\vdash_{\mathsf{g}}$ (cf. Lemma 5.2.10). Not widening the codomain cannot break preservation; it only strengthens the type.

2) Dyn-preserving unions. Define a predicate $\mathsf{hasDyn}(t)$ and an erasure $\mathsf{eraseDyn}(t)$ of top-level ? intersections:

$$\mathsf{hasDyn}(\tau) \text{ iff } \tau^{\Uparrow} \not\leq \tau^{\Downarrow};$$

Then the *dyn-aware join* $\bigvee^{\Delta}$ of a finite family $(t_k)_{k \in K}$ is

$$\bigvee\nolimits^{\Delta}_{k \in K} t_k \triangleq \begin{cases} ? \wedge \big(\bigvee\limits_{k \in K} t_k{}^{\Uparrow}\big) & \text{if } \exists k.\, \mathsf{hasDyn}(t_k), \\ \bigvee\limits_{k \in K} t_k & \text{otherwise.} \end{cases}$$

Use $\bigvee^{\Delta}$ in place of $\bigvee$ whenever unions are formed in gradual mode (e.g., case joins, if-then-else).

Subject reduction (sketch). Operational behaviour is unchanged; only the type assigned to a union becomes (at worst) more dynamic. Since $(? \wedge u) \leq u$ and $\mathsf{eraseDyn}(t) \leq t$, replacing $\bigvee$ by $\bigvee^{\Delta}$ yields a type below or equal to the original join. Preservation then follows from the original proof by subsumption monotonicity.

We conjecture that by adopting these two adjustments into our system we could obtain a refined behaviour for $\vdash_{\mathsf{g}}$. Given an expression e, if $\Gamma \vdash_{\mathsf{g}} e : t$ with t a *static* type, then this expression e will *not go wrong* in the same sense as when typed in $\vdash_{\mathsf{s}}$, that is, it will diverge or evaluate to a value of type t. We leave a detailed analysis of such an adjusted type system as future work.

Conclusion

This chapter introduced *safe-erasure gradual typing* for Core Elixir through a systematic development of four main components.

First, we presented the gradual type system (§ 5.1): we layered a gradual system $\vdash_{\mathsf{g}}$ on top of the static system $\vdash_{\mathsf{s}}$, added rules driven by consistent subtyping, and equipped the type language with *strong function types* $(t \to s)^{\star}$. The intent is to embrace ? without inserting casts or proxies in the compiled code, while preserving soundness and recovering as much static information as possible.

Second, we established the safety foundations (§ 5.2): we isolated the role of the ω-rules that model run-time failures and proved *progress*, *subject reduction*, and *Gradual Soundness* for the gradual judgment. Intuitively: if $\varnothing \vdash_{\mathrm{g}} e : t$ then evaluation either diverges, stops on a well-delimited ω error, or produces a value dynamically typable at t. This is the strongest guarantee compatible with erasing casts: success is preserved, failures remain explicit, and can be tied to specific use of unsafe typing rules.

Third, we developed the semantic theory for strong arrows (§ 5.3): we gave a relational account of strong arrows, introduced strict-domain/codomain constructors, and proved set-containment properties. This enabled decision procedures for subtyping in the presence of strong arrows through the subtyping algorithm (§ 5.3.2). We also hinted at how strong arrows arise as a composite of normal arrows with special arrows that enforce either their domain or their codomain (§ 5.3.3).

Fourth, we analyzed the practical integration challenges (§ 5.4): we clarified the limits of our gradual type system and its handling of dynamic interactions. Not every useful function can be inferred as strong; higher-order patterns and data-dependent behaviors often prevent that. Precision is intentionally sacrificed at the typed-untyped boundary: applications that *might* succeed receive ? (or $t \wedge ?$ under a strong arrow). We also showed that *statically typed code can go wrong* in the sense of § 5.4.1: hidden dynamism may still trigger ω at run time. The upside is that failures are *identified* (via ω) and information flows are explicit (via strong arrows and intersections), letting programmers recover precision with guards and ordinary refinements when needed.

In the next chapter, we move from explicit type tests ρ in pattern matching to develop a precise typed analysis for Elixir *patterns and guards*. As patterns and guards are *strong* (they check the type of the value they are applied to), the precision of this analysis will complement the strong arrow approach.

CHAPTER 6

TYPING GUARDS IN PATTERN MATCHING

Guards and pattern matching are Elixir's primary mechanism for refining control flow and data, subsuming the simple explicit type tests from the previous chapters. We formalize their core syntax and small-step semantics (including short-circuiting and error propagation), and design a guard analysis that computes, per clause, possibly- and surely-accepted types with an explicit exactness bit. These feed precise typing rules for case expressions, with static and ω-mode coverage guarantees, and support interface inference for multi-clause functions. Crucially, this precision drives the gradual system's strong function types: since guards encode actual dynamic checks, better guard analysis directly increases how many (and how precise) strong arrows we can infer.

Chapter Roadmap

- **Section 6.1 (Syntax and Semantics)**: Core Elixir pattern/guard syntax and operational semantics for matching and guard evaluation.
- **Section 6.2 (Typing Rules)**: accepted types and environment updates; setup for typing case-expressions.
- **Section 6.3 (Analysis $\rightarrow$ Accepted Types)**: guard-analysis judgment and its rules; derivation of (sure/possible) accepted types for case typing.
- **Section 6.4 (Soundness)**: key safety/necessity properties linking analysis to semantics.
- **Section 6.5 (Inference)**: interfaces inferred from guards for multi-clause functions; normalisation and gradual propagation.

$$
\begin{array}{llcl}
\textbf{Exprs} & e & ::= & \ldots \mid \texttt{case}\ e\ \overline{pg \to e} \mid \texttt{size}\ e \\
\textbf{Patterns} & p & ::= & c \mid x \mid \{\overline{p}\} \mid p \& p \\
\textbf{Guards} & g & ::= & a\,?\,\rho \mid a = a \mid a\ \texttt{!=}\ a \\
 & & \mid & a < a \mid g\ \texttt{and}\ g \mid g\ \texttt{or}\ g \\
\textbf{Atoms} & a & ::= & c \mid x \mid \{\overline{a}\} \mid \pi_a\, a \mid \texttt{size}\ a \\
\textbf{Tests} & \rho & ::= & c \mid b \mid \{\overline{\rho}\} \mid \{\overline{\rho}, ..\} \\
 & & \mid & \rho \vee \rho \mid \neg \rho
\end{array}
$$

(where variables occur at most once in each pattern)

Figure 6.1: Patterns and Guards

$$
\begin{array}{rll}
v/c = & \{\} & \text{if } v = c \\
v/x = & \{x \mapsto v\} & \\
\{v_1,..,v_n\}/\{p_1,..,p_n\} = & \bigcup_{i=1}^{n} \sigma_i & \text{if } v_i/p_i = \sigma_i \\
 & & \text{for all } i = 1..n \\
v/p = & \texttt{fail} & \text{otherwise} \\
v/(pg) = & \sigma & \text{if } v/p = \sigma \text{ and} \\
 & & g\sigma \hookrightarrow^{*} \texttt{true} \\
v/(pg) = & \texttt{fail} & \text{otherwise}
\end{array}
$$

Figure 6.2: Matching

6.1 Patterns and Guards in Elixir

In Elixir, patterns have the form of non-functional values (i.e. values in which no λ-abstraction occur) containing capture variables, whereas guards consist of complex expressions formed by boolean combinations (`and`, `or`, `not`) built on a limited set of expressions such as type tests (`is_integer`, `is_atom`, `is_tuple`, etc.), equality tests (`==`, `!=`), comparisons (`<`, `<=`, `>`, `>=`), data selection (`elem`, `hd`, `tl`, *map.key*), and size functions (`tuple_size`, `map_size`, `length`). The complete syntax for patterns and guards for (Featherweight) Elixir can be found in Section 13.1, Figure 13.2. To define our typed guard analysis, we introduce in Core Elixir a simplified syntax for patterns and guards given in Figure 6.1. The revised Core Elixir syntax for case expressions is $\texttt{case}\ e'\ \overline{pg \to e}$, where e' represents the expression that is matched and $\overline{pg \to e}$ denotes a list of branches. Each branch consists of a pair pg – formed by a pattern p and a guard g – and the corresponding expression e to be executed. Furthermore, we introduce a new expression $\texttt{size}\,e$ to calculate the size of a tuple, applicable in both expressions and guards, enhancing the complexity of guards to gauge the precision of our analysis.

Guards are constructed using three identifiers: guard atoms a, representing simplified expressions, type tests ($a\,?\,\rho$), and comparisons ($a = a$, $a \neq a$, $a < a$), which are combined using boolean operators `and` and `or`. The $a < a$ comparison uses the ordering of values $<_{\text{term}}$ derived from the total order on values used in Elixir (see Figure 6.4 for a formal definition). The test types ρ are extended to include union $\rho \vee \rho$ and negation $\neg\rho$, allowing for more expressive tests. For instance, it is now possible to verify whether a variable x is either an integer or a tuple ($x\,?\,\texttt{int} \vee \texttt{tuple}$) or to ascertain that it is not a tuple ($x\,?\,\neg\texttt{tuple}$).

This syntax closely resembles Elixir's concrete syntax; the main difference is that `not` is absent from guards, which is not restrictive: in Section 13.1 we outline a translation that encodes the `not` in the guards of Figure 6.1 by pushing negation into the leaves. In what follows, to improve readability, we sometimes use the syntax (p `when` g) to denote the pattern-guard pair pg.

The operational semantics defined in Figure 4.2 is extended with the rules in Figure 6.3 to account for pattern-matching and the size operator. The updated evaluation contexts and a new guard evaluation context are defined as follows:

$$\begin{array}{lll}
\textbf{Context} & \mathscr{E} & ::= \cdots \mid \texttt{size}\ \mathscr{E} \mid \texttt{case}\ \mathscr{E}\ \overline{pg \to e} \\
\textbf{Guard Context} & \mathscr{G} & ::= \square \mid \mathscr{G}\ \texttt{and}\ g \mid \mathscr{G}\ \texttt{or}\ g \mid \mathscr{G}\ ?\ t \mid \mathscr{G} < a \mid v < \mathscr{G} \\
 & & \mid \mathscr{G} = a \mid v = \mathscr{G} \mid \mathscr{G}\ \texttt{!=}\ a \mid v\ \texttt{!=}\ \mathscr{G}
\end{array}$$

This semantics relies on the functions for matching values to patterns v/p, and values to guarded patterns $v/(pg)$, as defined in Figure 6.2. When v is a value and p a pattern, v/p results in an environment σ that assigns the capture variables in p to corresponding matched values occurring in v. For example, v/x returns the environment where x is bound to v, and $\{v_1, v_2\}/\{x, y\}$ yields $x \mapsto v_1, y \mapsto v_2$. Similarly, $v/(pg)$ creates such an environment but also verifies that the guard g evaluates to `true` within the created environment; if v does not match p or the guard condition fails, the operation returns the token `fail`. For instance, $v/(x\ \texttt{when}\ x\ ?\ \texttt{int})$ results in $x \mapsto v$ if v is an integer and `fail` otherwise.

An important aspect of the semantics of pattern matching is that within a specific branch, if a reduction results in an error (i.e., reduces to an ω), the entire guard is considered to fail, and that branch is discarded. For instance, consider the guard $(\texttt{size}\ x = 2\ \texttt{or}\ x\ ?\ \texttt{int})$. If x is not a tuple, taking its size will lead to an error. Consequently, even if x is an integer, the guard will evaluate to `false` (as specified by rule [Context$_\omega$] in Figure 6.3), instead of `true` as could be expected from the disjunction. However, this does not imply a failure of the whole pattern matching expression since the failure of a guard is part of standard Elixir semantics: for instance consider the following definition:

```
1 def test2(x) when is_integer(elem(x,1)) or elem(x,0) == :int, do: elem(x,1)
2 def test2(x) when is_boolean(elem(x,0)) do: elem(x,0)
```

The application `test2({true})` makes the guard in line 1 fail, but the application succeeds returning `true`.

This behaviour is formalized by the Context rule in Figure 6.3, while other rules in this figure deal with Boolean operations, type tests, equality checks, and arithmetic operations within guard contexts. We use $\rightharpoonup$ for guard reductions, to distinguish them from expression reductions ($\hookrightarrow$).

It should also be noted that, in Elixir, both '`and`' and '`or`' operators exhibit short-circuit behaviour. Specifically, if the left-hand side of an `and`-guard evaluates to `false`, the right-hand side is not evaluated (as specified by rule [And$_\bot$]); similarly, if the left-hand side of an `or`-guard evaluates to `true`, the right-hand side is not evaluated (as defined by rule [Or$_\top$]).

6.2 Typing Rules

To type the expression $\texttt{case}\ e\ (p_i g_i \to e_i)_{i \le n}$ we aim to precisely type each branch's expression e_i by analyzing the set of values for which the pattern-guard pair $p_i g_i$ succeeds, that is, $\{v \in \textbf{Values} \mid v/p_i g_i \neq \texttt{fail}\}$. The best typing results are obtained when such a set coincides with the set of values of a given type. However, because of the presence of guards, not every such set of values corresponds perfectly to a type. For example, we have seen in line 155 the following guard:

$$
\begin{array}{llcll}
[\textsc{Case}] & \texttt{case}\ v\ \texttt{do}\ \big(p_i g_i \to e_i\big)_{i<n} & \hookrightarrow & e_j\sigma & \text{if } v/(p_j g_j) = \sigma \text{ and} \\
 & & & & \text{for all } i<j<n \quad v/(p_i g_i) = \texttt{fail} \\
[\textsc{Case}_\omega] & \texttt{case}\ v\ \texttt{do}\ \big(p_i g_i \to e_i\big)_{i<n} & \hookrightarrow & \omega_{\textsc{CaseEscape}} & \text{if } v/p_i g_i = \texttt{fail} \text{ for all } i<n \\
[\textsc{Size}] & \texttt{size}\ \{v_1,..,v_n\} & \hookrightarrow & n & \\
[\textsc{Size}_\omega] & \texttt{size}\ v & \hookrightarrow & \omega_{\textsc{Size}} & \text{if } v \neq \{\overline{v}\}
\end{array}
$$

$$
\begin{array}{lrcll}
[\textsc{And}_\top] & \texttt{true}\ \texttt{and}\ g & \hookrightarrow & g & \\
[\textsc{And}_\bot] & v\ \texttt{and}\ g & \hookrightarrow & \texttt{false} & \text{if } v \neq \texttt{true} \\
[\textsc{Or}_\top] & \texttt{true}\ \texttt{or}\ g & \hookrightarrow & \texttt{true} & \\
[\textsc{Or}_\bot] & \texttt{false}\ \texttt{or}\ g & \hookrightarrow & g & \\
[\textsc{Eq}_\top] & v = v' & \hookrightarrow & \texttt{true} & \text{if } v = v' \\
[\textsc{Eq}_\bot] & v = v' & \hookrightarrow & \texttt{false} & \text{else} \\
[\textsc{Lt}_\top] & v < v' & \hookrightarrow & \texttt{true} & \text{if } v <_{\text{term}} v' \\
[\textsc{Lt}_\bot] & v < v' & \hookrightarrow & \texttt{false} & \text{else} \\
[\textsc{NotEq}_\top] & v\ \texttt{!=}\ v' & \hookrightarrow & \texttt{true} & \text{if } v \neq v' \\
[\textsc{NotEq}_\bot] & v\ \texttt{!=}\ v' & \hookrightarrow & \texttt{false} & \text{else} \\
[\textsc{OfType}_\top] & v\,?\,t & \hookrightarrow & \texttt{true} & \text{if } v \in t \\
[\textsc{OfType}_\bot] & v\,?\,t & \hookrightarrow & \texttt{false} & \text{else} \\
[\textsc{Context}_g] & \mathscr{G}[g] & \hookrightarrow & \mathscr{G}[g'] & \text{if } g \hookrightarrow g' \\
[\textsc{Context}_a] & \mathscr{G}[a] & \hookrightarrow & \mathscr{G}[a'] & \text{if } a \hookrightarrow a' \\
[\textsc{Context}_\omega] & \mathscr{G}[a] & \hookrightarrow & \texttt{false} & \text{if } a \hookrightarrow \omega_p
\end{array}
$$

Figure 6.3: Pattern Matching and Guard Reductions

```
def test(x) when is_boolean(elem(x,0)) or elem(x,0) == elem(x,1), do: elem(x,0)
```

expressed in our formalism by the pattern-guard pair $(x\ \texttt{when}\ (\pi_0\,x\,?\,\texttt{bool})\ \texttt{or}\ (\pi_0\,x = \pi_1\,x))$—, which matches all tuples where either the first element is a Boolean, or the first two elements are identical: yet there does not exist a specific type that denotes exactly the set of all such tuples. To address this, given a pattern-guard pair pg, we approximate the set of values that match pg by defining two types termed as the *potentially accepted type* $\lbag pg \rbag$ (in our example it is $\{\texttt{bool},..\} \vee \{\mathbb{1},\mathbb{1},..\}$ since any value accepted by the pattern will belong to this type) and the *surely accepted type* $\rbag pg \lbag$ (here it is $\{\texttt{bool},..\}$ since all tuples starting with a Boolean are surely accepted). Through these types, we derive an approximating type t_i encompassing all values reaching e_i. When the matched expression is of type t, t_i is formulated as $(t \wedge \lbag p_i g_i \rbag) \setminus \bigvee_{j<i} \rbag p_j g_j \lbag$. In other words, the values in t_i are those that may be produced by e (i.e., those in t), and may be captured by $p_i g_i$ (i.e., those $\lbag p_i g_i \rbag$) and which are not surely captured by a preceding branch (i.e., minus those in $\rbag p_j g_j \lbag$ for all $j<i$). This type t_i can be utilized to generate the type environment under which e_i is typed. This environment, denoted as $t_i/_{p_i}$, assigns the deducible type of each capture variable of the pattern p_i, assuming the pattern matches a value in t_i. The definition of this environment is a standard concept in semantic subtyping and is detailed in Figure 6.6. A first coarse approximation of the typing discipline for pattern-matching expressions

$$\mathrm{rank}(v) = \begin{cases} 0 & v \text{ is an integer} \\ 1 & v \text{ is an atom} \\ 2 & v \text{ is a function value} \\ 3 & v \text{ is a tuple} \end{cases}$$

$$\begin{array}{rlcl}
\text{Integers:} & n <_{\text{intra}} n' & \iff & n < n' \\
\text{Atoms:} & a <_{\text{intra}} a' & \iff & \mathrm{str}(a) \text{ is lexicographically before } \mathrm{str}(a') \\
\text{Functions:} & f <_{\text{intra}} f' & \iff & f \text{ was created before } f' \\
\text{Tuples:} & (v_1, .., v_m) <_{\text{intra}} (v'_1, .., v'_{m'}) & \iff & m < m' \lor \\
 & & & (m = m' \land \exists i.\, v_i <_{\text{term}} v'_i \\
 & & & \quad \land \forall j < i.\, v_j = v'_j)
\end{array}$$

$$v <_{\text{term}} v' \;\text{ iff }\; \mathrm{rank}(v) < \mathrm{rank}(v') \lor (\mathrm{rank}(v) = \mathrm{rank}(v') \land v <_{\text{intra}} v')$$

Figure 6.4: Elixir-style value ordering

$$\begin{aligned}
\wr x \int_\Gamma &= \Gamma(x) \text{ if } x \in \mathtt{dom}(\Gamma) \\
\wr x \int_\Gamma &= \mathbb{1} \text{ if } x \notin \mathtt{dom}(\Gamma) \\
\wr \{ p_1, .., p_n \} \int_\Gamma &= \{ \wr p_1 \int_\Gamma, .., \wr p_n \int_\Gamma \} \\
\wr c \int_\Gamma &= c \\
\wr p_1 \,\&\, p_2 \int_\Gamma &= \wr p_1 \int_\Gamma \land \wr p_2 \int_\Gamma
\end{aligned}$$

Figure 6.5: Accepted types

If $t \leq \wr p \int$ then $t/_p$ is a map from the variables of p to types:

$$\begin{array}{lll}
t/_x(x) & = t & \\
t/_{\{p_1,\ldots,p_n\}}(x) & = t/_{p_i}(x) & \text{where } \exists i \text{ unique s.t. } x \in \mathrm{vars}(p_i) \\
t/_{p_1 \& p_2}(x) & = t/_{p_1}(x) & \text{if } x \in \mathsf{Vars}(p_1) \\
t/_{p_1 \& p_2}(x) & = t/_{p_2}(x) & \text{if } x \notin \mathsf{Vars}(p_1) \text{ and } x \in \mathsf{Vars}(p_2)
\end{array}$$

Figure 6.6: Typing Environments

is given by the following rule (given here only for presentation purposes but not included in the system):

$$(\text{case}_\omega(\text{coarse}))\ \frac{\Gamma \vdash e : t \quad (\forall i {\leq} n); (t_i \not\leq \mathbb{0} \Rightarrow \Gamma, t_i/_{p_i} \vdash e_i : s)}{\Gamma \vdash \mathsf{case}\ e\ (p_i g_i \to e_i)_{i \leq n} : s} \quad \begin{array}{l} t_i = (t \land \int p_i g_i \wr) \smallsetminus \bigvee_{j<i} \wr p_j g_j \int \\ t \leq \bigvee_{i \leq n} \int p_i g_i \wr \end{array}$$

$$\begin{array}{lll} \forall y \in \mathtt{dom}(\Gamma),\ \Gamma[x \mathrel{\hat{=}} t](y) & = \begin{cases} \Gamma(y) & \text{if } y \neq x \\ \Gamma(x) \wedge t & \text{if } y = x \end{cases} & \\ \Gamma[x \mathrel{\hat{=}} t]_p & = \big(\Gamma[x \mathrel{\hat{=}} t]\,,\, t'/p\big) & \text{where } t' = \rbag p \lbag_{\Gamma[x \mathrel{\hat{=}} t]} \end{array}$$

Figure 6.7: Environment Updates

The rule types a case-expression of n branches. For the i-th branch with pattern p_i and guard g_i, it computes t_i and produces the type environment ${}^{t_i}/_{p_i}$. This environment is used to type e_i only if $t_i \not\leq \mathbb{0}$ (i.e., t_i is not empty), thus ensuring that some values may reach the branch and checking for case redundancy. The side condition $t \leq \bigvee_{i<n} \lbag p_i g_i \rbag$ ensures that every value of type t (i.e., every possible result of e) may *potentially* be captured by some branch, addressing exhaustiveness. The rule is labeled with ω, indicating the emission of a warning, as the union $\bigvee_{i\leq n} \lbag p_i g_i \rbag$ might over-approximate the set of values captured by the case expression, risking run-time exhaustiveness failures. However, if $t \leq \bigvee_{i\leq n} \rbag p_i g_i \lbag$ holds, then there's no warning (cf. rule (case) later in this section), since all values in $\bigvee_{i\leq n} \rbag p_i g_i \lbag$ are captured by some pattern-guard pair, and so are those in t.

The typing rule for case expressions is actually more complicated than the one above, since it performs a finer-grained analysis of Elixir guards that is also used to compute their surely/potentially accepted types. Let us look at it in detail:

$$(\text{case}_\omega)\ \frac{\Gamma \vdash_{\mathsf{s}} e : t \qquad (\forall i{\leq}n)\ (\forall j{\leq}m_i)\ \ (t_{ij} \not\leq \mathbb{0} \ \Rightarrow\ \Gamma, {}^{t_{ij}}/_{p_i} \vdash_{\mathsf{s}} e_i : s)}{\Gamma \vdash_{\mathsf{s}} \mathtt{case}\ e\,\big(p_i g_i \to e_i\big)_{i\leq n} : s}\ t \leq \bigvee_{i\leq n} \lbag p_i g_i \rbag$$

$$\textit{where}\ \Gamma\,; t \vdash (p_i g_i)_{i\leq n} \rightsquigarrow (t_{ij}, \mathfrak{b}_{ij})_{i\leq n, j\leq m_i}$$

In contrast to the prior rule, the system now computes for each $p_i g_i$ pair, a list of m_i types $t_{i1}, ..., t_{im_i}$ which partitions the earlier t_i. The rule types each e_i expression m_i-times, each with a distinct environment ${}^{t_{ij}}/_{p_i}$. The t_{ij} at issue are derived from an auxiliary deduction system (formally defined in Section 6.3.3) $\Gamma; t \vdash (p_i g_i)_{i\leq n} \rightsquigarrow (t_{ij}, \mathfrak{b}_{ij})_{i\leq n, j\leq m_i}$. This system, detailed in the rest of this section, inspects each g_i for OR-clauses, and for each such clause it generates a pair $(t_i, \mathfrak{b}_i)$ that indicate the clause's type t_i and a Boolean flag $\mathfrak{b}_i$ indicating its exactness. For example, the guard $(\pi_0\, x \mathrel{?} \mathtt{bool}\ \mathtt{or}\ \pi_0\, x = \pi_1\, x)$ of our example is formed by two OR-clauses, and the analysis produces two pairs $(\{\,\mathtt{bool}\,,..\},\mathtt{true})$ and $(\{\,\mathbb{1},\mathbb{1}\,,..\},\mathtt{false})$: the first flag is $\mathtt{true}$ since the type for the first clause is exact ($\{\,\mathtt{bool}\,,..\}$ exactly contains all the values that match $\pi_0\, x \mathrel{?} \mathtt{bool}$); the second flag is $\mathtt{false}$ since the type is an approximation ($\{\,\mathbb{1},\mathbb{1}\,,..\}$ strictly contains all the values that match $\pi_0\, x = \pi_1\, x$). Likewise, the guard $(x \mathrel{?} \mathtt{int}\ \mathtt{or}\ \pi_0\, x \mathrel{?} \mathtt{int})$ will only produce exact types, namely $(\mathtt{int},\mathtt{true})$ and $(\{\,\mathtt{int}\,,..\},\mathtt{true})$. Guards are parsed in Elixir's evaluation order and potential clause failures. The analysis of guard g_i needs both Γ and p_i as it might use variables from either.

Given $\Gamma; t \vdash (p_i g_i)_{i\leq n} \rightsquigarrow (t_{ij}, \mathfrak{b}_{ij})_{i\leq n, j\leq m_i}$, the potentially accepted type for $p_i g_i$ is the union of all t_{ij}'s, while the surely accepted type for $p_i g_i$ is the union of all t_{ij}'s for which $\mathfrak{b}_{ij}$ is true. Thus, we have $\lbag p_i g_i \rbag = \bigvee_{j\leq m_i} t_{ij}$ and $\rbag p_i g_i \lbag = \bigvee_{\{j\leq m_i \mid \mathfrak{b}_{ij}\}} t_{ij}$. In our example, if g is the guard $(\pi_0\, x \mathrel{?} \mathtt{int}\ \mathtt{or}\ \pi_0\, x = \pi_1\, x)$, then $\lbag x g \rbag = \{\,\mathtt{bool}\,,..\} \vee \{\,\mathbb{1},\mathbb{1}\,,..\}$ and $\rbag x g \lbag = \{\,\mathtt{bool}\,,..\}$. Likewise, if g is the guard ($x\,?\,\texttt{int}$ or $\pi_0\,x\,?\,\texttt{int}$), then the potentially and surely accepted types of xg are the same, both being $\texttt{int} \vee \{\,\texttt{int}\,,..\,\}$, indicating that the approximation is exact. In this second case we can use a rule identical to the (case$_\omega$) rule but using a stricter side condition that ensures exhaustiveness and, thus, does not emit any warning:

$$\text{(case)}\ \frac{\Gamma \vdash_{\mathsf{s}} e : t \qquad (\forall i{\le}n)\,(\forall j{\le}m_i)\ \ (t_{ij} \not\leq \mathbb{0}\ \Rightarrow\ \Gamma, t_{ij}/_{p_i} \vdash_{\mathsf{s}} e_i : s)}{\Gamma \vdash_{\mathsf{s}} \texttt{case}\ e\,\big(p_i g_i \to e_i\big)_{i\le n} : s}\ t \leq \bigvee_{i\le n} \wr p_i g_i \int$$

$$\textit{where}\ \Gamma\,;t \vdash (p_i g_i)_{i\le n} \rightsquigarrow (t_{ij}, \mathfrak{b}_{ij})_{i\le n, j\le m_i}\ \text{and}\ \wr p_i g_i \int = \bigvee_{\{j\le m_i \,|\, \mathfrak{b}_{ij}\}} t_{ij}$$

When using this analysis, type safety depends on the side conditions used. Rule (case) with $t \leq \bigvee_{i\le n} \wr p_i g_i \int$ is safe for exhaustiveness, ensuring the same static guarantee as Theorem 4.3.7, as stated by the following updated version of the theorem:

Theorem 6.2.1 (Static Soundness)**.** *If $\varnothing \vdash_s e : t$ is derived without using ω-rules and with the (case) rule with condition $t \leq \bigvee_{i\le n} \wr p_i g_i \int$, then either $e \hookrightarrow^* v$ with $v : t$, or e diverges.*

Rule (case$_\omega$) will be used whenever the rule (case) fails—meaning that our guard analysis is not precise—raising a warning and adding $\omega_{\textsc{CaseEscape}}$ to the set of explicit runtime errors in Theorem 4.3.7, which becomes:

Theorem 6.2.2 (ω-Soundness)**.** *If $\varnothing \vdash_s e : t$ is derived using ω-rules and the (case) and (case$_\omega$) rules, then either $e \hookrightarrow^* v$ with $v : t$, or e diverges, or $e \hookrightarrow^* \omega_{\textsc{CaseEscape}}$ or $e \hookrightarrow^* \omega_{\textsc{OutOfRange}}$.*

These theorems will be proved in Section 6.4.3, after we introduce the necessary formalisation.

To complete the type system we need to add a rule for pattern matching expression in the gradual system:

$$\text{(case}_\star\text{)}\ \frac{\Gamma \vdash_{\mathsf{g}} e : t \qquad (\forall i{\le}n)\,(\forall j{\le}m_i)\ \ (t_{ij} \not\leq \mathbb{0}\ \Rightarrow\ \Gamma, t_{ij}/_{p_i} \vdash_{\mathsf{g}} e_i : s)}{\Gamma \vdash_{\mathsf{g}} \texttt{case}\ e\,\big(p_i g_i \to e_i\big)_{i\le n} : ?\wedge s}\ t \lesssim \bigvee_{i\le n} \int p_i g_i \wr$$

$$\textit{where}\ \Gamma\,;t \vdash (p_i g_i)_{i\le n} \rightsquigarrow (t_{ij}, \mathfrak{b}_{ij})_{i\le n, j\le m_i}\,\text{and}\int p_i g_i \wr = \bigvee_{j\le m_i} t_{ij}$$

The gradual rule (case$_\star$) simply checks that the scrutinee may be covered by some patterns, propagates the dynamic type, and does not modify the type safety of Theorem 5.2.15.

Theorem 6.2.3 (Gradual Soundness)**.** *If $\varnothing \vdash_g e : t$ using rules (case), (case$_\omega$), and (case$_\star$), then either $e \hookrightarrow^* v$ with $v \mathbin{⨟} t$, or e diverges, or there exists p such that $e \hookrightarrow^* \omega_p$.*

Remark (*Naive Type Narrowing*)

Let x be a variable of type $\mathbb{1}$ and consider the following code snippet:

$$\texttt{case}\,x\,(\{\,y,z\,,..\,\}\ \texttt{when}\ y\,?\,\texttt{int} \to \texttt{size}(x) + y, ...)$$

The current typing rules would reject this code since the `size` operator expects a tuple, but x is of type $\mathbb{1}$. This despite the fact that $\texttt{size}(x)$ will only be called if the pattern $\{\,y,z\,,..\,\}$, succeeds, implying x being a tuple. To solve this issue we need to refine the type of variables occurring in the matched expression. Given an expression e we define its skeleton $\texttt{sk}(e)$ as:

$$\begin{aligned} \texttt{sk}(x) &= x \\ \texttt{sk}(\{e_1,\dots,e_n\}) &= \{\texttt{sk}(e_1),\dots,\texttt{sk}(e_n)\} \\ \texttt{sk}(e) &= \mathbb{1} && \text{for any other expression} \end{aligned}$$

The skeleton of an expression is thus a pattern that matches the structure and variables of that expression while leaving out any functional parts. For example, the skeleton of a tuple formed by a variable and an application like $\{x, e(e_1,\dots,e_n)\}$ is $\{x, \mathbb{1}\}$ which is the pattern that captures in x the first projection matches any value in the second projection.

In practice, if e is being matched, its skeleton $\texttt{sk}(e)$ is added to the patterns of all branches. Thus, any type narrowing that occurs in their guard analysis is also applied to the variables of e. This is the justification for having intersections—noted &—in patterns: a value matches the intersection pattern $p_1 \,\&\, p_2$ iff it matches both p_1 and p_2; now every pattern p_i in $\texttt{case}\ e\ (p_i g_i \to e_i)_{i\leq n}$ can be compiled as $\texttt{sk}(e) \,\&\, p_i$. Then, we handle dependencies between variables (e.g., if pattern $x \,\&\, \{y, z, ..\}$ is followed by a guard $y\,?\,\texttt{int}$, then the type of x is refined to $\{\texttt{int}, \mathbb{1}, ..\}$), using an environment update $\Gamma[x \mathrel{\hat{=}} t]_p$ (see next section) that narrows the type of x in Γ to t, and uses pattern p to properly refine the type of other variables in Γ that depend on x. The pattern p can then simply be passed around alongside Γ in the guard analysis judgments. So in reality the guard analysis judgments should provide the information of the current pattern p under which a guard is being analyzed and, therefore, be of the form $\Gamma; p \vdash g \mapsto \mathscr{R}$, rather than $\Gamma \vdash g \mapsto \mathscr{R}$. Since it is a global dependency (no change is ever made to p) it is not necessary to propagate it in the typing rules. Furthermore its information is used in just two deduction rules (see rules [VAR] and [OR] in Section 6.4), therefore for clarity we omitted it everywhere apart from the single rule in which is used, since it would have cluttered the guard analysis rules. Nevertheless, it is a useful feature that avoids the problem described above, that we deal with in the guard analysis soundness proof (see Lemma 6.4.1), and that we implemented in the Elixir type checker.

6.3 Guard Analysis System

In this section we illustrate how to derive the judgment $\Gamma; t \vdash (p_i g_i)_{i\leq n} \rightsquigarrow (t_{ij}, \mathfrak{b}_{ij})_{i\leq n, j\leq m_i}$ that is central to typing case expressions and to infer function types from their guards. Our guard analysis derives two main judgments:

- *Guard Analysis Judgment*: $\Gamma \vdash g \mapsto \mathscr{R}$ represents the analysis of a single guard g under type environment Γ, producing a result $\mathscr{R}$ (see below) that describes the conditions under which the guard succeeds or fails.
- *Pattern-Guard Sequence Judgment*: $\Gamma; t \vdash (p_i g_i)_{i\leq n} \rightsquigarrow (t_{ij}, \mathfrak{b}_{ij})_{i\leq n, j\leq m_i}$ represent the analysis of a sequence of pattern-guard pairs under environment Γ with scrutinee type t, producing refined types and exactness flags for use in case expression typing.

The core of our guard analysis is the derivation the first judgments $\Gamma \vdash g \mapsto \mathscr{R}$, which associates to each guard a result $\mathscr{R}$ defined as follows:

$$\begin{array}{lrll} \textbf{Guard Results} & \mathscr{R} & ::= \overline{T} \mid \omega & \text{where } T ::= \{\mathscr{S}\,;\mathscr{T}\} \mid \{\mathscr{S}\,;\texttt{false}\} \\ \textbf{Environments} & \mathscr{S},\mathscr{T} & ::= (\Gamma, \mathfrak{b}) & \end{array}$$

6.3.1 Success Results

Whenever all the OR-clauses of a guard g may succeed, the guard analysis produces a list of *environment pairs* $\{\mathscr{S}\,;\mathscr{T}\}$, one for each OR-clause forming the guard. Each environment consists of a type environment Γ paired with a Boolean exactness flag $\mathfrak{b}$:

- On the left of each environment pair, the *safe environment* $\mathscr{S} = (\Gamma, \mathfrak{b})$ specifies a necessary condition on the types of the variables of the guard such that the guard clause does not error (i.e., does not evaluate to ω). When the flag $\mathfrak{b}$ of the environment is `true`, it means that the condition is also sufficient.
- On the right, the *success environment* $\mathscr{T} = (\Gamma, \mathfrak{b})$ specifies a necessary condition on the types of variables of the guard for the guard clause to evaluate to `true`. Again, $\mathfrak{b} =$ `true` indicates that this condition is also sufficient.

Example (*Imprecise Guards*)

For the guard $(x\,?\,\texttt{int})$ and $(x > y)$, comparisons never error in Elixir (total order), so the *safe environment* is $((x{:}\mathbb{1}, y{:}\mathbb{1}), \texttt{true})$. If the guard succeeds, then x is an integer, but the relational constraint $x > y$ cannot be captured by types; thus the *success environment* is $((x{:}\texttt{int}, y{:}\mathbb{1}), \texttt{false})$. In symbols:

$$\varnothing \vdash (x\,?\,\texttt{int}\,\texttt{and}\,x > y) \mapsto \{((x{:}\mathbb{1}, y{:}\mathbb{1}), \texttt{true}); ((x{:}\texttt{int}, y{:}\mathbb{1}), \texttt{false})\}.$$

When a guard is formed by several OR-clauses the analysis produces a list of environment pairs $\{\mathscr{S}\,;\mathscr{T}\}$, where each pair represents the way a clause may succeed taking into account the clauses that precede it. Concretely, the analysis of a disjunction guard g_1 or g_2 produces one pair describing the conditions under which g_1 succeeds, plus another pair describing the conditions where g_1 reduces to `false` *without erroring* and g_2 succeeds. The analysis can also produce a pair $\{\mathscr{S}\,; \texttt{false}\}$, corresponding to a clause that always evaluates to `false` within its safe environment $\mathscr{S}$. The presence of one such pair in a result does not indicate that the guard always fails, since other pairs in the list may be successful.

6.3.2 Failure Results

The cases for a faulty guard that either always errors (evaluates to ω) or always fails (evaluates to `false`) within its safe environment are captured by failure results $\mathscr{F}$.

$$\textbf{Failure Results} \qquad \mathscr{F} \quad ::= \quad \overline{\{\mathscr{S}\,; \texttt{false}\}} \mid \omega$$

Failure results are a subset of the guard results that, for presentation purposes, we single out by ranging them over with $\mathscr{F}$.

Rules Figure 6.4 gives the complete rules to derive $\Gamma \vdash g \mapsto \mathscr{R}$ judgments. Hereafter, we comment only on the most significant ones by unrolling a series of examples. Consider the guard $(\{\,x, y\,\}$ `when` $x\,?\,\texttt{tuple})$, that performs a type test on a capture *variable* x. The analysis of the guard $x\,?\,\texttt{tuple}$ is performed under the hypothesis that $x : \mathbb{1}$ and $y : \mathbb{1}$, since both capture variables do not have any further constraint (cf. rule [ACCEPT] at the end of this section). The guard analysis rule that handles this case is [VAR]

$$[\text{VAR}]\,\frac{\Gamma(x) \not\leq \rho \qquad \Gamma(x) \wedge \rho \neq \mathbb{0}}{\Gamma; p \vdash x \mathrel{?} \rho \mapsto \{(\Gamma, \texttt{true}); \big(\Gamma[x \mathrel{\hat=} \rho]_p, \texttt{true}\big)\}}$$

where the notation $\Gamma[x \mathrel{\hat=} t]$ denotes the environment obtained from Γ after refining the typing of x with t (i.e., ascribing it to $\Gamma(x) \wedge t$: see Figure 6.7 for the formal definition). This produces the judgment

$$(x{:}\mathbb{1}, y{:}\mathbb{1}) \vdash (x \mathrel{?} \texttt{tuple}) \mapsto \big\{\big((x{:}\mathbb{1}, y{:}\mathbb{1}), \texttt{true}\big) \,;\, \big((x{:}\texttt{tuple}, y{:}\mathbb{1}), \texttt{true}\big)\big\}$$

in which the first element of the result—the safe environment—leaves the type environment for variables unchanged, since this guard cannot error (paired with Boolean flag `true`, since this analysis is exact), while the second element—the success environment—contains $(x{:}\texttt{tuple}, y{:}\mathbb{1})$ indicating that the guard will succeed if and only if x is a tuple (and this condition is also sufficient as indicated again by the Boolean flag `true`).

If we refine the previous guard by adding a *conjunction*

$$\{\,x, y\,\}\ \texttt{when}\ (x \mathrel{?} \texttt{tuple})\ \texttt{and}\ (\texttt{size}\ x = 2)$$

then the new guard now specifically matches tuples of size 2. Its analysis is done by the following rule:

$$[\text{AND}]\,\frac{\Gamma \vdash g_1 \mapsto \{(\Phi_1, \mathfrak{b}_1); (\Delta_1, \mathfrak{c}_1)\} \quad \Delta_1 \vdash g_2 \mapsto \{(\Phi_2, \mathfrak{b}_2); (\Delta_2, \mathfrak{c}_2)\}}{\Gamma \vdash g_1\, \texttt{and}\, g_2 \mapsto \{\mathscr{S}; (\Delta_2, \mathfrak{c}_1 \,\&\, \mathfrak{c}_2)\}} \quad \mathscr{S} = \begin{cases}(\Phi_1, \mathfrak{b}_1) & \text{if } \mathfrak{b}_2 = \texttt{true} \text{ and } \Phi_2 = \Delta_1\\ (\Phi_2, \mathfrak{b}_1 \,\&\, \mathfrak{b}_2) & \text{otherwise}\end{cases}$$

In this rule, the success environment produced by the analysis of the first component $x \mathrel{?} \texttt{tuple}$ of the and (in our case, $\Delta_1 = (x : \texttt{tuple}, y : \mathbb{1})$) is used to analyze the second component ($\texttt{size}\ x = 2$), which is then handled by successive uses of the rules [EQ$_2$] and [SIZE]:

$$[\text{EQ}_2]\,\frac{\Gamma \vdash_{\mathsf{s}} a_2 : c \quad \Gamma \vdash a_1 \mathrel{?} c \mapsto \mathscr{R}}{\Gamma \vdash a_1 = a_2 \mapsto \mathscr{R}} \qquad [\text{SIZE}]\,\frac{\begin{array}{c}\Gamma \vdash a \mathrel{?} \texttt{tuple} \mapsto \{\_; (\Phi, \mathfrak{b})\}\\ \Phi \vdash a \mathrel{?} \texttt{tuple}^i \mapsto \{\_; \mathfrak{A}\}\end{array}}{\Gamma \vdash \texttt{size}\, a \mathrel{?} i \mapsto \{(\Phi, \mathfrak{b}) ; \mathfrak{A}\}}$$

where $\texttt{tuple}^i$ is the type of all the tuples of size i. Rule [EQ$_2$] corresponds to the best-case scenario of a guard equality: when one of the terms has a singleton type ($\Gamma \vdash a_2 : c$), a sufficient condition for both terms to be equal is that the other term gets this type as well ($\Gamma \vdash a_1 \mathrel{?} c$). In our example, this means doing the analysis $\Gamma \vdash \texttt{size}\ x \mathrel{?} 2$ with rule [SIZE]. This rule asks two questions (i.e., checks two premises): "can x be a tuple" (this produces a non-erroring environment), and "can x be a tuple of size 2?" (which in our case refines x to be of type $\{\,\mathbb{1}, \mathbb{1}\,\}$). The most general versions of these rules make approximations and can be found in Section 6.4 (Figure 6.11).

To go further, we can check that the second element of this tuple has type `int` by adding another conjunct to the guard

$$\{\,x, y\,\}\ \texttt{when}\ (x \mathrel{?} \texttt{tuple})\ \texttt{and}\ (\texttt{size}\ x = 2)\ \texttt{and}\ (\pi_1\, x \mathrel{?} \texttt{int})$$

Now, rule [PROJ] applies:

$$[\textsc{proj}]\ \frac{\Gamma \vdash_{\mathsf{s}} a' : i \quad \Gamma \vdash a \,?\, \texttt{tuple}^{>i} \mapsto \{\_ ; (\Phi, \mathfrak{b})\} \quad \Phi \vdash a \,?\, \{\, \overbrace{\mathbb{1}, \ldots, \mathbb{1}}^{i \text{ times}}, \rho\,, .. \,\} \mapsto \{\_ ; \mathfrak{A}\}}{\Gamma \vdash \pi_{a'}\, a \,?\, \rho \mapsto \{(\Phi, \mathfrak{b})\,;\, \mathfrak{A}\}}$$

where $\texttt{tuple}^{>i}$ represents tuples of size greater than i (e.g., $\texttt{tuple}^{>1} = \{\,\mathbb{1}, \mathbb{1}\,, ..\,\}$). This rule reads from left to right: after checking that the index a' is a singleton integer i (in our example, "1"), the non-erroring environment is computed by checking that the tuple has more than i elements. In our example, ($\texttt{size}\ x = 2$) has already refined x to be of type $\{\,\mathbb{1}, \mathbb{1}\,\}$. Finally, the success environment checks that the tuple is of size greater than i with t in i-th position (in our example, it has type $\{\,\mathbb{1}, \texttt{int}\,, ..\,\}$); since x was a tuple of size two, the intersection of those two types is $\{\,\mathbb{1}, \texttt{int}\,\}$.

In the case of a disjunction, a guard can succeed if its first component succeeds, or if the first fails (but does not error) and the second succeeds (guards being evaluated in a left-to-right order). Consider $\{\,x, y\,\}$ `when` ($x \,?\, \texttt{tuple}$) `and` ($\texttt{size}\ x = 2$) `or` ($y \,?\, \texttt{bool}$) whose analysis uses rule [OR]

$$[\textsc{or}]\ \frac{\begin{array}{c}\Gamma \vdash g_1 \mapsto \{(\Phi_1, \mathfrak{b}_1)\,;\, (\Delta_1, \mathfrak{c}_1)\} \\ \Gamma, {}^{t_i}/_{p} \vdash g_2 \mapsto \{(\Phi_2, \mathfrak{b}_2)\,;\, (\Delta_2, \mathfrak{c}_2)\}\end{array}}{\Gamma \vdash g_1 \,\texttt{or}\, g_2 \mapsto \{(\Phi_1, \mathfrak{b}_1)\,;\, (\Delta_1, \mathfrak{c}_1)\}, \{\mathscr{S}\,;\, (\Delta_2, \mathfrak{c}_1 \,\&\, \mathfrak{c}_2)\}} \quad \begin{array}{l} \mathscr{S} = \begin{cases} (\Phi_1, \mathfrak{b}_1) \text{ if } (\mathfrak{b}_2 = 1) \text{ and} \\ \qquad\qquad \left(\Phi_2 = \Gamma, {}^{t_i}/_{p}\right) \\ (\Phi_2, \mathfrak{b}_1 \,\&\, \mathfrak{b}_2) \ \text{otherwise} \end{cases} \\ t = \begin{cases} \llparenthesis p \rrparenthesis_{\Phi_1} \setminus \llparenthesis p \rrparenthesis_{\Delta_1} & \text{if } \mathfrak{c}_1 = 1 \\ \llparenthesis p \rrparenthesis_{\Phi_1} & \text{if } \mathfrak{c}_1 = 0 \end{cases} \end{array}$$

in which p is the pattern that $g_1 \,\texttt{or}\, g_2$ is guarded against. The first term of the `or`, that is, ($x \,?\, \texttt{tuple}$) `and` ($\texttt{size}\ x = 2$), is analyzed with rule [AND] given before, which produces $\big\{\big((x{:}\mathbb{1}, y{:}\mathbb{1}), \texttt{true}\big)\,;\, \big((x{:}\{\,\mathbb{1}, \mathbb{1}\,\}, y{:}\mathbb{1}), \texttt{true}\big)\big\}$. The second term, thus, is analyzed under the environment $(x{:}\neg\{\,\mathbb{1}, \mathbb{1}\,\}, y{:}\mathbb{1})$ which is obtained by subtracting the success environment of the first guard from its non-erroring one (i.e., we realize that, since tuples of size two make the first guard succeed, they will never reach the second guard). This is done by computing the type

$$t = \llparenthesis \{\,x, y\,\} \rrparenthesis_{x{:}\mathbb{1}, y{:}\mathbb{1}} \setminus \llparenthesis \{\,x, y\,\} \rrparenthesis_{x{:}\{\,\mathbb{1}, \mathbb{1}\,\}, y{:}\mathbb{1}} = \{\,\mathbb{1}, \mathbb{1}\,\} \setminus \{\,\{\,\mathbb{1}, \mathbb{1}\,\}, \mathbb{1}\,\} = \{\,\neg\{\,\mathbb{1}, \mathbb{1}\,\}, \mathbb{1}\,\}$$

where the notation $\llparenthesis p \rrparenthesis_{\Gamma}$ (defined in Figure 6.5) denotes the type of values that are accepted by a pattern p *and* which, when matched against p, bind the capture variables of p to types in Γ (e.g., $\llparenthesis \{\,x, y\,\} \rrparenthesis_{x{:}\texttt{int}, y{:}\texttt{bool}} = \{\,\texttt{int}, \texttt{bool}\,\}$). This choice of t is motivated by the fact that the analysis of the first term is *exact* (since the Boolean flag is `true`), therefore it is safe to assume that the values that make the first guard succeed, never end up in the second guard. Because this is a disjunction, the two ways that the guard succeeds are not mixed into a single environment, but split into two distinct solutions that are concatenated. Then, a little of administrative work on the Boolean flags ensures which results are exact and which ones are not.

So far our guards could not error, but it is a common feature in Elixir that guards that error short-circuit a branch of a case expression. For example, the guard

$$\{\,x, y\,\}\ \texttt{when}\ (\texttt{size}\ x = 2)\ \texttt{or}\ x \,?\, \texttt{bool}$$

only succeeds when the first projection of the matched value is a tuple of size two, and fails for all other values *including* when the first projection is a boolean (in which case `size` raises an error). This is handled by rule [OR] as well, by considering the non-erroring environment of a guard and using it as a base to analyze the second term of a disjunction. In our example, the non-erroring environment is $(x{:}\texttt{tuple}, y{:}\mathbb{1})$, and the second term is found instantly to be false. This will raise a warning, as a clause of a guard that only evaluates to false is a sign of a possible mistake.

6.3.3 Computation of accepted types

An accepted type $(t, \mathfrak{b})$ for a guarded pattern pg is a pair where t is a type and $\mathfrak{b}$ is a Boolean flag. If $\mathfrak{b}$ is `true`, then every value in t is accepted by pg, and every value accepted by pg is in t. If $\mathfrak{b}$ is `false`, then only the latter is true.

The guard analysis judgment $\Gamma \vdash g \mapsto \mathscr{R}$, is used to derive the judgment $\Gamma; t \vdash (p_i g_i)_{i\leq n} \leadsto (s_{ij}, \mathfrak{b}_{ij})_{i\leq n, j\leq m_i}$ that enumerates the accepted types to be used during the typing of a case expression. Only three rules suffice to derive this judgment: rule [ACCEPT] takes care of a single pattern-guard pair, and translates a list of possible success environments $(\Delta_i, \mathfrak{b}_i)_{i\leq n}$ into a list of pairs formed by an accepted type and its precision $(\wr p \int_{\Delta_i}, \mathfrak{b}_i)_{i\leq n}$. Guards that always fail are handled by rule [FAIL].

$$[\text{ACCEPT}]\ \frac{\Gamma, t/_p \vdash g \mapsto \{\_; (\Delta_i, \mathfrak{b}_i)\}_i}{\Gamma; t \vdash pg \leadsto \left(\wr p \int_{\Delta_i}, \mathfrak{b}_i\right)_i} \qquad [\text{FAIL}]\ \frac{\Gamma, t/_p \vdash g \mapsto \mathscr{F}}{\Gamma; t \vdash pg \leadsto (\mathbb{O}, \texttt{true})}$$

The sequence of successive guard-pattern pairs in a case expression is handled by [SEQ], which takes care to refine the possible types as the analysis advances, by subtracting from the potential type t the surely accepted types $\bigvee_{(s,\text{true})\in\mathscr{A}} s$ of the analysis of the current guard-pattern.

$$[\text{SEQ}]\ \frac{\Gamma; t \vdash pg \leadsto \mathscr{A} \qquad \Gamma; t \smallsetminus \left(\bigvee_{(s,\texttt{true})\in\mathscr{A}} s\right) \vdash \overline{pg} \leadsto \overline{\mathscr{A}}}{\Gamma; t \vdash pg\,\overline{pg} \leadsto \mathscr{A}\,\overline{\mathscr{A}}}$$

This last rule is then used to produce the auxiliary types $\Gamma; t \vdash (p_i g_i)_{i\leq n} \leadsto (t_{ij}, \mathfrak{b}_{ij})_{i\leq n, j\leq m_i}$ used in the typing rules for case expressions.

Most of the guard analysis rules have been explained in Section 6.3, and the workings of the other rules can be easily derived from them. Here we summarize the rules.

Figure 6.9 defines the main rules for guard analysis. These rules define how type information flows through complex guard expressions.

Figure 6.10 defines the boolean rules for guard analysis, which handle the logical combination of guards using AND and OR operators.

Figure 6.11 defines the approximation rules for guard analysis, which handle more complex guard expressions that involve projection operations and tuple size checks

Figure 6.12 defines the failure and false rules for guard analysis, which define cases where guards definitely fail or when guard evaluation results in false outcomes.

In Table 6.1 we number these rules for easy reference in the text. The numbering is consistent with the order of the rules in Figures 6.9, 6.10, 6.11, and 6.12.

6.4 Soundness of the Guard Analysis

Exhaustive Set of Guard Analysis Rules The safety analysis detailed in Chapter 4 is achieved on a calculus that uses type tests ρ instead of guarded patterns pg. The precise rules for pattern matching are introduced and explained in Section 6.2. We recall them in Figure 6.8. In order to prove the safety of these rules, we need to prove that the guard analysis we defined previously is correct and does not make unsafe type approximations. Doing that will require tying the operational semantics of guards to the guard analysis we defined previously. This will be the goal of Lemma 6.4.1, which will be used to establish Lemmas 6.4.3 and 6.4.2 that state that accepted types can be trusted and used to derive the type of variables within branches.

Lemma 6.4.1 (Safe/Success environment). *Let v be a value such that $v : t$ where $t \leq \wr p \int_\Gamma$. Let $\Gamma; p \vdash g \mapsto \mathscr{R}$ be a guard analysis judgment with $\Gamma \leq (t/p)$. Let $\{(\Phi, \mathfrak{b}) ; \mathfrak{A}\} \in \mathscr{R}$ where $\mathfrak{A}$ is either $(\Delta, \mathfrak{c})$ or `false`. Given $\sigma = v/p$, the following statements hold:*

$$(\Phi \leq \Gamma) \tag{6.4.1}$$

$$g\sigma \hookrightarrow^\star r \in \{\texttt{true}, \texttt{false}\} \implies v : \wr p \int_\Phi \tag{6.4.2}$$

$$\mathfrak{b} = \texttt{true} \text{ and } v : \wr p \int_\Phi \implies g\sigma \hookrightarrow^\star r \in \{\texttt{true}, \texttt{false}\} \tag{6.4.3}$$

If $\mathfrak{A} = (\Delta, \mathfrak{c})$ *then:*

$$(\Delta \leq \Phi) \tag{6.4.4}$$

$$g\sigma \hookrightarrow^\star \texttt{true} \implies (v : \wr p \int_\Delta) \tag{6.4.5}$$

$$\text{if } (\mathfrak{c} = \texttt{true}) \text{ then } (v : \wr p \int_\Delta) \implies g\sigma \hookrightarrow^\star \texttt{true} \tag{6.4.6}$$

If $\mathfrak{A} = \texttt{false}$ *then:*

$$g\sigma \hookrightarrow^\star \texttt{false} \tag{6.4.7}$$

If $\mathscr{R} = \omega$ *then:*

$$g\sigma \hookrightarrow^\star \mathtt{fail} \tag{6.4.8}$$

where $$\Gamma \leq \Gamma' \overset{def}{=} \forall x \in \texttt{dom}(\Gamma) \cap \texttt{dom}(\Gamma'), \Gamma(x) \leq \Gamma'(x)$$

Explanation. The purpose of Lemma 6.4.1 is to make a link between the operational semantics of guards (that reduces via $\hookrightarrow^\star$, which is the transitive closure of the guard small-step reduction $\hookrightarrow$ defined in 6.3) and the type of values that can be accepted by the guard. We have necessary conditions (6.4.2, 6.4.3) that say that, if a guard does not fail, or succeeds, then the value was of a given type. We have also sufficient conditions (6.4.5, 6.4.6) that say that, if a value was of a given type, then the guard will either not fail, or succeed. We maintain an ordering between type environments (6.4.1, 6.4.4) which allows us to ensure that, as the analysis progresses, types obtained are more and more precise and thus retain the properties obtained earlier in the refinement. For instance, if guard a ? `tuple` is found to succeed in environment Φ, and $\Delta \leq \Phi$, then $a\,?\,\texttt{tuple}$ will also succeed in environment Δ. Erroring and always failing guards are handled by additional conditions (6.4.8, 6.4.7).

Proof. We prove the four implications simultaneously by induction on the derivation $\mathscr{D} : \Gamma; p \vdash g \mapsto \mathscr{R}$. For every rule ρ of Figures 6.9-6.12 we assume, as induction hypothesis, that the lemma already holds for all premises of ρ, and show that it also holds for each $\{(\Phi, \mathfrak{b}) ; (\Delta, \mathfrak{c})\} \in \operatorname{concl}(\rho)$.

Operational notation. The small-step relation $v/(pg) \hookrightarrow^{\star} r$ evaluates the guard g on the value v previously matched by the pattern p and yields a result $r \in \{\texttt{true}, \texttt{false}, \texttt{fail}\}$. We write $v : t$ for the typing judgement $\varnothing \vdash_{\mathsf{s}} v : t$.

The proof proceeds by case analysis on the last rule used in $\mathscr{D}$.

Main guard analysis rules (1-7)

Rule 1: TRUE. $[\text{TRUE}]\,\dfrac{\Gamma \vdash_{\mathsf{s}} a : \rho}{\Gamma \vdash a\,?\,\rho \mapsto \{(\Gamma, \texttt{true}) ; (\Gamma, \texttt{true})\}}$

$g = a\,?\,\rho$.

(6.4.1) $(\Phi \leq \Gamma)$. $\Phi = \Gamma$ concludes.

(6.4.2) By hypothesis on v, we have $v : \wr p \int_{\Gamma}$. Since $\Phi = \Gamma$, we thus have $v : \wr p \int_{\Phi}$.

(6.4.3) Instead of proving $g\sigma \hookrightarrow^{\star} r \in \{\texttt{true}, \texttt{false}\}$, we prove below that $g\sigma \hookrightarrow^{\star} \texttt{true}$.

(6.4.4) $(\Delta \leq \Phi)$ since $\Delta = \Gamma$.

(6.4.5) By hypothesis on v, we have $v : \wr p \int_{\Gamma}$. Since $\Delta = \Gamma$, we thus have $v : \wr p \int_{\Delta}$.

(6.4.6) Given $\mathfrak{c} = \texttt{true}$, we want to prove $g\sigma \hookrightarrow^{\star} \texttt{true}$ where $\sigma = v/p$. Since $g = a\,?\,\rho$ and $\Gamma \vdash a : \rho$, this is trivial when $\sigma = \{\}$. Now, we want to know why this remains true when σ substitutes some variables inside of a. Let $x \in \texttt{fv}(p) \cap \texttt{fv}(a)$. We write $v' = \sigma(x)$ the value substituted into a. σ is obtained from v/p, by matching the parts of v with capture variables in p. Given that $v : t$, the types of the values of σ can be obtained by computing t/p. In particular, $v' : (t/p)(x)$. Our current environment Γ is, by hypothesis, a refinement of t/p. Thus, $\Gamma(x) \leq (t/p)(x)$. And given that $t \leq \wr p \int_{\Gamma}$, we also know the converse: that $(t/p)(x) \leq \Gamma(x)$. So we have $(\Gamma, x : (t/p)(x) \vdash_{\mathsf{s}} a : \rho)$ and $(\varnothing \vdash_{\mathsf{s}} v' : (t/p)(x))$. By substitution Lemma 4.3.3, we have $(\Gamma \vdash_{\mathsf{s}} [v'/x]a : \rho)$. Repeating this for every $x \in \texttt{fv}(p) \cap \texttt{fv}(a)$ yields $(\Gamma \vdash_{\mathsf{s}} a\sigma : \rho)$. Now, $(a\sigma)$ is statically typed with ρ, so by soundness of the static system 4.3.7 we know that it evaluates to a value v_a such that $v_a : \rho$. So $(a\sigma)\,?\,\rho$ reduces to $v_a\,?\,\rho$ which reduces to $\texttt{true}$ (by rule $\textsc{OfType}_{\top}$).

(6.4.7) Not applicable.

(6.4.8) Not applicable.

Rule 2: FALSE. $$[\text{FALSE}]\ \frac{\Gamma \vdash_{\mathsf{s}} a : s \qquad s \wedge \rho \simeq \mathbb{0}}{\Gamma \vdash a \mathrel{?} \rho \mapsto \{(\Gamma, \mathtt{true}) ; \mathtt{false}\}}$$
$g = a \mathrel{?} \rho$.
We prove each statement in order:

(6.4.1), (6.4.2) Clear since $\Phi = \Gamma$ and, by hypothesis, $v : \wr p \int_\Gamma$.

(6.4.3) We directly prove, below, that $g\sigma \hookrightarrow^\star \mathtt{false}$.

(6.4.4), (6.4.5) Clear since $\Delta = \Gamma$, thus $v : \wr p \int_\Delta$.

(6.4.6) As in the previous case for rule (true), by considering all variables $x \in \mathtt{fv}(a) \cap \mathtt{dom}(\sigma)$ we find that $\Gamma \vdash_{\mathsf{s}} a\sigma : s$. By soundness (Theorem 4.3.7), $a\sigma$ reduces to a value $v_a : s$. By inversion of (false), we have $s \wedge \rho \simeq \mathbb{0}$. Thus, $v_a \mathrel{?} \rho$ reduces to $\mathtt{false}$. Thus, $g\sigma \hookrightarrow^\star \mathtt{false}$.

(6.4.7) Not applicable.

(6.4.8) Not applicable.

Rule 3: VAR. $$[\text{VAR}]\ \frac{\Gamma(x) \not\leq \rho \qquad \Gamma(x) \wedge \rho \not\simeq \mathbb{0}}{\Gamma; p \vdash x \mathrel{?} \rho \mapsto \{(\Gamma, \mathtt{true}) ; \big(\Gamma[x \mathrel{\hat{=}} \rho]_p, \mathtt{true}\big)\}}$$
$g = x \mathrel{?} \rho$.

(6.4.1), (6.4.2) Clear since $\Phi = \Gamma$.

(6.4.3) The guard $x \mathrel{?} \rho$ cannot fail, so it reduces to $\{\mathtt{true}, \mathtt{false}\}$.

(6.4.4) By definition of an environment update, we have $\Gamma[x \mathrel{\hat{=}} \rho] \leq \Gamma$.

(6.4.5) We must have $x \in \mathtt{fv}(p)$ and $(x, v_x) \in \sigma$ where $v_x : (t/p)(x)$. If $(x \mathrel{?} \rho) \hookrightarrow^\star \mathtt{true}$, then $v_x \mathrel{?} \rho \hookrightarrow^\star \mathtt{true}$, i.e. $v_x : \rho$. We already knew that $v : \wr p \int_\Gamma$; refining Γ to be $\Gamma[x \mathrel{\hat{=}} \rho]$ means that x gets type $\Gamma(x) \wedge \rho$. So the part of v corresponding to x in p, which is v_x, has both type $\Gamma(x)$ and ρ. So $v_x : \wr p \int_{\Gamma[x \hat{=} \rho]}$.

(6.4.6) Conversely, if $v_x : \wr p \int_{\Gamma[x \hat{=} \rho]}$, then $v_x = (v/p)(x)$ is of type $\Gamma(x) \wedge \rho$, which makes the guard $v_x \mathrel{?} \rho$ evaluate to $\mathtt{true}$.

(6.4.7) Not applicable.

(6.4.8) Not applicable.

Rule 4: SIZE. $$[\text{SIZE}]\ \frac{\begin{array}{c}\Gamma \vdash a \mathrel{?} \mathtt{tuple} \mapsto \{\_ ; (\Phi, \mathfrak{b})\} \\ \Phi \vdash a \mathrel{?} \mathtt{tuple}^i \mapsto \{\_ ; \mathfrak{A}\}\end{array}}{\Gamma \vdash \mathtt{size}\, a \mathrel{?} i \mapsto \{(\Phi, \mathfrak{b}) ; \mathfrak{A}\}}$$
$g = \mathtt{size}\, a \mathrel{?} i$.

(6.4.1) By (IH 6.4.1 and 6.4.1) on the first premise and transitivity, $\Phi \leq \Gamma$.

(6.4.2) If $\mathtt{size}\, a \mathrel{?} i$ reduces to $\{\mathtt{true}, \mathtt{false}\}$, we must have $a \mathrel{?} \mathtt{tuple}$ that reduces to $\mathtt{true}$, because otherwise the guard reduction would $\mathtt{fail}$. Thus, by (IH 6.4.5), $v : \wr p \int_\Phi$.

(6.4.3) If $\mathfrak{b} = \mathtt{true}$ and $v : \wr p \int_\Phi$, then by (IH 6.4.6), we have $a\sigma \mathrel{?} \mathtt{tuple} \hookrightarrow^* \mathtt{true}$. Thus, $a\sigma$

reduces to a tuple value v', which suffices by definition of the reduction for size testing, to ensure that $\texttt{size}\, a\sigma \mathrel{?} i$ does not error, since $(\texttt{size}\, a)\sigma = \texttt{size}\,(a\sigma)$ reduces to $\texttt{size}\, v'$.

Case $\mathfrak{A} = (\Delta, \mathfrak{c})$

(6.4.4) By (IH 6.4.1) on the second premise, $\Delta \leq \Phi$.

(6.4.5) If $(\texttt{size}\, a \mathrel{?} i)\sigma$ reduces to $\texttt{true}$, $a\sigma$ necessarily reduces to a value v' which is a tuple of size exactly i, since the last reduction will check that v' has type $\texttt{tuple}^i$. Therefore, $(a \mathrel{?} \texttt{tuple}^i)\sigma$, which reduces to $v' \mathrel{?} \texttt{tuple}^i$, also reduces to $\texttt{true}$. Thus, by (IH 6.4.5), $v : \wr p \int_\Delta$.

(6.4.6) If $\mathfrak{c} = \texttt{true}$ and $v : \wr p \int_\Delta$,
then by (IH 6.4.6) on the second premise, $((a \mathrel{?} \texttt{tuple}^i)\sigma \hookrightarrow^\star \texttt{true})$, which implies that $((\texttt{size}\, a \mathrel{?} i)\sigma \hookrightarrow^\star \texttt{true})$

Case $\mathfrak{A} = \texttt{false}$

(6.4.7) In this case, by (IH 6.4.7), $(a \mathrel{?} \texttt{tuple}^i)\sigma$ reduces to $\texttt{false}$, hence $(\texttt{size}\, a \mathrel{?} i)\sigma$ reduces to $\texttt{false}$.

(6.4.8) Not applicable.

Rule 5: PROJ.

$$[\textsc{proj}]\ \frac{\Gamma \vdash_{\mathsf{s}} a' : i \quad \Gamma \vdash a \mathrel{?} \texttt{tuple}^{>i} \mapsto \{\_ ; (\Phi, \mathfrak{b})\} \quad \Phi \vdash a \mathrel{?} \{\overbrace{\mathbb{1}, \dots, \mathbb{1}}^{i \text{ times}}, \rho\, \texttt{,}\, ..\} \mapsto \{\_ ; \mathfrak{A}\}}{\Gamma \vdash \pi_{a'}\, a \mathrel{?} \rho \mapsto \{(\Phi, \mathfrak{b}) ; \mathfrak{A}\}}$$

$g = \pi_{a'}\, a \mathrel{?} \rho$.

(6.4.1) $(\Phi \leq \Gamma)$ by (IH 6.4.1 and 6.4.4) on the second premise and transitivity.

(6.4.2) If $(\pi_{a'}\, a \mathrel{?} \rho)\sigma$ reduces to $\{\texttt{true}, \texttt{false}\}$, then by the definition of the [PROJ] reduction rule, $a\sigma$ evaluates to a tuple of size at least i (since we know that $a'\sigma$ evaluates to i, since $\Gamma \vdash_{\mathsf{s}} a' : i$ in the first premise and the soundness of the type system). Thus, the guard $(a \mathrel{?} \texttt{tuple}^{>i})\sigma$ reduces to $\texttt{true}$, and by induction hypothesis (statement 6.4.5) we get $v : \wr p \int_\Phi$.

(6.4.3) If $\mathfrak{b} = \texttt{true}$ and $v : \wr p \int_\Phi$, then by induction hypothesis (statement 6.4.6), we get that $(a \mathrel{?} \texttt{tuple}^{>i})\sigma \hookrightarrow^\star \texttt{true}$, thus $a\sigma$ reduces to a tuple value of size at least i. This means that $(a \mathrel{?} \texttt{tuple}^{>i})\sigma$ does not error, and so $(\pi_{a'}\, a \mathrel{?} \rho)\sigma$ does not error either.

Case $\mathfrak{A} = (\Delta, \mathfrak{c})$

(6.4.4) By IH 6.4.1 on the third premise, we have $\Delta \leq \Phi$.

(6.4.5) If $(\pi_{a'}\, a \mathrel{?} \rho)\sigma$ reduces to $\texttt{true}$, then $a\sigma$ reduces to a tuple value of size more than i, with at position i a value v' of type ρ. Thus, $a\sigma \mathrel{?} \{\overbrace{\mathbb{1}, \dots, \mathbb{1}}^{i \text{ times}}, \rho\, \texttt{,}\, ..\}$ will also reduce to $\texttt{true}$. By IH 6.4.5, we get $v : \wr p \int_\Delta$.

(6.4.6) If $\mathfrak{c} = \texttt{true}$ and $v : \wr p \int_\Delta$, then by induction hypothesis on the third premise,

$(a\,?\,\{\overbrace{\mathbb{1},\ldots,\mathbb{1}}^{i\text{ times}},\rho\,\mathbf{,}\,..\})\,\sigma \hookrightarrow^\star \texttt{true}$, which implies that $(\pi_{a'}\,a\,?\,\rho)\,\sigma \hookrightarrow^\star \texttt{true}$.

Case $\mathfrak{A} = \texttt{false}$

(6.4.7) In this case, by IH, $(a\,?\,\{\overbrace{\mathbb{1},\ldots,\mathbb{1}}^{i\text{ times}},\rho\,\mathbf{,}\,..\})\,\sigma$ reduces to $\texttt{false}$. Since $a' : i$, it reduces to the integer i, thus $(\pi_{a'}\,a)$ reduces to $\pi_i\,a$. Now, by (IH 6.4.6) on the second premise, we know $a\,\sigma$ reduces to a tuple v_a of size larger than i. Thus $\pi_i\,v_a$. But we also know that, if v_a was a tuple of size larger than i with i-th element v_i, then the guard from the third premise would not reduce to $\texttt{false}$. Thus, v_i is not of type ρ, and $(\pi_{a'}\,a\,?\,\rho)\,\sigma$ reduces to $\texttt{false}$.

(6.4.8) Not applicable.

Rule 6: EQ$_1$. $[\text{EQ}_1]\ \dfrac{\Gamma \vdash_s a_1 : c \quad \Gamma \vdash a_2\,?\,c \mapsto \mathscr{R}}{\Gamma \vdash a_1 = a_2 \mapsto \mathscr{R}}$

$g = a_1 = a_2$.

Because the rule merely re-uses the analysis result of its guard premise $\Gamma \vdash a_2\,?\,c \mapsto \mathscr{R}$, each pair $\{(\Phi,\mathfrak{b});\mathfrak{A}\} \in \mathscr{R}$ appears unchanged in the conclusion. Everything to prove therefore comes from the induction hypotheses (IH) that already hold for that premise, together with static soundness for the well-typed term a_1.

(6.4.1) From the IH for the premise we already have $\Phi \leq \Gamma$.

(6.4.2) Suppose $(a_1 = a_2)\,\sigma \hookrightarrow^\star r \in \{\texttt{true},\texttt{false}\}$. The reduction sequence is $a_1\,\sigma \hookrightarrow c$ (no failure by static soundness of the typing judgement $\Gamma \vdash_s a_1 : c$), followed by the reduction of $a_2\,\sigma$ to a value (no failure, since $(a_1 = a_2)\,\sigma$ does not error). If the reduction of a_2 does not error, then neither does the reduction of $a_2\,?\,c$ and applying IH (6.4.2) to the second premise gives $v : \wr p \int_\Phi$.

(6.4.3) Assume $\mathfrak{b} = \texttt{true}$ and $v : \wr p \int_\Phi$. The IH for the premise gives $(a_2\,?\,c)\,\sigma \hookrightarrow^\star r_2 \in \{\texttt{true},\texttt{false}\}$. Again $a_1\,\sigma$ evaluates without failure (dynamic soundness), so the whole guard $(a_1 = a_2)\,\sigma$ does not error.

(6.4.4) If $\mathfrak{A} = (\Delta,\mathfrak{c})$ then the IH for the premise already provides $\Delta \leq \Phi$.

(6.4.5) Continuing under $\mathfrak{A} = (\Delta,\mathfrak{c})$: if $(a_1 = a_2)\,\sigma \hookrightarrow^\star \texttt{true}$ then, both $a_2\,\sigma$ and $a_1\,\sigma$ reduce to c. So, in particular, $(a_2\,?\,c)\,\sigma$ reduces to $\texttt{true}$; applying IH (6.4.5) yields $v : \wr p \int_\Delta$.

(6.4.6) Still under $\mathfrak{A} = (\Delta,\mathfrak{c})$: assume $\mathfrak{c} = \texttt{true}$ and $v : \wr p \int_\Delta$. IH (6.4.6) gives $(a_2\,?\,c)\,\sigma \hookrightarrow^\star \texttt{true}$. Since $a_1\,\sigma$ evaluates to a value $v_1 : c$, the final comparison succeeds, so $(a_1 = a_2)\,\sigma \hookrightarrow^\star \texttt{true}$.

(6.4.7) If $\mathfrak{A} = \texttt{false}$ the IH for the premise tells us $(a_2\,?\,c)\,\sigma \hookrightarrow^\star \texttt{false}$; which means that $a_2\,\sigma$ reduces to $v' \neq c$, hence $(a_1 = a_2)\sigma \hookrightarrow^\star \texttt{false}$.

(6.4.8) Straightforward by IH on the second premises since it implies that a_2 fails.

Rule 7: EQ$_2$. $[\text{EQ}_2]\,\dfrac{\Gamma \vdash_s a_2 : c \quad \Gamma \vdash a_1 ? c \mapsto \mathscr{R}}{\Gamma \vdash a_1 = a_2 \mapsto \mathscr{R}}$ This rule is symmetrical to EQ$_1$.

Rule 8: LT. $[\text{LT}]\,\dfrac{\Gamma \vdash_s a_0 : t_0 \quad \Gamma \vdash_s a_1 : t_1 \quad \mathsf{alwaysLess}(t_0, t_1)}{\Gamma \vdash a_0 < a_1 \mapsto \{(\Gamma, \mathtt{true}) ; (\Gamma, \mathtt{true})\}}$

$g = a_0 < a_1$.

(6.4.1) Straightforward since $\Phi = \Gamma$ concludes.

(6.4.2) By hypothesis, we already have $v : \wr p \int_\Gamma$.

(6.4.3) By static soundness of the typing judgements $\Gamma \vdash_s a_0 : t_0$ and $\Gamma \vdash_s a_1 : t_1$, as proven before, in the case for rule (TRUE), we have that $a_0\,\sigma$ and $a_1\,\sigma$ reduce to values v_0 and v_1 respectively. Thus, $g\,\sigma$ does not error.

(6.4.4) Straightforward since $\Delta = \Gamma$.

(6.4.5) We already have $v : \wr p \int_\Gamma$.

(6.4.6) We know $a_0\,\sigma$ and $a_1\,\sigma$ reduce to values v_0 and v_1 of types t_0 and t_1 respectively. The hypothesis that `alwaysLess`(t_0, t_1) implies that $v_0 < v_1$. Thus, $g\,\sigma$ reduces to `true`.

(6.4.7) Not applicable.

(6.4.8) Not applicable.

Rule 9: LTMAYBE. $[\text{LTMAYBE}]\,\dfrac{\Gamma \vdash_s a_0 : t_0 \quad \Gamma \vdash_s a_1 : t_1 \quad \mathsf{maybeLess}(t_0, t_1)}{\Gamma \vdash a_0 < a_1 \mapsto \{(\Gamma, \mathtt{true}) ; (\Gamma, \mathtt{false})\}}$

This proof is identical to the proof of LT, except for the condition (6.4.6) which is not applicable since $\mathfrak{c} = \mathtt{false}$.

Boolean rules

Rule 10: AND. $[\text{AND}]\,\dfrac{\begin{array}{c}\Gamma \vdash g_1 \mapsto \{(\Phi_i, \mathfrak{b}_i) ; (\Delta_i, \mathfrak{c}_i)\}_{i=1..n} \\ \forall i\ \Delta_i \vdash g_2 \mapsto \big\{(\Phi_{ij}, \mathfrak{b}_{ij}) ; (\Delta_{ij}, \mathfrak{c}_{ij})\big\}_{j=1..m_i}\end{array}}{\Gamma \vdash g_1 \,\mathsf{and}\, g_2 \mapsto \big\{\mathfrak{A}_{ij} ; (\Delta_{ij}, \mathfrak{c}_i \,\&\, \mathfrak{c}_{ij})\big\}_{ij}}$ $\mathfrak{A}_{ij} = \begin{cases} (\Phi_i, \mathfrak{b}_i) & \text{if } \mathfrak{b}_{ij} = \mathtt{true} \\ & \text{and } \Phi_{ij} = \Delta_i \\ (\Phi_{ij}, \mathfrak{b}_i \,\&\, \mathfrak{b}_{ij}) & \text{else} \end{cases}$

$g = g_1 \,\mathsf{and}\, g_2$.

Let $\{(\Phi, \mathfrak{b}) ; (\Delta, \mathfrak{c})\} \in \mathscr{R}$. Note that Φ could be obtained either from the first premise–corresponding to the case when, if the g_1 evaluates to `true`, then g_2 does not error, or from the second premise, which corresponds to the case when the safe environment depends both on g_1 and g_2. We distinguish the two cases by i) and ii). Let i_0, j_0 be the indices of the result picked, such that $\Delta = \Delta_{i_0 j_0}$.

(6.4.1) i) $(\Phi \leq \Gamma)$ by IH on the first premise.

ii) $(\Phi \leq \Gamma)$ by IH on the second premise.

(6.4.2) If $(g_1 \,\mathsf{and}\, g_2)\,\sigma$ reduces to $\{\mathtt{true}, \mathtt{false}\}$, then $g_1\,\sigma$ must reduce to $\{\mathtt{true}, \mathtt{false}\}$, so i) by IH on the first premise we obtain $v : \wr p \int_\Phi$. In the case of ii) then $v : \wr p \int_\Phi$ comes by (IH) on the second premise.

(6.4.3) If $\mathfrak{b} = \texttt{true}$ and $v : \lbag p \rbag_{\Phi}$, then

i) In this case, we want to prove that $(g_1 \,\texttt{and}\, g_2)\,\sigma$ reduces to $\{\texttt{true},\texttt{false}\}$ while knowing that $g_1\,\sigma$ reduces to $\{\texttt{true},\texttt{false}\}$ (by IH) and knowing that, due to the condition of the rule to pick the first premise and by (IH 6.4.3), if the guard succeeds then we are in an environment Δ_i which is a safe environment for g_2. Thus, $(g_1 \,\texttt{and}\, g_2)\,\sigma$ reduces to $\{\texttt{true},\texttt{false}\}$.

ii) In this case, we have a safe environment for g_2 Φ_{ij} which is a refinement of a Δ_i which is a success (thus, safe) environment for g_1. Thus, $g_1 \,\texttt{and}\, g_2$ is safe.

(6.4.4) We aim to prove that $\Delta \leq \Phi$.

i) In this case, $\Phi = \Phi_{i_0}$. By IH on the first premise, $\Delta_{i_0} \leq \Phi_{i_0}$. By IH on the second premise, $\Delta_{i_0 j_0} \leq \Delta_{i_0}$. Thus, by transitivity, $\Delta_{i_0 j_0} \leq \Phi_{i_0}$, which concludes.

ii) In this case, $\Phi = \Phi_{i_0 j_0}$. By direct IH on the second premise, $\Delta_{i_0 j_0} \leq \Phi_{i_0 j_0}$.

(6.4.5) Suppose $g\sigma \hookrightarrow^{\star} \texttt{true}$. We aim to prove that $v : \lbag p \rbag_{\Delta}$. Note that, if $g\sigma \hookrightarrow^{\star} \texttt{true}$, then $g_2\sigma \hookrightarrow^{\star} \texttt{true}$. Thus,

i) By IH on the second premise, $v : \lbag p \rbag_{\Delta_{i_0 j_0}}$.

ii) Same as for i).

(6.4.6) Suppose $\mathfrak{c} = \texttt{true}$ and $v : \lbag p \rbag_{\Delta}$. We want to prove that $g\sigma \hookrightarrow^{\star} \texttt{true}$. Both i) and ii) are similar. Since $\texttt{true} = \mathfrak{c} = \mathfrak{c}_{i_0} \mathbin{\&} \mathfrak{c}_{i_0 j_0}$, we have that $\mathfrak{c}_{i_0} = \texttt{true}$. To apply the (IH 6.4.6) on the first premise, we need to prove that $v : \lbag p \rbag_{\Delta_{i_0}}$. Since $\Delta_{i_0 j_0} \leq \Delta_{i_0}$, we have $v : \lbag p \rbag_{\Delta_{i_0 j_0}} \leq \lbag p \rbag_{\Delta_{i_0}}$. Thus $g_1\sigma \hookrightarrow^{\star} \texttt{true}$. We then apply the (IH 6.4.6) on the second premise, so $g_2\sigma \hookrightarrow^{\star} \texttt{true}$. Thus, $g\sigma \hookrightarrow^{\star} \texttt{true}$.

(6.4.7) Not applicable.

(6.4.8) Not applicable.

Rule 11:

$$[\textsc{or}]\ \frac{\begin{array}{c}\Gamma \vdash g_1 \mapsto \{(\Phi_i, \mathfrak{b}_i)\,;\,(\Delta_i, \mathfrak{c}_i)\}_{i=1..n} \\ \forall i\ \Gamma, {t_i}/{p} \vdash g_2 \mapsto \left\{(\Phi_{ij}, \mathfrak{b}_{ij})\,;\,(\Delta_{ij}, \mathfrak{c}_{ij})\right\}_{j=1..m_i}\end{array}}{\Gamma \vdash g_1 \,\texttt{or}\, g_2 \mapsto \{(\Phi_i, \mathfrak{b}_i)\,;\,(\Delta_i, \mathfrak{c}_i)\}_i \mathbin{@} \left\{\mathfrak{A}_{ij}\,;\,(\Delta_{ij}, \mathfrak{c}_i \mathbin{\&} \mathfrak{c}_{ij})\right\}_{ij}}$$

$$\mathfrak{A}_{ij} = \begin{cases} (\Phi_i, \mathfrak{b}_i) & \text{if } \mathfrak{b}_{ij} = \texttt{true} \\ & \quad \text{and } \Phi_{ij} = \Gamma, \left({t_i}/{p}\right) \\ (\Phi_{ij}, \mathfrak{b}_i \mathbin{\&} \mathfrak{b}_{ij}) & \text{else} \end{cases}$$

$$t_i = \begin{cases} \lbag p \rbag_{\Phi_i} \setminus \lbag p \rbag_{\Delta_i} & \text{if } \mathfrak{c}_i = \texttt{true} \\ \lbag p \rbag_{\Phi_i} & \text{if } \mathfrak{c}_i = \texttt{false} \end{cases}$$

$g = g_1 \,\texttt{or}\, g_2$.

(6.4.1) $(\Phi \leq \Gamma)$ by IH on both premises and transitivity.

(6.4.2) If $(g_1 \,\texttt{or}\, g_2)\,\sigma$ reduces to $\{\texttt{true},\texttt{false}\}$, then $g_1\,\sigma$ and $g_2\,\sigma$ must reduce to $\{\texttt{true},\texttt{false}\}$. By IH 6.4.2 on both premises, we get $v : \lbag p \rbag_{\Phi}$.

(6.4.3) If $\mathfrak{b} = \texttt{true}$ and $v : \lbag p \rbag_{\Phi}$, then by IH 6.4.3 on both premises, both $g_1\,\sigma$ and $g_2\,\sigma$ reduce to $\{\texttt{true},\texttt{false}\}$, so $(g_1 \,\texttt{or}\, g_2)\,\sigma$ reduces to $\{\texttt{true},\texttt{false}\}$.

<u>Case $\mathfrak{A} = (\Delta, \mathfrak{c})$</u>:

(6.4.4) $(\Delta \leq \Phi)$ by IH.

(6.4.5) If $(g_1 \,\texttt{or}\, g_2)\,\sigma$ reduces to `true`, then at least one of $g_1\,\sigma$ or $g_2\,\sigma$ reduces to `true`. By IH 6.4.5 on the corresponding premise, we get $v : \wr p \int_\Delta$.

(6.4.6) If $\mathfrak{c} = \texttt{true}$ and $v : \wr p \int_\Delta$, then by IH 6.4.6, at least one of $g_1\,\sigma$ or $g_2\,\sigma$ reduces to `true`, so $(g_1 \,\texttt{or}\, g_2)\,\sigma$ reduces to `true`.

<u>Case $\mathfrak{A} = \texttt{false}$</u>:

(6.4.7) : By IH, both $g_1\,\sigma$ and $g_2\,\sigma$ reduce to `false`, so $(g_1 \,\texttt{or}\, g_2)\,\sigma$ reduces to `false`.

(6.4.8) : Not applicable.

Approx rules

Rule 12: PROJ (approx). $$[\text{PROJ}_a]\;\frac{\begin{array}{c}\Gamma \vdash a' \,?\, \texttt{int} \mapsto \{\_ ; (\Phi, \mathfrak{b})\} \\ \Phi \vdash a \,?\, \texttt{tuple} \mapsto \{\_ ; (\Delta, \mathfrak{c})\}\end{array}}{\Gamma \vdash \pi_{a'}\, a \,?\, \rho \mapsto \{(\Delta, \texttt{false}) ; (\Delta, \texttt{false})\}}$$
$g = \pi_{a'}\, a \,?\, t$.

(6.4.1) By IH on the two premises we get $\Phi \leq \Gamma$ and $\Delta \leq \Phi$, thus by transitivity, $\Delta \leq \Gamma$.

(6.4.2) If $g\sigma$ terminates, then each premise guard terminates; by IH on the second premise we get $v : \wr p \int_\Delta$.

(6.4.3) Vacuous because $\mathfrak{b} = \texttt{false}$.

(6.4.4) Trivial: $\Delta = \Phi$.

(6.4.5) If $g\sigma \hookrightarrow^\star \texttt{true}$, then atom a' reduces to some integer i and atom a reduces to a tuple v_a of size at least i. Thus both guards $a' \,?\, \texttt{int}$ and $a \,?\, \texttt{tuple}$ reduce to `true`. By (IH 6.4.5) on the second premise, we get $v : \wr p \int_\Delta$.

(6.4.6) Vacuous because $\mathfrak{c} = \texttt{false}$.

(6.4.7) Not applicable.

(6.4.8) Not applicable.

Rule 13: EQ (approx). $$[\text{EQ}_a]\;\frac{\begin{array}{c}\Gamma \vdash a_0 \,?\, \mathbb{1} \mapsto \{\_ ; (\Phi, \mathfrak{b})\} \\ \Phi \vdash a_1 \,?\, \mathbb{1} \mapsto \{\_ ; (\Delta, \mathfrak{c})\}\end{array}}{\Gamma \vdash a_0 = a_1 \mapsto \{(\Delta, \mathfrak{b} \,\&\, \mathfrak{c}) ; (\Delta, \texttt{false})\}}$$
$g = (a_0 = a_1)$.

(6.4.1) By IH on the two premises we get $\Phi \leq \Gamma$ and $\Delta \leq \Phi$, thus by transitivity, $\Delta \leq \Gamma$.

(6.4.2) If $g\sigma$ terminates, then each premise guard terminates; by IH on the second premise we get $v : \wr p \int_\Delta$.

(6.4.3) If $\mathfrak{b} \,\&\, \mathfrak{c} = \texttt{true}$ and $v : \wr p \int_\Delta$, then by (IH 6.4.6) on the second premise, we get $a_1\,\sigma$ that reduces to $\{\texttt{true}, \texttt{false}\}$. Since $\Delta \leq \Phi$, then we also get $a_0\,\sigma$ that reduces to $\{\texttt{true}, \texttt{false}\}$. Thus, $(a_0 = a_1)\,\sigma$ reduces to $\{\texttt{true}, \texttt{false}\}$, since the equality guard does not fail when both sides reduce to $\{\texttt{true}, \texttt{false}\}$.

(6.4.4) Trivial: proven in case 6.4.1.

(6.4.5) If $g\sigma \hookrightarrow^{\star}$ `true`, then both atoms a_0 and a_1 reduce to the same value v. Thus both guards evaluate to `true`. By (IH 6.4.5) on the second premise, we get $v : \wr p \int_{\Delta}$.

(6.4.6) Vacuous because $\mathfrak{c} = \texttt{false}$.

(6.4.7) Not applicable.

Rule 14: SIZE (approx). $[\text{SIZE}_a]\ \dfrac{\Gamma \vdash a\,?\,\texttt{tuple} \mapsto \{\_;(\Delta,\mathfrak{c})\} \qquad \rho \wedge \texttt{int} \not\leq \mathbb{0}}{\Gamma \vdash \texttt{size}\,a\,?\,\rho \mapsto \{(\Delta,\mathfrak{c});(\Delta,\texttt{false})\}}$

$g = \texttt{size}\,a\,?\,t$.

(6.4.1) By IH we get $\Delta \leq \Gamma$.

(6.4.2) If $g\sigma$ terminates, then the guard $a\,?\,\texttt{tuple}$ terminates; by IH we get $v : \wr p \int_{\Delta}$.

(6.4.3) If $\mathfrak{c} = \texttt{true}$ and $v : \wr p \int_{\Delta}$, then by (IH 6.4.6), we get $a\sigma$ that reduces to `true`. Thus, $a\sigma$ reduces to a tuple, and the size guard terminates, and so does the type test.

(6.4.4) Trivial: proven in case 6.4.1.

(6.4.5) If $g\sigma \hookrightarrow^{\star}$ `true`, then atom a reduces to a tuple, thus the guard $a\,?\,\texttt{tuple}$ reduces to `true`. By (IH 6.4.5) on the second premise, we get $v : \wr p \int_{\Delta}$.

(6.4.6) Vacuous because $\mathfrak{c} = \texttt{false}$.

(6.4.7) Not applicable.

(6.4.8) Not applicable.

Rule 15: LT (approx). $[\text{LT}_a]\ \dfrac{\begin{array}{c}\Gamma \vdash a_0\,?\,\mathbb{1} \mapsto \{\_;(\Phi,\mathfrak{b})\} \\ \Phi \vdash a_1\,?\,\mathbb{1} \mapsto \{\_;(\Delta,\mathfrak{c})\}\end{array}}{\Gamma \vdash a_0 < a_1 \mapsto \{(\Delta,\mathfrak{b}\,\&\,\mathfrak{c});(\Delta,\texttt{false})\}}$

$g = a_0 < a_1$.

Same proof as for rule EQ (approx).

Failure rules

Rule 16: LT (fail). $[\text{LTFALSE}]\ \dfrac{\Gamma \vdash_{\mathsf{s}} a_0 : t_0 \quad \Gamma \vdash_{\mathsf{s}} a_1 : t_1 \quad \mathsf{neverLess}(t_0,t_1)}{\Gamma \vdash a_0 < a_1 \mapsto \{(\Gamma,\texttt{true});\texttt{false}\}}$

$g = a_0 < a_1$.

If $a_0\sigma$ and $a_1\sigma$ reduce to values v_0 and v_1 with types t_0 and t_1 respectively, and $\texttt{neverLess}(t_0,t_1)$ holds, then by construction of `neverLess`, we have that $v_0 \geq v_1$. Thus, $g\sigma$ reduces to `false`: the guard will never be satisfied whatever value p matches the pattern p.

Rule 17: SIZE$_\omega$. $[\text{SIZE}_\omega]\ \dfrac{\Gamma \vdash a\,?\,\texttt{tuple} \mapsto \mathscr{F}}{\Gamma \vdash \texttt{size}\,a\,?\,\rho \mapsto \omega}$

$g = \texttt{size}\,a\,?\,t$.

If $\mathscr{F} = \omega$, then $a\,?\,\texttt{tuple}\,\sigma$ reduces to `fail`, thus $a\sigma$ reduces to `fail`, thus $g\sigma$ reduces to `fail`.
If $\mathscr{F}$ contains some $\{\_ ; \texttt{false}\}$, then by (IH 6.4.7), we get $a\,?\,\texttt{tuple}\,\sigma$ that reduces to `false`. Thus, the size guard will never terminate on $a\sigma$, and so $g\sigma$ reduces to `fail`.

Rule 18: EQ$_\omega$. $[\text{EQ}_\omega]\ \dfrac{\Gamma \vdash a_i\ ?\ \mathbb{1} \mapsto \mathscr{F}}{\Gamma \vdash a_0 = a_1 \mapsto \omega}\ i \in \{0,1\}$
$g = a_0 = a_1$.
If either a_0 or a_1 reduces to `fail`, then $g\sigma$ reduces to `fail`.

Rule 19-20: PROJ$_\omega$ (tuple). $[\text{PROJ}^t_\omega]\ \dfrac{\Gamma \vdash a\ ?\ \texttt{tuple} \mapsto \mathscr{F}}{\Gamma \vdash \pi_{a'}\, a\,?\,\rho \mapsto \omega}$ $[\text{PROJ}^i_\omega]\ \dfrac{\Gamma \vdash a'\,?\,\texttt{int} \mapsto \mathscr{F}}{\Gamma \vdash \pi_{a'}\, a\,?\,\rho \mapsto \omega}$
$\underline{g = \pi_{a'}\, a\,?\,t}$. If either a' or a reduces to `fail`, then $g\sigma$ reduces to `fail`.

Rule 21: BOUND$_\omega$. $[\text{BOUND}_\omega]\ \dfrac{\Gamma \vdash_s a' : i \qquad \Gamma \vdash a\,?\,\texttt{tuple}^{>i} \mapsto \mathscr{F}}{\Gamma \vdash \pi_{a'}\, a\,?\,\rho \mapsto \omega}$
$g = \pi_{a'}\, a\,?\,t$.
If a never reduces to a tuple of size at least i, then $g\sigma$ always reduces to `fail`.

Rule 22: LT$_\omega$. $[\text{LT}_\omega]\ \dfrac{\Gamma \vdash a_i\ ?\ \mathbb{1} \mapsto \mathscr{F}}{\Gamma \vdash a_0 < a_1 \mapsto \omega}\ i \in \{0,1\}$
If either comparand always errors, then g always errors.

Rule 23: ORFALSE $[\text{ORF}]\ \dfrac{\begin{array}{c}\Gamma \vdash g_1 \mapsto \{(\Phi, \mathfrak{b}) ; \texttt{false}\} \\ \Phi \vdash g_2 \mapsto \mathscr{R}\end{array}}{\Gamma \vdash g_1\ \texttt{or}\ g_2 \mapsto \mathscr{R}}$
If a boolean guard always reduces to `false`, then the analysis of g_1 `or` g_2 always depends on the analysis of g_2.

Rule 24: OR$_\omega$. $[\text{OR}_\omega]\ \dfrac{\Gamma \vdash g_1 \mapsto \omega}{\Gamma \vdash g_1\ \texttt{or}\ g_2 \mapsto \omega}$ If g_1 always errors, then g_1 `or` g_2 always errors.

Rule 25: AND$_{\mathscr{F}}$-L (fail). $[\text{AND}_{\mathscr{F}}\text{-L}]\ \dfrac{\Gamma \vdash g_1 \mapsto \mathscr{F}}{\Gamma \vdash g_1\ \texttt{and}\ g_2 \mapsto \mathscr{F}}$
$g = g_1\ \texttt{and}\ g_2$.
If guard g_1 always fails or error, then both g_1 `and` g_2 and g_2 always fail or error.

Rule 26: AND$_{\mathscr{F}}$-R (all). $[\text{AND}_{\mathscr{F}}\text{-R}]\ \dfrac{\begin{array}{c}\Gamma \vdash g_1 \mapsto \{(\Phi_i, \mathfrak{b}_i) ; (\Delta_i, \mathfrak{c}_i)\}_{i \le n} \\ \forall i \le n\ \Delta_i \vdash g_2 \mapsto \mathscr{F}_i\end{array}}{\Gamma \vdash g_1\ \texttt{and}\ g_2 \mapsto \begin{cases}\omega & \text{if } \forall i, \mathscr{F}_i = \omega \\ \overline{\mathscr{F}}^j & j \text{ s.t. } \mathscr{F}_j \ne \omega\end{cases}}$

$g = g_1 \texttt{ and } g_2$.
If guard g_2 always fails or error, then guard $g_1 \texttt{ and } g_2$ always fails or error.

Since every rule has been considered, the induction is complete, proving the lemma.

□

Lemma 6.4.2 (Surely accepted types are sufficient)**.** *Given the guarded pattern sequence analysis* $\Gamma\,;t \vdash (p_i g_i)_{i\le n} \rightsquigarrow (t_{ij}, \mathfrak{b}_{ij})_{i\le n, j\le m_i}$*, for all* i, j *such that* $\mathfrak{b}_{ij} = \texttt{true}$ *and* $t_{ij} \not\le \mathbb{0}$*, for all value* v*,*

$$(v : t_{ij}) \implies \exists (i_0 \le i)\ s.t.\ (v/p_{i_0} g_{i_0} \ne \texttt{fail})$$

Proof. Since it is $[\textsc{accept}]\,\dfrac{\Gamma, t/p \vdash g \mapsto \{\_; (\Delta_i, \mathfrak{b}_i)\}_i}{\Gamma; t \vdash pg \rightsquigarrow \left(\wr p \int_{\Delta_i}, \mathfrak{b}_i\right)_i}$ that derives the types t_{ij} from environments Δ, we know that there is $(\Gamma, t/p) \vdash g \mapsto \{\_; (\Delta, \mathfrak{c})\}$ such that $t_{ij} = \wr p \int_\Delta$ and $\mathfrak{b}_{ij} = \mathfrak{c}$. By Lemma 6.4.1, statement 6.4.6, since $\mathfrak{b}_{ij} = \texttt{true}$ and $v : t_{ij} = \wr p \int_\Delta \le \wr p \int_{\Gamma, t/p}$, we know that the value v will necessarily be accepted by the pattern guard $(p_i g_i)$ (since $g\,(v/p_i)$ will reduce to $\texttt{true}$). Since pattern-matching is first-match, it could still be accepted by other branches, but it suffices to ensure that $(\exists i_0 \le i)\,;(v/(p_{i_0} g_{i_0}) \ne \texttt{fail})$.

□

Lemma 6.4.3 (Possibly accepted types are necessary)**.** *Given the guarded pattern sequence analysis* $\Gamma\,;t \vdash (p_i g_i)_{i\le n} \rightsquigarrow (t_{ij}, \mathfrak{b}_{ij})_{i\le n, j\le m_i}$*, for all* i, j *such that* $t_{ij} \not\le \mathbb{0}$*, for all value* v*,*

$$(v/(p_i g_i) \ne \texttt{fail}) \implies \exists j \le m_i\ s.t.\ (v : t_{ij})$$

Proof. Supposing $(v/(p_i g_i) \ne \texttt{fail})$ means that $v/p_i = \sigma$ and $g_i\sigma \hookrightarrow^\star \texttt{true}$. So $v/p_i = \sigma$ implies $v : \wr p_i \int_\Gamma$. By the rule ACCEPT, we know that $(\Gamma, t/p_i) \vdash g_i \mapsto \left\{\_; (\Delta_{ij}, \mathfrak{b}_{ij})\right\}_j$ where, for all i, j, $t_{ij} = \wr p_i \int_{\Delta_{ij}}$. Since $(\Gamma, t/p_i) \le (t/p_i)$, all the assumptions of Lemma 6.4.5 hold. Thus $v : \wr p_i \int_{\Delta_{ij}} = t_{ij}$.

□

6.4.1 Guard Comparison Analysis

Our system includes an advanced feature for analyzing comparison guards such as $a_1 < a_2$, $a_1 \le a_2$, $a_1 > a_2$, and $a_1 \ge a_2$. This analysis leverages the (total, but arbitrary) ordering on Elixir values to provide precise type information when comparison operations are used in guards.

The guard analysis rules for comparison operations are:

$$[\text{LT}]\ \frac{\Gamma \vdash_{\mathsf{s}} a_0 : t_0 \quad \Gamma \vdash_{\mathsf{s}} a_1 : t_1 \quad \mathsf{alwaysLess}(t_0, t_1)}{\Gamma \vdash a_0 < a_1 \mapsto \{(\Gamma, \mathtt{true}); (\Gamma, \mathtt{true})\}} \qquad [\text{LT}a]\ \frac{\begin{array}{c}\Gamma \vdash a_0 \mathbin{?} \mathbb{1} \mapsto \{\_; (\Phi, \mathfrak{b})\} \\ \Phi \vdash a_1 \mathbin{?} \mathbb{1} \mapsto \{\_; (\Delta, \mathfrak{c})\}\end{array}}{\Gamma \vdash a_0 < a_1 \mapsto \{(\Delta, \mathfrak{b} \,\&\, \mathfrak{c}); (\Delta, \mathtt{false})\}}$$

$$[\text{LTFALSE}]\ \frac{\Gamma \vdash_{\mathsf{s}} a_0 : t_0 \quad \Gamma \vdash_{\mathsf{s}} a_1 : t_1 \quad \mathsf{neverLess}(t_0, t_1)}{\Gamma \vdash a_0 < a_1 \mapsto \{(\Gamma, \mathtt{true}); \mathtt{false}\}}$$

These rules cover three scenarios for the comparison guard $a_1 < a_2$: always true when the maximum of a_1's type is strictly less than the minimum of a_2's type (first rule); a conservative approximation when the types overlap or the relationship is imprecise; and always false when the minimum of a_2's type is less than or equal to the maximum of a_1's type (third rule).

6.4.2 Strict Ordering of Types

We decide some comparison guards statically via a strict ordering on types, derived from Elixir's total term ordering in Figure 6.4.

Definition 6.4.4. *Type ordering.*
Define $t_1 <_{\text{type}} t_2$ iff for all $v_1 \in t_1$ and $v_2 \in t_2$ we have $v_1 <_{\text{term}} v_2$. We implement this with componentwise extrema and a global min/max:

- *Per-component candidates (with rank tokens $r_{\mathsf{int}} < r_{\mathsf{atom}} < r_{\mathsf{tuple}} < r_{\mathsf{fun}}$):*

$$(\min_{\mathsf{int}}(t), \max_{\mathsf{int}}(t)) \quad := \quad \begin{cases} (r_{\mathsf{int}}, r_{\mathsf{int}}) & t \wedge \mathtt{int} \not\leq \mathbb{O} \\ (\epsilon, \epsilon) & \textit{otherwise} \end{cases}$$

$$(\min_{\mathsf{atom}}(t), \max_{\mathsf{atom}}(t)) \quad := \quad \begin{cases} (\min A, \max A) & A = (t \wedge \mathtt{atom}) \textit{ finite, non-empty} \\ (r_{\mathsf{atom}}, r_{\mathsf{atom}}) & (t \wedge \mathtt{atom}) \textit{ cofinite} \\ (\epsilon, \epsilon) & \textit{otherwise} \end{cases}$$

$$(\min_{\mathsf{tuple}}(t), \max_{\mathsf{tuple}}(t)) \quad := \quad \begin{cases} (n_{\min}, n_{\max}) & t \wedge \mathtt{tuple} \textit{ has arities in } [n_{\min}, n_{\max}] \\ (r_{\mathsf{tuple}}, r_{\mathsf{tuple}}) & t \wedge \mathtt{tuple} \not\leq \mathbb{O} \\ (\epsilon, \epsilon) & \textit{otherwise} \end{cases}$$

$$(\min_{\mathsf{fun}}(t), \max_{\mathsf{fun}}(t)) \quad := \quad \begin{cases} (r_{\mathsf{fun}}, r_{\mathsf{fun}}) & t \wedge \mathtt{fun} \not\leq \mathbb{O} \\ (\epsilon, \epsilon) & \textit{otherwise} \end{cases}$$

- *Global extrema pick the least/greatest defined component by the rank order:*

$$\min(t) = \min\big(\{\min_c(t)\}_c \setminus \{\epsilon\}\big), \quad \max(t) = \max\big(\{\max_c(t)\}_c \setminus \{\epsilon\}\big).$$

- *Decide ordering by extrema: $t_1 <_{\text{type}} t_2$ iff $\max(t_1) <_{\text{term}} \min(t_2)$.*

$$\text{(case)}\ \frac{\Gamma \vdash e : t \quad (\forall i{\leq}n)\ (\forall j{\leq}m_i)\ \ (t_{ij} \not\leq \mathbb{0}\ \Rightarrow\ \Gamma, t_{ij}/p_i \vdash e_i : s)}{\Gamma \vdash \mathtt{case}\ e\,(p_i g_i \to e_i)_{i\leq n} : s}\ t \leq \bigvee_{i\leq n} \wr p_i g_i \int$$

$$\text{(case}_\omega\text{)}\ \frac{\Gamma \vdash e : t \quad (\forall i{\leq}n)\ (\forall j{\leq}m_i)\ \ (t_{ij} \not\leq \mathbb{0}\ \Rightarrow\ \Gamma, t_{ij}/p_i \vdash e_i : s)}{\Gamma \vdash \mathtt{case}\ e\,(p_i g_i \to e_i)_{i\leq n} : s}\ t \leq \bigvee_{i\leq n} \int p_i g_i \wr$$

$$\text{(case}_\star\text{)}\ \frac{\Gamma \vdash e : t \qquad (\forall i{\leq}n)\ (\forall j{\leq}m_i)\ \ (t_{ij} \not\leq \mathbb{0}\ \Rightarrow\ \Gamma, t_{ij}/p_i \vdash e_i : s)}{\Gamma \vdash \mathtt{case}\ e\,(p_i g_i \to e_i)_{i\leq n} : {?} \wedge s}\ t \,\tilde{\leq} \bigvee_{i\leq n} \int p_i g_i \wr$$

where $\Gamma ; t \vdash (p_i g_i)_{i\leq n} \rightsquigarrow (t_{ij}, \mathfrak{b}_{ij})_{i\leq n, j\leq m_i}$ and $\int p_i g_i \wr = \bigvee_{j\leq m_i} t_{ij}$ and $\wr p_i g_i \int = \bigvee_{\{j\leq m_i \mid \mathfrak{b}_{ij}\}} t_{ij}$

Figure 6.8: Case Typing Rules

Predicates used in rules. Write

$$\begin{aligned}
\mathsf{alwaysLess}(t_1, t_2) &\equiv t_1 <_{\text{type}} t_2 \\
\mathsf{neverLess}(t_1, t_2) &\equiv t_2 <_{\text{type}} t_1 \\
\mathsf{maybeLess}(t_1, t_2) &\equiv \neg\mathsf{alwaysLess} \wedge \neg\mathsf{neverLess}
\end{aligned}$$

This ordering is independent from subtyping (e.g., `int` $<_{\text{type}}$ `atom` while `int` $\not\leq$ `atom`) and is only used to decide the comparison-guard rules.

6.4.3 Typing with Guards: Safety

Given the results of the previous section, the theorem for type soundness can be extended to the case with guards.

Theorem 6.4.5 (Static Soundness)**.** *If* $\varnothing \vdash_s e : t$ *is derived without using* ω*-rules and with the (case) rule with condition* $t \leq \bigvee_{i\leq n} \wr p_i g_i \int$*, then either* $e \hookrightarrow^* v$ *with* $v : t$*, or* e *diverges.*

Proof. Proving this theorem requires updating the proofs of progress (4.3.4) and preservation (4.3.6). Here is a sketch of the proof:

1. For progress, if e is a case `case` $e\ \overline{pg \to e}$ that reduces to $\omega_{\text{CASEESCAPE}}$, then for all $i \leq n$, $v/p_i g_i =$ `fail`. This contradicts the fact that every $v \in \bigvee_{i\leq n} \wr p_i g_i \int$ is a surely accepted value, i.e. should be accepted by at least one guarded-pattern per Lemma 6.4.2
2. Preservation comes almost immediately from rule (case): since every branch is typed with the same type from the whole case (this is enabled by subtyping), a case reduction will preserve the typing. Making sure that we always reduce into a well-typed expression comes from Lemma 6.4.3 which ensures that our value (which is captured and substituted into the branch) is of type t_{ij} for some i, j; thus the Substitution Lemma 4.3.3 applies and the branch is well-typed. □

Theorem 6.4.6 (ω-Soundness)**.** *If $\varnothing \vdash_s e : t$ is derived using ω-rules and the (case) and (case$_\omega$) rules, then either $e \hookrightarrow^* v$ with $v : t$, or e diverges, or $e \hookrightarrow^* \omega_{\text{CaseEscape}}$ or $e \hookrightarrow^* \omega_{\text{OutOfRange}}$.*

Proof. Our analysis of the guarded-patterns is best-case/worst-case: when the surely accepted types do not coincide with the possibly accepted types, and are too small to type-check the program, we can use rule (case$_\omega$) to type-check. But this allows values to escape pattern matching, thus adding error $\omega_{\text{CaseEscape}}$ to the soundness result. □

Theorem 6.4.7 (Gradual Soundness)**.** *If $\varnothing \vdash_g e : t$ using rules (case), (case$_\omega$), and (case$_\star$), then either $e \hookrightarrow^* v$ with $v \between t$, or e diverges, or there exists p such that $e \hookrightarrow^* \omega_p$.*

Proof. Lemma 5.2.12, with the addition of rule

$$(\text{case}^\circ)\ \frac{\Gamma \vdash_{\mathsf{w}} e \between t \qquad (\forall i {\leq} n)\,(\forall j {\leq} m_i)\ \left(t_{ij} \not\leq \mathbb{0} \ \Rightarrow\ \Gamma, t_{ij}/_{p_i} \vdash_{\mathsf{s}} e_i \between s\right)}{\Gamma \vdash_{\mathsf{s}} \mathtt{case}\ e\,(p_i g_i \to e_i)_{i \leq n} \between s \wedge ?}$$

is still valid as it is a rule more general than (case), (case$_\omega$), and (case$_\star$) —see Fig. 6.8. Thus, the hypothesis $\varnothing \vdash_g e : t$ in the statement of the theorem implies $\varnothing \vdash_{\mathsf{w}} e \between t$. If we can update the ($\vdash_{\mathsf{w}}$)-based progress Lemma 5.2.1 and the preservation Lemma 5.2.11 with the additional case typing rules of Fig. 6.8, then the theorem holds as a simple corollary of these Lemmas. For progress, it is obvious as the new case expressions with guard evaluation do not get stuck anymore than expressions do. In case no guard succeeds, then explicit error $\omega_{\text{CaseEscape}}$ will output.
For preservation, we observe that the rule (case$^\circ$) preserves typing. Indeed, this rule introduces case expressions with guarded patterns, which evaluate to an *attainable* branch expression e_i ("attainable", which Lemma 6.4.3 ensures, is important here: if it was possible that the case expression evaluates to a branch that was not well-typed within the (case$^\circ$) rule because it was not true that $t_{ij} \leq \mathbb{0}$, then we would not be able to say anything about the result of the reduction, thus preservation would not hold). Then, by inversion on the typing rule, this branch expression is "$\between$"-well-typed with s, and by Lemma 5.2.6, it is also "$\between$"-well-typed with $s \wedge ?$. Thus, the reduction preserves the typing. □

6.5 Inference

The problem of inference for Elixir consists of finding a type for functions defined by several pattern-matching clauses, without any type annotation. Inference appears in this work mainly as a convenience tool: indeed, one could simply decide that every function must be annotated, and inference would not be required. In our case, it is both an interesting research question and a practical one: writing annotations for untyped code is not without effort, and inference can help by suggesting annotations to the programmer. In the case of anonymous functions, being able to infer their types means that annotating can be made optional (e.g., this is convenient when

Table 6.1: Guard Analysis Rules Numbering

Rule #	**Rule Name**	**Description**
1	true	Basic type check success
2	false	Type intersection is empty
3	var	Variable type refinement
4	size	Tuple size analysis
5	proj	Tuple projection
6	eq_1	Equality check (first form)
7	eq_2	Equality check (second form)
8	lt	Less than check
9	lt (maybe)	Less than check (maybe)
Boolean Operators		
10	and	Boolean AND operation
11	or	Boolean OR operation
Approximation Rules		
12	proj (approx)	Approximated projection
13	eq (approx)	Approximated equality
14	size (approx)	Approximated size check
15	lt (approx)	Approximated less than check
Failure Rules		
16	lt (fail)	Less than check failure
17	size_ω	Size operation failure
18	eq_ω	Equality operation failure
19	proj_ω (tuple)	Projection failure (not a tuple)
20	proj_ω (int)	Projection failure (index not int)
21	bound_ω	Projection failure (out of bounds)
22	lt_ω	Less than check failure
23	orFalse	OR with false branch
24	or_ω	OR with error branch
25	$\text{and}_{\mathscr{F}}$-L (fail)	AND with left branch failure
26	$\text{and}_{\mathscr{F}}$-R (all)	AND with all branches failing

$$[\textsc{true}]\ \frac{\Gamma \vdash_{\mathsf{s}} a : \rho}{\Gamma \vdash a \mathrel{?} \rho \mapsto \{(\Gamma, \mathtt{true})\,;\,(\Gamma, \mathtt{true})\}} \qquad [\textsc{false}]\ \frac{\Gamma \vdash_{\mathsf{s}} a : s \qquad s \wedge \rho \simeq \mathbb{0}}{\Gamma \vdash a \mathrel{?} \rho \mapsto \{(\Gamma, \mathtt{true})\,;\,\mathtt{false}\}}$$

$$[\textsc{var}]\ \frac{\Gamma(x) \not\leq \rho \qquad \Gamma(x) \wedge \rho \not\simeq \mathbb{0}}{\Gamma; p \vdash x \mathrel{?} \rho \mapsto \big\{(\Gamma, \mathtt{true})\,;\,\big(\Gamma[x \mathrel{\hat{=}} \rho]_p, \mathtt{true}\big)\big\}} \qquad [\textsc{size}]\ \frac{\begin{array}{c}\Gamma \vdash a \mathrel{?} \mathtt{tuple} \mapsto \{\_\,;\,(\Phi, \mathfrak{b})\}\\ \Phi \vdash a \mathrel{?} \mathtt{tuple}^{i} \mapsto \{\_\,;\,\mathfrak{A}\}\end{array}}{\Gamma \vdash \mathtt{size}\, a \mathrel{?} i \mapsto \{(\Phi, \mathfrak{b})\,;\,\mathfrak{A}\}}$$

$$[\textsc{proj}]\ \frac{\Gamma \vdash_{\mathsf{s}} a' : i \quad \Gamma \vdash a \mathrel{?} \mathtt{tuple}^{>i} \mapsto \{\_\,;\,(\Phi, \mathfrak{b})\} \quad \Phi \vdash a \mathrel{?} \{\overbrace{\mathbb{1}, \ldots, \mathbb{1}}^{i \text{ times}}, \rho, ..\} \mapsto \{\_\,;\,\mathfrak{A}\}}{\Gamma \vdash \pi_{a'}\, a \mathrel{?} \rho \mapsto \{(\Phi, \mathfrak{b})\,;\,\mathfrak{A}\}}$$

$$[\textsc{eq}_1]\ \frac{\Gamma \vdash_{\mathsf{s}} a_1 : c \quad \Gamma \vdash a_2 \mathrel{?} c \mapsto \mathscr{R}}{\Gamma \vdash a_1 = a_2 \mapsto \mathscr{R}} \qquad [\textsc{eq}_2]\ \frac{\Gamma \vdash_{\mathsf{s}} a_2 : c \quad \Gamma \vdash a_1 \mathrel{?} c \mapsto \mathscr{R}}{\Gamma \vdash a_1 = a_2 \mapsto \mathscr{R}}$$

$$[\textsc{lt}]\ \frac{\Gamma \vdash_{\mathsf{s}} a_0 : t_0 \quad \Gamma \vdash_{\mathsf{s}} a_1 : t_1 \quad \mathsf{alwaysLess}(t_0, t_1)}{\Gamma \vdash a_0 < a_1 \mapsto \{(\Gamma, \mathtt{true})\,;\,(\Gamma, \mathtt{true})\}}$$

$$[\textsc{ltMaybe}]\ \frac{\Gamma \vdash_{\mathsf{s}} a_0 : t_0 \quad \Gamma \vdash_{\mathsf{s}} a_1 : t_1 \quad \mathsf{maybeLess}(t_0, t_1)}{\Gamma \vdash a_0 < a_1 \mapsto \{(\Gamma, \mathtt{true})\,;\,(\Gamma, \mathtt{false})\}}$$

Figure 6.9: Guard Analysis: Main Rules

$$[\textsc{and}]\ \frac{\begin{array}{c}\Gamma \vdash g_1 \mapsto \{(\Phi_i, \mathfrak{b}_i)\,;\,(\Delta_i, \mathfrak{c}_i)\}_{i=1..n}\\ \forall i\ \Delta_i \vdash g_2 \mapsto \big\{(\Phi_{ij}, \mathfrak{b}_{ij})\,;\,(\Delta_{ij}, \mathfrak{c}_{ij})\big\}_{j=1..m_i}\end{array}}{\Gamma \vdash g_1\, \mathtt{and}\, g_2 \mapsto \big\{\mathfrak{A}_{ij}\,;\,(\Delta_{ij}, \mathfrak{c}_i \mathbin{\&} \mathfrak{c}_{ij})\big\}_{ij}} \quad \mathfrak{A}_{ij} = \begin{cases}(\Phi_i, \mathfrak{b}_i) & \text{if } \mathfrak{b}_{ij} = \mathtt{true}\\ & \text{and } \Phi_{ij} = \Delta_i\\ (\Phi_{ij}, \mathfrak{b}_i \mathbin{\&} \mathfrak{b}_{ij}) & \text{else}\end{cases}$$

$$[\textsc{or}]\ \frac{\begin{array}{c}\Gamma \vdash g_1 \mapsto \{(\Phi_i, \mathfrak{b}_i)\,;\,(\Delta_i, \mathfrak{c}_i)\}_{i=1..n}\\ \forall i\ \Gamma, {}^{t_i}/_{p} \vdash g_2 \mapsto \big\{(\Phi_{ij}, \mathfrak{b}_{ij})\,;\,(\Delta_{ij}, \mathfrak{c}_{ij})\big\}_{j=1..m_i}\end{array}}{\Gamma \vdash g_1\, \mathtt{or}\, g_2 \mapsto \{(\Phi_i, \mathfrak{b}_i)\,;\,(\Delta_i, \mathfrak{c}_i)\}_i \mathbin{@} \big\{\mathfrak{A}_{ij}\,;\,(\Delta_{ij}, \mathfrak{c}_i \mathbin{\&} \mathfrak{c}_{ij})\big\}_{ij}} \quad \begin{array}{l}\mathfrak{A}_{ij} = \begin{cases}(\Phi_i, \mathfrak{b}_i) & \text{if } \mathfrak{b}_{ij} = \mathtt{true}\\ & \text{and } \Phi_{ij} = \Gamma, \big({}^{t_i}/_{p}\big)\\ (\Phi_{ij}, \mathfrak{b}_i \mathbin{\&} \mathfrak{b}_{ij}) & \text{else}\end{cases}\\ t_i = \begin{cases}\llparenthesis p \rrparenthesis_{\Phi_i} \setminus \llparenthesis p \rrparenthesis_{\Delta_i} & \text{if } \mathfrak{c}_i = \mathtt{true}\\ \llparenthesis p \rrparenthesis_{\Phi_i} & \text{if } \mathfrak{c}_i = \mathtt{false}\end{cases}\end{array}$$

Figure 6.10: Guard Analysis: Boolean Rules

passing short anonymous functions created on the fly, to a module enumerable data, as done by the code in line 161). To study inference, we thus add to Core Elixir expressions, *non-annotated* λ-abstractions formed by a list of pattern-matching clauses:

$$e ::= \cdots \mid \lambda(\,\overline{pg \to e}\,)$$

Operationally, the expression $\lambda(\,\overline{pg \to e}\,)$ is equivalent to $\lambda x.\mathtt{case}\ x\,(\overline{pg \to e})$ where x is fresh and the interface of the λ-abstraction is omitted. This essentially means that a multi-clause function definition is encoded into a single function with a single case expression (which is exactly what the Elixir compiler does).

$$[\textsc{Proj}_a]\ \frac{\begin{array}{c}\Gamma \vdash a' \mathrel{?} \texttt{int} \mapsto \{\_ ; (\Phi, \mathfrak{b})\} \\ \Phi \vdash a \mathrel{?} \texttt{tuple} \mapsto \{\_ ; (\Delta, \mathfrak{c})\}\end{array}}{\Gamma \vdash \pi_{a'}\, a \mathrel{?} \rho \mapsto \{(\Delta, \texttt{false}) ; (\Delta, \texttt{false})\}} \qquad [\textsc{Eq}_a]\ \frac{\begin{array}{c}\Gamma \vdash a_0 \mathrel{?} \mathbb{1} \mapsto \{\_ ; (\Phi, \mathfrak{b})\} \\ \Phi \vdash a_1 \mathrel{?} \mathbb{1} \mapsto \{\_ ; (\Delta, \mathfrak{c})\}\end{array}}{\Gamma \vdash a_0 = a_1 \mapsto \{(\Delta, \mathfrak{b} \mathbin{\&} \mathfrak{c}) ; (\Delta, \texttt{false})\}}$$

$$[\textsc{Lt}_a]\ \frac{\begin{array}{c}\Gamma \vdash a_0 \mathrel{?} \mathbb{1} \mapsto \{\_ ; (\Phi, \mathfrak{b})\} \\ \Phi \vdash a_1 \mathrel{?} \mathbb{1} \mapsto \{\_ ; (\Delta, \mathfrak{c})\}\end{array}}{\Gamma \vdash a_0 < a_1 \mapsto \{(\Delta, \mathfrak{b} \mathbin{\&} \mathfrak{c}) ; (\Delta, \texttt{false})\}}$$

$$[\textsc{Size}_a]\ \frac{\Gamma \vdash a \mathrel{?} \texttt{tuple} \mapsto \{\_ ; (\Delta, \mathfrak{c})\} \qquad \rho \wedge \texttt{int} \not\leq \mathbb{0}}{\Gamma \vdash \texttt{size}\, a \mathrel{?} \rho \mapsto \{(\Delta, \mathfrak{c}) ; (\Delta, \texttt{false})\}}$$

Figure 6.11: Guard Analysis: Approximation Rules

$$[\textsc{Size}_\omega]\ \frac{\Gamma \vdash a \mathrel{?} \texttt{tuple} \mapsto \mathscr{F}}{\Gamma \vdash \texttt{size}\, a \mathrel{?} \rho \mapsto \omega} \qquad [\textsc{Eq}_\omega]\ \frac{\Gamma \vdash a_i \mathrel{?} \mathbb{1} \mapsto \mathscr{F}}{\Gamma \vdash a_0 = a_1 \mapsto \omega}\ i \in \{0,1\}$$

$$[\textsc{Proj}^t_\omega]\ \frac{\Gamma \vdash a \mathrel{?} \texttt{tuple} \mapsto \mathscr{F}}{\Gamma \vdash \pi_{a'}\, a \mathrel{?} \rho \mapsto \omega} \qquad [\textsc{Proj}^i_\omega]\ \frac{\Gamma \vdash a' \mathrel{?} \texttt{int} \mapsto \mathscr{F}}{\Gamma \vdash \pi_{a'}\, a \mathrel{?} \rho \mapsto \omega}$$

$$[\textsc{Bound}_\omega]\ \frac{\Gamma \vdash_{\mathsf{s}} a' : i \qquad \Gamma \vdash a \mathrel{?} \texttt{tuple}^{>i} \mapsto \mathscr{F}}{\Gamma \vdash \pi_{a'}\, a \mathrel{?} \rho \mapsto \omega} \qquad [\textsc{OrF}]\ \frac{\begin{array}{c}\Gamma \vdash g_1 \mapsto \{(\Phi, \mathfrak{b}) ; \texttt{false}\} \\ \Phi \vdash g_2 \mapsto \mathscr{R}\end{array}}{\Gamma \vdash g_1\, \texttt{or}\, g_2 \mapsto \mathscr{R}}$$

$$[\textsc{And}_{\mathscr{F}}\text{-L}]\ \frac{\Gamma \vdash g_1 \mapsto \mathscr{F}}{\Gamma \vdash g_1\, \texttt{and}\, g_2 \mapsto \mathscr{F}} \qquad [\textsc{And}_{\mathscr{F}}\text{-R}]\ \frac{\begin{array}{c}\Gamma \vdash g_1 \mapsto \{(\Phi_i, \mathfrak{b}_i) ; (\Delta_i, \mathfrak{c}_i)\}_{i \leq n} \\ \forall i \leq n\ \Delta_i \vdash g_2 \mapsto \mathscr{F}_i\end{array}}{\Gamma \vdash g_1\, \texttt{and}\, g_2 \mapsto \begin{cases}\omega & \text{if } \forall i, \mathscr{F}_i = \omega \\ \overline{\mathscr{F}}^j & j \text{ s.t. } \mathscr{F}_j \neq \omega\end{cases}}$$

$$[\textsc{ltFalse}]\ \frac{\Gamma \vdash_{\mathsf{s}} a_0 : t_0 \quad \Gamma \vdash_{\mathsf{s}} a_1 : t_1 \quad \mathsf{neverLess}(t_0, t_1)}{\Gamma \vdash a_0 < a_1 \mapsto \{(\Gamma, \texttt{true}) ; \texttt{false}\}}$$

Figure 6.12: Guard Analysis: False/Failure Rules

6.5.1 Inferring Interfaces from Guards

To infer the type of such functions, we apply the guard analysis from Section 6.1. For each clause, it yields a list of admissible input types t_i, covering all types the clause may accept. We type the function body at each t_i, obtaining t'_i, and assign the intersection type $\bigwedge_i (t_i \to t'_i)$ to the function. For example, consider the guard in the following function

$$\lambda(\, x\, \texttt{when}\, (x \mathrel{?} \texttt{int}\, \texttt{or}\, x \mathrel{?} \texttt{bool}) \to x\,)$$

its analysis produces the two accepted types `int` and `bool`; type-checking the function with `int` as input gives `int` as result, and likewise for `bool`. Hence, the inferred type is $(\texttt{int} \to \texttt{int}) \wedge (\texttt{bool} \to \texttt{bool})$. Formally, the new expression is typed by the rule (INFER) below

$$\frac{\Gamma ; \mathbb{1} \vdash (p_i g_i)_{i \leq n} \rightsquigarrow (t_{ij}, \mathfrak{b}_{ij})_{i \leq n, j \leq m_i}\ (\forall i\, \forall j) ; (\Gamma, x : t_{ij} \vdash_{\mathsf{g}} \texttt{case}\, x\, \texttt{do}\, (\overline{pg \to e}) : t'_{ij})}{\Gamma \vdash_{\mathsf{g}} \lambda(\, \overline{pg \to e}\,) : \bigwedge_{ij} (t_{ij} \to t'_{ij})}\ (\textsc{Infer})$$

where x is a fresh variable, $\mathbb{1}$ is chosen as its initial type (meaning that the argument could be of any type), and the Boolean flags $\mathfrak{b}_{ij}$ are discarded (the exactness analysis is not required).

The extension of the type system with inference is sound, since the typing rule (INFER) is a specific combination the rules for λ-abstractions and case-expressions, which are sound. The only difference is that interface checked for the λ-abstraction is determined by the guard analysis rather than written by the programmer.

In some cases, the domain inferred by the guard analysis for the function (i.e., $\bigvee_{ij} t_{ij}$) may not be precise. When this occurs, the programmer can assist the inference process by providing the domain type: it would then suffice to replace this type for $\mathbb{1}$ in the rule (INFER). For example, if we swap the order of the or-guards in the first clause of `test` in line 154, then the type inferred for the function would be the one in lines 156–159 but where the second arrow (line 157) has domain `{:int, ..}` instead of `{:int, term(), ..}`. Although the type checker would produce a warning (because of the use of (proj$_\omega$)), this type would accept as input `{:int}`, which fails. This can be avoided if the programmer provides the input type `{tuple(), tuple(), ..} or {boolean()}` to the inference process. This can be done by using partial annotations, as we proposed in Castagna et al. (2024a, Section 3.2). These are annotations that do not specify the output type leaving the system the task to deduce it. In the specific example we would use the following partial annotation:

```
$ {{tuple(), tuple(), ..} or {boolean()}} -> _
```

More generally, we can suppose that also the multi-clause functions may be annotated with an optional interface:

$$\begin{array}{llll} \textbf{Expressions} & e & ::= & \cdots \mid \lambda^{\mathbb{I}}(\overline{pg \to e}) \\ \textbf{Interfaces} & \mathbb{I} & ::= & \varepsilon \mid \{t_i \to t'_i\}_{i=1..n} \end{array}$$

where ε denotes the empty string. The previous λ-abstractions of the form $\lambda^{\mathbb{I}}x.e$ are then syntactic sugar for single clause functions with one pattern variable and no guards, that is, $\lambda^{\mathbb{I}}(x \to e)$. The typing rules for multi-clause λ-abstractions with a non-empty interface $\mathbb{I}$ are the same as the ones without interface, except that the domain of the interface is used instead of $\mathbb{1}$ and the type tested are the ones in the interface rather than those deduced by the guard analysis.

6.5.2 Dynamic propagation with inferred types.

Inferring static function types for existing code in a dynamic language can disrupt backwards compatibility, as existing code may rely on invariants that are not captured by types. Furthermore, in a set-theoretic type system, no property guarantees that a given inferred type is the most general; consider, for example, that the successor function could be given types $\texttt{int} \to \texttt{int}$ but also any variation of $(0 \to 1) \wedge (1 \to 2) \wedge ((\texttt{int} \setminus (0 \vee 1)) \to \texttt{int})$ using singleton types. While both types are correct and can be related by subtyping, it is the role of the programmers to choose the one that corresponds to their intent and to annotate the function accordingly.

Thus, we choose to introduce some flexibility so that inferred static types do not prematurely enforce this choice. We achieve this by adding a dynamic arrow intersection that points the full domain (the union of the t_i's) to ?, in rule (INFER$_\star$):

$$\frac{\Gamma;\mathbb{1}\vdash(p_i g_i)_{i\leq n}\rightsquigarrow(t_{ij},\mathfrak{b}_{ij})_{i\leq n,j\leq m_i}\ (\forall i\,\forall j);(\Gamma,x:t_{ij}\vdash_{\mathrm{g}}\mathtt{case}\,x\,\mathtt{do}\,(\overline{pg\to e}):t'_{ij})}{\Gamma\vdash\lambda(\,\overline{pg\to e}\,)\vdash_{\mathrm{g}}\bigwedge_{ij}(t_{ij}\to t'_{ij})\wedge(\bigvee_{ij}t_{ij}\to ?)}\ (\text{INFER}_\star)$$

Now '?' gets automatically intersected with each possible return type during function application, which will significantly loosen the discipline enforced by the type system since if ? is intersected with a type, it will allow applying a function to this type as long as the function's domain is *compatible* (as opposed to *a subtype* of) the type. While using rule (INFER$_\star$) by default appears necessary when typing a dynamic language, being able to type-check using only rule (INFER) gives a stronger type safety guarantee, as eliminating the use of '?' during type-checking (and thus, potentially, of gradual rules) yields a stronger type safety guarantee.

Multi-arity. *We have presented inference for single-arity functions, but the same principle straightforwardly applies to multi-arity functions presented in Section 4.4.1: anonymous functions become $\lambda(\,\overline{\overline{p}g\to e}\,)$, and the current guard analysis can be repurposed to produce accepted types for each argument by wrapping these arguments into a tuple pattern.*

6.5.3 Inference without precise analysis

Goal. Offer a lightweight inference that does not rely on the precise guard analysis. It operates only on patterns and their easily computed accepted types, and combines per-clause input/output behaviours with simple type operations, so it can be implemented quickly. This is the scheme currently used in the Elixir compiler (v1.19), as the precise guard analysis of this chapter has not yet been implemented.

Construction from patterns only. Consider a function $\lambda(\,(p_i g_i\to e_i)_{i=1..n}\,)$. For each clause i, let $t_i = \langle\!\!\lbrace p_i \rbrace\!\!\rangle$ be the accepted type of the pattern (ignoring precise guard refinements), and let $\Gamma,x:t_i\vdash_{\mathrm{g}} e_i : s_i$ be the typing of the body under that assumption. This yields arrows $t_i\to s_i$ for all i.

Because of first-match semantics, we cannot take the intersection $\bigwedge_i(t_i\to s_i)$: overlapping domains would require returning all results at once. For example:

```
def f({x, y}) when is_integer(x) -> :int       # {term(), term()} -> :int
def f({x, y}) when is_boolean(y) -> :bool      # {term(), term()} -> :bool
```

We must instead reflect that either clause may fire on overlapping inputs and therefore return the union of their results, e.g. `{term(), term()} -> (:int or :bool)`.

Two arrow construction choices. Given a set of inferred arrows $\mathscr{A}=\{(t_i\to s_i)\}_{i=1..n}$, there are two natural ways to account for overlaps among the domains t_i:

- *Precise arrow.* Split domains so they become pairwise disjoint, assigning to each piece the union of all returns of clauses that cover it. Concretely, we build a set of arrows $\mathscr{A}'$ by inserting each arrow into it one by one and maintaining this invariant: every arrow in $\mathscr{A}'$ is pairwise disjoint with every other arrow in $\mathscr{A}'$. To insert, say, arrow $t_i \to s_i$ into $\mathscr{A}'$, we compare it with every other arrow in $\mathscr{A}'$ and:

- for any empty intersection, we can insert it directly into $\mathscr{A}'$;
- for any non-empty intersection $t_i \wedge t_j$, emit $(t_i \wedge t_j \to s_i \vee s_j)$ and then recursively add the leftovers $(t_i \setminus t_j \to s_i)$ and $(t_j \setminus t_i \to s_j)$ to $\mathscr{A}'$

Consider the following example:

```
(integer() or atom() -> :foo) and (integer() or float() -> :bar)
```

this first strategy yields

```
(integer() -> :foo or :bar) and (atom() -> :foo) and (float() -> :bar)
```

This representation is semantically precise but requires computing differences ($\setminus$) and can blow up the number of clauses.

- *Fast arrow.* Avoid splitting domains: whenever two clauses overlap, broaden their return types by taking their union. This keeps each original domain intact and simply unions the returns across the entire overlap *cluster*. On the same example, we obtain

```
(integer() or atom()-> :foo or :bar) and (integer() or float()-> :foo or :bar)
```

Intuition. The procedure computes the transitive-closure of overlap among domains and unions the corresponding returns within each connected component. Let $i \sim j$ iff $t_i \wedge t_j \neq \mathbb{0}$, and let $[i]$ be the equivalence class of i in the transitive closure of $\sim$. The fast normalisation returns the arrows

$$\{\, t_i \to \bigvee_{k \in [i]} s_k \,\}_{i=1..n}.$$

This is a safe over-approximation of the precise result: whenever an input value lies in the overlap of multiple domains, any of their returns may be produced (due to lack of knowledge if the pattern captures all its inputs), hence returning the union is sound; when the input lies only in t_i, we may return a supertype of s_i, which is conservative but acceptable.

Why this is acceptable in a gradual setting. Recall that our inference rule (INFER_$\star$) already adds a dynamic arrow

$$\Big(\bigvee_i t_i\Big) \to ?$$

to the inferred interface. Therefore, during application, every inferred return is intersected with ?, so broadening returns via the fast normalisation does not compromise type safety, only precision. The benefit is that we avoid expensive difference computations and clause explosion, while still providing useful (and sound) annotations to guide developers and tools.

Multi-arity and patterns. The same fast normalisation applies verbatim to multi-arity functions: treat each clause's tuple of argument types as its domain t_i (cf. §4.4.1). Overlaps are computed on these tuple domains, and returns are unioned within each overlap cluster. It however adds to the complexity of the precise normalisation strategy, which computes differences between domains: indeed, the difference on multi-arity domains has to be transformed into a union of tuples before being usable to create function types. For instance $\{\mathbb{1}, \mathbb{1}\} \setminus \{\texttt{int}, \texttt{int}\}$ has to be transformed into $\{\texttt{int}, \neg\texttt{int}\} \vee \{\neg\texttt{int}, \mathbb{1}\}$ before being usable to create function types, through elimination of negations on tuples, which is its own separate process (see Theorem 4.5.3).

Takeaway.

- **Precise** normalisation yields disjoint domains and minimal returns but is potentially expensive (differences; clause blow-up).
- **Fast** normalisation keeps original domains and unions returns across overlap clusters; it is efficient, easy to implement, and integrates well with ($\textsc{Infer}_\star$) due to dynamic propagation.

Conclusion

This chapter explained how Elixir's patterns and guards integrate with our type system. We gave a compact syntax and operational account, derived accepted types and environment refinements for guarded patterns, used them to type case-expressions with exact and ω-mode guarantees, and showed how the same analysis supports interface inference for multi-clause functions.

In short, guards act as executable evidence: the more precise the guard analysis, the stronger the function types we can derive, while preserving soundness when precision is unavailable.

In the next chapter, we survey related work and position our approach in the literature, then outline future directions for Elixir, including typing message passing and a module system for behaviours.

CHAPTER 7

Discussion

“Luau is the first programming language to put the power of semantic subtyping in the hands of millions of creators.”

Alan Jeffrey, *Semantic Subtyping in Luau* (2022).

Related work

Erlang/Elixir Type Systems. The work that is most closely related to ours is by Schimpf et al. (2023a) who propose a type system for Erlang based on semantic subtyping, implement it as the Etylizer tool, and provide useful benchmarks regarding its expressiveness compared both to Dialyzer (2006) and Gradualizer (2020). The work by Schimpf et al. (2023a) is rather different from ours, since they adapt the existing theory of semantic subtyping to Erlang, while the point of our work is to show how to *extend* semantic subtyping with features motivated by or specific to Elixir: how to add gradual typing without modifying Elixir’s compilation and how to extract the most information from the expressive guards of Erlang/Elixir. Also, their work extensively uses type reconstruction, while we rely more on explicit type annotations, gradual typing, and the inference of guards accepted types.

Another related work is *eqWAlizer* (2022), an open-source Erlang type-checker developed by Meta and used to check the code of WhatsApp. It reads Erlang Typespecs with few exceptions. In particular, and contrary to what we do, they have distinct types for records and dictionaries, and empty lists are subsumed to lists. As in our system, they use generics (constrained by the same `when` keyword we use) with local type inference, type narrowing, and gradual typing. In particular, eqWAlizer uses the same subtyping and precision relations for gradual types as we do, since both approaches are based on Lanvin (2021) and Castagna et al. (2019). However, eqWAlizer techniques to gradually type Erlang expressions are quite different from ours (no dynamic propagation or strong arrows). Another important difference is that when typing overloaded functions with overloaded specs (i.e., our intersections of arrow types) eqWAlizer does not take into account the order of the clauses of the functions while their applications require the argument to be compatible with a unique clause. Thanks to negation types our approach takes into account the order of clauses, and applications are correctly typed even if the argument is compatible with several clauses: it thus implements a more precise type inference.

Dialyzer (2006), which serves as the current default to provide type inference in Erlang, is a static analysis tool that detects errors with a discipline of no false positive, while our static type system ensures soundness, that is, no false negative. Dialyzer lacks support for conditional types or intersection types to capture the relation between input and output types for functions, and record types are parsed but not used.

An actively developed alternative to type Erlang is Gradualizer (2020), which also supports Elixir programs through a translation frontend. The approach looks similar to ours, though it lacks subtyping, with gradual typing inspired from Lanvin (2021). But a comparison is difficult since it lacks a formalization or a detailed description.

Berger et al. (2024) compares those four tools (Etylizer, eqWAlizer, Gradualizer, and Dialyzer) regarding their expressivity and performance on the union of their respective test suites, and find that there is a high level of disagreement between Gradualizer, Etylizer, and eqWAlizer (25% - 45% of test cases across all test suites), concluding that there is yet a need for more semantic agreement on Erlang's type annotations and highlighting the challenging nature of Erlang's type language. While we have not systematically compared our work with those tools, in this work we tried to provide a precise treatment of the specificities of typing Erlang/Elixir programs, and releasing our efforts in the form of an industrial-grade type system for Elixir may help the community move forward on this topic.

Static typing for BEAM languages. The initial effort to type Erlang was by Marlow and Wadler (1997) who defined a type system based on solving subtyping constraints. This type system supports disjoint unions, a limited form of complement, and recursive types, but not general unions, intersections, or negations, as we do. The formalization lacks proofs for first-class function types, which is a solved problem in semantic subtyping. One issue with this work is that they infer constrained types which are quite large, which leads to the use of a heuristics-based simplification algorithm to make them more readable. Valliappan and Hughes (2018) describes a Hindley-Milner type system for Erlang, and Rajendrakumar and Bieniusa (2021) explores a bidirectional type system (without set-theoretic types) as a research contribution.

Numerous statically-typed languages constructed for the Erlang VM have emerged over time. Two examples, Hamler and purerl (2021; 2023), derived from *PureScript* (2013), incorporate a type system akin to Haskell's, including type classes. Notably, in Hamler, type classes are used to model Erlang OTP [1] behaviors. Another language, Caramel (2022), features a type system inspired from OCaml. Sesterl (2021) extends the trend by offering a module system inspired by Rossberg et al. (2014), utilizing functors to type Erlang OTP behaviors (a high priority in our future work list). Lastly, Gleam (2019) is a functional language utilizing well-proven static typing methodologies from the ML community dating back to the early 90s: a Hindley-Milner type system, as introduced by Hindley (1969) and Milner (1978), supplemented with a rudimentary form of row polymorphism in the meaning of Wand (1989) and Rémy (1989).

Parametric polymorphism. Concerning parametric polymorphism, while our work introduces novel theoretical developments for semantic subtyping in Elixir, integrating parametric polymorphism into the language requires no additional theoretical advances. Specifically, we can adapt the polymorphic system described in Castagna et al. (2014) and Castagna et al. (2015) (subsequently extended in Castagna et al. (2019) to include gradual types) to use it with Elixir. The Etylizer project (2023) serves as a practical demonstration of this adaptability, as it successfully ports the polymorphic system of Castagna et al. (2014) and Castagna et al. (2015) to Erlang. Given that Elixir and Erlang are sister languages compiled to the BEAM, this illustrates that this polymorphic theory is directly applicable as it is to Elixir. For this reason, our theoretical development does not include polymorphic types, focusing instead on the novel aspects specific to Elixir. That said, even if the theory does not need any addition, the eventual practical integration of polymorphism into Elixir—which initially will only be achievable through the systematic use of explicit type annotations without resorting to type reconstruction—will require a non-trivial work of implementing the tallying and constraint solving algorithms described in Castagna et al. (2019), as well as a deeper understanding of the techniques necessary for producing informative error messages and for pretty-printing the types resulting from these algorithms.

Maps. Two recent works extend the theory of semantic subtyping by specifically targeting Elixir's map types. The first work Castagna (2023b) introduces a system that seamlessly types maps used as *records*—i.e., maps with a fixed, statically known set of keys, where accessing an unknown key is a (type) error—and maps used as *dictionaries*, where the set of keys may vary dynamically, keys can be computed by expressions, and lookups of undefined keys do not result in an error. This work builds on the theory of record types for semantic subtyping developed in Frisch's PhD dissertation Frisch (2004), and is orthogonal to our contributions, making it directly compatible with the type system presented here. The second work by Castagna and Peyrot (2025b) extends the theory of semantic subtyping by adding *row polymorphism*, and motivates its use for typing Elixir's maps. Row polymorphism is particularly useful in Elixir because maps are a primary data structure for encoding structured data, including records, configurations, state representations, and structs. In many idiomatic patterns, maps are extended or partially updated across function boundaries (e.g., by adding new fields in pipeline expressions or merging

[1] Open Telecom Platform, a set of Erlang libraries and design principles for building distributed systems

configuration data), and row polymorphism enables the typing of such patterns without losing precision. As in the previous case, the system by Castagna and Peyrot (2025b) builds on the record type theory of Frisch and extends it with techniques borrowed from the polymorphic type system of CDuce (2014; 2015) to handle row variables. Like Castagna (2023b), this work is orthogonal to ours and can be easily integrated with our system without requiring changes to its core structure. Finally, Yildirim et al. (2025) extends the latter to allow map types to be parametric in *key domain* (in addition to being parametric in value domain, as is the case in row polymorphism).

Other languages using semantic subtyping. Elixir and Erlang are among the latest languages to embrace semantic subtyping techniques. Other languages in this category include CDuce (2003) which lacks gradual typing and guards, but supply the latter with powerful regular expression patterns; Ballerina (2019) which is a domain-specific language for network-aware applications whose emphasis is on the use of read-only and write-only types and shares with Elixir the typing of records given by Castagna (2023b); Lua*u* (*Lua*u 2022; Jeffrey, 2022) Roblox's gradually typed dialect of Lua, a dynamic scripting language for games with emphasis on performance, with a type system that switches to semantic subtyping when the original syntactic subtyping fails (Luau Team, 2023); Julia (2018) with a type system that is based on a combination of syntactic and semantic subtyping and sports an advanced type system for modules. Finally, Python's type specification (Python Software Foundation, 2025a) has recently started incorporating concepts from gradually typed semantic subtyping. Although this is not yet reflected in practice, there are promising attempts such as the `ty` type-checker that aim to include union, intersection, and negation types in Python (Astral Team, n.d.). While some of these languages employ gradual typing and/or guards, none share Elixir's emphasis on these features or, consequently, on the typing techniques we developed in this work. Nevertheless, we believe that portions of the work described in the last chapters could be adapted to these languages, particularly the techniques for safe erasure gradual typing (strong functions and dynamic propagation) and the extension of semantic subtyping to multi-arity function spaces.

Gradual typing in semantic subtyping. The thesis by Lanvin (2021) defines a semantic subtyping approach to gradual typing, which forms the basis of the gradual typing aspects of our system, since we borrow from Lanvin (2021) the definitions of subtyping, precision, and consistent subtyping for gradual types. The main difference with Lanvin (2021) is that he considers that sound gradual typing is achievable by inserting casts in the compiled code whenever necessary, while our work shows a way to adapt gradual typing to achieve soundness while remaining in a full erasure discipline. The relations defined by Lanvin (2021) are also implemented at Meta for the gradual typing of the (Erlang) code of WhatsApp via *eqWAlizer* (2022). Lanvin (2021) builds on and extends the work by Castagna *et al.* (2019) who show how to perform ML-like type reconstruction in a gradual setting with set-theoretic types. This absence of proper type reconstruction is one of the limitations of our work. To address it we count on adapting the results of Castagna et al. (2024b) on type inference for dynamic languages. An alternative option is to utilize the approach by Castagna et al. (2016) which employs traditional, less computationally demanding type reconstruction techniques than those in Castagna et al. (2024b), but lacks the capability to infer intersection types for functions.

Industrial solutions for typing dynamic languages. Industry systems have adopted gradual typing to facilitate the progressive adoption of typing disciplines. TypeScript (Bierman et al., 2014), built on JavaScript, is a most prominent example, offering a compile-time-only type system featuring erased annotations, structural types, unions, intersections, and control flow narrowing (Microsoft, 2025b). However, its subtyping relation for unions and intersections is defined syntactically rather than semantically, resulting in "incomplete" union and intersection types that do not always align with expectations/properties of the intended set-theoretic interpretation. Moreover, the subtyping relation was intentionally designed to be unsound in certain—quoting (Microsoft, 2025a)—"carefully considered" cases, which contrasts with our design philosophy. Despite these acknowledged limitations in soundness and completeness, TypeScript represents a clear success story in terms of both adoption and tooling ecosystem. Our work reproduces these key features while addressing soundness and completeness concerns, and providing additional benefits grounded in a robust theoretical framework. A particularly illustrative example of such benefits involves JavaScript's heavy reliance on dictionary data structures. As detailed in Castagna (2023b, Section 5), the syntactic approach employed by TypeScript (but also the one of Flow, see below) produces imprecise and ambiguous typing for these structures, necessitating various restrictions. In contrast, semantic subtyping provides more general and precise typing for dictionary structures while eliminating the need for such restrictions. Another key difference is that TypeScript features two types representing `dynamic()`: `any` and `unknown`, the first being permissive and unsound, while the second is not, as it forces checks at the use site.

Flow, by Chaudhuri et al. (2017) (built on JavaScript, too), has a similar surface, but with stricter inference (a feature that we only partially integrate into our system, as we present inference as a way to suggest more strictly enforced type annotations). It also has a smaller ecosystem. Hack (2014) (built on PHP) features a static type checker that is able to introduce runtime checks via HHVM (HipHop Virtual Machine) type hints, providing operators 'is' and 'as' to assert and cast types (casts are only supported for `bool`, `int`, `float` and `string`, as can be seen in Hack's documentation (2025)). It is a pragmatic construction that really compares to our leveraging of existing BEAM VM checks and guard mechanisms. The difference is that we use more expressive structural types, thus there is a greater distance, in Elixir than in Hack, between the types that are VM-checkable and those that feature in type annotations.

Sorbet (built on Ruby) is a gradual type system with an erasure discipline, a dual design similar to TypeScript with `T.untyped` (unsafe escape hatch) and `T.anything` (explicit check required), and it can optionally insert runtime checks for type assertions, similarly to Hack and to our approach. It supports union (`T.any`) and intersection (`T.all`) types, and uses the latter for control flow narrowing in a way very similar to our system. One main difference with our approach is that the Sorbet system is largely nominal as types are based on classes and modules rather than structural shapes. Another more substantial difference is that, as for TypeScript or Flow, the subtyping relation is syntactic rather than semantic. While Sorbet—unlike TypeScript—does not push unsoundness, and—like us—it explains containment in terms of a set-theoretic interpretation of types, it still does not fully comply with such an interpretation. For instance, Sorbet permits the intersection of two modules that export the same method with different signatures (i.e., types). Since the actual method depends on the order in which the two modules are imported, then the intersection of the types of these two modules should give this method the union of the two signatures. However, while Sorbet accepts the calls whose arguments are in the intersection of the domains of the signatures (as expected when applying a function with a union type), if the return types are incompatible then the call is rejected (instead of being typed with the union of the return types). Furthermore, if the intersection of the domains is empty, then a code specifying the intersection of the two modules is still accepted, despite the fact that the method in common to the modules can never be called. This shows that the basic subtyping properties of the union of two arrow types are not accounted for. It is a direct consequence of the syntactic nature of the subtyping relation, which does not completely capture the intended semantics of types.

In general, the major difference of our approach with industry solutions is that it is built on a sound foundation of gradual set-theoretic types, thus avoiding the pitfall of ad-hoc rules that are added later on: for instance, the overload call evaluation of Python (PEP 484) consists of a six-step algorithm (see the specification Python Software Foundation Team (2025)) that performs expansions to attempt to find matches with arguments; steps five and six were recently added to “determine whether all possible materializations of the argument’s type are assignable to the corresponding parameter type”. By contrast, overloaded (intersected) gradual function types are a native object of our type algebra. However, these systems are also not bound to a specific theory and can constantly evolve to match desired behaviors. While this enhances their flexibility, it also hinders their portability to other languages and complicates formal reasoning about their properties.

Positioning among gradual typing disciplines. We discussed above few major examples of industrial type systems for dynamic languages. A larger range of gradual type systems for industrial-grade dynamic languages have explored different trade-offs between soundness, precision, and runtime overhead. These have been thoroughly examined by Greenman, Dimoulas, and Felleisen (2023), who categorize the implemented semantics into three main disciplines: *erasure*, *transient*, and *natural*. In the *erasure* discipline, types are used only for static analysis and have no influence on the runtime behaviour of programs. The other two disciplines change the runtime behaviour of programs based on the static type information: the transient discipline inserts shape checks in the code that enforce the correspondence between top-level value constructors and the expected types, ensuring that the *typed portion* of the code cannot “go wrong”; the natural discipline goes further by inserting wrappers or proxies that enforce for the *entire program* the full type structure at runtime.

As noted by Greenman et al. (2023), the *erasure* discipline is by far the most popular design choice in industry, especially because its predictable behaviour and performance. But the current systems using this discipline implement it at the expense of type soundness. For example, in TypeScript type annotations are completely erased from the emitted JavaScript. However, as already pointed out, TypeScript does not guarantee type soundness and requires modifications to the compiler, such as additional static checks (Rastogi et al., 2015), to partially recover it. This issue also affects the already cited Flow and Hack, developed by Meta for JavaScript and PHP, respectively, as well as systems like Mypy (Lehtosalo et al., 2017) and the one defined by Rastogi, Chaudhuri, and Hosmer (2012).

Our work clearly belongs to the class of *erasure* disciplines, and improves its state of the art by introducing a fully erasable gradual type system that *retains* type soundness. We achieve this by statically identifying and type-checking *strong functions*, which can safely interact with dynamic code without compromising soundness. The presence of strong functions reduces the gap between the erasure and transient disciplines. Strong functions are, in effect, proxies for the shape checks of the transient discipline: instead of inserting these shape checks, as done by transient systems, our system statically identifies the presence of these checks within the scope of a function application. Of course, there are several inherent limitations to this approach. Checking the presence of shape checks instead of systematically inserting them means deducing—actually, enforcing—less precise types. Furthermore, a shape check is possible only if there is a corresponding guard that implements it, which is not always the case, thus reducing the number of functions that can be proven strong. Thus, the static detection of a strong function working on composite specifications such as `list(integer)` may be problematic, since there is no primitive guard in Elixir to assert that *all* elements of a list have a given type. However, these limitations seem to us acceptable trade-offs to retain both soundness and the advantages of the erasure discipline.

The soundness property enforced by our system bears resemblance to the notion of *open-world soundness* introduced for Reticulated Python by Vitousek et al. (2017), classified by Greenman et al. (2023) as a transient system, which guarantees that well-typed programs, when compiled from a gradually-typed surface language to an untyped target language, can safely interoperate with untyped code. In our case, Elixir's existing runtime behaviour already provides sufficient guarantees (via its virtual machine checks and guard mechanisms) to ensure a similar property when endowed with a gradual type system, but *without runtime wrappers or coercions*. Typed Racket, by Tobin-Hochstadt and Felleisen (2008) (implementing a *natural* discipline) takes a hybrid approach, using occurrence typing for internal precision and contracts at module boundaries to enforce soundness.

To summarize, transient and natural disciplines, such as Typed Racket and Reticulated Python, maintain soundness via runtime instrumentation, which introduces overhead and affects integration for existing untyped code. By contrast, our system achieves soundness without modifying the runtime, relying instead on Elixir's built-in runtime checks and our guard-aware static analysis. Combined with semantic subtyping and dynamic type propagation, our approach occupies a distinct point in the design space: it avoids the unsoundness of TypeScript, Flow, and Hack, and the runtime cost of systems like Reticulated Python and Typed Racket, though this comes at the cost of less precise typing in some dynamic contexts. Our *safe erasure* strategy enables Elixir developers to adopt types incrementally, without sacrificing compatibility, performance, or language semantics.

Future work

Message-passing

One key characteristic of Elixir is its concurrency and distribution system based on message-passing between lightweight threads called *processes*. Receiving messages from other processes is done through the `receive` construct, which relies on pattern matching and guards to match messages sitting in the process inbox. Typing the concurrency constructs and the actor model of Elixir is an obvious next step. Our type system is already capable of augmenting the code in `receive` with type information from guards, with narrowing and approximations. The *potentially accepted type* (cf. Section 1.1.4 c)) of the patterns and guards in receive operations can be used to define *interfaces* (i.e., types) for processes and thus type higher-order communications. A longer-term research project is to type processes with behavioral types, in the sense of Ancona et al. (2016), for example by adapting the theory of Mailbox Types by de'Liguoro and Padovani (2018) and Fowler et al. (2023).

Module system for behaviours

Behaviours Elixir provides first-class module support, enabling modules to be passed as arguments and returned as computational results. However, at runtime modules are represented as atoms subject to dynamic typing mechanisms. The language offers static type declarations for modules through *behaviours*, which serve as interface specifications within the Elixir ecosystem.

A behaviour specification comprises three components: type declarations (concrete or abstract), function callbacks defining expected signatures, and exported function definitions that remain invariant across implementations. A module implements a behaviour when it provides definitions for each mandatory callback, where implementing functions accept supersets of callback domains and return subsets of callback codomains.

Currently, Elixir employs behaviours for basic static module typing–modules that adopt behaviours but fail to implement required callbacks trigger compilation warnings. However, this represents a minimal approach: Elixir utilizes only the callback component for partial verification, with no runtime verification when modules are passed as values.

First-Class Behaviour Types The next phase involves integrating behaviours as first-class types within our type system, enabling comprehensive exploitation of static typing benefits when working with higher-order modules. This integration presents significant research challenges.

Behaviours may specify callback types using abstract types–nominal types whose concrete implementations are module-specific. To fully leverage behaviour-provided information, the type system must distinguish between different type roles: concrete types shared across implementations must be exported transparently; abstract types accessible only to callback functions must be exported opaquely; and implementation-specific types requiring public visibility must be parameterized within behaviours.

This work necessitates extending the semantic subtyping framework to accommodate existential types for packaged modules Russo (2000) and Rossberg (2018) or bounded existential types as implemented in Julia Zappa Nardelli et al. (2018) and Chung et al. (2019). Even approximating abstract types with `dynamic()` requires careful consideration of language design implications, particularly regarding backward compatibility constraints.

The GenServer Behaviour The GenServer behaviour illustrates these challenges through its fundamental role in Elixir's standard library. It provides structured client-server interactions through synchronous and asynchronous communication patterns, encapsulating common boilerplate for process supervision and message handling.

The following schematic presents the GenServer behaviour definition using Typespec annotations, with annotations highlighting the distinct type roles:

```
defmodule GenServer do
  # Types
  @type option() :: {:debug, debug()} | {:name, name()} | ...    % transparent
  @type result() :: {:reply, reply(), state()} | ...             % transparent
  @type state() :: term()                                        % opaque
  @type request() :: term()                                      % parameter
  @type reply() :: term()                                        % parameter
  % ... additional types ...

  # Callbacks
  @callback init(init_arg :: term()) :: {:ok, state()} | :error
  @optional_callback handle_call(request(), pid(), state()) :: result()
  @optional_callback handle_cast(request(), state()) :: result()
  % ... additional callbacks ...

  # Functions
  @spec start(module(), any(), options()) :: on_start()          % higher-order
  % ... additional functions ...
end
```

Listing 7.1: GenServer behaviour specification with type role annotations

The behaviour specification exhibits three categories of components with specific typing implications:

- **Types.** The types `option()` and `result()` are fully defined and shared across implementations, necessitating transparent export. The `state()` type represents internal server state, accessible exclusively to implementation-specific functions, requiring opaque export.

The `request()` and `reply()` types describe message structures that are implementation-specific yet publicly accessible, making them suitable for parameterization.

- **Callbacks.** Callback specifications exhibit mandatory and optional variants, analogous to optional and mandatory field distinctions in map types within our type system.
- **Functions.** The `start` function requires a module as its first argument, but this module must implement the GenServer behaviour, not merely any arbitrary module.

When behaviours are elevated to first-class module types, the GenServer behaviour would be specified using the following syntax:

```
defmodule type GenServer(request, reply) do
  # Types
  type option() = {:debug, debug()} | {:name, name()} | ...       %  transparent
  type result() = {:reply, reply(), state()} | ...                % transparent
  type state()                                                    % opaque
  % ... additional types ...

  # Callbacks
  callback init :: init_arg() -> {:ok, state()} | :error
  callback optional(handle_call) :: request(), pid(), state() -> result()
  callback optional(handle_cast) :: request(), state() -> result()
  % ... additional callbacks ...

  # Functions
  spec start :: GenServer(request, reply), init_arg(), options() -> on_start()
  % ... additional functions ...
end
```

Listing 7.2: Proposed GenServer behaviour as first-class type

In this formulation, `request` and `reply` serve as parameters of the module type GenServer. The types `request()` and `reply()` are transparently exported, while `state()` remains opaque. Optional callbacks are explicitly declared using syntax reminiscent of map type optional fields. The `start` function specification explicitly constrains its first argument to modules implementing the same GenServer type, ensuring type-safe module passing rather than treating modules as arbitrary atoms.

Part II

Pragmatic Typing for Dynamic Languages

CHAPTER 8

GRADUAL SET-THEORETIC TYPES AS API

8.1 Our approach: a theory-first system, built for reasoning

Type systems serve multiple purposes in modern programming:

- **Documentation:** Types serve as mechanized documentation. Even a purely informal language of types written as comments above the code helps convey programmer intent. A type system enforces these annotations and increases confidence that the specification is correct.
- **Style enforcement:** Types can be used to enforce a certain style of programming. For example, a type system can enforce that functions adhere to a specific signature or that data structures are used consistently.
- **Code tooling:** Types can be used to provide better tooling support, such as autocompletion, refactoring tools, and static analysis provided directly in an IDE as part of a standardized language server maintained by the community.
- **Type-driven development:** Types can be used to drive the development process. This is a common practice in functional programming languages, where types are used to guide the design of functions and data structures.
- **Bug finding:** A type system finds bugs, either statically or dynamically. A strict typing discipline guarantees that certain classes of runtime errors cannot occur. A gradual typing discipline finds type clashes that lead to certain errors. We study the continuum between the two.

A common thread runs through these goals: a type system allows both the programmer and the machine to reason about the code. This reasoning occurs under different rules of interaction.

The compiler's output, including type errors and warnings, reflects how the machine reasoned about the code.

Programmers also reason about the code and direct the compiler towards their views. In languages where type annotations are enforced, annotations act as a contract between the programmer and the compiler. With a language of types, programmers express intent both through what the code should do and through the operators the type language provides. Our specific approach adds a practical advantage: beyond representing data structures (tuples, maps, etc.) and base types (integers, booleans, etc.), the semantic subtyping framework lets programmers reason with familiar set intuitions. Conceiving types as sets of values, and using set connectives (union, intersection, negation) with simple words ("or", "and", "not") raises the level of abstraction without introducing more complex typing notions. This ease of use has a cost: designing a set-theoretic type system for a full-fledged language like Elixir is complex, from the design of types as an algebra with set connectives to the integration of these types into the language's existing features and paradigms.

8.2 A simple type system built on complex types

There is a striking mismatch between:

i) the complexity of constructing set-theoretic types—from their semantic interpretation to highly recursive decision procedures (obtained by solving set inclusions) and their implementation (the subject of Chapters 9 and 11) and

ii) the intuitive simplicity of set-theoretic types.

This tension also appears when contrasting the effort of building the "set-theoretic type engine" with the simplicity of wiring it into an actual type checker. In practice, there is rarely a need to stray from the declarative system of Chapters 4–5–6. Given a way to store types, perform subtyping and related checks, and compute type operators, implementing the rules is largely an administrative exercise of calling constructors and destructors. Even polymorphism becomes straightforward once tallying (which solves subtyping constraints) is delegated to the engine.

Set-theoretic types offer a firm ground for intuition that a language designer can iterate on. A case we return to is Elixir Protocols (whose typing is described in the introduction 1.1.3 and for which an example is found in Appendices A.4): simple union types already suffice to capture their dispatch behaviour, even though Protocols were not an initial design target (see Chapter 13).

Table 8.1 summarizes the correspondence between set-theoretic operations and our surface syntax.

We argue that set-theoretic types can be provided as an abstract API for use by the type-checker. What the checker needs from the engine is small and stable: (i) constructors for the set-theoretic connectives, base types and structural types, (ii) operators for structural types and function application, and (iii) a handful of queries: subtyping, equivalence, satisfiability/emptiness, and disjointness. With these in hand, standard rules—application, pattern matching, and

Set Operation	Mathematical	Our Type System
Union	$A \vee B$	`A or B`
Intersection	$A \wedge B$	`A and B`
Complement	$\mathscr{D} \smallsetminus A$	`not A`
Empty set	$\emptyset$	`none()`
Universe	$\mathscr{D}$	`term()`

Table 8.1: Correspondence between set-theoretic operations and type system syntax.

guards—reduce to a few calls to compute refinements and to test (non-)emptiness. A synthetic description of such an API can be seen in Figure 8.1.

```
# constructors
$ integer/0 :: (-> type)
$ atom/1    ::  (list(atom) -> type)       # e.g. atom([:ok, :error])
$ tuple/1   :: (list(type) -> type)
$ fun/2     ::   (list(type), type -> type)

# connectives
$ union/2        :: (type, type -> type)
$ intersection/2 :: (type, type -> type)
$ difference/2   :: (type, type -> type)

# operators on types
$ dom/1     :: (type -> type)
$ app/2     :: (type, list(type) -> type)
$ project/2 :: (type, type -> type)

# type queries
$ subtype?/2    :: (type, type -> type)
$ compatible?/2 :: (type, type -> type)    # can those two types coincide?
$ empty?/1      :: (type -> boolean) # true if the type is the empty set
$ disjoint?/2   :: (type, type -> boolean) # true if they never coincide
```

Listing 8.1: Types API

Conclusion

This chapter distills a theory-first, API-facing view of semantic subtyping: think in sets, compose with ∨, ∧, \, and rely on a small, stable vocabulary that tools can surface directly to programmers.

In the next chapter, Chapter 9, we leave the API surface for the type engine itself: we describe the modular type representation (`Descr`), component-wise set operations, and the gradual pair invariant that make these API operators executable inside a compiler. This architectural layer is what the following implementation chapters refine and specialize.

CHAPTER 9

BOOTSTRAPPING GRADUAL SET-THEORETIC TYPES

"Alain Frisch's dissertation is said to include detailed accounts of many of the lessons learned while working on CDuce, however this work is written in French and the authors of this work ne comprend pas le français."

Andrew M., *Advanced Logical Type Systems for Untyped Languages* (2019)

This chapter provides a high-level overview of the key implementation challenges and design decisions for set-theoretic types. The following chapters present each topic in detail: Chapter 10 covers type representation and gradual types, Chapter 11 addresses structural types, and Chapter 12 compares alternative representations.

The ℂDuce language presented in Frisch (2004) established the first blueprint for implementing set-theoretic types with semantic subtyping for functional languages. Despite this pioneering work, the dissemination of semantic subtyping has been slow, with few languages adopting the full approach until recently. Notable exceptions include Luau (2022), which implements semantic subtyping (with some approximations for function types), and emerging systems like Erlang's Gradualizer and Elixir's type system.

While semantic subtyping's core ideas gained traction, most implementations resorted to ad hoc techniques that failed to leverage the approach's full power. The semantic approach offers compelling advantages: (1) subtyping becomes a decidable total function; (2) type systems can directly utilize set-theoretic operations (e.g., using type difference to drive pattern matching); (3) programmers can employ set-theoretic connectives in type annotations without restriction.

The slow adoption stems from two primary factors beyond historical barriers (the original 2004 thesis was written in French):

- **Missing foundational features:** Critical capabilities such as polymorphism were only introduced in 2014 (Castagna et al., 2014; Castagna et al., 2015). Similarly, Castagna et al. (2019) established the theoretical foundation for gradual typing in semantic subtyping-essential for dynamic languages–but lacked concrete implementation guidance (see Chapter 10).
- **Implementation complexity:** The approach rests on solving set-containment inequalities and requires substantial theoretical machinery to formalize the type algebra. Many researchers question whether a fully set-theoretic approach is algorithmically feasible, particularly for complex features like polymorphism (noting, for example, Reynolds' (1984) influential claim that "Polymorphism is not set-theoretic" of which Castagna and Xu (2011) can be considered a formal refutation) and function types.

This chapter addresses both concerns and shows that

- implementing semantic subtyping does not require mastering its complete theoretical foundation—practical algorithms exist that are modular and approachable;
- the implementation is not only feasible and efficient (with well-understood trade-offs), but also *à la carte*: language designers can adopt components incrementally to match their specific needs.

9.1 Descr: Modular Type Representation (an overview)

The fundamental idea for implementing set-theoretic types is to consider each type as a union of disjoint *component* types (each of which being in DNF form, from Chapter 3). For example, the type `number` decomposes into two disjoint components: the base types `integer` and `float`. Our implementation represents each type as a record that tracks which type components are present–we call this record a *Descr* (short for "type descriptor").

The *key insight* is that, because components are mutually disjoint, set-theoretic operations (union, intersection, difference) can be computed *field by field* and then combined. This modularity makes the implementation both simple and extensible: to add a new base type, one simply introduces a corresponding field in the `Descr` record and defines the set operations for that field.

9.1.1 Anatomy of a Descr.

Drawing on the semantic subtyping approach of ℂDuce (2003), a *Descr* is a record with one field for each type component, where each field stores a representation of the subset of values of that type included in the overall type. To illustrate, consider four principal components—`integer`,

`atom`, `tuple`, and `function`—each represented differently. [1] Table 9.1, at the end of this section, summarizes the different types in the Descr that we describe.
We now describe each representation in more detail.

Integer type. Although Featherweight Elixir supports singleton integer types (e.g., `42`), our current implementation treats `integer` as *indivisible*: its only strict subtype is the empty type. Thus, every type in Elixir contains either all integer values or none. This is in contrast to the `atom` type (see next item), which supports singleton subtypes and more granular distinctions. The indivisible integer type is therefore represented by a simple Boolean flag indicating whether integer values are included or not.

Atom type. The case of `atom` is different from that of `integer`. In Elixir, it is common to use atoms in tuples as flags to determine the content of the rest of the tuple. We saw an example with the `:int` atom in Section 2; a more illustrative example is provided by *GenServer* (one of the most widely used modules in Elixir) whose options are passed as pairs where the first element is an atom indicating the option name and determining the type of the second element:

```
$ type option() = {:debug, debug()} or
                  {:name, name()} or
                  {:timeout or :hibernate_after, timeout()} or
                  {:spawn_opt, [Process.spawn_opt()]}
```

In view of the widespread use of this pattern, our implementation has supported atom singleton types from the outset. This means that, unlike `integer`, `atom` is not an indivisible type. However, every nonempty subset of the `atom` falls into one of the following two forms:

- a finite explicit set of atoms $\{a_1, \ldots, a_n\}$;
- the complement of a finite set of atoms (i.e., a co-finite set).

Thus, we can represent any type formed by atoms by a pair in which the first element is a flag that indicates whether the type is finite or cofinite, that is, (`:union`, $\{a_1, \ldots, a_n\}$) or (`:minus`, $\{a_1, \ldots, a_n\}$), and where `atom` corresponds to (`:minus`, { }). For example, the type `:foo or :bar` is represented as (`:union`, {`:foo`, `:bar`}), while the type of 'all atoms except `:foo` and `:bar`' (i.e. `atom() and not (:foo or :bar)`) is be (`:minus`, {`:foo`, `:bar`}). In particular, the type `atom`itself is represented as (`:minus`, {}), since it includes every atom (the complement of an empty set of exclusions).

[1]The current Elixir implementation includes additional base types: `binary()`, `float()`, `pid()`, `port()`, `reference()`, and `list()`, all handled with similar techniques, and uses *only decision trees* to represent structural types, like tuples. The reasons for the choice of decision trees will be explained in Chapter 11. For a detailed comparative analysis of alternative representations we explored (DNFs, compressed DNFs, union forms) and why they were ultimately not adopted, see Chapter 12.

Tuple type. Any tuple type can always be put in disjunctive normal form. Furthermore, it is always possible to eliminate intersections of tuple types (by intersecting the components), and similarly eliminate intersections of their negations (by distributing them over disjunctions: $(t_1, t_2) \wedge \neg(s_1, s_2) = (t_1 \wedge s_1,\ t_2 \wedge \neg s_2) \vee (t_1 \wedge \neg s_1,\ t_2)$, and likewise for larger arities). Based on this property, we may represent any tuple type as a list of descriptor tuples—a representation we call *union forms* (see Section 12.3 for a detailed analysis of this approach and its limitations). Each element of the list corresponds to one variant of a tuple (with a particular arity and component types). For example, a type consisting of "either a pair of an integer and a boolean, or a triple of an atom, an integer, and a term" would be stored as [`{int, bool}`, `{atom, int,` $\mathbb{1}$ `}`]. In general, a tuple component of a `Descr` is a list of tuple types $[(t_{1,1}, \ldots, t_{1,n_1}), \ldots, (t_{k,1}, \ldots, t_{k,n_k})]$, each tuple $(\cdot)$ representing one alternative shape included in the type.

Function type. In semantic subtyping, every subtype of `function` is a propositional logic formula where literals are atomic arrow types $(t_1, \ldots, t_n) \to t$ or their negations $\neg\,((t_1, \ldots, t_n) \to t)$. We can store such a formula using a binary decision diagram (BDD), that is, a binary tree where each node is labeled (for a comprehensive complexity analysis of BDDs and alternative representations, see Chapter 12). and the leaves are labeled either 0 or 1. The formula represented

$$\begin{tikzcd}
 & & (\texttt{int} \to \texttt{int}) \arrow[dl] \arrow[dr, "\neg"] & \\
 & (\texttt{bool} \to \texttt{bool}) \arrow[dl] \arrow[dr, "\neg"] & & 0 \\
1 & & 0 &
\end{tikzcd}$$

Figure 9.1: BDD of a (`int` → `int`) ∧ (`bool` → `bool`).

`int`	`true`
`atom`	`(:union, {:foo, :bar})`
`tuple`	[`{int, ¬int}`, `{¬int,` $\mathbb{1}$ `}`]
`fun`	as in Figure 9.1

Figure 9.2: Components of a `Descr`.

by a BDD is the union of all paths from the root to the 1-labeled leaves, where each path corresponds to the conjunction of the literals along the path, either positively (if taking a left branch after the literal) or negatively (if taking a right branch after the literal). For example, the type (`int` → `int`) ∧ (`bool` → `bool`) will be represented by the BDD shown in Figure 9.1, where there is only one path that leads to a leaf labeled 1.

The type represented by a Descr is the union of all the atomic types present in its fields. For example, the type `int` ∨ (`:foo or :bar`) ∨ (`{`$\mathbb{1}$`,`$\mathbb{1}$`}` ∖ `{int, int}`) ∨ ((`int` → `int`) ∧ (`bool` → `bool`)) will be represented by its four components described in Figure 9.2, where we distribute the negation to obtain the tuple type.

Note

Note also that the negation of any atomic type is represented by a descriptor that saturates the other component types: for instance, `¬int` ∨ `¬atom` is represented by a descriptor with the `integer` field set to `false`, the `atom` field set to (`:union`, {}), the `tuple` field set to [{$\mathbb{1}$, $\mathbb{1}$}], and the `fun` field set to the BDD for $\mathbb{1} \to \mathbb{1}$.

Table 9.1: Representation of type components in Descr

Type	**Representation**	**Rationale & Properties**
`integer`	Boolean flag	Indivisible (no singleton integer types in current implementation); either contains all integers or none. Represented by a simple bit.
`atom`	Finite/co-finite set: $(tag, \{a_1, \ldots, a_n\})$	Common in pattern matching (e.g., GenServer options). Tag indicates finite set (`:union`) or its complement (`:negation`). Example: (`:union`, {`:foo`, `:bar`})
`tuple`	List of tuple types: $[(t_{1,1}, \ldots, t_{1,n_1}), \ldots]$	Any tuple type can be put in DNF. Positive intersections can be computed directly. Negations can be eliminated by distributing over components. Each list element represents one alternative arity/shape (see Section 12.3 for details).
`function`	Binary Decision Diagram (BDD)	Function types are propositional formulas over arrow types $(t_1, \ldots, t_n) \to t$. BDDs provide canonical representation. Example in Figure 9.1.

9.1.2 Component-wise Set Operations.

The `Descr` structure provides an efficient way to perform type operations because each atomic field can be computed separately. In particular, since the atomic components are mutually disjoint, operations on types can be carried out *field by field*. The intersection of two types, for example, is computed by intersecting their respective `integer` flags, intersecting their `atom` components (which involves intersection of finite/cofinite sets of atoms), intersecting their lists of tuple types (by distributing intersections across corresponding tuple components) and intersecting their function BDDs. The resulting fields together form the `Descr` of the intersected type. Likewise, a union is performed by taking the union of each field separately (boolean OR for the `integer` field, union of atom sets for the `atom` field, concatenation and normalization of tuple lists for the `tuple` field, and BDD OR for the `function` field). Set difference is handled analogously (e.g., subtracting on each field in isolation). This component-wise approach greatly simplifies implementation and is crucial for the performance of the type-checker.

Suppose that `Descr` is an Elixir map with four fields: `integer`, `atom`, `tuple`, and `function`, and that we have a function `intersection/3` that takes a field key, two values, and computes their intersection according to which component is being intersected. Then, the implementation of intersection for `Descr` is straightforward:

```elixir
def intersection(type1, type2) do
  Map.merge(type1, type2, &intersection/3)
end
```

Listing 9.1: Implementation of intersection for Descr

The implementation enumerates all triples `(field, value1, value2)` and computes `intersection(field, value1, value2)` for each triple, making it the new value for the field in the result.[2] The implementation of `intersection/3` can be trivial, as for `:integer` where it computes `value1 and value2`, or a bit more involved for BDDs (see the detailed discussion in Chapter 10).

9.1.3 Extension to Gradual Types

Importantly, this modular design extends gracefully to *gradual types*. As described in Chapter 3, every gradual type t can be characterized by two extremes: a minimal static approximation and a maximal static approximation, via the equivalence.

$$\tau \simeq \tau^{\Downarrow} \vee \left(? \wedge \tau^{\Uparrow}\right)$$

In other words, any gradual type can be represented as the union of a purely static part ($t^{\Downarrow}$, the set of values that *will* be present at run time) and a dynamic part (or static over-approximation) constrained by a static bound ($? \wedge t^{\Uparrow}$, representing values that *may* occur at run time, up to the static limit $t^{\Uparrow}$). We leverage this property by implementing each gradual type as a pair of `Descr` structures: one for $t^{\Downarrow}$ and one for $t^{\Uparrow}$. In particular, Chapter 10 shows that to make operations on these pairs modular, we can maintain the invariant that the first descriptor is always a subtype of the second (i.e. $t^{\Downarrow} \leq t^{\Uparrow}$). This ensures that all values guaranteed by the static part are also covered by the dynamic part. If we enforce this invariant when constructing gradual types, then operations on gradual types can be performed component-wise on each of the two descriptors, just as with static types, and the result automatically satisfies the invariant. For example, consider the gradual type $t = \texttt{int} \vee (? \wedge \texttt{bool})$, which means "statically an integer or possibly a boolean at runtime." We represent t as a pair of descriptors $(\mathsf{D}_{\text{stat}}, \mathsf{D}_{\text{dyn}})$ where D_{stat} describes `int` and D_{dyn} describes $\texttt{int} \vee \texttt{bool}$. Note that $\mathsf{D}_{\text{stat}} \leq \mathsf{D}_{\text{dyn}}$ (since `int` is a subtype of $\texttt{int} \vee \texttt{bool}$). Now, any operation on t (say, intersection with another gradual type s) can be carried out by intersecting D_{stat} with s's static descriptor and D_{dyn} with s's dynamic descriptor, yielding a new pair that still respects the invariant. By contrast, if we attempted to represent t as `(int, bool)` (static int, dynamic booleans only), the invariant would be broken and a naive component-wise operation could yield incorrect results. In general, a purely static type (with no `?`) is simply represented

[2]Note that we consider that the fields of all types are always defined in the `Descr`, possibly containing the empty type if a given component is absent. In the Elixir implementation, we remove (for performance reasons) the empty fields from the `Descr`, thus intersecting cannot be done directly by the merge, which keeps the fields of the first argument if they are not present in the second. Instead, the behaviour when intersecting two `Descr`s is to remove the field if it is not present in either argument.

by two identical descriptors (its static and dynamic parts coincide), which trivially satisfies the invariant.

9.2 Implementation Details

The overview presented in this chapter establishes the core architectural principles behind our type representation. However, the successful implementation of semantic subtyping requires careful attention to the details of how each component type is represented and how operations on these components are performed efficiently.

Chapter 10 provides the comprehensive treatment of these implementation details, including:

- **Base type representation** (Section 10.1): How fundamental types like integers, floats, and binaries are represented using bit vectors for maximum efficiency.
- **Co-finite type handling** (Section 10.2): The algorithms for representing and manipulating infinite domains like atoms using finite/co-finite set representations.
- **Gradual type implementation** (Section 10.3): The complete treatment of how gradual types are realized using pair representations, including the formal justification and invariant preservation.

These implementation details are crucial for understanding how the modular `Descr` architecture translates into practical, efficient algorithms. While this chapter has established the foundational principles, Chapter 10 demonstrates how those principles are realized in concrete data structures and algorithms. Additionally, Chapter 12 provides a comparative analysis of alternative representations (DNFs, compressed DNFs, union forms) that we evaluated but ultimately chose not to adopt for structural types in the Elixir implementation.

Conclusion

This chapter established the foundational architecture for implementing semantic subtyping systems. We introduced the core `Descr` representation, which provides a modular, component-wise approach to type representation that enables efficient set-theoretic operations while maintaining extensibility.

The key insight presented here is that by representing types as records with disjoint component fields, we can perform complex set operations (union, intersection, difference) simply by computing these operations field-by-field and then combining the results. This modularity proves essential for both static and gradual typing, where the same architectural principles apply directly to gradual types through a pair representation that maintains crucial invariants.

While this overview chapter has established the theoretical foundation and high-level design principles, the success of any semantic subtyping implementation depends on the careful attention to detail in how these components are actually realized. Chapter 10 provides the comprehensive treatment of these implementation details, showing how the modular architecture translates into efficient, practical algorithms for base types, co-finite sets, and gradual type pairs. Chapter 11 details the decision tree approach to implement structural types, and its application to Elixir types: functions, tuples, (proper/improper) lists and maps. Chapter 12 complements this by presenting a detailed comparative analysis of alternative data structures (DNFs, compressed DNFs, union forms) that we explored but ultimately did not adopt, explaining the trade-offs that informed our final design choices.

This separation between architectural principles (this chapter), implementation details (Chapters 10 and 11), and alternative approaches (Chapter 12) reflects our belief that successful type system implementation requires both a clear high-level vision and meticulous attention to the algorithms that realize that vision, as well as an understanding of the design space and trade-offs involved.

We now turn to Chapter 10, which details the concrete data structures and algorithms that realize this architecture.

CHAPTER 10

TYPE REPRESENTATION DETAILS

This chapter provides the detailed implementation of the type representation components introduced in the overview of Chapter 9. While Chapter 9 established the overall `Descr` architecture and its modular design principles, this chapter focuses on the base type and singleton (literal type) components, and on the representation of gradual types. The next chapter, Chapter 11, will focus on the structural type components.

Chapter Roadmap

- **Section 10.1 (Base types)** Encode the six primitive domains as a 6-bit mask; give constant-time set operators and a minimal Elixir implementation.
- **Section 10.2 (Finite/cofinite literals)** Introduce the $\langle \mathit{mode}, S\rangle$ normal form for infinite literal domains; derive closed-form rules for $\wedge$, $\vee$, $\setminus$ and illustrate on atoms.
- **Section 10.3 (Gradual types as pairs)** Use the representation theorem to view τ as $[\tau^{\Downarrow}, \tau^{\Uparrow}]$; enforce the validity invariant $\tau^{\Downarrow} \leq \tau^{\Uparrow}$; contrast naive laws (Prop. 10.3.4) with the invariant-preserving algebra (Fig. 10.1).
  - **Invariant-preserving algebra.** Prove component-wise union/intersection/difference for valid pairs (Prop. 10.3.7).
  - **Alternative disjoint encoding.** Present $\langle \tau^{\Downarrow}, \tau^{\Uparrow} \setminus \tau^{\Downarrow}\rangle$ and its disjointness invariant; give normalized operators and explain why we do not adopt it in practice.
  - **Concrete Elixir integration.** Add a `:dyn` field to `Descr` to carry the dynamic component; define component-wise operators and show a direct implementation of difference (Code 10.2)

10.1 Base Types

The simplest component type is indivisible base types. Elixir provides six fundamental base types that serve as the atomic building blocks of its type system:

- `binary` – binary data (e.g., strings and binary sequences)
- `integer` – integer numbers
- `float` – floating-point numbers
- `pid` – process identifiers
- `port` – I/O port identifiers
- `reference` – unique reference values

In a `Descr`, each of these base types is represented by a Boolean field that indicates whether values of that type are included. For example, the type `binary() or float()` would have the `binary` field set to `true` and the `float` field set to `true`, while all other base-type fields (`integer`, `pid`, `port`, `reference`) would be `false`.

Set-theoretic operations on base-type fields correspond directly to boolean operations. For two types t_1 and t_2, and for a particular base type (say b) with truth values b_1 in t_1 and b_2 in t_2, we have:

$$\begin{array}{ll} \text{Intersection:} & (t_1 \wedge t_2)_b = b_1 \&\& b_2, \\ \text{Union:} & (t_1 \vee t_2)_b = b_1 \mid\mid b_2, \\ \text{Negation:} & (\neg\, t_1)_b = \sim\sim b_1, \\ \text{Difference:} & (t_1 \setminus t_2)_b = b_1 \&\& (\sim\sim b_2), \end{array}$$

where &&, ||, and ~~ on the right-hand side are the usual boolean AND, OR, and NOT.

Efficient bit-level representation. Because each base type corresponds to a bit, we can implement these operations extremely efficiently using bit-level operations. For instance, we can assign a fixed bit position to each of the six base types (e.g., bit 0 for `binary`, bit 1 for `integer`, bit 2 for `float`, etc.). Then a set of base types can be represented as a 6-bit number where `1` in position i indicates the presence of that base type and `0` indicates its absence. Under this encoding, the union of two types corresponds to a bitwise OR, intersection corresponds to bitwise AND, negation corresponds to bitwise NOT (within the 6-bit domain), and difference corresponds to bitwise AND with the complement.

For example, using the bit ordering (binary=0, integer=1, float=2, pid=3, port=4, reference=5), the type `binary() or float()` is represented by the bit pattern `101000` (bits 0 and 2 set to 1). The type `integer() or float()` is `010100` (bits 1 and 2 set). The intersection of these two, `101000 AND 010100`, yields `000000` (no bits set), which correctly represents the empty type (since there is no common element between "binary or floating" and "integer or floating"). The union, `101000 OR 010100`, yields `111100`, representing the type `binary() or integer() or float()` (bits 0,1,2 set).

A minimal implementation of this approach is shown below.

```
defmodule BaseTypes do
```

```
  import Bitwise

  # Bit positions for each base type (0-indexed)
  @binary_bit 0    # binary  -> bit 0
  @integer_bit 1   # integer -> bit 1
  @float_bit 2     # float   -> bit 2
  # ...and so on for pid (3), port (4), reference (5)

  # Bit masks for each base type (using left shift)
  @binary_mask   1 <<< @binary_bit    # 0b000001
  @integer_mask  1 <<< @integer_bit   # 0b000010
  @float_mask    1 <<< @float_bit     # 0b000100
  # ...and so on for pid, port, reference

  def intersection(type1, type2), do: type1 &&& type2
  def union(type1, type2), do: type1 ||| type2
  def difference(type1, type2), do: type1 &&& ~~~type2
  def subtype?(type1, type2), do: difference(type1, type2) == 0
end
```

Listing 10.1: Minimal implementation of base set-theoretic types with bit vectors

This bit-level representation makes operations on basic unions of these fundamental types extremely fast. Moreover, the implementation is *progressive* and modular: initially, a given type can be represented solely by a bit mask (which means that we are treating it as 'all or nothing' for each base type). If later we need to refine that type (for example, to distinguish particular atoms or exclude certain integers), we can *promote* the representation by replacing the simple bit with a more complex data structure for that base type. One such data structure is the co-finite set representation, described next.

10.2 Finite/Cofinite Types (or Literal Types)

Base types like `binary`, `integer`, and `float` are indivisible in our implementation–a type either includes all values of that kind or none. However, for other infinite domains such as atoms, we need finer granularity to express types such as "all atoms except `:a` and `:b`." This section presents a finite/cofinite representation that handles such types efficiently.

The finite/cofinite representation uses a pair structure $\langle tag, S\rangle$ where $tag \in \{\texttt{:union}, \texttt{:negation}\}$ indicates the representation mode and S is a finite set of values from the infinite domain. The interpretation is as follow (recall that constants are both values and types):

$$\langle \texttt{:union}, S\rangle \equiv \bigvee_{c\in S} c \qquad\qquad \langle \texttt{:negation}, S\rangle \equiv \mathit{domain} \setminus \bigvee_{c\in S} c$$

This representation is complete for any subset of an infinite domain: an arbitrary set of values is either finite (or can be made finite by excluding a co-finite part) or co-finite (or can be made co-finite by excluding a finite part). For example, in the domain of atoms we have:

$$\begin{aligned}\langle \texttt{:union}, \{\texttt{:a}, \texttt{:b}\}\rangle &\equiv \texttt{:a} \vee \texttt{:b},\\ \langle \texttt{:negation}, \{\texttt{:c}, \texttt{:d}\}\rangle &\equiv \texttt{atom} \setminus (\texttt{:c} \vee \texttt{:d}),\\ \langle \texttt{:negation}, \emptyset\rangle &\equiv \texttt{atom} \quad \text{(all atoms)},\\ \langle \texttt{:union}, \emptyset\rangle &\equiv \mathbb{0} \quad \text{(empty set)}.\end{aligned}$$

The key benefit of the finite/cofinite format is that set operations can be performed by simple manipulations of the finite set S and the mode tag. Suppose that we have two co-finite types $X_1 = \langle mode_1, S_1\rangle$ and $X_2 = \langle mode_2, S_2\rangle$ over the same domain. Then their intersection, union, and difference are given by the following rules:

Intersection $X_1 \wedge X_2$:

$$\begin{aligned}\langle \texttt{:union}, S_1\rangle \wedge \langle \texttt{:union}, S_2\rangle &= \langle \texttt{:union}, S_1 \cap S_2\rangle,\\ \langle \texttt{:negation}, S_1\rangle \wedge \langle \texttt{:negation}, S_2\rangle &= \langle \texttt{:negation}, S_1 \cup S_2\rangle,\\ \langle \texttt{:union}, S_1\rangle \wedge \langle \texttt{:negation}, S_2\rangle &= \langle \texttt{:union}, S_1 \setminus S_2\rangle,\\ \langle \texttt{:negation}, S_1\rangle \wedge \langle \texttt{:union}, S_2\rangle &= \langle \texttt{:union}, S_2 \setminus S_1\rangle.\end{aligned}$$

Union $X_1 \vee X_2$:

$$\begin{aligned}\langle \texttt{:union}, S_1\rangle \vee \langle \texttt{:union}, S_2\rangle &= \langle \texttt{:union}, S_1 \cup S_2\rangle,\\ \langle \texttt{:negation}, S_1\rangle \vee \langle \texttt{:negation}, S_2\rangle &= \langle \texttt{:negation}, S_1 \cap S_2\rangle,\\ \langle \texttt{:union}, S_1\rangle \vee \langle \texttt{:negation}, S_2\rangle &= \langle \texttt{:negation}, S_2 \setminus S_1\rangle,\\ \langle \texttt{:negation}, S_1\rangle \vee \langle \texttt{:union}, S_2\rangle &= \langle \texttt{:negation}, S_1 \setminus S_2\rangle.\end{aligned}$$

Difference $X_1 \setminus X_2$:

$$\begin{aligned}\langle \texttt{:union}, S_1\rangle \setminus \langle \texttt{:union}, S_2\rangle &= \langle \texttt{:union}, S_1 \setminus S_2\rangle,\\ \langle \texttt{:negation}, S_1\rangle \setminus \langle \texttt{:negation}, S_2\rangle &= \langle \texttt{:union}, S_2 \setminus S_1\rangle,\\ \langle \texttt{:union}, S_1\rangle \setminus \langle \texttt{:negation}, S_2\rangle &= \langle \texttt{:union}, S_1 \cap S_2\rangle,\\ \langle \texttt{:negation}, S_1\rangle \setminus \langle \texttt{:union}, S_2\rangle &= \langle \texttt{:negation}, S_1 \cup S_2\rangle.\end{aligned}$$

In these formulas, operations like $\cap$, $\cup$, $\setminus$ on the right-hand side refer to standard set operations on the finite sets S_1 and S_2. Intuitively, the rules can be understood as follows: (1) the intersection of two finite sets is the finite set of elements they share, whereas the intersection of two co-finite sets is co-finite, excluding exactly those elements excluded by either operand; (2) the union of two finite sets is finite (the union of their elements), while the union of two co-finite sets remains co-finite (excluding only elements excluded by both operands); (3) for difference, subtracting a finite set from a finite set yields a finite set (removing those elements), but subtracting a co-finite set from a co-finite set yields a finite set of elements that the second excludes but the first does not. Each mixed case yields a finite set, reflecting that a finite set minus a co-finite set (or vice versa, in intersection or union) results in a finite set.

The finite/cofinite representation is general and not limited to atoms; the same approach can be applied to any base type (for instance, to represent arbitrary subsets of `integer()` or `binary()` types) by substituting the relevant domain and using the corresponding finite sets of integers, strings, etc.

Example (*Cofinite intersection*)

Consider the intersection of $X_1 = \langle \texttt{:union}, \{\texttt{:a}, \texttt{:b}\}\rangle$ (the type $\texttt{:a} \vee \texttt{:b}$) and $X_2 = \langle \texttt{:negation}, \{\texttt{:b}, \texttt{:c}\}\rangle$ (all atoms except `:b` and `:c`).
Applying the intersection rule for $\langle \texttt{:union}, S_1\rangle \wedge \langle \texttt{:negation}, S_2\rangle$:

$$\begin{aligned} X_1 \wedge X_2 &= \langle \texttt{:union}, \{\texttt{:a}, \texttt{:b}\}\rangle \wedge \langle \texttt{:negation}, \{\texttt{:b}, \texttt{:c}\}\rangle \\ &= \langle \texttt{:union}, \{\texttt{:a}, \texttt{:b}\} \setminus \{\texttt{:b}, \texttt{:c}\}\rangle \\ &= \langle \texttt{:union}, \{\texttt{:a}\}\rangle \end{aligned}$$

This correctly computes the atom type `:a`, the only value belonging to both X_1 and X_2.

10.3 Gradual Types: Pair Representation

Gradual types—those that mention the special unknown type ?—fit naturally into a set-theoretic system via a pair-of-types view. We write gradual types as τ and static types as t, s. Each gradual type has two static approximations: $\tau^{\Downarrow}$ (also written $\tau^{\Downarrow}$), the *minimal* static approximation (values that definitely belong), and $\tau^{\Uparrow}$ (also written $\tau^{\Uparrow}$), the *maximal* one (values that may occur at runtime).

By the Representation Theorem (Chapter 3, 3.2.6), every gradual type satisfies $\tau \simeq \tau^{\Downarrow} \vee (? \wedge \tau^{\Uparrow})$. We therefore represent τ by the pair $[\tau^{\Downarrow}, \tau^{\Uparrow}]$, meaning that in our representation the ? type always appears at the top level.

Equivalently, view a gradual type as the interval $[\tau^{\Downarrow}, \tau^{\Uparrow}]$. For instance, $\texttt{int} \vee ?$ corresponds to the pair $(\texttt{int}, \texttt{term})$.

10.3.1 Pair Representation and the Invariant

This representation as pairs enables the use of the full syntax of gradual types, as defined in the introduction to semantic subtyping (Def. 3.1.1):

Definition 10.3.1 (Gradual Types)**.** *Let c range over constants, b over base types.*

$$\tau \ ::= \ ? \mid c \mid b \mid \tau \times \tau \mid \tau \to \tau \mid \tau \vee \tau \mid \neg\tau \mid \mathbb{0}$$

Here, ? may appear anywhere in the type syntax, but every time a gradual type is constructed it is immediately translated to the pair form. For example, $\{?, \texttt{int}\} \simeq [\{\mathbb{0}, \texttt{int}\}, \{\mathbb{1}, \texttt{int}\}] \simeq ? \wedge \{\mathbb{1}, \texttt{int}\}$, since any tuple with an empty component denotes the empty type. This translation avoids searching for nested ? and lets us reuse static relations on extremal materializations, e.g., $\tau_1 \le \tau_2$ iff ${\tau_1}^{\Downarrow} \le {\tau_2}^{\Downarrow}$ and ${\tau_1}^{\Uparrow} \le {\tau_2}^{\Uparrow}$.

This is convenient as it solves two issues that would appear if we represented gradual types with arbitrarily nested ? types: 1) since the ? type can appear anywhere, it is not quick to determine when a type is gradual or not, as the type could be nested deeply within it; and 2) we have described in Chapter 4 how to design a static type system for Core Elixir, with relations defined on static types. Those relations extend nicely to gradual types by using the static relations *on extremal materializations*, for example $\tau_1 \leq \tau_2$ if ${\tau_1}^{\Downarrow} \leq {\tau_2}^{\Downarrow}$ and ${\tau_1}^{\Uparrow} \leq {\tau_2}^{\Uparrow}$. If ? were nested within τ, computing those materializations would require traversing the type to replace ? with $\mathbb{0}$ or $\mathbb{1}$.

If we were to represent every gradual type using a pair, we must ensure that the pair indeed encodes the extremal materializations ($\tau^{\Downarrow}$ and $\tau^{\Uparrow}$). This is not automatic. Consider $\tau = \texttt{int} \vee (? \wedge \texttt{bool})$. A tempting pair is $[\,\texttt{int}\,,\texttt{bool}\,]$, but materializing gives:

- $\tau^{\Downarrow} = \texttt{int}$ (correct)
- $\tau^{\Uparrow} = \texttt{int} \vee (\mathbb{1} \wedge \texttt{bool}) = \texttt{int} \vee \texttt{bool}$ (not `bool`)

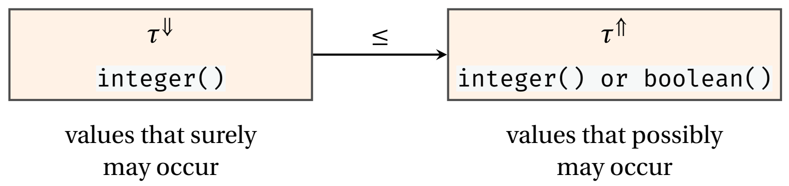
Thus, $[\,\texttt{int}\,,\texttt{int} \vee \texttt{bool}\,]$ is the correct pair, not $[\,\texttt{int}\,,\texttt{bool}\,]$. To rule out such mismatches, we enforce the invariant stated below.

Definition 10.3.2 (Gradual Type as Pair)**.** *A* gradual type τ *is represented as a pair* $[\,t_\sigma\,,t_\delta\,]$ *where:*

- t_σ *is the* static component*, a static type describing values guaranteed to occur at runtime;*
- t_δ *is the* dynamic component*, a static type describing dynamic values that may occur at runtime*

Example: $\tau =$ `integer()` $\vee$ (`?` $\wedge$ `boolean()`)

"Statically an integer, may be a boolean at runtime"



Invariant: $\tau^{\Downarrow} \leq \tau^{\Uparrow}$
Operations: Component-wise on both descriptors
Static type: $\tau^{\Downarrow} = \tau^{\Uparrow}$ (identical descriptors)

Figure 10.1: Gradual type pair representation

A gradual pair $[\,t_\sigma\,,t_\delta\,]$ denotes $t_\sigma \vee (? \wedge t_\delta)$. For example, $\texttt{int} \vee (? \wedge \texttt{bool})$ says that integers are certainly present, and booleans may appear at runtime. This supports best-/worst-case reasoning: a call is safe only if the callee's domain covers both components. For arithmetic, we obtain concise rules:

- If both components are numeric (e.g., $[\,$`int`$\,,\,$`int or float`$\,]$), addition is well-typed and yields a number.
- If the static part is numeric but the dynamic part may not be (e.g., $[\,$`int`$\,,\,$`int or bool`$\,]$), addition is allowed; strict modes may warn or reject.
- If the static part is non-numeric and nonempty (e.g., $[\,$`bool`$\,,\,$`bool`$\,]$), addition is rejected.
- If the static part is empty, the dynamic part must intersect numeric types (e.g., $[\,\mathbb{0}\,,\,$`int or atom`$\,]$); otherwise reject (e.g., $[\,\mathbb{0}\,,\,$`atom`$\,]$).

A subtlety of the pair view is that component-wise comparison can be misleading. For example, $\tau_1 = \texttt{int} \vee (? \wedge \texttt{bool})$ and $\tau_2 = \texttt{int} \vee (? \wedge (\texttt{bool} \vee \texttt{int}))$ are equivalent, even though $\texttt{int} \not\simeq \texttt{int} \vee \texttt{bool}$. The reason is that statically present values are also present dynamically.

To clarify this issue, we begin by recalling the definition of subtyping on gradual types we gave in Chapter 3:

Definition 3.2.7 (Lifted gradual subtyping)**.** *We define the* semantic gradual subtyping *relation* $\leq$ *between gradual types as follows:*

$$\tau_1 \leq \tau_2 \overset{def}{\Leftrightarrow} \begin{cases} \tau_1{}^{\Downarrow} \leq \tau_2{}^{\Downarrow} \\ \tau_1{}^{\Uparrow} \leq \tau_2{}^{\Uparrow} \end{cases}$$

and the semantic gradual equivalence *relation* $\simeq$ *as* $\tau_1 \simeq \tau_2 \Leftrightarrow (\tau_1 \leq \tau_2)$ *and* $(\tau_2 \leq \tau_1)$.

Given $\tau = [\, t_\sigma \,,\, t_\delta \,]$, computing the extrema compared above means replacing the instances of ? judiciously. For the maximal extrema, we replace the only covariant occurrence of ? with $\mathbb{1}$, thus:

$$\tau^{\Uparrow} \simeq t_\sigma \vee (\mathbb{1} \wedge t_\delta) \simeq t_\sigma \vee t_\delta$$

and, conversely, for the minimal extrema, we replace ? with $\mathbb{0}$:

$$\tau^{\Downarrow} \simeq t_\sigma \vee (\mathbb{0} \wedge t_\delta) \simeq t_\sigma$$

Going back to the definition of subtyping of above with the computed formulas for the extrema, we see that subtyping on pairs can be defined as follows:

Definition 10.3.3 (Naive Gradual Subtyping on Pairs)**.**

$$[\, t_1 \,,\, t_2 \,] \leq [\, s_1 \,,\, s_2 \,] \overset{def}{\Leftrightarrow} \begin{cases} t_1 \leq s_1 \\ t_1 \vee t_2 \leq s_1 \vee s_2 \end{cases}$$

Similar considerations apply to set operations on arbitrary pairs; for instance:

$$[\, t_1 \,,\, t_2 \,] \vee [\, s_1 \,,\, s_2 \,] = [\, t_1 \vee s_1 \,,\, (t_1 \vee t_2) \vee (s_1 \vee s_2) \,]$$

In summary, adding also intersection and difference, the base operations of subtyping and set operations on pairs of types come up as:

Property 10.3.4 (Naive Gradual Set Operations on Pairs)**.**

$$\begin{aligned}
[\, t_1 \,,\, t_2 \,] \le [\, s_1 \,,\, s_2 \,] &\iff (t_1 \le s_1) \wedge (t_1 \vee t_2 \le s_1 \vee s_2) \\
[\, t_1 \,,\, t_2 \,] \vee [\, s_1 \,,\, s_2 \,] &\simeq [\, t_1 \vee s_1 \,,\, (t_1 \vee t_2) \vee (s_1 \vee s_2) \,] \\
[\, t_1 \,,\, t_2 \,] \wedge [\, s_1 \,,\, s_2 \,] &\simeq [\, t_1 \wedge s_1 \,,\, (t_1 \vee t_2) \wedge (s_1 \vee s_2) \,] \\
[\, t_1 \,,\, t_2 \,] \setminus [\, s_1 \,,\, s_2 \,] &\simeq [\, t_1 \setminus (s_1 \vee s_2) \,,\, (t_1 \vee t_2) \setminus s_1 \,]
\end{aligned}$$

Proof. Writing $[\, t_1 \,,\, t_2 \,]$ as $t_1 \vee (? \wedge t_2)$, minimal and maximal materializations act by replacing covariant (resp. contravariant) occurrences of ? with $\mathbb{1}$ (resp. $\mathbb{0}$). For difference, let $\tau \overset{\text{def}}{=} (t_1 \vee (? \wedge t_2)) \setminus (s_1 \vee (? \wedge s_2))$. Then, with contravariance of the subtrahend:

$$\tau^{\Downarrow} \simeq (t_1 \vee (\mathbb{0} \wedge t_2)) \setminus (s_1 \vee (\mathbb{1} \wedge s_2)) \simeq t_1 \setminus (s_1 \vee s_2) \tau^{\Uparrow} \simeq (t_1 \vee (\mathbb{1} \wedge t_2)) \setminus (s_1 \vee (\mathbb{0} \wedge s_2)) \simeq (t_1 \vee t_2) \setminus s_1 \quad \square$$

However, these formulas involve many unions and intersections. To recover simple, component-wise rules, we maintain pairs in the *valid form* $[\, \tau^{\Downarrow} \,,\, \tau^{\Uparrow} \,]$, characterized by the following invariant:

Invariant 10.3.5. *A gradual pair type* $[\, t_1 \,,\, t_2 \,]$ *is valid iff it satisfies* $t_1 \le t_2$.

Intuitively, every value that will appear statically may also appear dynamically.

Property 10.3.6 (Extremal materializations are the valid pairs)**.** *A gradual type* τ *admits a valid pair* $[\, t_1 \,,\, t_2 \,]$ *as its representation iff* $\tau^{\Downarrow} \simeq t_1$ *and* $\tau^{\Uparrow} \simeq t_2$.

Proof. Since $\tau^{\Downarrow} \le \tau^{\Uparrow}$, the pair $[\, \tau^{\Downarrow} \,,\, \tau^{\Uparrow} \,]$ is valid. The converse follows from the definition of materialization. □

10.3.2 Invariant-Preserving Operations

Having established the invariant $\tau^{\Downarrow} \le \tau^{\Uparrow}$ for valid pairs, we now show that set-theoretic operations preserve this property. In other terms, component-wise set-theoretic operations on valid pairs preserve validity.

Proposition 10.3.7 (Efficient Set Operations on Valid Pairs)**.** *Let* $[\, t_1 \,,\, t_2 \,]$ *and* $[\, s_1 \,,\, s_2 \,]$ *be valid gradual types (i.e.,* $t_1 \le t_2$ *and* $s_1 \le s_2$*). Then:*

$$\begin{aligned}
[\, t_1 \,,\, t_2 \,] \vee [\, s_1 \,,\, s_2 \,] &\overset{\text{def}}{=} [\, t_1 \vee s_1 \,,\, t_2 \vee s_2 \,], \\
[\, t_1 \,,\, t_2 \,] \wedge [\, s_1 \,,\, s_2 \,] &\overset{\text{def}}{=} [\, t_1 \wedge s_1 \,,\, t_2 \wedge s_2 \,], \\
[\, t_1 \,,\, t_2 \,] \setminus [\, s_1 \,,\, s_2 \,] &\overset{\text{def}}{=} [\, t_1 \setminus s_2 \,,\, t_2 \setminus s_1 \,]
\end{aligned}$$

are valid gradual types.

Proof Sketch. The key insight is that for valid pairs, the dynamic component equals the union of both components: $t_2 \simeq t_1 \vee t_2$ and $s_2 \simeq s_1 \vee s_2$. This allows us to derive the results directly from Property 10.3.4.
For union: $t_1 \vee s_1 \leq t_2 \vee s_2$ follows from the monotonicity of union and the validity of the input pairs.
For intersection: $t_1 \wedge s_1 \leq t_2 \wedge s_2$ follows from the monotonicity of intersection.
For difference: $t_1 \setminus s_2 \leq t_2 \setminus s_1$ holds because $t_1 \leq t_2$ and removing larger sets from the right component preserves the ordering. □

Subtyping also follows directly from the definition of subtyping for extremal materializations.

Property 10.3.8 (Subtyping on Valid Pairs)**.** *Let* $[\, t_1 \,,\, t_2 \,]$ *and* $[\, s_1 \,,\, s_2 \,]$ *be valid gradual types. Then:*

$$[\, t_1 \,,\, t_2 \,] \leq [\, s_1 \,,\, s_2 \,] \iff (t_1 \leq s_1) \wedge (t_2 \leq s_2)$$

Lemma 10.3.9 (Closure under Constructors)**.** *If gradual types are represented as valid pairs, then all set operations and type operators (e.g., subtyping, compatibility, emptiness) applied component-wise yield valid pairs.*

Corollary 10.3.10. *Gradual typing operators can be systematically defined in terms of the corresponding operations on static types applied component-wise.*

10.3.3 Alternative Encoding: Disjoint Pairs

While the valid pair representation $[\, \tau^{\Downarrow} \,,\, \tau^{\Uparrow} \,]$ provides elegant set-theoretic operations, an alternative approach we considered is to store the static component and exactly the additional dynamic component. We discuss this disjoint pair representation briefly for completeness, though we did not adopt it in practice. We represent a gradual type τ as the disjoint pair $\langle \tau^{\Downarrow}, \tau^{\Uparrow} \setminus \tau^{\Downarrow} \rangle$, storing the static component and exactly (but not more than) the additional dynamic component.

Definition 10.3.11 (Disjoint Pair Representation)**.** *A gradual type* τ *can be represented as the pair* $\langle t_\sigma, t_\delta \rangle$ *where:*

- $t_\sigma = \tau^{\Downarrow}$ *is the static component;*
- $t_\delta = \tau^{\Uparrow} \setminus \tau^{\Downarrow}$ *is the strictly dynamic component.*

This representation enjoys a key invariant that makes it attractive for certain applications:

Invariant 10.3.12 (Disjointness Invariant)**.** *A disjoint pair type* $\langle t_1, t_2 \rangle$ *is* valid *iff it satisfies* $t_1 \wedge t_2 \simeq \mathbb{0}$*. We may also call* $\langle t_1, t_2 \rangle$ *a* valid gradual type.

This invariant ensures that the static and dynamic components are disjoint, which provides two key benefits:

Non-redundant representation. Because the static and dynamic parts do not overlap, a gradual type can often be printed or displayed without repeating information. For example, consider the gradual type $\tau = \texttt{float()} \vee (? \wedge \texttt{integer()})$ (statically a float, but might be an integer). Under the $(\tau^{\Downarrow}, \tau^{\Uparrow})$ scheme, τ would be represented as $\left[\texttt{float()}\,\mathbf{,}\,\texttt{float()} \vee \texttt{integer()}\right]$, which if naively printed might appear as "`float() or (dynamic() and (float() or integer()))`," redundantly mentioning 'float()'. In the disjoint representation, we would store τ as $\langle\texttt{float()}, \texttt{integer()}\rangle$, which directly corresponds to the more concise description "`float() or (dynamic() and integer())`." In general, the disjoint form ensures that no static type portion is ever duplicated in the dynamic portion.

Static-type detection. It is trivial to test whether a given gradual type contains no ?: this is exactly the case when the second component of the pair is $\mathbb{0}$. In the disjoint encoding, $\langle t, s\rangle$ represents a purely static type if and only if $s \simeq \mathbb{0}$ (because that means there are no "extra" dynamic values beyond the static ones).

Despite these advantages, the disjoint representation is more complicated to maintain across arbitrary type operations. Ensuring the invariant $T \wedge D \simeq \mathbb{0}$ after each operation requires additional normalization steps. For example, if we denote a gradual type in disjoint form as $\langle t_1, t_2\rangle$ (with $t_1 \wedge t_2 = \mathbb{0}$) and another as $\langle s_1, s_2\rangle$ (with $s_1 \wedge s_2 = \mathbb{0}$), then the fundamental operations must be defined with care to keep the result disjoint:

Property 10.3.13 (Set Operations for Disjoint Pairs)**.**

$$\langle t_1, t_2\rangle \vee \langle s_1, s_2\rangle \simeq \langle t_1 \vee s_1, (t_1 \vee t_2) \vee (s_1 \vee s_2) \setminus (t_1 \vee s_1)\rangle$$
$$\langle t_1, t_2\rangle \wedge \langle s_1, s_2\rangle \simeq \langle t_1 \wedge s_1, (t_1 \vee t_2) \wedge (s_1 \vee s_2) \setminus (t_1 \wedge s_1)\rangle$$
$$\langle t_1, t_2\rangle \setminus \langle s_1, s_2\rangle \simeq \langle t_1 \setminus (s_1 \vee s_2), (t_1 \vee t_2) \setminus s_1 \setminus (t_1 \setminus (s_1 \vee s_2))\rangle$$

Proof. We derive these operations by converting to the valid representation, performing the operation, then normalizing back to maintain disjointness:
For union: $\langle t_1, t_2\rangle \simeq \left[\, t_1 \,\mathbf{,}\, t_1 \vee t_2 \,\right]$ and $\langle s_1, s_2\rangle \simeq \left[\, s_1 \,\mathbf{,}\, s_1 \vee s_2 \,\right]$. The union becomes:

$$\begin{aligned}\left[\, t_1 \,\mathbf{,}\, t_1 \vee t_2 \,\right] \vee \left[\, s_1 \,\mathbf{,}\, s_1 \vee s_2 \,\right] &\simeq \left[\, t_1 \vee s_1 \,\mathbf{,}\, (t_1 \vee t_2) \vee (s_1 \vee s_2) \,\right] \\ &\simeq \langle t_1 \vee s_1, (t_1 \vee t_2) \vee (s_1 \vee s_2) \setminus (t_1 \vee s_1)\rangle\end{aligned}$$

The normalization step $(t_1 \vee t_2) \vee (s_1 \vee s_2) \setminus (t_1 \vee s_1)$ ensures that the result satisfies the disjointness invariant. The other operations follow similarly by expanding to the valid representation, applying the standard operations from Lemma 10.3.4, and then normalizing back to maintain disjointness. □

Each of these definitions computes a preliminary result (as if using the $(\tau^{\Downarrow}, \tau^{\Uparrow})$ rules) and then subtracts out any overlap between what would be the new static part and new dynamic part. For instance, in the union: $t_1 \vee s_1$ is the new static part, and we initially consider all values that either type could have $((t_1 \vee t_2) \vee (s_1 \vee s_2))$ as a candidate for the dynamic part, but we then remove $(t_1 \vee s_1)$ from it because those values are now accounted for in the static part. Similar logic applies to intersection and difference, where we remove the intersection $t_1 \wedge s_1$ or the portion of t_1 that remains after subtraction, respectively, from the would-be dynamic component.

These extra set difference operations are what make the disjoint approach more computationally expensive. Essentially, every time we combine types, we must immediately "re-normalize" the result to maintain disjointness. We derived the above formulas by converting disjoint pairs to the $(\tau^{\Downarrow}, \tau^{\Uparrow})$ form, applying the simpler component-wise operations, and then converting back by splitting off the overlap. While correct, this process means additional work on each operation.

Takeaway. Our belief is that the benefits of the disjoint representation do not outweigh its costs. The redundancy it avoids is mostly superficial (affecting the printed form of types but not their meaning), and a pretty-printer can always post-process a $[\tau^{\Downarrow}, \tau^{\Uparrow}]$ representation to eliminate obvious duplications if needed. On the other hand, the performance penalty of constantly performing set differences can be significant, especially given that type operations are ubiquitous in the compiler's type inference and checking phases. Moreover, the disjoint invariant can lead to counterintuitive intermediate forms. For example, one might expect the type `dynamic() or integer()` to simplify to just `dynamic()` (since `dynamic()` conceptually includes all values, including integers). In the disjoint scheme, that type would be represented as ⟨`integer()`,(`not integer`)⟩, which if naively interpreted looks like "integer or dynamic" again; careful simplification is required to realize that it is equivalent to just `dynamic()`. Similarly, taking the union of two gradual types in disjoint form can temporarily produce a type whose dynamic part overlaps with the static part, requiring an immediate cleanup step (as reflected in the formulas above).

For these reasons, our implementation sticks to the simpler $[\tau^{\Downarrow}, \tau^{\Uparrow}]$ pair representation for gradual types. It guarantees a unique form for each gradual type (thanks to the $\tau^{\Downarrow} \leq \tau^{\Uparrow}$ invariant) and allows all operations to reuse efficient static-type algorithms in a straightforward way. The small amount of redundancy in the representation (and in pretty-printed output) is an acceptable trade-off for the gains in implementation simplicity and performance.

10.3.4 Concrete Elixir implementation

While the pair representation is fine, if we wanted to keep the `Descr` form of a record type, is it also doable. For that recall that the `Descr` represents types as a union of non-overlapping components. This perfectly corresponds to the disjoint pair representation, which encodes τ as $\langle \tau^{\Downarrow}, \tau^{\Uparrow} \setminus \tau^{\Downarrow} \rangle$. This suggests the idea to add a new field `:dyn` to the `Descr` structure, which represents the dynamic part of the type (that which is intersected with ?), while the rest of the `Descr` (that is, the new `Descr` minus the `:dynamic` field) represents the static part of the type. For example, type `integer()` ∨ (? ∧ `float()`) would be represented as:

`%{:integer ⇒ true, :dyn ⇒ %{:float ⇒ true}}`

We can push this idea and, instead of putting the disjoint dynamic part $(\tau^{\Uparrow} \setminus \tau^{\Downarrow})$ in the `:dyn` field, directly store $\tau^{\Uparrow}$ in the `:dyn` field (which introduces a slight repetition of the static part, but with the benefit of being able to perform set operations directly on the components). Proposition 10.3.7 ensures that both intersection and union can be performed directly on the components, by delegating to the dynamic component the corresponding static operations (say, `intersection_static` and `union_static`), while difference (see Code 10.2) can be done with minimal efforts by swapping them.

```elixir
def difference(left, right) do
  # pop the (dyn, static) pair from each gradual type
  {left_dynamic, left_static} = Map.pop(left, :dyn, left)
  {right_dynamic, right_static} = Map.pop(right, :dyn, right)
  # compute the new dyn. part
  dynamic_part = difference_static(left_dynamic, right_static)
  # insert the new dyn. part back into the new static part
  Map.put(difference_static(left_static, right_dynamic), :dyn, dynamic_part)
end
```

Listing 10.2: Difference operation on gradual types in Elixir.

Conclusion

This chapter presented efficient representations for base types, literal types, and gradual types. Base types use bit-vectors for constant-time operations; literal domains employ finite/cofinite pairs that reduce set operations to finite set manipulations; gradual types rely on the invariant-preserving pair $[\tau^{\Downarrow}, \tau^{\Uparrow}]$, enabling component-wise operations that reuse static algorithms.

The unifying principle is *progressive refinement*: begin with the simplest abstraction (bits) and promote components to richer structures only when precision demands it, preserving the same component-wise algebra throughout. This approach integrates cleanly into the modular `Descr` architecture via a dedicated `:dyn` field for gradual types.

The next chapter applies these same design principles to structural types (functions, tuples, lists, maps).

CHAPTER 11

IMPLEMENTING STRUCTURAL TYPES

This chapter implements structural types. We use Binary Decision Diagrams (BDDs) as the core Boolean backbone and introduce Ternary Decision Diagrams (TDDs) to avoid exponential blow-ups in unions. We then specialize functions, tuples, lists, and maps in Elixir within the modular `Descr` architecture.

Chapter Roadmap

- **Section 11.1 (Binary Decision Diagrams):** Core representation for structural types with polynomial-time Boolean operations and Elixir examples.
- **Section 11.2 (Ternary Decision Diagrams):** Lazy decision trees that prevent exponential blow-up across consecutive unions while preserving semantics.
- **Section 11.3 (Application to Elixir):** Production integration via the modular `Descr` design, with specialized representations:
  - Multi-arity functions as co-finite TDDs (§11.3.1)
  - Tuples as open/closed structures (§11.3.2)
  - Lists as proper/improper structures (§11.3.3)
  - Maps as hybrid record-dictionary structures (§11.3.4)

11.1 Background on Binary Decision Diagrams

Frisch (2004) proposes the use of binary decision diagrams ($\mathscr{B}$) as a data structure to represent set-theoretic types. Their use is recalled and expanded in Castagna (2016). Binary decision diagrams were introduced by Akers (1978) and further developed by Bryant (1986). We recall here their main ideas and algorithms.

Note

As we discuss in Chapter 12, BDDs emerged as our optimal choice after evaluating several alternative representations, including disjunctive normal forms (DNF) and union forms. The key advantage of BDDs lies in the symmetry of their implementation of set operations union, intersection, difference: these all offer the same complexity (polynomial in the size of the input trees); while DNFs only offer polynomial union and intersection, and difference is exponential (see Chapter 12).

Definition 11.1.1 (Binary Decision Diagram)**.** *A binary decision diagram ($\mathcal{B}$) represents a propositional formula on positive or negative types.*
The syntax of $\mathcal{B}$ is defined inductively as follows:

$$\mathcal{B} ::= 1 \mid 0 \mid n\,?\,\mathcal{B} : \mathcal{B}$$

where n (nodes of the BDD) ranges over atomic types, that is, types with a top-level constructor, e.g. functions $(t_1, \ldots, t_n) \to t$, tuples $\{\, t_1, \ldots, t_n \,\}$, maps (introduced in Section 1.1.5, the shape of their nodes is given in Section 11.3.4), lists (in Section 11.3.3).

The interpretation of $\mathcal{B}$ in set-theoretic semantics is given by:

$$\begin{aligned}
[\![1]\!] &= \mathbb{1} \\
[\![0]\!] &= \mathbb{O} \\
[\![(n\,?\,\mathcal{B}_1 : \mathcal{B}_2)]\!] &= ([\![n]\!] \wedge [\![\mathcal{B}_1]\!]) \vee (\neg[\![n]\!] \wedge [\![\mathcal{B}_2]\!])
\end{aligned}$$

Thus, $\mathcal{B}$ encodes a union of intersections of positive and negative atomic types. For example, $\mathcal{B} = (\texttt{int} \to \texttt{int})\,?\,(\texttt{bool} \to \texttt{bool})\,?\,1 : 0 : 0$ encodes the type $(\texttt{int} \to \texttt{int}) \wedge (\texttt{bool} \to \texttt{bool})$. This is represented graphically in Figure 11.1.

$$\begin{tikzcd}
& & (\texttt{int} \to \texttt{int}) \arrow[dl] \arrow[dr, "\neg"] & \\
& (\texttt{bool} \to \texttt{bool}) \arrow[dl] \arrow[dr, "\neg"] & & 0 \\
1 & & 0 &
\end{tikzcd}$$

Figure 11.1: $\mathcal{B}$ representing $(\texttt{int} \to \texttt{int}) \wedge (\texttt{bool} \to \texttt{bool})$

A key insight from Frisch (2004) is that the types representing different kinds of data (tuples, records, pairs) are pairwise disjoint. Consequently, each kind can be represented by a separate $\mathcal{B}$.

Therefore, a type is represented by a triple of $\mathcal{B}$s, one for each kind of data: tuples, records, and pairs. The set-theoretic operations are then performed component-wise on these $\mathcal{B}$.

Set Operations on BDDs – Frisch (2004) Let $\mathscr{B}$, $\mathscr{B}_1$, and $\mathscr{B}_2$ denote generic BDDs, where $\mathscr{B}_1 = n_1 \mathbin{?} C_1 \mathbin{:} D_1$ and $\mathscr{B}_2 = n_2 \mathbin{?} C_2 \mathbin{:} D_2$. We suppose given an arbitrary order on atomic types $n_1 < n_2$[1], and we maintain an invariant that the nodes are strictly increasing with respect to this order. This ensures that no atomic type is duplicated along each path from the root to a leaf (that is, among each disjunct of the DNF). The unions, intersections, and differences of BDDs are defined as follows.

The base cases for set operations are:

$$\begin{aligned}
&1 \vee \mathscr{B} = \mathscr{B} \vee 1 = 1\\
&0 \vee \mathscr{B} = \mathscr{B} \vee 0 = \mathscr{B}\\
&1 \wedge \mathscr{B} = \mathscr{B} \wedge 1 = \mathscr{B}\\
&0 \wedge \mathscr{B} = \mathscr{B} \wedge 0 = 0\\
&\mathscr{B} \setminus 1 = 0\\
&\mathscr{B} \setminus 0 = \mathscr{B}\\
&0 \setminus \mathscr{B} = 0\\
&1 \setminus \mathscr{B} = \neg\mathscr{B}
\end{aligned}$$

For BDDs with internal nodes, the operations are defined recursively based on the ordering of literals:

$$\textbf{Union}: \qquad \mathscr{B}_1 \vee \mathscr{B}_2 = \begin{cases} n_1 \mathbin{?} (C_1 \vee C_2) \mathbin{:} (D_1 \vee D_2) & \text{if } n_1 = n_2\\ n_1 \mathbin{?} (C_1 \vee \mathscr{B}_2) \mathbin{:} (D_1 \vee \mathscr{B}_2) & \text{if } n_1 < n_2\\ n_2 \mathbin{?} (\mathscr{B}_1 \vee C_2) \mathbin{:} (\mathscr{B}_1 \vee D_2) & \text{if } n_1 > n_2 \end{cases}$$

$$\textbf{Intersection}: \qquad \mathscr{B}_1 \wedge \mathscr{B}_2 = \begin{cases} n_1 \mathbin{?} (C_1 \wedge C_2) \mathbin{:} (D_1 \wedge D_2) & \text{if } n_1 = n_2\\ n_1 \mathbin{?} (C_1 \wedge \mathscr{B}_2) \mathbin{:} (D_1 \wedge \mathscr{B}_2) & \text{if } n_1 < n_2\\ n_2 \mathbin{?} (\mathscr{B}_1 \wedge C_2) \mathbin{:} (\mathscr{B}_1 \wedge D_2) & \text{if } n_1 > n_2 \end{cases}$$

$$\textbf{Difference}: \qquad \mathscr{B}_1 \setminus \mathscr{B}_2 = \begin{cases} n_1 \mathbin{?} (C_1 \setminus C_2) \mathbin{:} (D_1 \setminus D_2) & \text{if } n_1 = n_2\\ n_1 \mathbin{?} (C_1 \setminus \mathscr{B}_2) \mathbin{:} (D_1 \setminus \mathscr{B}_2) & \text{if } n_1 < n_2\\ n_2 \mathbin{?} (\mathscr{B}_1 \setminus C_2) \mathbin{:} (\mathscr{B}_1 \setminus D_2) & \text{if } n_1 > n_2 \end{cases}$$

$$\textbf{Negation}: \qquad \neg\mathscr{B} = \begin{cases} 0 & \text{if } \mathscr{B} = 1\\ 1 & \text{if } \mathscr{B} = 0\\ n \mathbin{?} \neg C \mathbin{:} \neg D & \text{if } \mathscr{B} = n \mathbin{?} C \mathbin{:} D \end{cases}$$

[1]For the Elixir implementation, we use the order on all terms defined formally in Figure 6.4.

Key insight. The principle of BDDs is to provide a natural way to represent disjunctive normal forms (DNF), which were proved in Chapter 3 to be an effective universal form for types. Recall that a DNF is a disjunction of conjunctions of literals, written as:

$$\bigvee_{i\in I} \bigwedge_{n\in P_i} n \wedge \bigwedge_{n\in N_i} \neg n$$

where n ranges over 'atomic types' (type nodes) such as function types $(t_1,\dots,t_n) \to t$, tuples $\{\, t_1,\dots,t_n \,\}$, maps and lists. These are atomic types because they have not been composed via set operations (for example, `int` $\vee$ `float` is not atomic since it applies union to the atomic types `int` and `float`).

The interesting part is that, given some $\mathscr{B} = n\,?\,C : D$, its interpretation is naturally that of a DNF: suppose by induction that the interpretation of BDDs is a DNF, then the interpretation of $\mathscr{B}$ is: $(n \wedge [\![C]\!]) \vee (\neg n \wedge [\![D]\!])$, which is directly equivalent to a DNF by sweeping the literals n and $\neg n$ under the disjunctions of $[\![C]\!]$ and $[\![D]\!]$.

Simplifications. After any operation, we simplify the BDD by the standard rewrite

$$n\,?\,\mathscr{B} : \mathscr{B} \implies \mathscr{B},$$

which is in time $\mathcal{O}(|\mathscr{B}|)$. A less costly simplification, which we use in practice, is the special case:

$$n\,?\,0 : 0 \implies 0,$$

Typical apply(op, $\mathscr{B}_1$, $\mathscr{B}_2$) behavior and intuition for BDD operations Consider two BDDs whose variables respect the same total order. Let $n_1 < \cdots < n_k$ be the variables that appear in $\mathscr{B}_1$ and $m_1 < \cdots < m_\ell$ those that appear in $\mathscr{B}_2$. In the common non-interleaving situation where either $n_1 < \cdots < n_k < m_1 < \cdots < m_\ell$ (or the symmetric case), the standard apply algorithm `apply`$(op, \mathscr{B}_1, \mathscr{B}_2)$ simplifies substantially. In the first case, it never recurses into subgraphs of $\mathscr{B}_2$: it traverses $\mathscr{B}_1$ and, at leaves, combines with $\mathscr{B}_2$ (see Figures 11.2-11.3).

Concretely: for the union operation $(\vee)$, replace every 0-leaf of $\mathscr{B}_1$ with $\mathscr{B}_2$, with cost $\mathcal{O}(|\mathscr{B}_1|)$; for intersection $(\wedge)$, replace every 1-leaf of $\mathscr{B}_1$ with $\mathscr{B}_2$; and for difference $(\setminus)$, implemented as $a \wedge \neg b$, replace every 1-leaf of $\mathscr{B}_1$ with $\neg\mathscr{B}_2$.

If instead $m_1 < \cdots < m_\ell < n_1 < \cdots < n_k$, the same reasoning applies with the roles swapped, yielding time $\mathcal{O}(|\mathscr{B}_2|)$. This non-interleaving case offers a simple mental model; when variables interleave, `apply` must explore both BDDs.

11.1.1 Implementation of union

We assume that the ordered BDD nodes have the shape $n\,?\,L : R$ with test node n and branches L and R, 0 and 1 are `:bdd_bot` and `:bdd_top`. In Elixir:

```
$type bdd = :bdd_bot or :bdd_top or {atomic(), bdd(), bdd()}
```

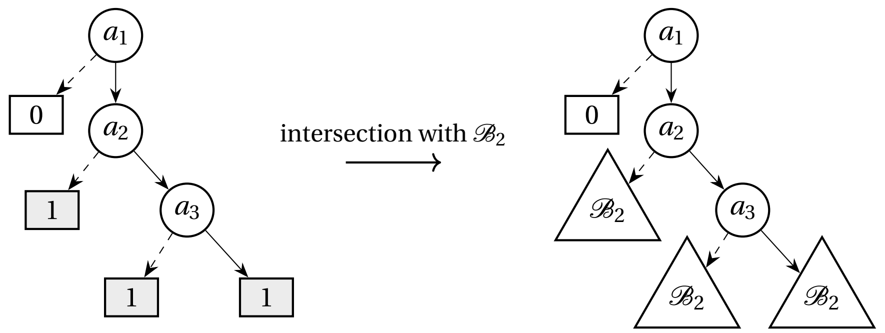


Figure 11.2: Intersection without interleaving ($a_1 < \cdots < a_n < b_1 < \cdots < b_m$): on the left BDD$_1$ (with 1 leaves), on the right the result of $\wedge$ with $\mathscr{B}_2$ (each 1 leaf replaced by $\mathscr{B}_2$).

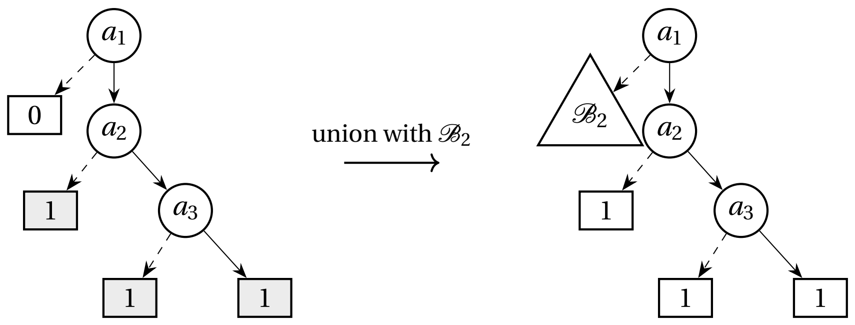


Figure 11.3: Union without interleaving ($a_1 < \cdots < a_n < b_1 < \cdots < b_m$): on the left BDD$_1$ (with 1 leaves), on the right the result of $\vee$ with $\mathscr{B}_2$ (each 0 leaf replaced by $\mathscr{B}_2$).

where `:atomic()` represents the atomic types (functions, tuples, maps, lists). Then the function for union is:

```elixir
# Leaf cases
defp bdd_union(:bdd_bot, other), do: other
defp bdd_union(other, :bdd_bot), do: other
defp bdd_union(:bdd_top, _), do: :bdd_top
defp bdd_union(_, :bdd_top), do: :bdd_top

# Node cases
defp bdd_union(bdd1 = {lit1, l1, r1}, bdd2 = {lit2, l2, r2}) do
  cond do
    lit1 < lit2 -> {lit1, bdd_union(l1, bdd2), bdd_union(r1, bdd2)}
    lit1 > lit2 -> {lit2, bdd_union(bdd1, l2), bdd_union(bdd1, r2)}
    true ->  {lit1, bdd_union(l1, l2), bdd_union(r1, r2)}
  end
end
```

Listing 11.1: Elixir implementation of BDD union

We notice the disjunction on the order of literals. By doing this for every operation, we make sure that along a path, the rank of literals is strictly increasing. Thus, literals are not duplicated down.

By branching on the (total) order of literals, every root-to-leaf path strictly increases in literal rank, ensuring that no literal is tested twice along a path.

11.1.2 Complexity of BDD operations

Let $\mathscr{B}_1$ and $\mathscr{B}_2$ be BDDs. Because operations recurse down the trees and stop at leaves, base cases are constant time (except for $1 \smallsetminus \mathscr{B}$, which first negates $\mathscr{B}$). Writing $|\mathscr{B}|$ for the number of nodes, the union algorithm of Listing 11.1 runs in worst case $\mathscr{O}(|\mathscr{B}_1| \cdot |\mathscr{B}_2|)$; this is the standard bound proved by induction on $|\mathscr{B}_1| + |\mathscr{B}_2|$.

The same worst-case bound $\mathscr{O}(|\mathscr{B}_1| \cdot |\mathscr{B}_2|)$ holds for $\wedge$ and difference (with the usual non interleaving speed ups recalled earlier). Negation swaps leaves and runs in $\mathscr{O}(|\mathscr{B}|)$.

11.2 Ternary Decision Diagrams

Binary decision diagrams can grow exponentially in size when performing consecutive union operations. This limitation was identified by Frisch (2004), who proposed ternary decision trees to encode lazy unions using an additional middle branch. This section presents ternary decision diagrams (TDDs, also called lazy BDDs because they lazily encodes unions in their middle node) as an efficient solution to this problem.

11.2.1 Definition and Semantics

Definition 11.2.1 (Ternary Decision Diagram – Frisch (2004))**.** *A ternary decision diagram (TDD) node is represented as $\mathscr{T} = n\,?\,C : U : D$, where:*

- *n is a type node (atomic type)*
- *C is the* constrained *branch (taken when n holds)*
- *U is the* uncertain *branch (union that can be taken regardless of n)*
- *D is the* dual *branch (taken when n does not hold)*

The semantics of a TDD node is:

$$[\![\mathscr{T}]\!] = ([\![n]\!] \cap [\![C]\!]) \cup [\![U]\!] \cup ((\mathscr{D} \setminus [\![n]\!]) \cap [\![D]\!])$$

The key insight is that the middle branch U allows unions to be kept lazy, postponing expansion until needed for intersection or difference operations. For instance, given two atomic types $n_1 < n_2$, encoding $n_1 \vee n_2$ as a BDD yields $n_1\,?\,1 : (n_2\,?\,1 : 0)$, interpreted as $n_1 \vee (\neg n_1 \wedge n_2)$. This is equivalent to $n_1 \vee n_2$ because the $\neg n_1$ constraint is redundant: any value in both n_1 and n_2 is already captured by the first disjunct. By comparison, a TDD can represent the same type as $n_1\,?\,1 : (n_2\,?\,1 : 0 : 0) : 0$, which directly encodes $n_1 \vee n_2$ without the disjointness expansion.

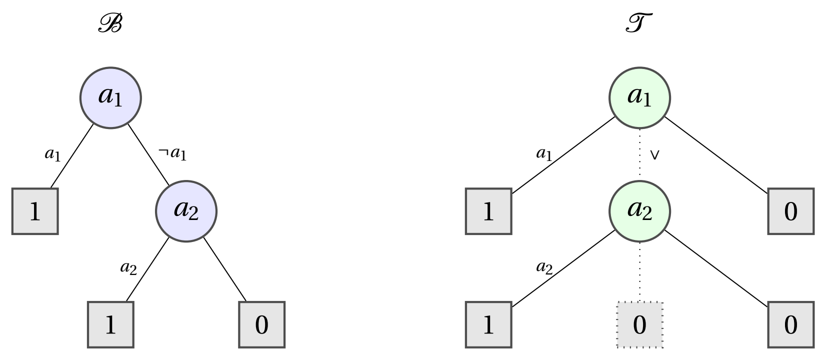


Figure 11.4: BDD vs. TDD union example.

11.2.2 Operations

Let $\mathscr{T}_1 = n_1 \,?\, C_1 : U_1 : D_1$ and $\mathscr{T}_2 = n_2 \,?\, C_2 : U_2 : D_2$ be two TDDs. Frisch (2004) define the following operations:

- **Union**:

$$\mathscr{T}_1 \vee \mathscr{T}_2 = \begin{cases} n_1 \,?\, C_1 \vee C_2 : U_1 \vee U_2 : D_1 \vee D_2 & \text{if } n_1 = n_2, \\ n_1 \,?\, C_1 : U_1 \vee \mathscr{T}_2 : D_1 & \text{if } n_1 < n_2, \\ n_2 \,?\, C_2 : (\mathscr{T}_1 \vee U_2) : D_2 & \text{if } n_1 > n_2; \end{cases}$$

- **Intersection**:

$$\mathscr{T}_1 \wedge \mathscr{T}_2 = \begin{cases} n_1 \,?\, (C_1 \vee U_1) \wedge (C_2 \vee U_2) : 0 : (D_1 \vee U_1) \wedge (D_2 \vee U_2) & \text{if } n_1 = n_2, \\ n_1 \,?\, (C_1 \wedge \mathscr{T}_2) : (U_1 \wedge \mathscr{T}_2) : (D_1 \wedge \mathscr{T}_2) & \text{if } n_1 < n_2, \\ n_2 \,?\, (\mathscr{T}_1 \wedge C_2) : (\mathscr{T}_1 \wedge U_2) : (\mathscr{T}_1 \wedge D_2) & \text{if } n_1 > n_2; \end{cases}$$

- **Difference**:

$$\mathscr{T}_1 \setminus \mathscr{T}_2 = \begin{cases} n_1 \,?\, (C_1 \vee U_1) \smallsetminus (C_2 \vee U_2) : 0 : (D_1 \vee U_1) \smallsetminus (D_2 \vee U_2) & \text{if } n_1 = n_2, \\ n_1 \,?\, (C_1 \vee U_1) \smallsetminus \mathscr{T}_2 : 0 : (D_1 \cup U_1) \smallsetminus \mathscr{T}_2 & \text{if } n_1 < n_2, \\ n_2 \,?\, \mathscr{T}_1 \smallsetminus (C_2 \vee U_2) : 0 : \mathscr{T}_1 \smallsetminus (D_2 \vee U_2) & \text{if } n_1 > n_2; \end{cases}$$

We derive more precise ways to perform these operations, as well as some additional special cases that reduce the size of the resulting TDD. The union remains the same, it is for intersection and difference that we do these computations.

More precise intersection We suppose given two TDDs $\mathscr{T}_1 = n_1 \,?\, C_1 : U_1 : D_1$ and $\mathscr{T}_2 = n_2 \,?\, C_2 : U_2 : D_2$. Consider the semantic equality on TDDs: $\mathscr{T}_1 \simeq \mathscr{T}_2$ if and only if $[\![\mathscr{T}_1]\!] = [\![\mathscr{T}_2]\!]$.

Theorem 11.2.2. *If $n_1 = n_2$, then writing $n = n_1 = n_2$, we have*

$$\mathscr{T}_1 \wedge \mathscr{T}_2 \simeq n \,?\, C_1 \wedge (C_2 \vee U_2) \vee (U_1 \wedge C_2) : (U_1 \wedge U_2) : D_1 \wedge (U_2 \vee D_2) \vee (U_1 \wedge D_2)$$

Proof. We compute the semantics of $\mathscr{T}_1 \wedge \mathscr{T}_2$. To help following the computation, we denote set union as $+$, set intersection as $\cdot$. We also denote $c_1 = [\![C_1]\!]$, $u_1 = [\![U_1]\!]$, $d_1 = [\![D_1]\!]$, $c_2 = [\![C_2]\!]$, $u_2 = [\![U_2]\!]$, $d_2 = [\![D_2]\!]$.
Notice that $[\![n]\!] \cdot [\![\neg n]\!] = [\![n \wedge \neg n]\!] = \varnothing$. Also $[\![n]\!] \cdot [\![n]\!] = [\![n]\!]$ and $[\![\neg n]\!] \cdot [\![\neg n]\!] = [\![\neg n]\!]$.

$$\begin{aligned}
[\![\mathscr{T}_1 \wedge \mathscr{T}_2]\!] &= [\![n_1 \mathbin{?} C_1 : U_1 : D_1]\!] \wedge [\![n_2 \mathbin{?} C_2 : U_2 : D_2]\!] \\
&= ([\![n]\!] \cdot c_1 + u_1 + [\![\neg n]\!] \cdot d_1) \cdot ([\![n]\!] \cdot c_2 + u_2 + [\![\neg n]\!] \cdot d_2) \\
&= [\![n]\!] \cdot c_1 c_2 + [\![n]\!] \cdot c_1 u_2 + [\![n]\!] \cdot u_1 c_2 + u_1 u_2 + [\![\neg n]\!] \cdot u_1 d_2 + [\![\neg n]\!] \cdot d_1 u_2 + [\![\neg n]\!] \cdot d_1 d_2 \\
&= [\![n]\!] \left(c_1 (c_2 + u_2) + u_1 c_2\right) + u_1 u_2 + [\![\neg n]\!] \left(d_1 (u_2 + d_2) + u_1 d_2\right) \\
&= [\![n \mathbin{?} C_1 \wedge (C_2 \vee U_2) \vee (U_1 \wedge C_2) : U_1 \wedge U_2 : D_1 \wedge (U_2 \vee D_2) \vee (U_1 \wedge D_2)]\!] \quad \square
\end{aligned}$$

More precise difference

Theorem 11.2.3. *If $U_2 = 0$ and $n_1 = n_2$, then writing $n = n_1$ we have*

$$\mathscr{T}_1 \setminus \mathscr{T}_2 \simeq n \mathbin{?} (C_1 \vee U_1) \wedge \neg C_2 : (U_1 \wedge \neg D_2 \wedge \neg C_2) : (D_1 \vee U_1) \wedge \neg D_2$$

Proof. We compute the semantics of $\mathscr{T}_1 \setminus \mathscr{T}_2$ when $U_2 = 0$. To help following the computation, we denote set union as $+$, set intersection as $\cdot$. We also denote $c_1 = [\![C_1]\!]$, $u_1 = [\![U_1]\!]$, $d_1 = [\![D_1]\!]$, $c_2 = [\![C_2]\!]$, $u_2 = [\![U_2]\!] = \varnothing$, $d_2 = [\![D_2]\!]$, $\overline{c_2} = \overline{[\![C_2]\!]}$, $\overline{d_2} = \overline{[\![D_2]\!]}$.
Notice that $[\![n]\!] \cdot [\![\neg n]\!] = [\![n \wedge \neg n]\!] = \varnothing$. Also $[\![n]\!] \cdot [\![n]\!] = [\![n]\!]$ and $[\![\neg n]\!] \cdot [\![\neg n]\!] = [\![\neg n]\!]$.

$$\begin{aligned}
[\![\mathscr{T}_1 \setminus \mathscr{T}_2]\!] &= [\![n \mathbin{?} C_1 : U_1 : D_1]\!] \setminus [\![n \mathbin{?} C_2 : 0 : D_2]\!] \\
&= [\![n \mathbin{?} C_1 : U_1 : D_1]\!] \cap \overline{([\![n]\!] \cap [\![C_2]\!]) \cup ([\![\neg n]\!] \cap [\![D_2]\!])} \\
&= [\![n \mathbin{?} C_1 : U_1 : D_1]\!] \cap (\overline{[\![n]\!]} \cup \overline{[\![C_2]\!]}) \cap (\overline{[\![\neg n]\!]} \cup \overline{[\![D_2]\!]}) \\
&= ([\![n]\!] \cdot c_1 + u_1 + [\![\neg n]\!] \cdot d_1) \cdot ([\![\neg n]\!] + \overline{[\![C_2]\!]}) \cdot ([\![n]\!] + \overline{[\![D_2]\!]}) \\
&= ([\![n]\!] \cdot c_1 + u_1 + [\![\neg n]\!] \cdot d_1) \cdot ([\![\neg n]\!] \cdot \overline{d_2} + \overline{c_2} \cdot \overline{d_2} + [\![n]\!] \cdot \overline{c_2}) \\
&= [\![n]\!] \left(c_1 (\overline{c_2} \cdot \overline{d_2} + \overline{c_2}) + \overline{c_2} u_1\right) \\
&\quad + u_1 \cdot \overline{c_2} \cdot \overline{d_2} \\
&\quad + [\![\neg n]\!] \left(\overline{d_2} \cdot (u_1 + d_1) + d_1 \overline{c_2} \cdot \overline{d_2}\right) \\
&= [\![n]\!] (c_1 \overline{c_2} + \overline{c_2} u_1) + u_1 \cdot \overline{c_2} \cdot \overline{d_2} + [\![\neg n]\!] \left(\overline{d_2} \cdot (u_1 + d_1)\right)
\end{aligned}$$

where we conclude by factorizing on $\overline{c_2}$ and recognizing the three parts of the TDD. □

Summary of improved TDD operations The union operation maintains laziness by attaching $\mathscr{B}_2$ to the middle branch when $n_1 < n_2$, avoiding expansion of C_1 or D_1. Intersection and difference operations may expand unions by forming $(C_i \cup U_i)$ combinations when type nodes coincide.

Figure 11.5: Summary of TDD operations. Leaves 0 and 1 behave as in ordinary BDDs. $\mathcal{T}_1 = n_1 \,?\, C_1 : U_1 : D_1$ and $\mathcal{T}_2 = n_2 \,?\, C_2 : U_2 : D_2$.

Op.	**Definition**	**Condition**
	$n_1 \,?\, (C_1 \vee C_2) : (U_1 \vee U_2) : (D_1 \vee D_2)$	$n_1 = n_2$
$\mathcal{T}_1 \vee \mathcal{T}_2$	$n_1 \,?\, C_1 : (U_1 \vee \mathcal{T}_2) : D_1$	$n_1 < n_2$
	$n_2 \,?\, C_2 : (U_2 \vee \mathcal{T}_1) : D_2$	$n_1 > n_2$
	$n_1 \,?\, (C_1 \wedge (C_2 \vee U_2)) \vee (U_1 \wedge C_2) : (U_1 \wedge U_2) : (D_1 \wedge (U_2 \vee D_2)) \vee (U_1 \wedge D_2)$	$n_1 = n_2$
$\mathcal{T}_1 \wedge \mathcal{T}_2$	$n_1 \,?\, (C_1 \wedge \mathcal{T}_2) : (U_1 \wedge \mathcal{T}_2) : (D_1 \wedge \mathcal{T}_2)$	$n_1 < n_2$
	$n_2 \,?\, (\mathcal{T}_1 \wedge C_2) : (\mathcal{T}_1 \wedge U_2) : (\mathcal{T}_1 \wedge D_2)$	$n_1 > n_2$
	$n_1 \,?\, (C_1 \vee U_1) \wedge \neg C_2 : (U_1 \wedge \neg D_2 \wedge \neg C_2) : (D_1 \vee U_1) \wedge \neg D_2$	$n_1 = n_2$ and $U_2 = 0$
$\mathcal{T}_1 \smallsetminus \mathcal{T}_2$	$n_1 \,?\, (C_1 \vee U_1) \smallsetminus (C_2 \vee U_2) : 0 : (D_1 \vee U_1) \smallsetminus (D_2 \vee U_2)$	$n_1 = n_2$
	$n_1 \,?\, (C_1 \vee U_1) \smallsetminus \mathcal{T}_2 : 0 : (D_1 \vee U_1) \smallsetminus \mathcal{T}_2$	$n_1 < n_2$
	$n_2 \,?\, (\mathcal{T}_1 \vee U_2) \smallsetminus C_2 : 0 : (\mathcal{T}_1 \vee U_2) \smallsetminus C_2$	$n_1 > n_2$

11.2.3 Extracting DNF from TDDs

The connection between TDDs and the Disjunctive Normal Form (DNF) established in Chapter 3 is fundamental to understanding why TDDs provide an efficient representation for structural types. While TDDs maintain a compact form optimized for set operations, there are situations where we need to recover the underlying DNF representation, that is, for:

- **Emptiness decision**: Determining whether a type represents the empty set. This also decides subtyping, which comes as deciding the emptiness of the differnce of two types.
- **Type operators**: Implementing specific type operations that require explicit intersection of literals

DNF extraction algorithm. Given a TDD $\mathcal{T} = n \,?\, C : U : D$, we can extract its DNF representation through a systematic path traversal. The algorithm collects all paths from the root to leaves that terminate at 1, where each path represents a conjunction of literals:

- When taking the *left branch* (constrained branch): add the type node n as a *positive literal* to the intersection
- When taking the *right branch* (dual branch): add the type node n as a *negative literal* $\neg n$ to the intersection]
- When taking the *middle branch* (uncertain branch): no literal is added (the union is included regardless of the type node)

```
defp bdd_to_dnf(bdd), do: bdd_to_dnf([], [], [], bdd)

defp bdd_to_dnf(acc, _pos, _neg, :bdd_bot), do: acc
defp bdd_to_dnf(acc, pos, neg, :bdd_top), do: [{pos, neg} | acc]

defp bdd_to_dnf(acc, pos, neg, {_, _} = lit) do
  [{[lit | pos], neg} | acc]
end

# Lazy node: {lit, C, U, D}  =  (lit /\ C) \/ U \/ (~lit /\ D)
defp bdd_to_dnf(acc, pos, neg, {lit, c, u, d}) do
  # U is a bdd in itself, we accumulate its lines first
  bdd_to_dnf(acc, pos, neg, u)
  # Constrained part
  |> bdd_to_dnf([lit | pos], neg, c)
  # Dual part
  |> bdd_to_dnf(pos, [lit | neg], d)
end
```

Listing 11.2: Elixir implementation of DNF extraction from TDD

Example (*DNF extraction.*)

Consider the TDD representing $(\mathtt{int} \to \mathtt{int}) \vee (\mathtt{bool} \to \mathtt{bool})$:

$$(\mathtt{int} \to \mathtt{int})\,?\,1 : ((\mathtt{bool} \to \mathtt{bool})\,?\,1 : 0 : 0) : 0$$

Following the DNF extraction algorithm in Listing 11.2:

- **Path 1**: Root → left → 1: yields $(\mathtt{int} \to \mathtt{int})$
- **Path 2**: Root → middle → left → 1: yields $(\mathtt{bool} \to \mathtt{bool})$

The extracted DNF is: $(\mathtt{int} \to \mathtt{int}) \vee (\mathtt{bool} \to \mathtt{bool})$

Complexity considerations. The DNF extraction process is linear in the number of nodes in the TDD.

Integration with emptiness decision. The emptiness decision algorithm for record types (Theorem 11.3.6) leverages this DNF extraction process. When determining if a structural type is empty, we extract the DNF form and check for contradictions between positive and negative literals, enabling precise emptiness detection without full normalization.

11.2.4 Computing Type Operators

The method to compute operators for **static** types follows a systematic three-step process, leveraging the DNF representation established in Section 11.2.3:

1. **Extract DNF from tree**: Convert the type representation (TDD) to its Disjunctive Normal Form using the algorithm in Listing 11.2.
2. **Remove empty clauses**: Apply emptiness decision algorithms (such as Theorem 11.3.6) on single clauses to eliminate empty ones.
3. **Compute relevant operation**: Apply the specific type operator (intersection, union, difference, domain, application result) on the cleaned DNF form.

For instance, to compute the domain of a function type, we extract the DNF and then apply the domain operator given in Definition 3.3.4.

To compute operators for **gradual** types, recall from §3.3.2 that we can lift operators from static types to gradual types. If F is a **monotonic** operator known to be increasing for its types (for instance, since tuple elements are covariant, then projection is increasing), then the gradual version of F is given by computing F on the extrema of type τ and composing them with:

$$\tilde{F}(\tau) = F(\tau^{\Downarrow}) \wedge (? \wedge F(\tau^{\Uparrow}))$$

and if F is decreasing for $\leq$ (this is the case for the domain operator):

$$\tilde{F}(\tau) = F(\tau^{\Uparrow}) \vee (? \wedge F(\tau^{\Downarrow}))$$

In case of doubt on the monotonicity of an operator, it is always possible to use formula:

$$\tilde{F}(\tau) = (F(\tau^{\Downarrow}) \wedge F(\tau^{\Uparrow})) \vee (? \wedge (F(\tau^{\Downarrow}) \vee F(\tau^{\Uparrow})))$$

11.3 Application to Elixir

These techniques have been successfully deployed in the Elixir compiler. The type-checker uses a hybrid approach: bit vectors for base types, co-finite sets for atoms, TDDs for function and (since v1.19) tuples, lists and maps, . The gradual pair representation seamlessly integrates with all of these via the `Descr` abstraction. In practice on large industrial codebases (e.g., multi-million-line projects at Remote), compilation stays within our budget: type checking typically adds under a couple of milliseconds per module, and decision-diagram operations rarely appear as a bottleneck.

11.3.1 Multi-arity function as finite/co-finite TDDs

The current implementation of function types in Elixir uses a ternary decision tree with atomic components $(t_1, \ldots, t_n) \to t$ where n ranges in $\mathbb{N}$[2]. While this approach is sufficient for our needs, it can be improved by noticing that function types of distinct arity are, in Elixir as well as in our interpretation (they are parts of $\mathscr{P}_f\left(\mathscr{D}_{\mho}{}^n \times \mathscr{D}_{\Omega}\right)$), completely disjoint. Thus, a tree which intersects nodes of distinct arities is automatically empty. In the code, this means checking, in several places, the number of arguments in atomic types within a TDD, to prune those useless types. An alternative, which was not merged into the compiler but has been tested and successfully implemented in a fork, is to represented each arity of functions by a distinct TDD. Set operations are then done component-by-component, in the same modular way that the `Descr` does for different component types. We could thus add new fields to the `Descr`, named`{:fun, arity}` where `arity` is some positive integer. However, we want to retain one feature: to be able to

[2]Actually, n ranges from 0 to 255, which is the maximum arity for an Elixir/Erlang function

represent the negation of "all functions" minus a union of functions of distinct arity. Therefore, we take the approach of putting, in the unified field `:fun` of the `Descr`, a co-finite type which consists of a pair`{:union, tdds}` or`{:negation, tdds}`. The idea is the following: a function type is either a union of functions of disjoint arities (including negation of a given arity, which stay contained in their own TDD), or it is the "type of all functions" `fun` minus a couple of function types.

Definition 11.3.1 (Fun field for multi-arity function)**.** *Let $n \in \mathbb{N}$ and let for all $k \in [0, n]$, Φ_k denote a TDD of atomic function types of arity k: $(t_1, \dots, t_k) \to t$ (note that for $k = 0$ this is $\to t$). The field dedicated to* `:fun` *is either*

$$\text{i) Finite} \quad \{\texttt{:union}, \bigvee_{k=0}^{n} \Phi_k\} \qquad \text{ii) Co-finite} \quad \{\texttt{:negation}, \bigvee_{k=0}^{n} \Phi_k\}$$

with interpretations:

$$\left[\!\!\left[\{\texttt{:union}, \bigvee_{k=0}^{n} \Phi_k\} \right]\!\!\right] = \bigcup_{k=0}^{n} [\![\Phi_k]\!] \qquad \left[\!\!\left[\{\texttt{:negation}, \bigvee_{k=0}^{n} \Phi_k\} \right]\!\!\right] = [\![\texttt{fun}]\!] \smallsetminus \left(\bigcup_{k=0}^{n} [\![\Phi_k]\!] \right)$$

Justification of the co-finite representation. This representation relies on the disjointness of function types of different arities. Let $T = \bigvee_{k=0}^{n} \Phi_k$ and $T' = \bigvee_{k=0}^{n} \Phi'_k$ where each Φ_k and Φ'_k represents function types of arity k (using $\mathbb{O}$ for absent arities).
The key operations are:

- Union: $T \vee T' = \bigvee_{k=0}^{n} (\Phi_k \vee \Phi'_k)$ by disjointness of arities
- Intersection: $T \wedge T' = \bigvee_{k=0}^{n} (\Phi_k \wedge \Phi'_k)$ since functions of different arities have empty intersection
- Difference: $T \setminus T' = \bigvee_{k=0}^{n} (\Phi_k \setminus \Phi'_k)$ similarly by disjointness

For co-finite representations ($S = (\texttt{fun} \smallsetminus \bigvee_{k=0}^{n} \Psi_k)$) , the operations reduce to the corresponding operations on the finite witness sets, as established in the co-finite types section (Section 10.2). □

11.3.2 Tuples as open/closed structures

Elixir tuples are represented using a TDD ($\mathscr{T}$) whose nodes are of the form`{:closed, elements}` and`{:open, elements}`. This specialized node structure distinguishes between *open* and *closed* tuples, which is crucial for handling tuple types that may have additional elements beyond those explicitly specified.

Definition 11.3.2 (Tuple Representation)**.** *A tuple type is represented as a TDD ($\mathscr{T}$) where each node is either:*

- `{:open, elements}` *where* `elements` *is a list of types* $[t_1, \dots, t_n]$ *representing the types of the first n elements, with the possibility of additional elements of any type beyond position n.*

- `{:closed, elements}` *where* `elements` *is a list of types* $[t_1, \ldots, t_n]$ *representing exactly the types of all n elements, with no additional elements allowed.*

For example, the type `{integer(), string()}` (a closed tuple of exactly two elements) is represented as `{:closed, [integer(), string()]}`, while `{integer(), string(), ..}` (an open tuple with at least two elements) is represented as `{:open, [integer(), string()]}`.

Set operations on tuple types are performed by usual operations on ternary decision trees.

The open/closed distinction enables precise typing of operations like `elem/2`, where knowing whether a tuple can have additional elements affects the soundness of type inference.

Note

Contrary to multi-arity functions, which have a distinct arity, because of open tuples we cannot consider that we have a disjoint union of tuples of a given arity. This is why the idea of the previous Section 11.3.1 is not used for tuples. In a language without open tuples, it would be possible to represent tuples of a given arity as a single TDD, which would simplify some operations (for instance, we need to check the arity of the closed tuples when intersecting them).

11.3.3 Lists as proper/improper structures

Elixir lists are represented using a TDD ($\mathcal{T}$) whose nodes are pairs of types `{list_type, last_type}` where `list_type` is the type of elements and `last_type` is the type of the last element. This specialized structure captures both the type of list elements and the type of the final tail, enabling precise typing of both proper lists (ending with `[]`) and improper lists (ending with a non-list value).

Definition 11.3.3 (List Representation)**.** *A list type is represented as a TDD ($\mathcal{T}$) where each node is a pair* `{list_type, last_type}` *where:*

- `list_type` *is the type of all elements in the list*
- `last_type` *is the type of the final element, which is* `[]` *for proper lists, or some other type for improper lists*

For example:

- `list(integer())` (proper list of integers) is represented as `{integer(), []}`
- `list(integer(), atom())` (improper list of integers ending with an atom) is represented as `{integer(), atom()}`
- `list(integer(), integer())` (improper list where the tail is also an integer) is represented as `{integer(), integer()}`

This representation enables precise typing of list operations:

- `hd/1` returns `elements_type or last_type` (since the list may be empty, we need to consider the last type), unless the `last_type` is `[]`, in which case it returns `elements_type` ;
- `tl/1` returns `list(elements_type, last_type)`

Set operations on list types are performed by usual operations on ternary decision trees.

In particular, $list(e,t)$ is the type defined by the equation $X = (e,X) \vee t$, and $list(e) = list(e,[])$.

Emptiness Decision Algorithm for List Types For list types represented as pairs $\{t_{\text{elem}}, t_{\text{last}}\}$, we can derive an emptiness decision that does not require deriving equations from sets. This bypasses the usual construction of types with a proper set intersection, but that is because we rely on a pair of types which are already set-theoretic. Instead, we define the necessary nodes, here pairs of types for list types, and reason directly on the DNF to find an emptiness condition for each clause.

The emptiness decision problem for list types is to determine whether:

$$\bigvee_{i\in I}\Big(\bigwedge_{\{t_{\text{elem}},t_{\text{last}}\}\in P_i}\{t_{\text{elem}}, t_{\text{last}}\} \wedge \bigwedge_{\{s_{\text{elem}},s_{\text{last}}\}\in N_i}\neg\{s_{\text{elem}}, s_{\text{last}}\}\Big) \simeq \mathbb{0}$$

The key insight is that list types can be intersected by simply intersecting their element and last types:

$$\{t_{\text{elem}}, t_{\text{last}}\} \wedge \{s_{\text{elem}}, s_{\text{last}}\} = \{t_{\text{elem}} \wedge s_{\text{elem}}, t_{\text{last}} \wedge s_{\text{last}}\}$$

By iterating this componentwise intersection inside each clause, the conjunction of all positive list literals collapses to a single list literal whose components are the meets of the corresponding parts. Concretely, for each clause index i:

$$\bigwedge_{\{t_{\text{elem}},t_{\text{last}}\}\in P_i}\{t_{\text{elem}}, t_{\text{last}}\} = \{L_{\text{elem},i},\ L_{\text{last},i}\}$$
$$\text{where}\quad L_{\text{elem},i} \triangleq \bigwedge_{\{t_{\text{elem}},t_{\text{last}}\}\in P_i} t_{\text{elem}},\quad L_{\text{last},i} \triangleq \bigwedge_{\{t_{\text{elem}},t_{\text{last}}\}\in P_i} t_{\text{last}}$$

Therefore, the problem becomes:

$$\bigvee_{i\in I}\Big(\{L_{\text{elem},i}, L_{\text{last},i}\} \wedge \bigwedge_{\{s_{\text{elem}},s_{\text{last}}\}\in N_i}\neg\{s_{\text{elem}}, s_{\text{last}}\}\Big) \simeq \mathbb{0}$$

For emptiness decision, we need to determine when a list type minus several negated list types becomes empty. Consider the following observations:

1. Negated list types whose element type is not a supertype of the positive element type can be ignored. For example, lists of numbers (integers and floats) minus lists of integers ([1,2,3]) still contain lists with integers interspersed with at least one float ([1,1.5,2,2.5]) (so they are definitely not lists of floats).
2. Only negated list types with element types that are supertypes of the positive element type can potentially make the type empty.

Theorem 11.3.4 (Emptiness Decision Algorithm for List Types)**.** *Let $\Phi(L_{elem}, L_{last}, N)$ be the emptiness decision function for list types, where L_{elem} and L_{last} are the intersection of positive* *list element and last types respectively, and N is a set of negative list types. The function Φ is defined as:*

$$\Phi(L_{elem}, L_{last}, N) = (L_{elem} \leq \mathbb{0}) \ \texttt{or} \ (L_{last} \leq \mathbb{0}) \ \texttt{or} \ \Phi'(L_{elem}, L_{last}, N)$$

where Φ' is defined recursively as:

$$\begin{aligned}\Phi'(L_{elem}, L_{last}, N \cup \{\{N_{elem}, N_{last}\}\}) = \texttt{if} \ & L_{elem} \leq N_{elem} \\ \texttt{then} \ & (L_{last} \setminus N_{last} \leq \mathbb{0}) \ \texttt{or} \ \Phi'(L_{elem}, L_{last} \setminus N_{last}, N) \\ \texttt{else} \ & \Phi'(L_{elem}, L_{last}, N)\end{aligned}$$

and $\Phi'(L_{elem}, L_{last}, \emptyset) = false$.

Proof. The algorithm works by systematically eliminating negative constraints. For each negative list type $\{N_{\text{elem}}, N_{\text{last}}\}$:

- If $L_{\text{elem}} \leq N_{\text{elem}}$, then the negative constraint applies. The remaining lists must have last elements not in N_{last}, so we compute $L_{\text{last}} \setminus N_{\text{last}}$.
- If this difference is empty, then all possible lists are eliminated by this negative constraint, making the overall type empty.
- If $L_{\text{elem}} \not\leq N_{\text{elem}}$, then the negative constraint doesn't apply and can be ignored.

The base cases empty(L_{elem}) and empty(L_{last}) handle the trivial cases where no lists can be formed. □

11.3.4 Maps as hybrid record-dictionary

Elixir maps are represented using a TDD ($\mathcal{T}$) whose nodes are pairs of types`{fields, domains}` where `fields` is the association of keys to types and `domains` is the association of domain types to value types (see Section 1.1.5 for more context). This specialized structure captures both the specific field mappings and the domain-based mappings, enabling precise typing of hybrid record-dictionary structures.

Definition 11.3.5 (Map Representation)**.** *A map type is represented as a TDD ($\mathcal{T}$) where each node is a pair* `{fields, domains}` *where:*

- `fields` *holds the association of specific keys to their value types*
- `domains` *holds the association of domain types (e.g.,* `atom()`*,* `integer()`*) to their corresponding value types*

This node is closed outside the declared domains*: any key that is neither listed in* `fields` *nor whose type belongs to one of the domain types in* `domains` *is absent.*

Castagna (2023b) treats with precision the problem of typing Elixir records, which are a hybrid of records and dictionaries (see Section 1.1.5 of our introduction for a motivation of this structure). It also provides a representation for maps, not as a disjunctive normal form, but as a union of triples of the form {`fields`, `domains`, `extra`} where `fields` holds the association of keys to types, `domains` holds the value of the domain types (for instance, integers mapping to atoms), and `extra` represents some extra key/type pairs which are, technically, used for completeness of the representation, and appear only when eliminating explicit negations of positive maps.

In our implementation, we use a TDD form (not a union, then) which keeps explicit negations. An example of such a node is the 2-tuple:

```
{%{a: integer(), b: string()}, %{atom: atom(), integer: float()}}
```

which represents the type

```
%{a: integer(), b: string(), atom() => atom(), integer() => float()}
```

that has two defined keys, `:a` and `:b`, and maps any present other atom key (resp. any integer key) to an atom (resp. a float). In particular, it is closed outside these domains: any key whose type is neither an atom nor an integer is undefined.

Emptiness Decision Algorithm for Record Types We then implement the emptiness decision algorithm presented in Castagna (2023b). We recall it here in the form of function Φ:

Quote (*Follow-up to Lemma 4.7 in Castagna (2023b)*)

Theorem 11.3.6 (Emptiness Decision Algorithm for Record Types). *Let* $\Phi(R_\circ, N)$ *be the emptiness decision function for record types, where* $R_\circ$ *is a (positive) record type and* N *is a set of (negative) record types. The function* Φ *is defined recursively as follows:*

$$\begin{aligned}\Phi(R_\circ, N \cup \{R\}) = {} & \mathit{if}\, \forall k \in K.(\mathit{def}(R_\circ))_k \leq (\mathit{def}(R))_k \\ & \mathit{then}\ \ \forall \ell \in L.(R_\circ(\ell) \leq R(\ell)\ \mathit{or}\, \Phi(R_\circ \wedge \%\{\ell : \mathit{not}(R(\ell))\}, N)) \\ & \mathit{else}\, \Phi(R^\circ, N)\end{aligned}$$

where

- If R = {`fields`, `domains`}, the operation $\mathrm{def}(R)_k$ fetches the type of field k in the domain record `domains`.
- Operation $R(\ell)$ for a given key ℓ fetches the type of the value associated with ℓ in the field record `fields` *if it exists*; *otherwise*, it fetches the type of the default value associated with this key in the domain record `domains` (for instance, if the atom key `:age` is not present in `fields`, then the default type for atom keys `atom` is fetched from `domains`).
- $\%\{\ldots, \ell : \mathrm{not}(R(\ell))\}$ is an open map with key ℓ set to type $\mathrm{not}(R(\ell))$.
- $\mathrm{not}(t)$ is the negation of a type in the context of a possibly absent type denoted by $\bot$. Formally, $\mathrm{not}(t) = (\neg t) \vee \bot$ and $\mathrm{not}(t \vee \bot) = (\neg t)$ (for details, see Castagna (2023b)).

Implementing the function above is simply a matter of extracting the DNF $\bigvee_{i \in I}(L_i, N_i)$ from the TDD of maps. Then, for each clause (L, N), compute the intersection $R_\circ$ of positive maps in L, then Φ will decide, for each clause, if it is empty or not.

Conclusion

The modular architecture established in Chapter 9 proved essential for integrating these structural type representations. By extending the `Descr` abstraction with specialized fields for each structural type, we maintained the component-wise set operations that make the implementation both efficient and extensible. This approach enables language designers to adopt structural typing incrementally, choosing the appropriate representation for each type domain based on their specific requirements and performance constraints.

However, BDDs were not our first choice. During the development of the Elixir type system, we experimented with several alternative representations–Disjunctive Normal Forms (DNFs), compressed DNFs, and union forms for specific type domains. Each approach presented different trade-offs in terms of computational complexity and practical performance. Chapter 12 provides a comprehensive analysis of these alternatives, explaining why TDDs ultimately emerged as the optimal choice for our implementation.

CHAPTER 12

STRUCTURAL TYPE ALTERNATIVES: A COMPARATIVE ANALYSIS

The implementation of structural types in the Elixir type system hinges on the choice of data structure. While Chapter 11 details our final implementation using Binary Decision Diagrams (BDDs) and their lazy variants (TDDs), this chapter reviews the alternative representations explored during development.

Chapter Roadmap

- **Section 12.1 (DNF)** Discusses representing types directly as lists of pairs of positive and negative atomic types, which is the DNF representation.
- **Section 12.2 (Simplified DNF)** Presents a compressed DNF representation, which is applicable only to types where intersection can be computed directly (e.g., maps and tuples, but not functions).
- **Section 12.3 (Union Form)** Examines union forms (lists of type tuples) suitable for representing types on which negations can be fully eliminated (e.g., tuples).

Chronological context. The transition from DNFs (v1.18) to BDDs/TDDs (v1.19) followed a concrete performance arc observed during compiler development. Early v1.19 releases adopted lazy BDDs for function types to rein in union-heavy blowups but still regressed in a subset of projects from users with large codebases until we revised the equal-node formulas for intersection and difference to preserve laziness in the middle branch. The ablation in Tables 12.3 and 12.2 summarizes these milestones quantitatively and is further discussed in Chapter 14. These data underpin the design guidance given here (e.g., why difference-heavy workloads force BDDs) and make explicit where lazy representations help or hinder.

Each representation—DNF, compressed DNF, and union forms—strikes a different balance between computational cost and memory use. In practice, this choice governs the cost of the core operations $\vee$, $\wedge$, $\setminus$, and $\neg$. DNFs perform well for $\vee$ and $\wedge$, but they grow quickly under $\setminus$, which makes them ill-suited when pattern matching triggers frequent differences.

12.1 Disjunctive Normal Forms (DNFs)

BDDs provide an efficient representation when all operations (including difference and negation) are required. However, for certain type domains—particularly maps and tuples—alternative representations may be advantageous when difference operations are infrequent. This section describes Disjunctive Normal Forms (DNFs), which are well-suited for union and intersection but inefficient for difference.

A fundamental result is that every type can be expressed as a DNF: a union of conjunctions of positive and negative atomic types. This yields a natural representation for each map type as a list of pairs, each consisting of lists of positive and negative atomic types (that is, types with a top-level *constructor* such as $(t_1, \ldots, t_n) \to t$, as opposed to a set *connective* such as $t_1 \wedge t_2$).

```
$ type atomic_list() = list(atomic())
$ type dnf() = [{atomic_list(), atomic_list()}]
```

When encoding maps as a DNF, each map atomic type is a pair $\langle tag, fields \rangle$, where $tag \in$ `{:open, :closed}` indicates whether the map is open (allowing additional fields) or closed (containing exactly the specified fields), and *fields* is a mapping from keys to types. For example, the type `%{..., a: integer()} and not %{b: atom()}` is represented as a single-element list:

```
[{[{:open, %{:a => integer()}}], [{:closed, %{:b => atom()}}]}]
```

Note

Historical note. In version 1.18 of the Elixir compiler, map and tuple types were represented using DNFs. In version 1.19, we transitioned to (lazy) BDDs because function inference made extensive use of type differences. Although DNF is efficient for intersection and union, difference operations have exponential complexity, rendering DNF impractical for this use case, as analyzed below.

12.1.1 Redundancy in DNFs

Recall that $\mathscr{M}_1 = \bigvee_{i=1}^{n} (L_i, N_i)$ represents the type $\bigvee_{i=1}^{n} \bigwedge_{a \in L_i} a \wedge \bigwedge_{a \in N_i} \neg a$, which is interpreted as the domain subset $\bigcup_{i=1}^{n} \bigcap_{a \in L_i} [\![a]\!] \wedge \bigcap_{a \in N_i} \neg [\![a]\!]$.

Redundancy in DNFs arises primarily in two ways:

- If clauses (L_i, N_i) and (L'_i, N'_i) satisfy $L'_i \subseteq L_i$ and $N'_i \subseteq N_i$, then (L_i, N_i) is redundant and can be removed, keeping only (L'_i, N'_i) (the subset clause is more general, so in a union it subsumes the other).

- If a clause (L, N) satisfies $L \cap N \neq \varnothing$—that is, some atomic type appears both positively and negatively—the clause is empty and can be discarded.

Avoiding these two cases during union, intersection, and difference operations is crucial to prevent the generation of redundant clauses.

12.1.2 Union of DNFs

12.1.2 a) Naive union $\vee$ on DNFs

If a DNF is a finite disjunction of pairs (L, N) (with L, N finite sets of atomic types, respectively positive and negative), then

$$\mathscr{M}_1 = \bigvee_{i=1}^{n} (L_i, N_i) \qquad \text{and} \qquad \mathscr{M}_2 = \bigvee_{j=1}^{m} (L'_j, N'_j)$$

satisfy

$$\mathscr{M}_1 \vee \mathscr{M}_2 \simeq \bigvee_{i=1}^{n} (L_i, N_i) \vee \bigvee_{j=1}^{m} (L'_j, N'_j). \tag{$\vee_1$}$$

In other words, $\vee$ *on DNFs is list union.* An immediate problem is that repeatedly performing unions on the same DNF causes unbounded growth.

12.1.2 b) Semantics of $\vee$ on DNFs

Although representing DNFs as lists of pairs is straightforward, recalling their semantics helps reason about simplification. Recall that

$$\mathscr{M}_1 = \bigvee_{i=1}^{n} (L_i, N_i) \quad \text{represents} \quad \bigvee_{i=1}^{n} \Big(\bigwedge_{a \in L_i} a \wedge \bigwedge_{a \in N_i} \neg a \Big),$$

interpreted as the domain subset

$$\bigcup_{i=1}^{n} \Big(\bigcap_{a \in L_i} [\![a]\!] \cap \bigcap_{a \in N_i} \overline{[\![a]\!]} \Big).$$

Adding to this disjunction a pair (L', N') such that $L' \subseteq L_i$ and $N' \subseteq N_i$ does not change the represented set, while adding a pair with $L' \supseteq L_i$ and $N' \supseteq N_i$ renders the former redundant.

To define a more concise union operation on DNFs, we need to ensure that we do not add redundant clauses. To do so, we filter redundant pairs through the predicate filter, so that we consider only the clauses in $\text{filter}(\{(L_i, N_i) \mid i = 1, \ldots, n\} \cup \{(L_i, N_i) \mid i = 1, \ldots, k\})$ where, for any set of clauses S, we have

$$(L, N) \in \text{filter}(S) \iff \forall (L', N') \in S, (L' \subseteq L) \wedge (N' \subseteq N) \implies (L', N') = (L, N)$$

We then have

$$\mathscr{M}_1 \vee \mathscr{M}_2 \simeq \bigvee_{(L,N)\in\text{filter}(\mathscr{M}_1\cup\mathscr{M}_2)} (L,N) \tag{$\vee_2$}$$

This corresponds to the semantic union $[\![\mathscr{M}_1]\!] \cup [\![\mathscr{M}_2]\!]$. The trade-off is between removing no semantic duplicates (formula $\vee_1$), which may lead to redundancy, and paying a computational cost to simplify the DNF (formula $\vee_2$) in settings where set operations are applied repeatedly.

Size model. Let the *size* of a pair be $\|(L,N)\| \triangleq |L| + |N|$. For a DNF $\mathscr{M} = \bigvee_{k=1}^{r}(L_k, N_k)$, define the number of clauses and the total size of the DNF as:

$$\#\mathscr{M} \triangleq r \qquad \text{and} \qquad \|\mathscr{M}\| \triangleq \sum_{k=1}^{r} \|(L_k, N_k)\|.$$

Let $\ell_{\max} \triangleq \max\big(\max_i \|(L_i, N_i)\|,\ \max_j \|(L'_j, N'_j)\|\big)$ denote the maximum pair size across both inputs.

Proposition 12.1.1 (Complexity of DNF union without subsumption)**.** *Given $\mathscr{M}_1$ and $\mathscr{M}_2$ as above, computing $\mathscr{M}_1 \vee \mathscr{M}_2$ as set-union of pairs under structural equality has the following costs, depending on the deduplication strategy.*

(A) ***Hashed dedup.*** *Build a hash-set from the pairs of $\mathscr{M}_1$, then insert pairs of $\mathscr{M}_2$:*

$$T_{\text{HASH}}(n,m) \in \Theta\big((n+m)\cdot\ell_{\max}\big) \quad \textit{(expected time)}$$

Here the $\ell_{\max}$ factor accounts for hashing/equality on pairs.

(B) ***Naive list-difference, e.g.* `dnf1 ++ (dnf2 -- dnf1)`*.*** *BEAM-style structural equality on lists is linear in their length, thus*

$$T_{\text{NAIVE}}(n,m) \in \Theta\big(n\,m\cdot\ell_{\max}\big) \quad \textit{time}$$

In both cases the output size is at most $n+m$, so $\#(\mathscr{M}_1 \vee \mathscr{M}_2) \le n+m$.

Proof sketch. For (A), inserting n cubes to build the set and then m membership checks/insertions is linear in the number of cubes modulo the per-cube cost of hashing/equality, which is $O(\ell_{\max})$. For (B), list subtraction scans `dnf1` for each element of `dnf2`, yielding $n\cdot m$ equality comparisons, each costing $O(\ell_{\max})$. □

Corollary 12.1.2 (Idempotence cost)**.** *Computing $\mathscr{M} \vee \mathscr{M}$ with hashed deduplication runs in $\Theta(\#\mathscr{M}\cdot\ell_{\max})$ and returns $\mathscr{M}$, so repeated unions do not increase the size. With naive list-difference, the cost is $\Theta\big((\#\mathscr{M})^2\cdot\ell_{\max}\big)$, which suggests that making a special case for direct structural equality (with $\#\mathscr{M}\cdot\ell_{\max}$ cost) between the two DNFs is worthwhile.*

Remark 12.1.3 (If one also performs subsumption/canonicalization). *If the implementation additionally removes subsumed pairs (i.e., deletes (L,N) when there exists (L',N') with $L' \subseteq L$ and $N' \subseteq N$), a straightforward pairwise check leads to*

$$T_{\text{SUBS}}(n,m) \in \Theta\big((n+m)^2 \cdot \ell_{\max}\big),$$

which dominates the costs above.

12.1.3 Intersection of DNFs

If a DNF is a finite disjunction of pairs (L,N) (with L,N finite sets of atomic types, respectively positive and negative), then

$$\mathcal{M}_1 = \bigvee_{i=1}^{n} (L_i, N_i) \qquad \text{and} \qquad \mathcal{M}_2 = \bigvee_{j=1}^{m} (L'_j, N'_j)$$

satisfy

$$\mathcal{M}_1 \wedge \mathcal{M}_2 \;=\; \bigvee_{i=1}^{n} \bigvee_{j=1}^{m} (L_i \cup L'_j, N_i \cup N'_j) \tag{$\wedge_1$}$$

In other words, $\wedge$ *on DNFs is cartesian product.* However, this cartesian product creates many semantically redundant clauses.

12.1.3 a) Semantics of $\wedge$ on DNFs

Assume each clause (L,N) is *normalized* (finite sets, $L \cap N = \varnothing$). For

$$\mathcal{M}_1 = \bigvee_{i=1}^{n} (L_i, N_i) \qquad \text{and} \qquad \mathcal{M}_2 = \bigvee_{j=1}^{m} (L'_j, N'_j),$$

write the *consistency predicate*

$$\text{cons}\big((L,N),(L',N')\big) \;:\Longleftrightarrow\; (L \cap N' = \varnothing) \;\cap\; (L' \cap N = \varnothing).$$

Then

$$\mathcal{M}_1 \wedge \mathcal{M}_2 \;=\; \bigvee_{\substack{1 \le i \le n \\ 1 \le j \le m \\ \text{cons}((L_i,N_i),(L'_j,N'_j))}} \big(L_i \cup L'_j,\; N_i \cup N'_j\big). \tag{$\wedge_2$}$$

Intuitively, $\wedge$ on DNFs is a *conflict-filtered cartesian product*: inconsistent pairs vanish; consistent pairs merge by set union of their atomic types.

Size model. Reuse the size model from the union case: $\|(L,N)\| = |L| + |N|$, $\#\mathscr{M}$ is the number of clauses, $\|\mathscr{M}\| = \sum \|(L_k, N_k)\|$, and let $\ell_{\max}$ be the maximum clause size over both inputs. Let K_{cons} be the number of consistent pairs (i,j); clearly $K_{\text{cons}} \le nm$ and $\#(\mathscr{M}_1 \wedge \mathscr{M}_2) \le K_{\text{cons}}$ (before subsumption).

Proposition 12.1.4 (Complexity of DNF intersection without subsumption)**.** *Computing $\mathscr{M}_1 \wedge \mathscr{M}_2$ by enumerating consistent pairs and merging them admits the following bounds.*

(A) ***Cross-product with hashed dedup.*** *Check consistency for every (i,j) and insert each consistent merge into a hash-set:*

$$T^{\wedge}_{\text{HASH}}(n,m) \in \Theta\big(nm \cdot \ell_{\max}\big) \quad \textit{(expected time).}$$

Per-pair cost covers (i) conflict tests $(L_i \cap N'_j = \varnothing)$ and $(L'_j \cap N_i = \varnothing)$ and (ii) hashing/equality for the merged clause; both are $O(\ell_{\max})$ with standard set structures.

(B) ***Naive linear dedup of outputs.*** *If one appends each consistent merge to a list and tests membership by linear scan,*

$$T^{\wedge}_{\text{NAIVE}}(n,m) \in \Theta\big(nm \cdot \ell_{\max} + K^2_{\text{cons}} \cdot \ell_{\max}\big).$$

This is dominated by quadratic dedup when many distinct merges are produced.

In all cases the worst-case *output size is $\#(\mathscr{M}_1 \wedge \mathscr{M}_2) \le nm$ and this lower-bounds time by $\Omega(nm)$ since those clauses must be materialized.*

Proof sketch. For (A), we do nm consistency checks and at most K_{cons} insertions. With hashed sets, both checks and (amortized) inserts cost $O(\ell_{\max})$, yielding $\Theta(nm\ell_{\max})$. For (B), generating K_{cons} candidates and scanning an output that grows linearly causes $\sum_{t=1}^{K_{\text{cons}}} t = \Theta(K^2_{\text{cons}})$ equality checks, each $O(\ell_{\max})$. □

Corollary 12.1.5 (Idempotence vs. redundancy)**.** *Logically, $\mathscr{M} \wedge \mathscr{M} = \mathscr{M}$. Operationally, the cartesian-product construction emits*

$$\{(L_k, N_k) \mid 1 \le k \le \#\mathscr{M}\} \;\cup\; \{(L_i \cup L_j,\, N_i \cup N_j) \mid i \ne j,\ \text{cons}((L_i,N_i),(L_j,N_j))\}$$

where every cross-merge is more specific *and therefore subsumed by (L_i,N_i) and (L_j,N_j). Thus: with subsumption/canonicalization the result collapses back to $\mathscr{M}$; without it, the size can inflate up to n^2 distinct clauses (of which $n(n-1)$ are semantically redundant).*

Remark 12.1.6 (If one also performs subsumption/canonicalization)**.** *As for union, removing subsumed clauses naively entails $T_{\text{SUBS}}(K_{\text{cons}}) \in \Theta\big(K^2_{\text{cons}} \cdot \ell_{\max}\big)$.*

Remark 12.1.7 (Structural subsumption vs. semantic subsumption)**.** *In an implementation, it is also possible to interpret the intersection of atomic types $p_1 \overset{def}{=} \bigwedge_{a\in L} a, p_2 \overset{def}{=} \bigwedge_{a\in L'}$ and the union of negations $n_1 \overset{def}{=} \bigvee_{a\in N} a, n_2 \overset{def}{=} \bigvee_{a\in N'}$ in order to:*

- *check if each $p_1 \setminus n_1$ is empty (if yes, remove it)*
- *check if $(p_1 \setminus n_1) \leq (p_2 \setminus n_2)$ (in which case we only keep the clause (L', N')) for all $(L, N), (L', N')$*

They both require computing many additional intersection and unions. The first is a heavy process (n use of emptiness decision). The second is more heavy (n^2 subtyping checks) and we never implemented it for these reasons (although a proper benchmark would be interesting, as clauses removal can greatly reduce the cost of subtyping decision).

12.1.3 b) Exponential growth in map intersection is inevitable

As shown above, for the DNF representation of maps, intersection is implemented as a filtered cartesian product (using the cons predicate), from which we can also eliminate structural duplicates (that is, (L, N) is subsumed by (L', N') iff $L \subseteq L'$ and $N \subseteq N'$) or semantically empty clauses. Unfortunately, it is straightforward to construct a map type whose size grows exponentially (as a power of two) by iterating intersections.

To illustrate this exponential growth, consider the following sequence of map intersections:

$$\begin{aligned}
M_0 &= \texttt{\%}\{a : \text{integer}()\} \\
M_1 &= \texttt{\%}\{b : \text{atom}()\}\,\texttt{or}\,\texttt{\%}\{c : \text{float}()\} \\
M_2 &= \texttt{\%}\{d : \text{boolean}()\}\,\texttt{or}\,\texttt{\%}\{e : \text{pid}()\} \\
&\vdots \\
M_n &= \texttt{\%}\{k_{2n} : t_{2n}\}\,\texttt{or}\,\texttt{\%}\{k_{2n+1} : t_{2n+1}\}
\end{aligned}$$

where the keys k_{2i} and k_{2i+1} are pairwise disjoint with all of the previous keys. When we compute $M_0 \wedge M_1 \wedge M_2 \wedge \cdots \wedge M_n$, each intersection with a union of two maps creates a cartesian product. If we use the filtered version of formula ($\vee_2$), the result after n iterations contains 2^n DNF clauses, as shown in the experimental results (computed with Elixir v1.18.4):

Iteration	1	2	3	4	5	6	7	8	9	10	11	12	13
DNF Clauses	2	4	8	16	32	64	128	256	512	1024	2048	4096	8192

This exponential growth is not a flaw in the implementation, but an inevitable consequence of the semantic completeness of set-theoretic types. When we express complex disjunctive constraints such as "a map that has either field A with type X or field B with type Y," and then intersect this with another similar constraint, we must account for all combinations that satisfy both constraints.

The mathematical explanation for this growth is that each union in the DNF represents a disjunctive choice, and when we intersect two disjunctions, we must consider all pairwise combinations:

$$\left(\bigvee_{i=1}^{m} A_i\right) \wedge \left(\bigvee_{j=1}^{n} B_j\right) = \bigvee_{i=1}^{m}\bigvee_{j=1}^{n}(A_i \wedge B_j)$$

This creates $m \times n$ terms in the result. When each operand is itself a union of two terms, the result doubles in size with each intersection.

This exponential growth reflects the inherent complexity of certain type relationships in a set-theoretic type system. Some disjunctions intrinsically require an exponential number of terms to represent all valid combinations. This is not a limitation of our implementation, but a property of the type system's expressiveness.

In practice, several factors mitigate this exponential growth:

- Most real-world programs do not involve such systematic intersections
- The symbolic representation of negations avoids even worse exponential growth from difference operations
- Type inference algorithms can often simplify complex types before they reach this pathological case

This analysis clarifies why the implementation warns that "intersection is a cartesian product": it is a fundamental algorithmic property reflecting the semantic complexity of set-theoretic type operations.

12.1.4 DNF difference

The difference operation on DNFs is significantly more complex than union and intersection. Given two DNFs $\mathcal{M}_1 = \bigvee_{i=1}^{n}(L_i, N_i)$ and $\mathcal{M}_2 = \bigvee_{j=1}^{m}(L'_j, N'_j)$, computing $\mathcal{M}_1 \setminus \mathcal{M}_2$ requires applying De Morgan's law and distributing negation:

$$\mathcal{M}_1 \setminus \mathcal{M}_2 = \mathcal{M}_1 \wedge \neg\mathcal{M}_2 = \mathcal{M}_1 \wedge \neg\left(\bigvee_{j=1}^{m}(L'_j, N'_j)\right) = \mathcal{M}_1 \wedge \bigwedge_{j=1}^{m}\neg(L'_j, N'_j)$$

Each $\neg(L'_j, N'_j)$ expands to $\bigvee_{a\in L'_j}\neg a \vee \bigvee_{a\in N'_j} a$, generating a very large number of terms. The final result intersects $\mathcal{M}_1$ with this expanded form, causing a large blow-up in time and space.

This growth renders DNFs impractical for type inference scenarios that rely heavily on difference operations, such as pattern matching analysis where it is necessary to compute "remaining types" after matching specific patterns.

12.2 Compressed DNFs

For certain type domains where intersection can be computed efficiently, a *compressed DNF* representation can be used to maintain the DNF structure while computing intersections on the positive atomic types.

In a compressed DNF, rather than maintaining separate positive atomic types L_i in each clause (L_i, N_i), we precompute their intersection and store the result as a single compressed atomic type. Formally, for a DNF $\mathscr{M} = \bigvee_{i=1}^{n}(L_i, N_i)$ where $L_i = \{a_{i,1}, a_{i,2}, \dots, a_{i,k_i}\}$, the compressed form becomes:

$$\mathscr{M}_{\text{compressed}} = \bigvee_{i=1}^{n} (a_i, N_i) \qquad \text{where} \qquad a_i = \bigwedge_{j=1}^{k_i} a_{i,j}$$

This transformation is effective only when the intersection operation $\bigwedge_{j=1}^{k_i} a_{i,j}$ can be computed efficiently and yields a well-defined result.

Base types have straightforward intersections; map types follow rules for open/closed maps (Section 12.1.3); tuple intersection is component-wise with arity constraints; list intersection preserves structural properties. A notable exception is function types, where intersection does not simplify to a single arrow form. The key advantage of is that pre-computing the positive intersection can eliminate terms, if we detect those as being empty.

12.2.1 The case of maps: compressed DNFs

If a map type is represented as a DNF, it is a disjunction of pairs (L, N) where L is a set of map atomic types and N is a set of negated map atomic types. Concretely, for map types, the atomic types are as follows:

For all $a \in L$, a is either $\langle$`:open`,$\textit{fields}_1\rangle$ or $\langle$`:closed`,$\textit{fields}_2\rangle$; here, `:open` denotes an open record type (e.g., `%{..., a: integer()}` denotes maps that bind atom `a` to an integer type and may have any other field), while `:closed` denotes a closed record type (e.g., `%{a: integer()}` denotes maps that bind atom `a` to an integer type and no other field).

12.2.1 a) Intersection of map atomic types

Intersected atomic maps always yield another atomic map:

Open-Open Intersection. When both maps are open, the result is open and contains the union of all fields with intersected types:

$$\langle \texttt{:open}, \textit{fields}_1\rangle \wedge \langle \texttt{:open}, \textit{fields}_2\rangle = \langle \texttt{:open}, \textit{fields}_1 \sqcup \textit{fields}_2\rangle$$

where $\sqcup$ denotes the symmetric merge operation that intersects types for common keys.

Closed-Closed Intersection. When both maps are closed, the result is closed and must contain exactly the intersection of fields:

$$\langle \texttt{:closed}, \textit{fields}_1\rangle \wedge \langle \texttt{:closed}, \textit{fields}_2\rangle = \langle \texttt{:closed}, \textit{fields}_1 \sqcap \textit{fields}_2\rangle$$

where $\sqcap$ denotes the symmetric intersection that requires all keys from both maps to be present with intersecting types. If any key is missing from either map, the intersection is empty.

Open-Closed Intersection. When one map is open and the other is closed, the result is closed, and all fields from the open map must be present in the closed map (except for optional fields):

$$\langle\texttt{:open}, \textit{fields}_o\rangle \wedge \langle\texttt{:closed}, \textit{fields}_c\rangle = \langle\texttt{:closed}, \textit{fields}_c \sqcap \textit{fields}_o\rangle$$

Thus, the form $\mathscr{M} = \bigvee_{i=1}^{n}(L_i, N_i)$ can be compressed to $\bigvee_{i=1}^{n}(a_i, N_i)$, where a_i is the (non-empty) computed intersection of the atomic types in L_i. Adopting this compressed form for the DNF of maps allows for more efficient intersection. The intersection algorithm handles these cases by:

1. Computing the cartesian product of all map atomic type pairs from both DNFs
2. For each pair, attempting to intersect the atomic types using the rules above
3. If the intersection is empty (yields $\mathbb{0}$), discard the result
4. Merging the negated atomic types from both operands
5. Eliminating duplicates and ensuring that the result is not present in the negation list

This approach maintains symbolic negations, avoiding the expensive distribution of difference operations that would otherwise create an exponential number of map atomic types. Keeping negations symbolic enables efficient intersection while preserving semantic integrity.

12.3 Union Forms for Tuples

As shown in Theorem 4.5.3, every tuple type t can be put into a union form such that, given two finite families of atomic tuples $\mathscr{C}$ and $\mathscr{O}$, we have:

$$t \simeq \bigvee_{\{\overline{s}\}\in\mathscr{C}} \{\overline{s}\} \vee \bigvee_{\{\overline{r}\texttt{,}\,..\}\in\mathscr{O}} \{\overline{r}\texttt{,}\,..\}$$

We describe succintly the set-theoretic operations on union forms for tuples.

Normalization. Normalize $(\mathscr{C},\mathscr{O})$ after every operation:

- Drop empty atomic types (any empty component makes the whole atomic type empty).
- Remove subsumed atomic types using the tuple order:
  - Closed vs closed: drop $\{t_1,\dots,t_n\}$ if for some $\{u_1,\dots,u_n\}$ we have $\forall i.\ t_i \le u_i$.
  - Closed vs open: drop $\{t_1,\dots,t_n\}$ if there exists $\{u_1,\dots,u_\ell\texttt{,}\,..\}$ with $\ell \le n$ and $\forall i \le \ell.\ t_i \le u_i$.
  - Open vs open: drop $\{s_1,\dots,s_k\texttt{,}\,..\}$ if there exists $\{u_1,\dots,u_\ell\texttt{,}\,..\}$ with $\ell \le k$ and $\forall i \le \ell.\ s_i \le u_i$.

Union.

- Concatenate families: $(\mathscr{C}_1 \cup \mathscr{C}_2,\ \mathscr{O}_1 \cup \mathscr{O}_2)$, then *Normalize*.

Intersection. Compute pairwise on atomic types; collect results; then *Normalize.*

- Closed ∧ Closed (same arity n):

$$\{\, t_1,\dots,t_n \,\} \;\wedge\; \{\, u_1,\dots,u_n \,\} \;=\; \{\, t_1 \wedge u_1,\; \dots,\; t_n \wedge u_n \,\}.$$

  If arities differ, the result is $\mathbb{0}$.

- Closed ∧ Open:

$$\{\, t_1,\dots,t_n \,\} \;\wedge\; \{\, u_1,\dots,u_\ell \,\mathbf{,}\, .. \,\} \;=\; \begin{cases} \mathbb{0} & \text{if } n < \ell, \\ \{\, t_1 \wedge u_1,\; \dots,\; t_\ell \wedge u_\ell,\; t_{\ell+1},\dots,t_n \,\} & \text{if } n \geq \ell. \end{cases}$$

- Open ∧ Open: let $L \overset{\text{def}}{=} \max(k,\ell)$, and define $s'_i \overset{\text{def}}{=} (i \leq k ?\, s_i : \mathbb{1})$, $u'_i \overset{\text{def}}{=} (i \leq \ell ?\, u_i : \mathbb{1})$.

$$\{\, s_1,\dots,s_k \,\mathbf{,}\, .. \,\} \;\wedge\; \{\, u_1,\dots,u_\ell \,\mathbf{,}\, .. \,\} \;=\; \{\, s'_1 \wedge u'_1,\; \dots,\; s'_L \wedge u'_L \,\mathbf{,}\, .. \,\}.$$

Difference.

- Do not implement ad hoc rules here. Reduce differences by invoking the decomposition theorem (Theorem 4.5.3) to eliminate tuple negatives and obtain a finite union of closed/open tuple atomic types, then *Normalize.*

12.3.1 An invariant for the union form.

Note that difference elimination produces tuples that are pairwise disjoint; if we maintained the tuples in a state where they are pairwise disjoint, we would not need to check for duplicates, as the intersection formula, under this invariant, creates disjoint tuples. We would only need to prune the empty ones.

The remaining question is how to maintain disjointness under union. This can be achieved by observing that any union of tuples can be decomposed as a union of three pairwise disjoint tuples:

$$\begin{aligned} \{t_1,\dots,t_n\} \vee \{s_1,\dots,s_n\} = {} & \{t_1 \wedge s_1, t_2 \vee s_2, \dots, t_n \vee s_n\} \\ & \vee \{s_1 \setminus t_1, s_2, \dots, s_n\} \\ & \vee \{t_1 \setminus s_1, t_2, \dots, t_n\} \end{aligned}$$

where we prune, among the three tuples, those that are empty because either $t_1 \wedge s_1$, $t_1 \setminus s_1$, or $s_1 \setminus t_1$ is empty. This provides a technique to insert a tuple $t = \{t_1,\dots,t_n\}$ into a set S of tuples that are pairwise disjoint on their first component: we search for tuples $\{s_1,\dots,s_n\} \in S$ for which $t_1 \wedge s_1$ is not empty. If there are none, we simply add t to S. If there is one, we generate the three tuples above, directly insert the first and the second (they are disjoint from the rest of S as their first component is a subtype of s_1), then recursively insert the third tuple into S. This process terminates as each recursive call uses a strictly smaller first component.

Maintaining this invariant appears costly; we include it only to illustrate that alternative approaches are possible. Nguyen (2008) (Section 4.2.3) studies different decompositions for unions of pairs, and the existence of a unique maximal decomposition.

12.3.2 Arity Constraints

An important advantage of union forms for tuples is that they naturally handle arity constraints. Tuples of different arities are disjoint, so:

$$\{\, t_1, \ldots, t_n \,\} \land \{\, u_1, \ldots, u_m \,\} = \begin{cases} \{\, t_1 \land u_1, \ldots, t_n \land u_n \,\} & \text{if } n = m \\ \emptyset & \text{if } n \neq m \end{cases}$$

This property enables efficient emptiness checks and automatic pruning of impossible combinations during intersection.

12.3.3 Limitations and Trade-offs

While union forms are effective for tuples with small arities and limited unions, they suffer from exponential growth when dealing with complex disjunctions. The intersection operation can create $O(k \times m)$ tuple atomic types, which becomes problematic for large union types.

Additionally, difference operations on union forms exhibit the same exponential complexity as DNFs, requiring expansion of negations. This limitation ultimately led us to adopt BDDs for tuple types as well, particularly when pattern matching analysis required frequent difference operations.

12.4 Comparison

To complete the comparative analysis, we examine the complexity characteristics of BDDs and their lazy variants (TDDs), which ultimately became our chosen representations.

12.4.1 BDD Complexity

BDDs provide polynomial-time complexity for all Boolean operations. For BDDs $\mathscr{B}_1$ and $\mathscr{B}_2$ of sizes $|\mathscr{B}_1|$ and $|\mathscr{B}_2|$ respectively:

- **Union**: $O(|\mathscr{B}_1| \cdot |\mathscr{B}_2|)$ time and space
- **Intersection**: $O(|\mathscr{B}_1| \cdot |\mathscr{B}_2|)$ time and space
- **Difference**: $O(|\mathscr{B}_1| \cdot |\mathscr{B}_2|)$ time and space
- **Negation**: $O(|\mathscr{B}|)$ time and space

The key insight is that BDD operations use recursive decomposition with memoization and canonical ordering to try to mitigate exponential blow-up (which can still happen in the worst case).

12.4.2 Lazy BDD (TDD) Complexity

Lazy BDDs introduce a middle branch to defer union expansion, resulting in different complexity characteristics:

- **Union**: $O(|\mathscr{B}_1|)$ time when $\mathscr{B}_2$ is attached to the middle branch (lazy case)
- **Intersection**: $O(|\mathscr{B}_1| \cdot |\mathscr{B}_2|)$ time, may require union expansion
- **Difference**: $O(|\mathscr{B}_1| \cdot |\mathscr{B}_2|)$ time, may require union expansion
- **Negation**: $O(|\mathscr{B}|)$ time and space

The lazy approach prevents exponential growth during consecutive unions, but intersection and difference may still require expanding deferred unions, which can diminish the benefit.

12.4.3 Memory Usage Patterns

The four representations exhibit distinct memory usage patterns:

Representation	**Growth**	**Notes**
BDDs	Polynomial	Grows with Boolean operation complexity
TDDs	Polynomial	Lower for unions; spikes during $\wedge$/$\setminus$
DNFs	Exponential	Difference operations cause blow-up
Union forms	Quadratic	Intersection operations cause blow-up

Table 12.1: Memory usage patterns by representation.

12.4.4 Practical Performance Considerations

In practice, the choice between BDDs and TDDs depends on the operation mix:

- **Union-heavy workloads**: TDDs provide better performance due to lazy evaluation
- **Intersection-heavy workloads**: DNFs may be more efficient due to immediate evaluation
- **Mixed workloads**: TDDs provide more predictable performance characteristics
- **Pattern matching analysis**: TDDs are essential due to frequent difference operations

We have discussed the inherent trade-offs among these representations. Some constructions, notably iterated map intersections in DNF, can grow exponentially in the worst case (§12.1.3). This is not a flaw of the data structures but a semantic reality of expressive set-theoretic types. Nevertheless, it motivates practical mitigations (such as memoization, early subsumption checks, and shape-preserving simplifications) that we employ in the implementation. Our general conclusion is that, for most composite types, a BDD-based representation is the simplest and most efficient choice unless difference operations are rarely used in the system (which is not the case if, for example, differences are used to type pattern matching).

To support the choice of representation, we provide a table of trade-offs in Table 12.4.

Empirical summary tables. For completeness, we include two compact tables that measure compilation times for some important Elixir libraries (see Appendix A.5 for a description of the libraries; more details on the benchmarks will be given in Chapter 14).

- Table 12.2 contrasts v1.18 and v1.19 end-to-end times on representative projects.
- Table 12.3 reports type-checking time across implementation milestones (lazy BDD adoption and equal-node fixes).

Table 12.2: Type-checking performance: Elixir v1.18 (BDD) vs v1.19 (TDD) on selected projects. *Type-check* columns show type-checking time in seconds; *Total* columns show full compilation time; *%* columns show percentage of compilation time spent type-checking.

Project	Type-check (s)		Δ (s)	Total (s)		% type-check	
	v1.18	v1.19		v1.18	v1.19	v1.18	v1.19
Remote	11.116	19.476	+8.360	707.598	228.801	1.6	8.5
Livebook	0.177	0.102	−0.075	4.112	3.051	4.3	3.3
Credo	0.059	0.027	−0.032	1.305	1.095	4.5	2.4
Phoenix	0.049	0.029	−0.020	0.525	0.461	9.3	6.3
Hex	0.091	0.065	−0.026	1.339	1.232	6.8	5.3

Table 12.3: Ablation study: type-checking time (seconds) across BDD/TDD implementation milestones.

Implementation milestone	Livebook	Phoenix	Credo
v1.18 baseline (BDD), OTP 28.1	0.098	0.027	0.021
v1.19 with lazy BDDs (functions only)	0.452	0.024	0.031
TDD everywhere (fixed intersection)	0.220	0.023	0.029
Stabilized lazy BDD intersection	0.225	0.023	0.029
Latest: map-line emptiness short-circuit	0.154	0.022	0.033

Conclusion

Choosing a representation. Based on the analysis and practical experience with the Elixir type-checker, we offer the following guidance:

- **Use TDDs** when difference (\) and negation (¬) operations are frequent. TDDs handle all operations in polynomial time. This is the default choice for function types and, since Elixir 1.19, for map types. Using TDDs over BDDs is preferable as soon as there exist consecutive unions in the system. We have not found a clear argument for using BDDs over TDDs, other than simplicity of the implementation.
- **Use DNFs** only when difference and negation are *never* used (except to compute subtyping, which uses difference). DNFs are efficient for union and intersection but become exponentially large with difference operations. This representation was used for maps in Elixir 1.18 but was abandoned due to the need to infer precise intersection types for pattern matching, which made heavy use of difference operations.

Table 12.4: Complexity comparison of type representations. n_1, n_2 denote number of clauses (DNF) or size (BDD/TDD); ℓ denotes maximum clause size.

Operation	**Metric**	**DNF**	**BDD**	**TDD (lazy)**
$\wedge$	time	$O(n_1 n_2 \ell)$	$O(\lvert\mathscr{B}_1\rvert \cdot \lvert\mathscr{B}_2\rvert)$	$O(\lvert\mathscr{B}_1\rvert \cdot \lvert\mathscr{B}_2\rvert)$
	output size	$O(n_1 n_2)$	$O(\lvert\mathscr{B}_1\rvert \cdot \lvert\mathscr{B}_2\rvert)$	$O(\lvert\mathscr{B}_1\rvert \cdot \lvert\mathscr{B}_2\rvert)$
$\vee$	time	$O((n_1 + n_2)\ell)$	$O(\lvert\mathscr{B}_1\rvert \cdot \lvert\mathscr{B}_2\rvert)$	$O(\lvert\mathscr{B}_1\rvert)$ (lazy)
	output size	$O(n_1 + n_2)$	$O(\lvert\mathscr{B}_1\rvert \cdot \lvert\mathscr{B}_2\rvert)$	$O(\lvert\mathscr{B}_1\rvert + 1)$ (lazy)
$\setminus$	time	exponential	$O(\lvert\mathscr{B}_1\rvert \cdot \lvert\mathscr{B}_2\rvert)$	$O(\lvert\mathscr{B}_1\rvert \cdot \lvert\mathscr{B}_2\rvert)$
	output size	exponential	$O(\lvert\mathscr{B}_1\rvert \cdot \lvert\mathscr{B}_2\rvert)$	$O(\lvert\mathscr{B}_1\rvert \cdot \lvert\mathscr{B}_2\rvert)$
$\neg$	time	exponential	$O(\lvert\mathscr{B}\rvert)$	$O(\lvert\mathscr{B}\rvert)$
	output size	exponential	$O(\lvert\mathscr{B}\rvert)$	$O(\lvert\mathscr{B}\rvert)$

Key: DNF = Disjunctive Normal Form (list of clauses); BDD = Binary Decision Diagram; TDD = Ternary Decision Diagram (lazy union in middle branch).

- **Hybrid approach**: Different components in a `Descr` can use different representations. For example, functions may use TDDs, while tuples use union forms. An interesting feature would be to allow parameterization of this representation within a type-checker; that way, different codebases might take advantage of those different trade-offs.

CHAPTER 13

IMPLEMENTING THE TYPE SYSTEM

Throughout this thesis, we have shown how to type Elixir using an idealized version of the language called Core Elixir. We described the type system and its soundness, and detailed how to implement the type engine that powers the manipulation of set-theoretic types. In Chapter 9 we described the techniques required to implement the *type engine* for semantic subtyping: representation of set-theoretic types, set operations (intersection, union, difference), type relations (mainly subtyping—here we also use *consistent subtyping* and *compatibility*) and type operators. But we did not go through the specificities of how to use this engine to type-check Elixir programs using the typing rules of Chapters 4, 5 and 6. These typing rules are declarative; in the context of semantic subtyping, their algorithmic counterpart consists of the corresponding engine operator.

In this chapter, we fill some gaps between those formal type systems and their implementation in the Elixir compiler.

In §13.1 we relate Core Elixir to Featherweight Elixir (FW-Elixir), a strict subset of the full Elixir language with some non-trivial differences from Core Elixir, such as a richer guard syntax that includes negation. We then discuss the implementation of the typing rules of the declarative (although, quasi-algorithmic) type system that was outlined in Part I.

In §13.2 we discuss the implementation of the typing rules for function type application.

Chapter Roadmap

- **Section 13.1 (Bridging)**: Relates Core Elixir to FW-Elixir and compiles guards.
- **Section 13.2 (Application)**: Implements static and gradual function application.

$$\begin{array}{llcl}
\textbf{Expressions} & e & ::= & c \mid x \mid \lambda^{\mathbb{I}}(\overline{\overline{p}g \to e}) \mid e(\overline{e}) \mid \{\overline{e}\} \mid \pi_e\, e \mid \texttt{case}\ e\ \overline{\overline{p}g \to e} \mid e + e \\
\textbf{Patterns} & p & ::= & c \mid x \mid \{\overline{p}\} \\
\textbf{Guards} & g & ::= & a\,?\,\tau \mid a = a \mid a \mathrel{!=} a \mid a < a \mid g\ \texttt{and}\ g \mid g\ \texttt{or}\ g \\
\textbf{Guard atoms} & a & ::= & c \mid x \mid \pi_a\, a \mid \texttt{size}\, a \mid \{\overline{a}\} \\
\textbf{Test types} & \tau & ::= & b \mid c \mid \texttt{fun}_n \mid \{\overline{\tau}\} \mid \{\overline{\tau}\} \mid \tau \vee \tau \mid \neg\tau \\
\textbf{Base types} & b & ::= & \texttt{int} \mid \texttt{bool} \mid \texttt{fun} \mid \texttt{tuple} \\
\textbf{Types} & t & ::= & b \mid c \mid \overline{t} \to t \mid \{\overline{t}\} \mid \{\overline{t}, ..\} \mid t \vee t \mid \neg t \mid ? \\
\textbf{Interfaces} & \mathbb{I} & ::= & \varepsilon \mid \{t_i \to t'_i\}_{i=1..n}
\end{array}$$

Figure 13.1: Core Elixir

$$\begin{array}{llcl}
\textbf{Annotations} & \mathtt{A} & ::= & \texttt{\$T} \mid \varepsilon \\
\textbf{Expressions} & \mathtt{E} & ::= & \mathtt{L} \mid x \mid \mathtt{A}\ \texttt{fn}\ \overline{\overline{\mathtt{P}}\ \texttt{when G -> E}}\ \texttt{end} \mid \mathtt{E}(\mathtt{E}_1,..,\mathtt{E}_n) \mid \mathtt{E} + \mathtt{E} \\
 & & \mid & \texttt{case E}\ \overline{\texttt{P when G -> E}} \mid \{\mathtt{E}_1,..,\mathtt{E}_n\} \mid \texttt{elem(E,E)} \\
\textbf{Singletons} & \mathtt{L} & ::= & n \mid k \mid \{\overline{\mathtt{L}}\} \\
\textbf{Patterns} & \mathtt{P} & ::= & \mathtt{L} \mid x \mid \{\mathtt{P}_1,..,\mathtt{P}_n\} \\
\textbf{Guards} & \mathtt{G} & ::= & \mathtt{D} \mid \mathtt{C} \mid \texttt{not G} \mid \texttt{G and G} \mid \texttt{G or G} \mid \texttt{G == G} \mid \texttt{G != G} \\
 & & \mid & \texttt{G < G} \mid \texttt{G <= G} \mid \texttt{G > G} \mid \texttt{G >= G} \\
\textbf{Selectors} & \mathtt{D} & ::= & \mathtt{L} \mid x \mid \texttt{elem(D,D)} \mid \texttt{tuple\_size(D)} \mid \{\overline{\mathtt{D}}\} \\
\textbf{Checks} & \mathtt{C} & ::= & \texttt{is\_integer(D)} \mid \texttt{is\_atom(D)} \mid \texttt{is\_tuple(D)} \\
 & & \mid & \texttt{is\_function(D)} \mid \texttt{is\_function(D,}n\texttt{)} \\
\textbf{Base types} & \mathtt{B} & ::= & \texttt{integer()} \mid \texttt{atom()} \mid \texttt{function()} \mid \texttt{tuple()} \\
\textbf{Types} & \mathtt{T} & ::= & \mathtt{B} \mid \mathtt{L} \mid \overline{\mathtt{T}} \to \mathtt{T} \mid \{\overline{\mathtt{T}}\} \mid \{\overline{\mathtt{T}}, ..\} \mid \texttt{T or T} \mid \texttt{T and T} \mid \texttt{not T} \\
 & & \mid & \texttt{none()} \mid \texttt{term()} \mid \texttt{dynamic()}
\end{array}$$

(where x ranges over variables, n ranges over integers, and k ranges over atoms)

Figure 13.2: Featherweight Elixir

13.1 Bridging Core Elixir and Featherweight Elixir

Figure 13.1 presents the full syntax of Core Elixir, which encompasses all the language features discussed in the main body of the thesis. This includes expressions, patterns, guards, guard atoms, test types, base types, and types.

To bridge the gap between Core Elixir and a more concrete Elixir syntax, we introduce Featherweight Elixir (FW-Elixir). FW-Elixir is a strict subset of Elixir that covers all the language features discussed in this thesis, including tuples, anonymous and multi-arity functions, case-expressions with patterns and guards, and more. Figure 13.2 presents the formal syntax for FW-Elixir.

All the examples provided throughout the thesis are valid syntax for both Elixir and FW-Elixir: all FW-Elixir syntax is also valid syntax. Furthermore, FW-Elixir extends beyond the examples written in Core Elixir, as it allows for negated guards, which are absent from Core Elixir. For instance, consider the following alternative definition of the `negate` function:

```
def negate(x) when not(is_function(x) or is_tuple(x) or is_integer(x)) do
    not x
end
```

This definition is equivalent to that one:

```
def negate(x) when is_boolean(x), do: not x
```

assuming `integer()` and `boolean()` are the only basic types used in FW-Elixir. The type `boolean()` is the complement of `integer() or function() or tuple()` (where the type `function()` denotes the type of all functions) and all values of type `integer()` are captured by the first clause of `negate`. Both version are valid FW-Elixir syntax, but while the former can be rendered in Core Elixir, the latter cannot.

Guard Compilation While there is a clear correspondence between the expression syntax of FW-Elixir and Core Elixir, the relationship between their guard systems requires more careful treatment as we show next.

The definitions of the expressions in Figure 13.1 and 13.2, there is a clear 1-1 correspondence between them. For instance, the `second_strong` function we defined in the introduction (line 144) corresponds to the following Core Elixir expression:

$$\lambda^{\{\{\mathbb{1},\mathtt{int}\}\to\mathtt{int}\}}(x).\mathtt{case}\,x\,(\,\{\mathbb{1},\mathtt{int}\}\to\pi_1\,x\,)$$

It is therefore easy to transpose the typing and reduction rules defined for Core Elixir expressions to the expressions of FW-Elixir. The correspondence for guards and patterns of the two languages is instead more nuanced. In some cases this correspondence is clear: for example, it is clear that the second clause of the `test` function in the introduction (line 155) can be expressed in Core Elixir as a branch of a case-expression with pattern x and guard:

$$(\,x\,?\,\mathtt{bool}\,)\;\mathtt{or}\;(\,\pi_0\,x = \pi_1\,x\,)$$

However, the correspondence between guards in the two languages is more subtle. While Core Elixir type tests are more general than those that can be performed by FW-Elixir guards, the latter appear more expressive since they include negation, which is absent from Core Elixir. For instance, in FW-Elixir one can define:

```
def first(x) when not(tuple_size(x)==0), do: elem(x,0)
```

This cannot be directly expressed in Core Elixir with negation, although the equivalent guard `tuple_size(x)!=0` can be used.

We adapt Core Elixir guard and pattern analysis to FW-Elixir by compiling FW-Elixir guards into Core Elixir: push negations to leaves via De Morgan rules and re-encode relations not present in Core Elixir using the available ones. For example, `not(tuple_size(x)==0)`

$$
\begin{aligned}
\mathbf{T}_{\mathsf{G}}(\mathsf{D}) &= \mathbf{T}_{\mathsf{D}}(\mathsf{D})\,?\,\mathtt{true}\\
\mathbf{T}_{\mathsf{G}}(\mathsf{C}) &= \mathbf{T}_{\mathsf{C}}(\mathsf{C})\\
\mathbf{T}_{\mathsf{G}}(\mathsf{G}_1 \text{ and } \mathsf{G}_2) &= \mathbf{T}_{\mathsf{G}}(\mathsf{G}_1) \text{ and } \mathbf{T}_{\mathsf{G}}(\mathsf{G}_2)\\
\mathbf{T}_{\mathsf{G}}(\mathsf{G}_1 \text{ or } \mathsf{G}_2) &= \mathbf{T}_{\mathsf{G}}(\mathsf{G}_1) \text{ or } \mathbf{T}_{\mathsf{G}}(\mathsf{G}_2)\\
\mathbf{T}_{\mathsf{G}}(\text{not}\,\mathsf{G}) &= \mathbf{N}_{\mathsf{G}}(\mathsf{G})\\
\mathbf{T}_{\mathsf{G}}(\mathsf{G}_1 == \mathsf{G}_2) &= \mathbf{T}_{\mathsf{G}}(\mathsf{G}_1) = \mathbf{T}_{\mathsf{G}}(\mathsf{G}_2)\\
\mathbf{T}_{\mathsf{G}}(\mathsf{G}_1\,!= \mathsf{G}_2) &= \mathbf{T}_{\mathsf{G}}(\mathsf{G}_1)\,!= \mathbf{T}_{\mathsf{G}}(\mathsf{G}_2)\\
\mathbf{T}_{\mathsf{G}}(\mathsf{G}_1 < \mathsf{G}_2) &= \mathbf{T}_{\mathsf{G}}(\mathsf{G}_1) < \mathbf{T}_{\mathsf{G}}(\mathsf{G}_2)\\
\mathbf{T}_{\mathsf{G}}(\mathsf{G}_1 > \mathsf{G}_2) &= \mathbf{T}_{\mathsf{G}}(\mathsf{G}_2) < \mathbf{T}_{\mathsf{G}}(\mathsf{G}_1)\\
\mathbf{T}_{\mathsf{G}}(\mathsf{G}_1 <= \mathsf{G}_2) &= \mathbf{T}_{\mathsf{G}}(\mathsf{G}_1) < \mathbf{T}_{\mathsf{G}}(\mathsf{G}_2)\\
&\quad \mathtt{or}\ \ \mathbf{T}_{\mathsf{G}}(\mathsf{G}_1) = \mathbf{T}_{\mathsf{G}}(\mathsf{G}_2)\\
\mathbf{T}_{\mathsf{G}}(\mathsf{G}_1 >= \mathsf{G}_2) &= \mathbf{T}_{\mathsf{G}}(\mathsf{G}_2) < \mathbf{T}_{\mathsf{G}}(\mathsf{G}_1)\\
&\quad \mathtt{or}\ \ \mathbf{T}_{\mathsf{G}}(\mathsf{G}_1) = \mathbf{T}_{\mathsf{G}}(\mathsf{G}_2)
\end{aligned}
\qquad
\begin{aligned}
\mathbf{T}_{\mathsf{D}}(\mathtt{elem}(\mathsf{D}_1,\mathsf{D}_2)) &= \pi_{\mathbf{T}_{\mathsf{D}}(\mathsf{D}_1)}\mathbf{T}_{\mathsf{D}}(\mathsf{D}_2)\\
\mathbf{T}_{\mathsf{D}}(\mathtt{tuple\_size}(\mathsf{D})) &= \mathtt{size}\,\mathbf{T}_{\mathsf{D}}(\mathsf{D})\\
\mathbf{T}_{\mathsf{D}}(\{\mathsf{D}_1,..,\mathsf{D}_n\}) &= \{\mathbf{T}_{\mathsf{D}}(\mathsf{D}_1),..,\mathbf{T}_{\mathsf{D}}(\mathsf{D}_n)\}\\[1ex]
\mathbf{T}_{\mathsf{C}}(\mathtt{is\_integer}(\mathsf{D})) &= \mathbf{T}_{\mathsf{D}}(\mathsf{D})\,?\,\mathtt{int}\\
\mathbf{T}_{\mathsf{C}}(\mathtt{is\_atom}(\mathsf{D})) &= \mathbf{T}_{\mathsf{D}}(\mathsf{D})\,?\,\mathtt{atom}\\
\mathbf{T}_{\mathsf{C}}(\mathtt{is\_tuple}(\mathsf{D})) &= \mathbf{T}_{\mathsf{D}}(\mathsf{D})\,?\,\mathtt{tuple}\\
\mathbf{T}_{\mathsf{C}}(\mathtt{is\_function}(\mathsf{D})) &= \mathbf{T}_{\mathsf{D}}(\mathsf{D})\,?\,\mathtt{fun}\\
\mathbf{T}_{\mathsf{C}}(\mathtt{is\_function}(\mathsf{D},n)) &= \mathbf{T}_{\mathsf{D}}(\mathsf{D})\,?\,\mathtt{fun}_n
\end{aligned}
$$

$$
\begin{aligned}
\mathbf{N}_{\mathsf{G}}(\mathsf{D}) &= \mathbf{T}_{\mathsf{D}}(\mathsf{D})\,?\,\mathtt{false}\\
\mathbf{N}_{\mathsf{G}}(\mathsf{C}) &= \mathbf{N}_{\mathsf{C}}(\mathsf{C})\\
\mathbf{N}_{\mathsf{G}}(\mathsf{G}_1 \text{ and } \mathsf{G}_2) &= \mathbf{N}_{\mathsf{G}}(\mathsf{G}_1) \text{ or } \mathbf{N}_{\mathsf{G}}(\mathsf{G}_2)\\
\mathbf{N}_{\mathsf{G}}(\mathsf{G}_1 \text{ or } \mathsf{G}_2) &= \mathbf{N}_{\mathsf{G}}(\mathsf{G}_1) \text{ and } \mathbf{N}_{\mathsf{G}}(\mathsf{G}_2)\\
\mathbf{N}_{\mathsf{G}}(\text{not}\,\mathsf{G}) &= \mathbf{T}_{\mathsf{G}}(\mathsf{G})\\
\mathbf{N}_{\mathsf{G}}(\mathsf{G}_1 == \mathsf{G}_2) &= \mathbf{T}_{\mathsf{G}}(\mathsf{G}_1)\,!= \mathbf{T}_{\mathsf{G}}(\mathsf{G}_2)\\
\mathbf{N}_{\mathsf{G}}(\mathsf{G}_1\,!= \mathsf{G}_2) &= \mathbf{T}_{\mathsf{G}}(\mathsf{G}_1) = \mathbf{T}_{\mathsf{G}}(\mathsf{G}_2)\\
\mathbf{N}_{\mathsf{G}}(\mathsf{G}_1 < \mathsf{G}_2) &= \mathbf{T}_{\mathsf{G}}(\mathsf{G}_2) < \mathbf{T}_{\mathsf{G}}(\mathsf{G}_1)\\
&\quad \mathtt{or}\ \ \mathbf{T}_{\mathsf{G}}(\mathsf{G}_1) = \mathbf{T}_{\mathsf{G}}(\mathsf{G}_2)\\
\mathbf{N}_{\mathsf{G}}(\mathsf{G}_1 > \mathsf{G}_2) &= \mathbf{T}_{\mathsf{G}}(\mathsf{G}_1) < \mathbf{T}_{\mathsf{G}}(\mathsf{G}_2)\\
&\quad \mathtt{or}\ \ \mathbf{T}_{\mathsf{G}}(\mathsf{G}_1) = \mathbf{T}_{\mathsf{G}}(\mathsf{G}_2)\\
\mathbf{N}_{\mathsf{G}}(\mathsf{G}_1 <= \mathsf{G}_2) &= \mathbf{T}_{\mathsf{G}}(\mathsf{G}_2) < \mathbf{T}_{\mathsf{G}}(\mathsf{G}_1)\\
\mathbf{N}_{\mathsf{G}}(\mathsf{G}_1 >= \mathsf{G}_2) &= \mathbf{T}_{\mathsf{G}}(\mathsf{G}_1) < \mathbf{T}_{\mathsf{G}}(\mathsf{G}_2)
\end{aligned}
\qquad
\begin{aligned}
\mathbf{N}_{\mathsf{C}}(\mathtt{is\_integer}(\mathsf{D})) &= \mathbf{T}_{\mathsf{D}}(\mathsf{D})\,?\,(\neg\mathtt{int})\\
\mathbf{N}_{\mathsf{C}}(\mathtt{is\_atom}(\mathsf{D})) &= \mathbf{T}_{\mathsf{D}}(\mathsf{D})\,?\,(\neg\mathtt{atom})\\
\mathbf{N}_{\mathsf{C}}(\mathtt{is\_tuple}(\mathsf{D})) &= \mathbf{T}_{\mathsf{D}}(\mathsf{D})\,?\,(\neg\mathtt{tuple})\\
\mathbf{N}_{\mathsf{C}}(\mathtt{is\_function}(\mathsf{D})) &= \mathbf{T}_{\mathsf{D}}(\mathsf{D})\,?\,(\neg\mathtt{fun})\\
\mathbf{N}_{\mathsf{C}}(\mathtt{is\_function}(\mathsf{D},n)) &= \mathbf{T}_{\mathsf{D}}(\mathsf{D})\,?\,(\neg\mathtt{fun}_n)
\end{aligned}
$$

Figure 13.3: Guard compilation

compiles to $\mathtt{size}(x)\,!\!=\!0$, and `not(is_function(x) or is_tuple(x))` compiles to $(x\,?\,\neg\mathtt{fun})$ and $(x\,?\,\neg\mathtt{tuple})$. Likewise, `x >= y` compiles to $(y < x)$ $\mathtt{or}$ $(x = y)$.

The compilation preserves guard success: a FW-Elixir value succeeds on a guard if and only if the corresponding Core Elixir value succeeds on the compiled guard.

Guard compilation, defined in Figure 13.3, is given by defining two mutually recursive functions from the set $\mathscr{G}_{\text{Elixir}}$ of Elixir concrete guards of FW-Elixir to the set $\mathscr{G}_{\text{Core}}$ of Core Elixir guards (syntax in Figure 6.1). Precisely, the $\mathbf{T}_{\mathsf{G}} : \mathscr{G}_{\text{Elixir}} \to \mathscr{G}_{\text{Core}}$ function compiles a concrete guard into a core guard, and the $\mathbf{N}_{\mathsf{G}} : \mathscr{G}_{\text{Elixir}} \to \mathscr{G}_{\text{Core}}$ does so as well, but also pushes down a logical negation into the guard, which means that, say, a type-check of `int` becomes a type-check of ¬`int`, and that conjunctions and disjunctions are swapped using De Morgan rules. There is no $\mathbf{N}_{\mathsf{G}}$ defined on selectors `D`: $\mathbf{N}_{\mathsf{G}}$ is just an auxiliary function for $\mathbf{T}_{\mathsf{G}}$, which does not call it on `D` productions (a selector `D` is directly translated into checking whether it has the singleton type `true`). Notice that the compilation defines the semantic of relations present only in FW-Elixir, such as '`>=`', which is compiled as the union of '`<`' and '`=`'. In particular the negation of $\mathsf{G}_1 < \mathsf{G}_2$—i.e., $\mathbf{N}_{\mathsf{G}}(\mathsf{G}_1 < \mathsf{G}_2)$—is compiled as (the compilation of) $(\mathsf{G}_2 < \mathsf{G}_1)$ $\mathtt{or}$ $(\mathsf{G}_1 = \mathsf{G}_2)$ which is correct since all (FW-)Elixir values are totally ordered (cf. Figure 6.4).

FW-Elixir	**Core Elixir (compiled)**
`not(is_function(x) or is_tuple(x))`	$(x\,?\,\neg\mathtt{fun})$ and $(x\,?\,\neg\mathtt{tuple})$
`not(tuple_size(x) == 0)`	$\mathtt{size}\ x\ !{=}\ 0$
`x >= y`	$y < x$ or $x = y$

Table 13.1: Examples of FW-Elixir guards and their compiled Core Elixir counterparts.

Lemma 13.1.1 (Compilation preserves guard success)**.** *For every FW-Elixir guard $g \in \mathcal{G}_{Elixir}$ and every environment/value interpretation, a value succeeds on g if and only if it succeeds on* $\mathbf{T}_G(g)$.

Proof. By mutual induction on $\mathbf{T}_G(g)$ and $\mathbf{N}_G(g)$, assuming the same semantics for the guards in both Elixir and FW-Elixir (with the added obvious reduction for the not-guard). □

Examples (FW-Elixir vs Core Elixir). Table 13.1 summarizes a few illustrative translations.

13.2 Computing function type application

Having established the correspondence between FW-Elixir and Core Elixir through guard compilation, we now turn to the implementation of the typing rules themselves. The type system presented in Chapters 4, 5 and 6 is declarative. Its implementation is not straightforward, as it relies on details of the representation of set-theoretic types in semantic subtyping. This implies that some rules are implemented with additional conditions, or by using type operators more refined than those given by the theoretical system.

This is particularly the case for the rules used to type function applications, specialized to account for Elixir's strict evaluation semantics. We first describe how the application rule (app) is implemented and then examine the consequences on (app$_?$).

13.2.1 Gradual function application

In the static declarative system presented in Chapter 4 and 5, we use a declarative rule for typing function application:

$$(\text{app})\ \frac{\Gamma \vdash_s e : t_1 \to t_2 \quad \Gamma \vdash_s e' : t_1}{\Gamma \vdash_s e(e') : t_2}$$

In the algorithmic system with static types, checking that a function can be applied to an argument requires the domain operator $\mathtt{dom}(\cdot)$ over function types (Def. 3.3.4), and we use the application result operator $\circ$ (Def. 3.3.5) to compute t_2.

The gradual rule for application is the same, but we need to lift the static operators to do so. Lanvin (2021) does this for both, defining the gradual domain as:

$$\mathtt{dom}(\tau) = \mathtt{dom}(\tau^{\Uparrow}) \vee (? \wedge \mathtt{dom}(\tau^{\Downarrow}))$$

and the gradual result operator $\circ$ as:

$$\tau \circ \sigma = \tau^{\Downarrow} \circ \sigma^{\Uparrow} \vee (? \wedge (\tau^{\Uparrow} \circ \sigma^{\Downarrow}))$$

With the addition of the interpretation of arrows, to $\mho$ in the domain, which now makes it possible to distinguish e.g. types $\mathbb{0} \to \mathbb{1}$, $\mathbb{0} \to \mathtt{int}$, $\mathbb{0} \to \mathbb{0}$.

Remark (*Interesting interpretations*)

Let us verify that indeed $\mathbb{0} \to \mathbb{0} \leq \mathbb{0} \to \mathbb{1}$ but $\mathbb{0} \to \mathbb{1} \not\leq \mathbb{0} \to \mathbb{0}$.

$$\begin{aligned}
[\![\mathbb{0} \to \mathbb{0}]\!] &= \mathscr{P}_f\left(\overline{(\varnothing \cup \{\mho\}) \times \overline{\varnothing}^{\mathscr{D}_\Omega}}^{\mathscr{D}_\mho \times \mathscr{D}_\Omega}\right) \\
&= \mathscr{P}_f\left(\overline{\{\mho\} \times \mathscr{D}_\Omega}^{\mathscr{D}_\mho \times \mathscr{D}_\Omega}\right) \\
&= \mathscr{P}_f\left(\mathscr{D} \times \mathscr{D}_\Omega\right)
\end{aligned}$$

On the other hand,

$$\begin{aligned}
[\![\mathbb{0} \to \mathbb{1}]\!] &= \mathscr{P}_f\left(\overline{(\varnothing \cup \{\mho\}) \times \overline{\mathscr{D}_\Omega}^{\mathscr{D}_\Omega}}^{\mathscr{D}_\mho \times \mathscr{D}_\Omega}\right) \\
&= \mathscr{P}_f\left(\overline{\{\mho\} \times \varnothing}^{\mathscr{D}_\mho \times \mathscr{D}_\Omega}\right) \\
&= \mathscr{P}_f\left(\overline{\varnothing}^{\mathscr{D}_\mho \times \mathscr{D}_\Omega}\right) \\
&= \mathscr{P}_f\left(\mathscr{D}_\mho \times \mathscr{D}_\Omega\right)
\end{aligned}$$

and we see that $[\![\mathbb{0} \to \mathbb{0}]\!] = \mathscr{P}_f(\mathscr{D} \times \mathscr{D}_\Omega) \subsetneq \mathscr{P}_f(\mathscr{D}_\mho \times \mathscr{D}_\Omega) = [\![\mathbb{0} \to \mathbb{1}]\!]$.

If these arrows were identified, as it is in previous versions of semantic subtyping, the most precise result of applying a function of type $\mathbb{0} \to t$ would be $\mathbb{0}$. With $\mho$, we can infer t. Lanvin (2021) thus adds a special case to Def. 3.3.5 when the argument is $\mathbb{0}$:

Definition 13.2.1 (Result operator (Lanvin 2021))**.** *For every type $t \leq \mathbb{0} \to \mathbb{1}$ and every type s such that $s \leq \mathtt{dom}(t)$, we define the result type of t applied to s, noted $t \circ s$ as:*

$$\begin{aligned}
t \circ s &= \bigvee_{i \in I^+} \bigvee_{\substack{Q \subsetneq P_i \\ s \not\leq \bigvee_{s_i \to t_i \in Q} s_i}} \bigwedge_{s_i \to t_i \in P_i \setminus Q} t_i \quad \text{if } s \neq \mathbb{0} \\
&= \bigvee_{i \in I} \bigwedge_{s_i \to t_i \in P_i} t_i \quad \text{otherwise}
\end{aligned}$$

where

$$t \simeq \bigwedge_{i \in I^+} \left(\bigwedge_{(s_i, t_i) \in P_i} s_i \to t_i \wedge \bigwedge_{(s'_i, t'_i) \in N_i} \neg(s'_i \to t'_i) \right)$$

Computing gradual application (via subtyping) We recall the statement (notation as in Lanvin (2021)), which we will show to be *incorrect*:

Proposition 13.2.2 (Proposition A.23 in Lanvin (2021)). *For all $\tau, \tau', \sigma \in \mathscr{T}_{gradual}$ such that $\tau \leq \mathbb{0} \to \mathbb{1}$ and $\sigma \leq \mathtt{dom}(\tau)$,*

$$\text{(i) } \tau \leq \sigma \to \tau \circ \sigma \qquad \text{(ii) if } \tau \leq \sigma \to \tau' \text{ then } \tau \circ \sigma \leq \tau'$$

Counterexample. Let

$$\tau = ? \wedge (\mathtt{int} \to \mathtt{int}) \wedge (\mathtt{bool} \to \mathtt{bool}) \quad \text{and} \quad \sigma = ? \wedge \mathtt{int}$$

yielding

$$\tau^{\Downarrow} = \mathbb{0}, \quad \tau^{\Uparrow} = (\mathtt{int} \to \mathtt{int}) \wedge (\mathtt{bool} \to \mathtt{bool}), \quad \sigma^{\Downarrow} = \mathbb{0}, \quad \sigma^{\Uparrow} = \mathtt{int}$$

Then $\tau \leq \mathbb{0} \to \mathbb{1}$ obviously. Notice that, in the definition of domain 3.3.4, computing the domain of $\mathbb{0}$ means that $I = \varnothing$, so the domain is an intersection over an empty set, which is by convention equal to $\mathbb{1}$. This is consistent with the fact that the domain is contravariant, so it is maximal for the smallest type ($\mathbb{0}$). Semantically, this means that any argument can be applied to a function of type $\mathbb{0}$, since this function raises anyway (call-by-value semantics). Therefore, we have

$$\mathtt{dom}(\tau) = \mathtt{dom}(\tau^{\Uparrow}) \vee (? \wedge \mathtt{dom}(\tau^{\Downarrow})) = (\mathtt{int} \vee \mathtt{bool}) \vee ?$$

So we have $\sigma = ? \wedge \mathtt{int} \leq \mathtt{dom}(\tau)$. Therefore, we compute $\tau \circ \sigma = (\tau^{\Downarrow} \circ \sigma^{\Uparrow}) \vee (? \wedge (\tau^{\Uparrow} \circ \sigma^{\Downarrow}))$.

- First, $\tau^{\Downarrow} \circ \sigma^{\Uparrow}$: here $\sigma^{\Uparrow} = \mathtt{int} \leq \mathtt{dom}(\tau^{\Downarrow}) = \mathtt{dom}(\mathbb{0}) = \mathbb{1}$, hence by Def. 3.3.5, $\tau^{\Downarrow} \circ \sigma^{\Uparrow} = \mathbb{0}$.
- Second, $\tau^{\Uparrow} \circ \sigma^{\Downarrow}$: since $\sigma^{\Downarrow} = \mathbb{0} \leq \mathtt{dom}(\tau^{\Uparrow}) = \mathtt{int} \vee \mathtt{bool}$, the amended Def. 13.2.1 gives $\mathbb{0}$.

Thus, $\tau^{\Downarrow} \circ \sigma^{\Uparrow} = \mathbb{0}$ and $\tau^{\Uparrow} \circ \sigma^{\Downarrow} = \mathbb{0}$, so $\tau \circ \sigma = \mathbb{0}$. This violates the condition of the proposition, as it is not true that "$\tau \leq \sigma \to (\tau \circ \sigma)$", that is, it is not true that $? \wedge (\mathtt{int} \to \mathtt{int}) \wedge (\mathtt{bool} \to \mathtt{bool})$ is a subtype of $(? \wedge \mathtt{int}) \to \mathbb{0}$, because it is not true (by Definition of subtyping on gradual types 3.2.7) that their maximal extrema are subtypes, which is that $(\mathtt{int} \to \mathtt{int}) \wedge (\mathtt{bool} \to \mathtt{bool}) \not\leq (\mathbb{0} \to \mathbb{0})$. Indeed, we have computed $[\![\mathbb{0} \to \mathbb{0}]\!] = \mathscr{P}_f(\mathscr{D} \times \mathscr{D}_\Omega)$ in Remark 13.2.1. On the other hand, we have

$$[\![\mathtt{int} \to \mathtt{int}]\!] = \mathscr{P}_f\left(\overline{([\![\mathtt{int}]\!] \cup \{\mho\}) \times \overline{\mathtt{int}}^{\mathscr{D}_\Omega}}^{\mathscr{D}_\mho \times \mathscr{D}_\Omega}\right)$$
$$[\![\mathtt{bool} \to \mathtt{bool}]\!] = \mathscr{P}_f\left(\overline{([\![\mathtt{bool}]\!] \cup \{\mho\}) \times \overline{\mathtt{bool}}^{\mathscr{D}_\Omega}}^{\mathscr{D}_\mho \times \mathscr{D}_\Omega}\right)$$

To find a concrete element that belongs to both $[\![\mathtt{int} \to \mathtt{int}]\!]$ and $[\![\mathtt{bool} \to \mathtt{bool}]\!]$ but not to $[\![\mathbb{0} \to \mathbb{0}]\!]$, consider the pair $(\mho, \Omega) \in \mathscr{D}_\mho \times \mathscr{D}_\Omega$.

This pair belongs to $[\![\mathtt{int} \to \mathtt{int}]\!]$ because $\mho \in [\![\mathtt{int}]\!] \cup \{\mho\}$ and $\Omega \in \overline{[\![\mathtt{int}]\!]}^{\mathscr{D}_\Omega}$ (since $\Omega \notin [\![\mathtt{int}]\!]$). Similarly, it belongs to $[\![\mathtt{bool} \to \mathtt{bool}]\!]$ because $\mho \in [\![\mathtt{bool}]\!] \cup \{\mho\}$ and $\Omega \in \overline{[\![\mathtt{bool}]\!]}^{\mathscr{D}_\Omega}$.

However, $(\mho, \Omega) \notin [\![\mathbb{0} \to \mathbb{0}]\!] = \mathscr{P}_f(\mathscr{D} \times \mathscr{D}_\Omega)$ because $\mho \notin \mathscr{D}$ (by definition, $\mho$ is a special element not in the base domain). We have thus found a concrete τ and σ such that $\tau \not\leq \sigma \to (\tau \circ \sigma)$, which directly contradicts Proposition 13.2.2.

$$\tau \not\leq \sigma \to (\tau \circ \sigma) \qquad \text{(the analysis shows } \tau^{\Uparrow} \not\leq \sigma \to (\tau \circ \sigma)^{\Uparrow}\text{)} \qquad \square$$

Source of the discrepancy. The proof of A.23 in Lanvin (2021), found in appendix, relies on its static version, Proposition 6.13, which we recall here:

Proposition 13.2.3 (Proposition 6.13 in Lanvin (2021))**.** *For all $t, t', s \in \mathscr{T}_{static}$ such that $t \leq \mathbb{0} \to \mathbb{1}$ and $s \leq \texttt{dom}(t)$,*

$$\textit{(i)}\ t \leq s \to t \circ s \qquad \textit{(ii) if}\ t \leq s \to t'\ \textit{then}\ t \circ s \leq t' \circ s$$

In particular, it invokes Proposition 6.13 on $t = (\texttt{int} \to \texttt{int}) \wedge (\texttt{bool} \to \texttt{bool})$ and $s = \mathbb{0}$. This proposition appears right after defining the result operator with an $\mathbb{0}$ argument as special case, that gives $t \circ s = \mathbb{0}$. But it is not true that $(\texttt{int} \to \texttt{int}) \wedge (\texttt{bool} \to \texttt{bool}) \leq \mathbb{0} \to \mathbb{0}$ (it would be true if $\mathbb{0} \to \mathbb{0} \simeq \mathbb{0} \to \mathbb{1}$, that is, without having the $\mho$ element). This issue only comes up with the empty argument.

Our fix. Lanvin (2021) defines the gradual result operator $\circ$:

$$\tau \circ \sigma = \tau^{\Downarrow} \circ \sigma^{\Uparrow} \vee (? \wedge (\tau^{\Uparrow} \circ \sigma^{\Downarrow}))$$

and justifies this definition by arguing that "the type of the result of an application is covariant in the type of the function but contravariant in the type of the argument: for a function of type τ and an argument of type σ, the smallest possible result is obtained when τ is the smallest and σ is the largest, and conversely."

We note that the contravariance claim for the argument is generally incorrect: given the function $(\texttt{int} \to \texttt{int}) \wedge (\texttt{bool} \to \texttt{bool})$, whose domain is $\texttt{int} \vee \texttt{bool}$, when applied to $\texttt{int}$, it yields $\texttt{int}$, and when applied to the *supertype* $\texttt{int} \vee \texttt{bool}$, it yields $\texttt{int} \vee \texttt{bool}$: the result is *covariant* in the argument. In general, for gradual application, provided that it is type-safe (which is verified by condition $\sigma^{\Uparrow} \leq \texttt{dom}(\tau^{\Uparrow})$) to apply $\tau^{\Uparrow}$ to $\sigma^{\Uparrow}$, the type $\tau^{\Uparrow} \circ \sigma^{\Uparrow}$ is a supertype of $\tau^{\Uparrow} \circ \sigma^{\Downarrow}$. We thus replace the definition of the gradual result operator $\circ$ by:

$$\tau \circ \sigma = \tau^{\Downarrow} \circ \sigma^{\Downarrow} \vee (? \wedge (\tau^{\Uparrow} \circ \sigma^{\Uparrow}))$$

Note that the corrected proposition still requires $\sigma^{\Downarrow} \neq \mathbb{0}$ as a side condition: when $\sigma^{\Downarrow} = \mathbb{0}$, the check $\tau \leq \sigma \to (\tau \circ \sigma)$ reduces to $\tau^{\Uparrow} \leq \mathbb{0} \to (\tau \circ \sigma)^{\uparrow}$, which fails for any standard function type due to $\mho$ (the pair $(\mho, \Omega)$ belongs to $[\![\tau^{\Uparrow}]\!]$ but not to $[\![\mathbb{0} \to t]\!]$ for $t \neq \mathbb{1}$). This is not a deficiency of the formula but a fundamental consequence of distinguishing $\mathbb{0} \to t$ from $\mathbb{0} \to \mathbb{1}$ via $\mho$. In practice, when $\sigma^{\Downarrow} = \mathbb{0}$, the typing is handled by the rule (app$_?$) using compatibility and the extended gradual domain (§13.2), which bypasses this proposition entirely.

13.2.2 Sanity checking of dynamic function application (Rule (app$_?$))

Consider the application of a function of type `bool` → `bool` to an argument of type ? ∧ `int`. Intuitively, we want this application to be rejected: the function works only with Boolean arguments while the type ? ∧ `int` indicates that the argument can only evaluate to integer values.

The argument type ? ∧ `int` is not uncommon, since the presence of the dynamic type may result from dynamic propagation for enhanced flexibility (to allow using such an argument where a strict subtype of `int` is expected). However, in this case the flexibility is excessive: since ? ∧ `int` can materialize to $\mathbb{O}$, we could naively apply rule (app$_?$) and deduce the type ? for the application, instead of rejecting it.

The logic of this deduction is that if the argument materializes to $\mathbb{O}$ (i.e., the argument is a diverging expression), then the application is sound–which is correct, since the application will never return. However, accepting an application as sound solely because its argument–and thus the entire application–*might* diverge is nonsensical: if the argument actually diverges, then we do not care what the type-checker deduced.

Therefore, in practice we implement the rule (app$_?$) to reject applications when the only solution is to materialize the argument type to $\mathbb{O}$. We achieve this by introducing a *compatibility* relation, which is a restricted version of the consistent subtyping relation, and using it in our implementation of (app$_?$) instead of the usual consistent subtyping relation. This yields a stricter and still sound typing of gradual function applications.

Recall that *consistent subtyping*, denoted by $\lesssim$, is a relation between two types t_1 and t_2 such that if $t_1 \lesssim t_2$, then there exist t_1', t_2' where $t_1 \preccurlyeq t_1'$, $t_2 \preccurlyeq t_2'$, and $t_1' \leq t_2'$.

When we add the restriction that t_1' cannot be empty, we obtain the *compatibility* relation, denoted by $t_1 \sqsubseteq_c t_2$. Formally:

Definition 13.2.4 (Compatibility)**.**

$$t_1 \sqsubseteq_c t_2 \textit{ iff } (\textit{if } {t_1}^{\Downarrow} \leq \mathbb{O} \textit{ then } ({t_1}^{\Uparrow} \wedge {t_2}^{\Uparrow} \not\leq \mathbb{O}) \textit{ else } {t_1}^{\Downarrow} \leq {t_2}^{\Uparrow}) \textit{ or } (t_1 \simeq \mathbb{O} \wedge t_2 \simeq \mathbb{O}) \tag{13.2.1}$$

Compatibility clearly implies consistent subtyping: if t_1 is compatible with t_2, then either the first case of the conditional holds (in which case ${t_1}^{\Downarrow}$ is empty, so ${t_1}^{\Downarrow} \leq {t_2}^{\Uparrow}$), or the second case holds, which directly corresponds to consistent subtyping. The final disjunct in the definition (13.2.1) ensures that the empty type is compatible with itself, which is necessary for subtyping to imply compatibility.

In practice, this means that (app$_?$) accepts a materialization of the argument type to $\mathbb{O}$ only if the function explicitly expects arguments of type $\mathbb{O}$. While defining functions with empty domains is of limited interest in Elixir, this may be crucial in other dynamic languages.

For instance, `ty`—a type-checker for Python developed by Astral Astral Team (n.d.)—uses a function `assert_never()` that expects an argument of type $\mathbb{O}$ to check exhaustiveness of nested if/elif commands: the checked expression is passed in the last else-branch to `assert_never()` to statically verify that its type narrowed to the empty type (this is a standard function in the Python type system specification – see the specification Python Software Foundation (2025b)).

Even though this pattern is less common in Elixir, we have nevertheless implemented the compatibility relation for the Elixir compiler as given in (13.2.1).

As in the case for the (app) rule, the implementation of the ($\text{app}_?$) rule checks the compatibility (rather than subtyping) between the argument type and the domain of the function type as defined in Lanvin's gradual extension.

However, there is a subtlety in using the gradual domain for the ($\text{app}_?$) rule. Recall that the rule ($\text{app}_?$) allows both the function and the argument type to materialize, and that we use the compatibility relation to check whether the argument type is in the domain of the function type, while preventing the argument from materializing to $\mathbb{0}$. For this check, we need the domain of the function type on the right-hand side of compatibility.

Lanvin's gradual domain formula $\mathtt{dom}(\tau) = \mathtt{dom}(\tau^{\Uparrow}) \vee (? \wedge \mathtt{dom}(\tau^{\Downarrow}))$ works correctly when $\tau^{\Uparrow} \leq \mathtt{fun}$, since the static domain operator (Def. 3.3.4) requires its argument to be a function type. For instance, the gradual domain of $? \wedge (\mathtt{int} \to \mathtt{int})$ is $\mathtt{dom}(\mathtt{int} \to \mathtt{int}) \vee (? \wedge \mathtt{dom}(\mathbb{0})) = \mathtt{int} \vee (? \wedge \mathbb{1}) = \mathtt{int} \vee ?$, correctly reflecting that integers are certainly accepted while other values may also work dynamically.

However, when $\tau^{\Uparrow} \not\leq \mathtt{fun}$, the static domain $\mathtt{dom}(\tau^{\Uparrow})$ is undefined. This is the case for $\tau = ?$, where $?^{\Uparrow} = \mathbb{1}$: a function of type ? may accept any argument at runtime, so we expect $\widetilde{\mathtt{dom}}(?) = ?$, but Lanvin's formula cannot be applied.

To handle this, we extend the gradual domain by projecting $t^{\Uparrow}$ onto its function part before computing the domain:

Definition 13.2.5 (Extended gradual domain)**.**

$$\widetilde{\mathtt{dom}}(t) = \mathtt{dom}(t^{\Uparrow} \wedge \mathit{fun}) \vee (? \wedge \mathtt{dom}(t^{\Downarrow}))$$

where $\mathtt{dom}(\cdot)$ is the static domain operator (Def. 3.3.4), and the precondition is $t^{\Downarrow} \leq \mathtt{fun}$ (so that $\mathtt{dom}(t^{\Downarrow})$ is defined). When $t^{\Uparrow} \leq \mathtt{fun}$, this reduces to Lanvin's formula. When $t^{\Uparrow} \not\leq \mathtt{fun}$, the $\wedge\,\mathtt{fun}$ projection extracts the function part of $t^{\Uparrow}$ before computing its domain, while the second component $? \wedge \mathtt{dom}(t^{\Downarrow})$ provides the dynamic flexibility. For example:

- $\widetilde{\mathtt{dom}}(?) = \mathtt{dom}(\mathbb{1} \wedge \mathtt{fun}) \vee (? \wedge \mathtt{dom}(\mathbb{0})) = \mathtt{dom}(\mathbb{0} \to \mathbb{1}) \vee (? \wedge \mathbb{1}) = \mathbb{0} \vee ? = ?$.
- $\widetilde{\mathtt{dom}}((\mathtt{int} \to \mathtt{int}) \vee (? \wedge \mathtt{int})) = \mathtt{dom}(((\mathtt{int} \to \mathtt{int}) \vee \mathtt{int}) \wedge \mathtt{fun}) \vee (? \wedge \mathtt{dom}(\mathtt{int} \to \mathtt{int})) = \mathtt{dom}(\mathtt{int} \to \mathtt{int}) \vee (? \wedge \mathtt{int}) = \mathtt{int}$, since the non-function part $? \wedge \mathtt{int}$ is correctly discarded by the $\wedge\,\mathtt{fun}$ projection.

Now the condition for rule ($\mathit{app}_?$) is that the argument type t_2 must be compatible with the gradual domain of the function type t_1, that is $t_2 \sqsubseteq_c \widetilde{\mathtt{dom}}(t_1)$. Safety is preserved since the application is typed by a supertype of ? (the type deduced by the ($\text{app}_?$) rule), which is always safe.

Conclusion

We connected Core Elixir to a concrete fragment (FW-Elixir) and showed how guard negation can be compiled away, aligning surface syntax with our core analysis. We then implemented function application, first in the static setting and then in the gradual one, introducing the compatibility check and the gradual domain to precisely handle function application in a gradual setting. Finally, we outlined how these operators are realized in the Elixir compiler.

What comes next We now turn to Chapter 14: an empirical study of the system in the wild. We measure compilation and type-checking overheads on major Elixir codebases, summarize the rollout across releases, and report developer feedback and issues uncovered in practice.

CHAPTER 14

EVALUATION

"Elixir is, officially, a gradually typed language."

José Valim (Valim, 2024a)

Elixir Integration Status.

All the features presented in this thesis have been progressively implemented in Elixir by José Valim and myself since version 1.17 (included). Version 1.17, released in June 2024, incorporated gradual set-theoretic types for atoms, maps, and few basic indivisible types; version 1.18 covered more types and type-checked function calls performing type inference for patterns and return types. The current v1.19 implementation covers all language constructs and includes basic, atom, tuple, list, map, and function types, as well as the typing of *protocols* (akin to Haskell type classes). In v1.19 type inference for functions is refined to take partially into account the type of the operators the functions use. Its completion is planned for the next v1.20 release. The current implementation of the type system does not require any modification to the syntax of Elixir. The first modifications of the syntax are planned for the v1.21 with the explicit annotations for *structs* (which are named maps with a predefined set of keys, used as data containers) and, subsequently, the $-prefixed function type annotations demonstrated in our examples. The next steps of the integration of the type system into Elixir are detailed in a regularly updated roadmap in the Elixir documentation (Elixir Core Team, 2024). Each release follows an approximately six-month development cycle. The type system integration is performed with parsimony and minimal disruption to existing workflows. Currently, the type checker produces only warnings rather than hard errors, ensuring that the type system does not interfere with established Elixir development practices.

Timeline. The performance trajectory from v1.18 to v1.19 mirrors the data-structure evolution described in Chapter 12: DNFs sufficed until function inference introduced pervasive differences; initial BDD/TDD adoption curbed worst-case unions but regressed on intersections of equal nodes; fixing the equal-node formulas restored laziness in the union branch and eliminated the regressions. We ground this account in two summary tables introduced in Chapter 12: PT1 (Table 12.2) contrasts v1.18 vs v1.19 on representative projects, while PT2 (Table 12.3) ablates compiler milestones. We refer to these when interpreting the results below.

We used release candidate version 1.18-rc0 in September 2024 to perform a first assessment of the overhead introduced by our typechecker on five substantial and extensively tested codebases: Remote is one of the largest Elixir codebases with over a million lines of code (this is a conservative estimation, only the exact number of modules is known); Credo, Livebook, and Phoenix are among the most popular Elixir packages; and Hex is the package manager for the Elixir ecosystem. We describe those codebases in more detail in Appendix A.5.

Codebase	LoC	Files	Modules	Type Checking Time	Total Compilation Time	%
Remote	>1,000,000	>10,000	18,059	11.116 s	707.598 s	1.6
Livebook	61093	254	299	0.177 s	4.112 s	4.3
Credo	29181	252	264	0.059 s	1.305 s	4.5
Phoenix	21389	71	88	0.049 s	0.525 s	9.3
Hex	15632	196	241	0.091 s	1.339 s	6.8

The “type-checking time” includes type-checking, checking for deprecated APIs, and verifying function definitions. These overheads are comparable to previously existing ad hoc checks embedded within the Elixir compiler, which have since been replaced by our type system. Note that type-checking essentially consists of checking subtyping relations and that the complexity of subtype checking for arrow types is not greater than that for tuple types or map types (included in v1.18). Thus, although the v1.18 implementation of gradual set-theoretic types did not include arrow types, it already provided a precise assessment of the performance of the monomorphic system presented here, demonstrating that our implementation scales effectively, handling large codebases with minimal performance impact.

These results were confirmed by repeating the evaluation 8 months later with version 1.19, which adds function types, performs function type inference, includes protocol checking, and performs a more complete analysis of map types. We computed the same metrics but on the codebases as they had evolved (since in some cases with closed source code we were given access only to the latest version). Both the general compilation time and the type-checking time were significantly reduced due to improvements in both the general Elixir compiler and in the implementation of the type system, despite the additional type checks implemented in it.

Methodology and setup

We report wall-clock times for a clean compile of each project. For each configuration, we executed 5 consecutive clean builds and report the median. Caches were cleared between runs; network access was disabled. All measurements were taken on the same workstation (Darwin 24.6.0, OTP 28.1 unless noted, Elixir versions as indicated), with no competing workloads. The end-to-end "type-checking time" aggregates the static analyses performed during compilation (type checking, deprecation checks, verification of function definitions) as reported by the compiler.

Threats to validity

Project evolution between v1.18 and v1.19 introduces confounders (LoC, modules, and dependencies changed). We mitigate this by (i) reporting both type-checking and total compilation times, (ii) performing an ablation across compiler milestones (PT2) to attribute improvements to the data-structure fixes (lazy BDD equal-node formulas), and (iii) pinning OTP where noted. Results are stable across runs; absolute values may vary on different hardware.

The table below repeats the same metrics (LoC, files, modules, type-checking time, total compilation time, and the percentage share) for the same set of projects, recompiled with Elixir v1.19 using the same methodology. It updates the earlier v1.18-rc0 snapshot to reflect both compiler and type-system improvements on the codebases as they evolved, enabling a direct comparison with Table 12.2.

Codebase	LoC	Files	Modules	Type Checking Time	Total Compilation Time	%
Remote	>1,000,000	>10,000	24292	19.476 s	228.801 s	8.5
Livebook	61,170	256	258	0.102 s	3.051 s	3.3
Credo	28,493	251	314	0.027 s	1.095 s	2.4
Phoenix	22,497	74	110	0.029 s	0.461 s	6.3
Hex	15,476	196	207	0.065 s	1.232 s	5.3

The codebase for Remote is the only one that has grown significantly since the previous evaluation, with a 35% increase in the number of modules. The type-checking time for Remote has increased by 75% compared to the previous evaluation, and now it takes nearly 20 seconds to type-check 24 thousands modules. This, combined with the substantial improvement of the compiler (which reduced the total compilation time by 68% despite the increased number of modules), explains the much higher percentage of type-checking time relative to total compilation time. Remarkably, the new typing features introduced in v1.19 found a type error in the Remote codebase undetected by the previous versions. Specifically, the type-checker detected a call of the `String.Chars` protocol that may be called with a tuple argument in an error branch of some logger. Since the `String.Chars` protocol does not provide an implementation for tuples, this would result in a runtime failure if the error branch is ever executed. This error was undetected by tests because the calling of an error branch is extremely uncommon, which explains why it slipped through: classic needle in a haystack.

The initial implementation of our type system in Elixir 1.17, has been distributed since June 2024. Both versions 1.17 and 1.18 have already garnered positive feedback from developers (at the moment of writing we do not have any feedback on the 1.19 other than our own tests yet, since it is just a few days old). Particularly appreciated was the fact that the type system does not require any syntactic change to Elixir, and it was able to uncover previously hidden errors without introducing significant overhead and without requiring Elixir developers to explicitly add type annotations. For existing codebases with reasonable test coverage, most type system reports came from uncovering dead code—code which won't ever be executed—but also few actual type mismatch bugs. The type system has already found issues in major projects like Phoenix, Livebook, Postgrex, and Flame, particularly identifying unused function clauses and unreachable code paths. Some bugs had persisted undetected in production code for over two years. For instance, our type system found a critical bug in the Phoenix framework—which is used by over 14,600 public websites—where `String.to_atom/1` was being called with incompatible types, and since then it has been fixed (José Valim, 2024). In the Postgrex library, the type system identified 15 lines of dead code that were removed across multiple functions (Valim, 2024c); the guard analysis implemented in version 1.18 uncovered 2 different dead code paths in the Flame library used to provide modular scaling for CPU-intensive Elixir programs (Valim, 2024b); a type mismatch in the Live-view library allowed the removal of 6 lines of dead code (LiveView, 2024). Although these few examples may seem negligible, it is important to note that Elixir projects generally have good test coverage, and that these are bugs found in *production code* that has already been merged and passed through testing, code reviews, sometimes even Quality Assurance (QA) processes—code that has been running in production for months or even years.

Chapter 15
Conclusion

This PhD project started with a very pragmatic goal: to type Elixir. Our solution is based on set-theoretic types, and required not only additions to the theory—to account for multi-arity function types, tuple types, and the rich pattern-matching idioms of Elixir—but also novel techniques for gradual typing that preserve Elixir's runtime semantics. We insist there on the word *addition*: throughout our progress, the semantic subtyping framework has remained the same, and only the types have been extended to serve for new data-types; proving thus the robustness of the framework. This robustness is not merely theoretical: it has enabled us to build a practical type system that is now integrated into the official Elixir compiler and is being used by developers in production codebases.

Contributions. Our work extends the semantic subtyping framework along several dimensions:

- **Multi-arity function and tuple types:** We adapted the semantic subtyping framework to account for Elixir/Erlang's notion of function arity, introducing explicit arity in function type syntax and extending subtyping rules accordingly. This required non-trivial modifications to the set-theoretic interpretation of function spaces, reframing subtyping as a set-containment problem for multi-arity functions (as well as for open/closed tuple types).
- **Guard analysis and pattern matching:** We developed a sophisticated typing technique to analyze guards (boolean expressions in pattern matches and function definitions), interpreting guard conditions as type constraints that refine static types within each branch. This enables exhaustiveness and redundancy checking for pattern-match clauses, as well as precise type narrowing that captures Elixir's flow-sensitive programming idioms.
- **Dynamic type integration:** We incorporated the `dynamic()` type to represent untyped parts

of code, following gradual typing principles. This allows untyped Elixir code to interoperate safely with typed code through runtime checks, ensuring that gradually adding types does not break program behaviour (the gradual guarantee).

- **Strong function types:** Our most novel contribution came from a very practical observation: that very similar functions, in Elixir, had in fact different semantics in a gradual setting according to whether or not their arguments were guarded. Following this observation led us to developing the notion of *strong functions*, which act as type-level proxies that account for runtime type checks (guards or pattern matches) in the static function type. Strong arrows allow the type system to be sound (no false assurances about the absence of type errors) while also being precise, without needing to alter the compiled code. In the terminology of Garcia et al. (2021), this achieves *safe erasure* in gradual typing: we do not change program semantics to accommodate types, yet we retain strong type guarantees. This is another tool in the toolbox of gradual semantic subtyping, which we may not have yet fully exploited.

Implementation and Evaluation. We have strived to give a sufficient account of the techniques used to implement a set-theoretic type system. However, in such a complex work, there will always be a large part of specific design, choices and trade-offs. We chose to represent some of those: which data-structures should be used to represent types, what typing discipline works best, which type relations should be relied on, etc. Our implementation experience, documented in Part II of this thesis, demonstrates that the type system scales effectively: type-checking overhead remains under 10% even for codebases exceeding one million lines of code, as shown in Chapter 14. The modular design of our type engine, with its stable API (Chapter 8), has enabled incremental integration into the Elixir compiler, ensuring minimal disruption to existing development workflows.

The type system has been progressively integrated into Elixir since version 1.17, with function types, protocol checking, and comprehensive map analysis added in subsequent releases. As of version 1.19, the implementation covers all core language constructs and provides type inference for patterns and return types. The system currently produces warnings rather than hard errors, allowing developers to adopt types gradually without breaking existing code. This pragmatic approach has been essential for community acceptance. Future releases will introduce explicit type annotations on top of functions and polymorphism, which will start having a greater impact on the language as developers gain more control over type specifications and the type system becomes more expressive. For now, most of the time spent type-checking is spent on inferring function types (and checking their complex return types), and more annotations will save up some time in that regard.

Significance and Impact. The significance of this work extends beyond Elixir. Because Elixir's semantics are largely a superset of Erlang's, our type system effectively types Erlang as well, covering all Erlang idioms in addition to Elixir's extensions. The semantic subtyping framework we extended has proven robust enough to handle the diverse and dynamic idioms of a production language, demonstrating that set-theoretic types are not merely a theoretical curiosity but a practical foundation for gradual typing in dynamic languages.

Our work also contributes to the broader gradual typing literature by confronting its notions to an industrial setting, one in which we have to achieve sound gradual typing without modifying runtime semantics. The strong function technique we developed provides a way to leverage existing runtime checks for static type guarantees and integrate those into the type language, which may be applicable to other gradually typed languages.

Future Directions. While we have achieved our primary goal of typing Elixir, numerous avenues remain for future work. The module system represents a high priority: Elixir's module system, with its hot code swapping capabilities and dynamic loading, presents unique challenges for static typing that we have not yet addressed. Other directions include typing meta-programming, improving error messages, integrating with editor tooling for better developer experience, and developing techniques for easing the transition of large codebases (such as automated suggestion of type annotations or coding patterns to gradually introduce types).

Performance optimizations remain an ongoing concern. While our current implementation achieves acceptable overhead, further improvements could be made through more sophisticated caching strategies, parallelization of type checking, or refinement of our data structures for specific use cases. The exploration of alternative representations (as documented in Chapter 12) suggests that different trade-offs may be beneficial for different code patterns.

Concluding Remarks. In conclusion, this thesis demonstrates that semantic subtyping provides a robust foundation for gradual typing in dynamic languages, and that Elixir can be successfully typed without sacrificing its dynamic spirit or requiring changes to its runtime semantics. The combination of theoretical rigor and practical implementation has produced a type system that is both mathematically sound and pragmatically useful. As the type system continues to evolve within the Elixir compiler, we hope that this work will inspire further research into set-theoretic types and gradual typing, and that it will contribute to making Elixir codebases more reliable and maintainable for the growing community of developers who rely on this language for building robust, concurrent systems.

Bibliography

Akers, SB (1978). “Binary Decision Diagrams”. In: *IEEE Transactions on Computers* 27.6, pp. 509–516 (cit. on p. 226).

Ancona, D, V Bono, M Bravetti, G Castagna, J Campos, PM Deniélou, S Gay, N Gesbert, E Giachino, R Hu, EB Johnsen, F Martins, V Mascardi, F Montesi, N Ng, R Neykova, L Padovani, V Vasconcelos, and N Yoshida (2016). “Behavioral Types in Programming Languages”. In: *Foundations and Trends in Programming Languages* 3 (2-3), pp. 95–230. DOI: 10.1561/2500000031 (cit. on p. 196).

Armstrong, J (2003). “Making Reliable Distributed Systems in the Presence of Software Errors”. PhD thesis. Royal Institute of Technology (KTH), Stockholm. URL: http://erlang.org/download/armstrong_thesis_2003.pdf (cit. on p. 22).

Ballerina (2019). *Ballerina*. Accessed on Feb 28, 2024. URL: https://ballerina.io/ (cit. on p. 192).

Stenman, E (2024). *The Erlang Runtime System*. https://blog.stenmans.org/theBeamBook/. Accessed on February 28, 2024. URL: https://blog.stenmans.org/theBeamBook/ (cit. on p. 21).

Benzaken, V, G Castagna, and A Frisch (2003). “CDuce: an XML-Centric General-Purpose Language”. In: *ICFP ’03, 8th ACM International Conference on Functional Programming*. Uppsala, Sweden: ACM Press, pp. 51–63. DOI: 10.1145/944705.944711 (cit. on pp. 25, 206).

Berger, F, A Schimpf, A Bieniusa, and S Wehr (2024). “Same Same but Different: A Comparative Analysis of Static Type Checkers in Erlang”. In: *Proceedings of the 23rd ACM SIGPLAN International Workshop on Erlang*. Erlang 2024. Milan, Italy: Association for Computing Machinery, pp. 2–12. DOI: 10.1145/3677995.3678189. URL: https://doi.org/10.1145/3677995.3678189 (cit. on p. 190).

Bezanson, J, J Chen, B Chung, S Karpinski, VB Shah, J Vitek, and L Zoubritzky (2018). “Julia: Dynamism and Performance Reconciled by Design”. In: *Proc. ACM Program. Lang.* 2.OOPSLA. DOI: 10.1145/3276490. URL: https://doi.org/10.1145/3276490 (cit. on p. 192).

Bierman, G, M Abadi, and M Torgersen (2014). “Understanding TypeScript”. In: *ECOOP 2014 – Object-Oriented Programming*. Ed. by R Jones. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 257–281 (cit. on p. 193).

Bryant, RE (1986). “Graph-Based Algorithms for Boolean Function Manipulation”. In: *IEEE Transactions on Computers* 35.8, pp. 677–691 (cit. on p. 226).

Caramel (2022). https://github.com/leostera/caramel (cit. on pp. 24, 191).

Castagna, G (2016). *Covariance and contravariance: a fresh look at an old issue.* Tech. rep. Technical report, CNRS (cit. on pp. 82, 225).

Castagna, G (2020). "Covariance and Controvariance: a fresh look at an old issue (a primer in advanced type systems for learning functional programmers)". In: *Logical Methods in Computer Science* Volume 16, Issue 1. DOI: 10.23638/LMCS-16(1:15)2020. URL: https://lmcs.episciences.org/6098 (cit. on p. 100).

Castagna, G (2023a). "Programming with Union, Intersection, and Negation Types". In: *Essays Dedicated to Philip Wadler*. Also available as arXiv:2111.03354. Springer. DOI: 10.1007/978-3-031-34518-0_12. URL: https://link.springer.com/chapter/10.1007/978-3-031-34518-0_12 (cit. on pp. 24, 53, 100).

Castagna, G (2023b). "Typing Records, Maps, and Structs". In: *Proc. ACM Program. Lang.* 7.ICFP. DOI: 10.1145/3607838 (cit. on pp. 41, 99, 113, 191–193, 239, 240).

Castagna, G, G Duboc, and J Valim (2024a). "The Design Principles of the Elixir Type System". In: *The Art, Science, and Engineering of Programming* 8.2. DOI: 10.22152/programming-journal.org/2024/8/4 (cit. on pp. 53, 122, 184).

Castagna, G and A Frisch (2005). "A gentle introduction to semantic subtyping". In: *Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pp. 198–199. DOI: 10.1145/1069774.1069793 (cit. on pp. 25, 100).

Castagna, G, V Lanvin, T Petrucciani, and JG Siek (2019). "Gradual typing: a new perspective". In: *Proceedings of the ACM on Programming Languages* 3.POPL, pp. 1–32. DOI: 10.1145/3290329 (cit. on pp. 42, 44, 58, 63, 64, 190–192, 206).

Castagna, G, M Laurent, and K Nguyen (2024b). "Polymorphic Type Inference for Dynamic Languages". In: *Proc. ACM Program. Lang.* 8.POPL. DOI: 10.1145/3632882 (cit. on p. 193).

Castagna, G, K Nguyen, Z Xu, and P Abate (2015). "Polymorphic functions with set-theoretic types. Part 2: local type inference and type reconstruction". In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* POPL '15. New York, NY, USA: Association for Computing Machinery, pp. 289–302. DOI: 10.1145/2676726.2676991 (cit. on pp. 30, 53, 63, 64, 191, 192, 206).

Castagna, G, K Nguyen, Z Xu, H Im, S Lenglet, and L Padovani (2014). "Polymorphic Functions with Set-Theoretic Types. Part 1: Syntax, Semantics, and Evaluation". In: *Proceedings of the 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* POPL '14. New York, NY, USA: Association for Computing Machinery, pp. 5–17. DOI: 10.1145/2676726.2676991 (cit. on pp. 30, 53, 63, 64, 191, 192, 206).

Castagna, G, T Petrucciani, and K Nguyen (2016). "Set-Theoretic Types for Polymorphic Variants". In: *ICFP '16, 21st ACM SIGPLAN International Conference on Functional Programming*, pp. 378–391. DOI: 10.1145/2951913.2951928 (cit. on p. 193).

Castagna, G and L Peyrot (2025a). "Polymorphic Records for Dynamic Languages". In: *Proc. ACM Program. Lang.* 9.OOPSLA1. DOI: 10.1145/3720497. URL: https://doi.org/10.1145/3720497 (cit. on pp. 64, 99).

Castagna, G and L Peyrot (2025b). "Polymorphic Records for Dynamic Languages". In: *Proceedings of the ACM on Programming Languages* 9.OOPSLA1, pp. 1464–1491 (cit. on pp. 191, 192).

Castagna, G and Z Xu (2011). "Set-theoretic foundation of parametric polymorphism and subtyping". In: *SIGPLAN Not.* 46.9, pp. 94–106. DOI: `10.1145/2034574.2034788`. URL: `https://doi.org/10.1145/2034574.2034788` (cit. on pp. 63, 206).

CDuce (2003). Accessed on July 14, 2025. URL: `https://www.cduce.org/` (cit. on pp. 25, 192).

Cesarini, F and S Thompson (2009). *Erlang Programming: A Concurrent Approach to Software Development*. O'Reilly Media (cit. on p. 22).

Chaudhuri, A, P Vekris, S Goldman, M Roch, and G Levi (2017). "Fast and precise type checking for JavaScript". In: *Proceedings of the ACM on Programming Languages* 1.OOPSLA, 48:1–48:30. DOI: `10.1145/3133872` (cit. on p. 193).

Chung, B, FZ Nardelli, and J Vitek (2019). "Julia's Efficient Algorithm for Subtyping Unions and Covariant Tuples (Pearl)". In: *33rd European Conference on Object-Oriented Programming, ECOOP 2019, July 15-19, 2019, London, United Kingdom*. Vol. 134. LIPIcs, 24:1–24:15. DOI: `10.4230/LIPIcs.ECOOP.2019.24`. URL: `https://doi.org/10.4230/LIPIcs.ECOOP.2019.24` (cit. on p. 197).

contributors, e (2025). *Module types in Elixir core library*. Accessed: 2025-10-07. URL: `https://github.com/elixir-lang/elixir/tree/main/lib/elixir/lib/module/types` (cit. on p. 82).

de'Liguoro, U and L Padovani (2018). "Mailbox Types for Unordered Interactions". In: *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*. Vol. 109. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 15:1–15:28. DOI: `10.4230/LIPIcs.ECOOP.2018.15`. URL: `http://drops.dagstuhl.de/opus/volltexte/2018/9220` (cit. on p. 196).

Elixir (2012). `https://elixir-lang.org/` (cit. on p. 21).

Elixir Core Team (2024). *Elixir documentation: Gradual set-theoretic types*. `https://hexdocs.pm/elixir/gradual-set-theoretic-types.html`. URL: `https://hexdocs.pm/elixir/gradual-set-theoretic-types.html` (cit. on p. 271).

eqWAlizer (2022). `https://github.com/WhatsApp/eqwalizer` (cit. on pp. 189, 192).

Fowler, S, DP Attard, F Sowul, SJ Gay, and P Trinder (2023). "Special Delivery: Programming with Mailbox Types". In: *Proc. ACM Program. Lang.* 7.ICFP. DOI: `10.1145/3607832`. URL: `https://doi.org/10.1145/3607832` (cit. on p. 196).

Frisch, A (2004). "Théorie, conception et réalisation d'un langage de programmation adapté à XML". PhD thesis. PhD thesis, Université Paris 7 (cit. on pp. 9, 63, 66, 75, 82, 83, 99, 103, 109, 113, 132, 140, 191, 205, 225–227, 230, 231).

Frisch, A, G Castagna, and V Benzaken (2008). "Semantic Subtyping: dealing set-theoretically with function, union, intersection, and negation types". In: *Journal of the ACM* 55.4, pp. 1–64. DOI: `10.1145/1391289.1391293` (cit. on pp. 25, 33, 94, 100).

Gleam (2019). `https://gleam.run/` (cit. on pp. 24, 191).

Greenman, B, C Dimoulas, and M Felleisen (2023). “Typed-Untyped Interactions: A Comparative Analysis”. en. In: *ACM Transactions on Programming Languages and Systems* 45.1, pp. 1–54. DOI: 10.1145/3579833 (cit. on pp. 7, 147, 194, 195).

Hamler (2021). https://www.hamler-lang.org/ (cit. on pp. 24, 190).

HHVM Team (2014). *Hack Documentation*. https://docs.hhvm.com/hack/. (Visited on 05/14/2019) (cit. on p. 193).

Hindley, JR (1969). “The Principal Type-Scheme of an Object in Combinatory Logic”. In: *Transactions of the American Mathematical Society* 146, pp. 29–60 (cit. on p. 191).

Jeffrey, A (2022). *Semantic Subtyping in Lua*u. Blog post. Accessed on May 6th 2023. URL: https://blog.roblox.com/2022/11/semantic-subtyping-luau (cit. on pp. 192, 205).

José Valim (2024). *Bug fix commit: “Remove dead code found by the type system"*. URL: https : / / github . com / phoenixframework / phoenix / commit / 34d0ffef6aebcb5d4f210978aabca53b0e57f1ae (cit. on p. 274).

Kent, AM (2019). “Advanced Logical Type Systems for Untyped Languages”. PhD thesis. Indiana University (cit. on pp. 82, 205).

Lanvin, V (2021). “A semantic foundation for gradual set-theoretic types”. PhD thesis. Université Paris Cité (cit. on pp. 43, 44, 63–66, 70, 75, 79–82, 86, 88, 149, 190, 192, 263–266, 293).

Lazarek, L, B Greenman, M Felleisen, and C Dimoulas (2021). “How to Evaluate Blame for Gradual Types”. In: *Proc. ACM Program. Lang.* 5.ICFP. DOI: 10.1145/3473573. URL: https://doi.org/10.1145/3473573 (cit. on pp. 26, 276).

Lehtosalo, J, Gv Rossum, I Levkivskyi, MJ Sullivan, D Fisher, G Price, M Lee, N Seyfer, R Barton, S Ilinskiy, et al. (2017). “Mypy-optional static typing for python”. In: *URL: http://mypy-lang.org/[cited 2021-11-30]* (cit. on p. 195).

Lindahl, T and K Sagonas (2006). “Practical type inference based on success typings”. In: *ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming*. New York, NY, USA: Association for Computing Machinery, pp. 167–178 (cit. on pp. 23, 189, 190).

LiveView (2024). *Bug fix commit: “Fix bug found by typesystem"*. URL: https : / / github . com / phoenixframework / phoenix _ live _ view / commit / 6c6e2aaf6a01957cc6bb8a27d2513bff273e8ca2 (cit. on p. 274).

*Lua*u (2022). https://luau-lang.org/ (cit. on pp. 64, 192).

Luau Team (2023). *Lua*u *.594 Release*. GitHub release. Released on September the 8th 2023. URL: https://github.com/luau-lang/luau/releases/tag/0.594 (cit. on p. 192).

Marlow, S and P Wadler (1997). “A practical subtyping system for Erlang”. In: *ACM SIGPLAN Notices* 32.8, pp. 136–149 (cit. on pp. 23, 190).

Microsoft (2025a). *A Note on Soundness — TypeScript Handbook*. Accessed: 2025-09-23. URL: https : / / www . typescriptlang . org / docs / handbook / type - compatibility . html#a-note-on-soundness (cit. on p. 193).

Microsoft (2025b). *Narrowing — TypeScript Handbook*. https://www.typescriptlang.org/docs/handbook/2/narrowing.html. Accessed: 2025-09-23 (cit. on p. 193).

Milner, R (1978). “A theory of type polymorphism in programming”. In: *Journal of Computer and System Sciences* 17.3, pp. 348–375. DOI: 10.1016/0022-0000(78)90014-4 (cit. on p. 191).

Nguyen, K (2008). “Langage de combinateurs pour XML: conception, typage, implantation”. PhD thesis. Paris 11 (cit. on p. 253).

Pure Erlang (2023). https://github.com/purerl/purerl (cit. on p. 190).

PureScript (2013). https://www.purescript.org/ (cit. on p. 190).

Python Software Foundation (2025a). *Type System Concepts*. URL: https://typing.python.org/en/latest/spec/concepts.html#type-system-concepts (visited on 06/05/2025) (cit. on p. 192).

Python Software Foundation (2025b). *Unreachable Code and Exhaustiveness Checking – typing documentation*. https://typing.python.org/en/latest/guides/unreachable.html. Accessed: 2025-10-10 (cit. on p. 267).

Python Software Foundation Team (2025). *Overloads — Specification for the Python Type System*. https://typing.python.org/en/latest/spec/overload.html. Accessed: 2025-09-23 (cit. on p. 194).

Rajendrakumar, NV and A Bieniusa (2021). “Bidirectional typing for Erlang”. In: *Proceedings of the 20th ACM SIGPLAN International Workshop on Erlang*, pp. 54–63 (cit. on p. 190).

Rastogi, A, A Chaudhuri, and B Hosmer (2012). “The ins and outs of gradual type inference”. In: *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’12. Philadelphia, PA, USA: Association for Computing Machinery, pp. 481–494. DOI: 10.1145/2103656.2103714 (cit. on p. 195).

Rastogi, A, N Swamy, C Fournet, G Bierman, and P Vekris (2015). “Safe & efficient gradual typing for TypeScript”. In: *POPL ’15*. ACM, pp. 167–180 (cit. on p. 195).

Rémy, D (1989). “Type Checking Records and Variants in a Natural Extension of ML”. In: *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’89. Austin, Texas, USA: Association for Computing Machinery, pp. 77–88. DOI: 10.1145/75277.75284. URL: https://doi.org/10.1145/75277.75284 (cit. on p. 191).

Reynolds, JC (1984). “Polymorphism is not set-theoretic”. In: *International Symposium on Semantics of Data Types*. Springer, pp. 145–156 (cit. on p. 206).

Rossberg, A (2018). “1ML - Core and modules united”. In: *J. Funct. Program.* 28, e22. DOI: 10.1017/S0956796818000205. URL: https://doi.org/10.1017/S0956796818000205 (cit. on p. 197).

Rossberg, A, C Russo, and D Dreyer (2014). “F-ing modules”. In: *Journal of functional programming* 24.5, pp. 529–607 (cit. on p. 191).

Russo, CV (2000). “First-Class Structures for Standard ML”. In: *Proceedings of the 9th European Symposium on Programming Languages and Systems*. ESOP ’00. Berlin, Heidelberg: Springer-Verlag, pp. 336–350 (cit. on p. 197).

Schimpf, A, S Wehr, and A Bieniusa (2023a). “Set-theoretic Types for Erlang”. In: *Proc. of IFL 2022*. Copenhagen, Denmark: ACM. DOI: https://doi.org/10.1145/3587216.3587220 (cit. on pp. 7, 189).

Schimpf, A, S Wehr, and A Bieniusa (2023b). *Set-theoretic Types for Erlang.* DOI: 10.48550/arXiv.2302.12783. URL: https://arxiv.org/abs/2302.12783 (cit. on p. 191).

Sesterl (2021). https://github.com/gfngfn/Sesterl (cit. on p. 191).

Siek, JG and W Taha (2006). "Gradual typing for functional languages". In: *Scheme and Functional Programming Workshop.* Vol. 6, pp. 81–92 (cit. on pp. 42–44).

Svenningsson, J (2020). *Gradualizer.* https://github.com/josefs/Gradualizer (cit. on pp. 189, 190).

Team, H (2025). *Expressions and Operators: Type Assertions – Hack.* https://docs.hhvm.com/hack/expressions-and-operators/type-assertions. Accessed: 2025-09-23 (cit. on p. 193).

Tobin-Hochstadt, S and M Felleisen (2008). "The design and implementation of Typed Scheme". In: *POPL '08.* ACM. New York, NY, USA: Association for Computing Machinery, pp. 395–406 (cit. on p. 195).

Astral Team (n.d.). *Ty: a next-generation type-checker for Python.* URL: https://github.com/astral-sh/ty (cit. on pp. 192, 267).

TypeScript Team (2025). *Literal Types — TypeScript Handbook.* Accessed: 2025-10-10. URL: https://www.typescriptlang.org/docs/handbook/2/everyday-types.html#literal-types (cit. on p. 66).

Typespec (2023). https://www.erlang.org/doc/reference_manual/typespec.html. (Visited on 09/22/2023) (cit. on p. 26).

Valim, J (2024a). *"Elixir is, officially, a gradually typed language."*. Accessed: 2025-10-07. URL: https://x.com/josevalim/status/1744395345872683471 (cit. on p. 271).

Valim, J (2024b). *Bug fix commit: "Remove dead code found by Elixir v1.18".* URL: https://github.com/phoenixframework/flame/commit/0c0c2875e42952d2691cbdb7928fc32f4715e746 (cit. on p. 274).

Valim, J (2024c). *Bug fix commit: "Remove dead code found by typesystem".* URL: https://github.com/elixir-ecto/postgrex/commit/3308f277f455ec64f2d0d7be6263f77f295b1325 (cit. on p. 274).

Valliappan, N and J Hughes (2018). "Typing the wild in Erlang". In: *Proceedings of the 17th ACM SIGPLAN International Workshop on Erlang*, pp. 49–60 (cit. on pp. 24, 190).

Vitousek, MM, C Swords, and JG Siek (2017). "Big types in little runtime: open-world soundness and collaborative blame for gradual type systems". In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages.* POPL '17. New York, NY, USA: Association for Computing Machinery, pp. 762–774. DOI: 10.1145/3009837.3009849. URL: https://dl.acm.org/doi/10.1145/3009837.3009849 (cit. on p. 195).

Wand, M (1989). "Type inference for record concatenation and multiple inheritance". In: *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, pp. 92–97. DOI: 10.1109/LICS.1989.39162 (cit. on p. 191).

Yildirim, E, A Schimpf, S Wehr, and A Bieniusa (2025). *Semantic Subtyping for Maps in Erlang.* arXiv: 2508.00482 [cs.PL]. URL: https://arxiv.org/abs/2508.00482 (cit. on p. 192).

Zappa Nardelli, F, J Belyakova, A Pelenitsyn, B Chung, J Bezanson, and J Vitek (2018). “Julia Subtyping: A Rational Reconstruction”. In: *Proc. ACM Program. Lang.* 2.OOPSLA. DOI: 10.1145/3276483. URL: https://doi.org/10.1145/3276483 (cit. on p. 197).

APPENDIX A

ELIXIR AT A GLANCE

Elixir is a dynamic, **functional** programming language built on the Erlang VM, designed for building **scalable and maintainable** applications[1]. Its syntax is deliberately **approachable**, drawing inspiration from Ruby[2], yet it embraces the **concurrency** and fault-tolerance of Erlang. Here is a self-contained overview of Elixir's syntax and core concepts.

A.1 Approachable Syntax & Metaprogramming

Elixir uses a keyword-based syntax, where **blocks** of code are delimited by `do ... end` (instead of braces). Definitions use keywords `defmodule`, `def`, `def`. Modules are named in `CamelCase`, functions and variables in `snake_case`. **Function definitions** support inline clauses with `do:` for one-liners. The **return value** of a function is the last expression evaluated (no explicit `return`). Comments use `#`. Elixir also introduces a **pipeline operator** (`|>`), which threads an expression's result as the first argument of the next function call, enabling a linear flow of transformations. By convention, predicate functions often end with `?` and bang-functions (those that raise on errors) with `!`.

```elixir
# Defining a module with public and private functions
defmodule MathUtils do
  @moduledoc "Math utilities module (example of a module docstring)"
  @my_const 10   # module attribute (constant)

  # Public function with multiline body:
```

[1] https://elixir-lang.org/

[2] https://underjord.io/unpacking-elixir-syntax.html

```
  def add(a, b) do
    new_val = a + b
    new_val + @my_const   # implicit return of last expression
  end

  # Function with guard clause (executed only if guard condition true)
  def zero?(x) when x == 0, do: true
  def zero?(_x), do: false   # underscore ignores argument

  # Private function with default argument and one-line definition:
  defp increment(x, step \\ 1), do: x + step
end
```

Elixir features **metaprogramming** through macros which are inspired by Lisp, operating on the **Abstract Syntax Tree (AST)** of the code at compile time. Thus, the language can easily be extended with new constructs. At their core, macros are special functions that **return quoted AST nodes** to be inserted into the program during compilation[3]. Macros use `quote` and `unquote` to construct and inject AST fragments, and enable **Domain-Specific Languages (DSLs)** in Elixir's ecosystem. For instance, the Phoenix web framework's router and the Ecto library's schema definitions are implemented as macros that inject functions and logic, giving those APIs a declarative feel. To illustrate, here is macro `unless` (as it exists in Elixir's standard library):

```
defmodule MyMacros do
  defmacro unless(condition, do: block) do
    quote do
      if not unquote(condition) do  # inject the negated condition
        unquote(block)       # inject the block of code to execute
      end
    end
  end
end

require MyMacros
# Using the custom unless macro (executes the block as 2 > 5 is false)
MyMacros.unless 2 > 5, do: IO.puts("2 is not greater than 5")
```

Here `defmacro` defines a macro that generates an `if` expression at compile time.

A.2 Function Programming Paradigm: immutability and concurrency

Elixir is a **purely functional** language: it emphasizes **immutability** and pure functions. All data structures in Elixir are immutable : once created, they cannot be modified in place. Instead, functions create new data from old data. For example, updating a list or map produces a new list or map, leaving the original unchanged.

[3] https://elixirschool.com/en/lessons/advanced/metaprogramming

All variables are effectively single-assignment; what looks like reassigning a variable actually binds a new value to a new variable (the old value persists if another reference exists). .

Concurrency in Elixir is based on the **Actor model** (from Erlang). The runtime (the BEAM VM) supports millions of lightweight processes (green threads) that communicate via message passing and share no mutable state. With immutability, **multiple processes can operate on the same data without fear of unexpected modifications**, avoiding the need for locks on shared data . Elixir processes are isolated and fail independently; if one crashes, it does not take down others. Instead, supervisor processes can restart them (a philosophy of "let it crash" for fault tolerance). This makes building **highly concurrent, fault-tolerant systems** straightforward. Spawning a new process and sending a message to it looks like:

```
parent = self()   # current process PID
spawn(fn ->
  send(parent, {:greeting, "Hello from a new process"})
end)

# The parent process waits for a message:
receive do
  {:greeting, msg} -> IO.puts(msg)
end
```

Here `spawn(fn -> ... end)` starts a new process running the given anonymous function. We asynchronously `send` a message (an immutable tuple) to the parent, which `receive`s and pattern-matches on the message. Because processes do not share memory and communicate by copying data, common race conditions (related to shared mutable state) are eliminated by design. This concurrency model, combined with immutability, allows Elixir to **scale** vertically (many cores) and horizontally (distributed nodes) with relative ease. Under the hood, the BEAM schedules processes across cores, and it is possible to have millions of processes running concurrently.

A.3 Pattern Matching

Pattern matching allows to concisely destructure and match on the shape of data. In Elixir, the single **match operator** `=` means "match the pattern on the left with the value on the right." For example, writing `{a, b} = {:ok, 42}` will succeed if the right side is a 2-tuple, binding `a = :ok` and `b = 42`; but if the shapes do not align, a runtime `MatchError` is thrown. Patterns can include literal values (which must equal for the match to succeed), and the wildcard _ to ignore parts of data. Pattern matching is used **everywhere**: in variable binding, in `case` expressions, and even in **function definitions**. Elixir integrates pattern matching directly into function heads: it is possible to **define multiple function clauses, each with different parameter patterns**, and Elixir will dispatch to the first one that matches the arguments. This leads to a declarative style where the function's clauses act as an implicit conditional based on data shape. For example:

```elixir
# Pattern matching in a case expression:
case File.read("example.txt") do
  {:ok, contents} -> IO.puts("File size: #{byte_size(contents)} bytes")
  {:error, :enoent} -> IO.puts("File not found")
  {:error, reason} -> IO.puts("File error: #{reason}")
end

# Pattern matching in function clauses (recursive list sum):
defmodule Math do
  def sum_list([]), do: 0
  def sum_list([head | tail]), do: head + sum_list(tail)
end
IO.puts Math.sum_list([1,2,3])   # Output: 6
```

In the `case` above, each branch matches a different tuple shape from `File.read/1` (which returns either `{:ok, binary}` or `{:error, reason}`). The pattern-matched variables (`contents`, `reason`) are bound for use in the corresponding branch. In the function `sum_list/1`, we provide two clauses: one for the empty list (base case) and one that matches a non-empty list, simultaneously unpacking it into `head` and `tail`. This recursive definition is succinct and clear due to pattern matching on the list structure (`[head | tail]`). Elixir also supports **guards**, which are boolean expressions after the `when` keyword, to further refine pattern matches (e.g., numeric comparisons, type checks).

A.4 Protocols (Ad Hoc Polymorphism)

Elixir's **protocols** provide a mechanism for polymorphism (similar to type classes in Haskell) by allowing behaviour to be defined based on a data type *without* modifying that data type's original code. A protocol is a **contract** that defines a set of functions that can have different implementations depending on the type of the argument. At runtime, Elixir will **dispatch** a protocol function call to the correct implementation based on the type of the value provided as its first argument[4]. This enables *ad hoc* polymorphism. For example, Elixir has a built-in protocol `String.Chars` which requires a `to_string/1` function to be defined for each type implementing the `String.Chars` protocol. It is implemented for types like integers, floats, so that calling `to_string(value)` produces a sensible string representation[5] . Calling `to_string` on a type that has no implementation will raise a `Protocol.UndefinedError`. Developers can both **define their own protocols** and **provide implementations** for them on any data type (even built-in ones or custom structs). Defining a protocol is done with `defprotocol`, and implementations with `defimpl`. Here is an example protocol and a couple of implementations:

```elixir
# Define a protocol that provides a pretty-print string conversion
```

[4] https://hexdocs.pm/elixir/protocols.html
[5] https://elixirschool.com/en/lessons/advanced/protocols

```elixir
defprotocol Pretty do
  @doc "Converts the given value to a pretty string"
  def to_string(val)
end

# Implement Pretty for integer and atom types
defimpl Pretty, for: Integer do
  def to_string(val), do: "Int(#{val})"
end

defimpl Pretty, for: Atom do
  def to_string(val), do: ":" <> Atom.to_string(val)
end

IO.puts Pretty.to_string(5)    # Prints "Int(5)"
IO.puts Pretty.to_string(:ok)  # Prints ":ok"
```

This form of polymorphism is **open** (anyone can add a new implementation for a new type at any time) and **dynamic** (the selection happens at runtime based on the value's type).

A.5 Ecosystem and Tooling

Part of Elixir's appeal is its **robust tooling and thriving ecosystem**. Some of the main tools and libraries that are part of the ecosystem are:

- **Mix** — Elixir's built-in **build and project management tool**. Mix handles project scaffolding, compilation, running tests, managing dependencies, and more[6]. For example, `mix new my_app` creates a new project with a ready-to-go structure, `mix compile` compiles it, `mix test` runs tests, etc. Mix also provides an easy way to define custom tasks and is the interface to running Elixir code (via `mix run`) in projects. It is comparable to tools like Maven (Java) or Bundler/Rake (Ruby) but tailored to Elixir's needs.
- **Hex** — the **package manager** for Elixir. Hex is a central repository (hex.pm) where the community publishes libraries (packages are called "deps" in Elixir). Mix uses Hex to fetch and manage dependencies; declaring a dependency in your `mix.exs` file and running `mix deps.get` will pull it from Hex.
- **Phoenix** — the main **web development framework** for Elixir. Phoenix (first released in 2014) uses Elixir's strengths like concurrency and functional syntax to build scalable real-time web applications. It follows the Model-View-Controller pattern[7] and includes features like **LiveView**, to build interactive real-time UIs without writing JavaScript. With the BEAM VM's ability to handle massive concurrency, Phoenix can serve **large numbers of simultaneous connections**, and it is a centerpiece of the Elixir ecosystem.

[6] https://hexdocs.pm/mix/Mix.html
[7] https://en.wikipedia.org/wiki/Phoenix_(web_framework)

- **Ecto** — the main **database library** for Elixir, providing a composable query DSL and mapping between databases and Elixir data structures, similar to an ORM, but in a functional style. Ecto is often used with Phoenix for handling database interactions (migrations, validations, etc.), and it leverages Elixir's syntax (using macros to define schemas and relationships in a declarative way).
- **ExUnit** — Elixir's built-in **unit testing framework**. It provides everything needed for testing, including a test runner (`mix test`), test case definitions via macros (`ExUnit.Case`), and features like concurrent test execution (`async: true`).
- **Other Tools & Libraries:** Nerves, for example, is a framework for building embedded IoT systems with Elixir, leveraging the fault-tolerance of the BEAM on hardware. The community has produced libraries for many need: recently, for machine learning (e.g. Axon) and the coding notebook project Livebook, and Credo for static code analysis and linting.

Appendix

B Technical details

B.1 Set-theoretic interpretation of gradual types

The gradual interpretation domain $\mathcal{D}_g$, defined by Lanvin (2021). We replaced the original blame labels (blame ℓ) by named errors ω_p.

Definition B.1.1 (Interpretation domain $\mathcal{D}_g$)**.** *The* interpretation domain $\mathcal{D}_g$ *is the set of finite terms d produced inductively by the following grammar*

$$\begin{aligned}
d &::= c^g \mid (d,d)^g \mid \{(S,\delta),\dots,(S,\delta)\}^g \\
S &::= \{\mathfrak{d},\dots,\mathfrak{d}\} && S \text{ non empty} \\
\mathfrak{d} &::= d \mid \mho \\
\delta &::= d \mid \Omega \mid \omega_p \\
g &::= \,!\mid ?
\end{aligned}$$

where c ranges over the set $\mathcal{C}$ of constants, p ranges over runtime error names, and where Ω and $\mho$ are such that $\Omega \notin \mathcal{D}_g$ and $\mho \notin \mathcal{D}_g$. We also write $\mathcal{D}_g^\Omega = \mathcal{D}_g \cup \{\Omega\} \cup \mathcal{B}$ and $\mathcal{D}_g^\mho = \mathcal{P}_{fin}(\mathcal{D}_g \cup \{\mho\}) \setminus \{\varnothing\}$. We will also commonly use R to range over the set $\mathcal{P}_{fin}(\mathcal{D}_g^\mho \times \mathcal{D}_g^\Omega)$ so that R^g ranges over the finite relations in $\mathcal{D}_g$ with tag g. We may also use ξ to range over both $\mathfrak{d}$ and δ, that is, over elements of $\mathcal{D}_g^\Omega \cup \{\mho\}$.

Definition B.1.2 (Set-theoretic interpretation of types in $\mathcal{D}_g$)**.** *We define a binary predicate $\xi : \tau$ where $\xi \in \mathcal{D}_g^\Omega \cup \{\mho\}$ and $\tau \in \mathcal{T}_{gradual}$, by induction on the pair (ξ,τ) ordered lexicographically.*

The predicate is defined as follows:

$$\begin{aligned}
\omega_p : \tau &= \mathit{true} \\
d : ? &= (tag(d) = ?) \\
c^g : b &= c \in \mathscr{C}(b) \\
(d_1, d_2)^g : \tau_1 \times \tau_2 &= (d_1^g : \tau_1)\ \mathit{and}\ (d_2^g : \tau_2) \\
\{(S_1, \delta_1), \ldots, (S_n, \delta_n)\}^g : \tau_1 \to \tau_2 &= \forall i \in \{1..n\}.\ \mathit{if}\ (\exists \mathfrak{d} \in S_i.\ \mathfrak{d}^{\overline{g}} : \tau_1)\ \mathit{then}\ (\delta_i^g : \tau_2) \\
d : \tau_1 \vee \tau_2 &= (d : \tau_1)\ \mathit{or}\ (d : \tau_2) \\
d : \neg\tau &= (tag(d) = g)\ \mathit{and\ not}\ (d^{\overline{g}} : \tau) \\
\mho : \tau &= \mathit{true} \\
\delta : \tau &= \mathit{false} \qquad\qquad \mathit{otherwise}
\end{aligned}$$

Where the involutory operation $\bar{.} : \{!, ?\} \to \{!, ?\}$ *reverses a tag:* $\bar{!} = ?$ *and* $\bar{?} = !$. *We define the* tagged set-theoretic interpretation *of gradual types* $[\![.]\!]_g : \mathscr{T}_{gradual} \to \mathscr{P}(\mathscr{D}_g)$ *as* $[\![\tau]\!]_g = \{d \in \mathscr{D}_g \mid d : \tau\}$. *Finally, we define the* full set-theoretic interpretation $[\![\tau]\!]$ *as the union of both tagged interpretations:*

$$[\![\tau]\!] = \{d \in \mathscr{D} \mid d^! : \tau\} \cup \{d \in \mathscr{D} \mid d^? : \tau\}$$

The set-theoretic interpretation of gradual types is quite similar to the interpretation of static types (Definition 3.1.3). The major difference is the propagation of tags, which amounts to reversing the tag on the element being checked according to the variance of the type at hand.