Université Paris Cité

INSTITUT DE RECHERCHE EN INFORMATIQUE FONDAMENTALE

LMF Laboratoire Méthodes Formelles

UNIVERSITÉ PARIS CITÉ

ÉCOLE DOCTORALE 386
SCIENCES MATHÉMATIQUES DE PARIS CENTRE

Polymorphic type inference for dynamic languages

Reconstructing types for systems combining parametric, ad hoc, and subtyping polymorphism

---

Thèse de doctorat par

Mickaël LAURENT

Spécialité : Informatique

Thèse dirigée par Giuseppe Castagna
et co-encadrée par Kim Nguyen

Préparée à l'IRIF (UMR 8243) et au LMF (UMR 9021)

Présentée et soutenue publiquement le 21 Juin 2024

Directeur :	Giuseppe Castagna	-	Directeur de recherche (IRIF, Paris)
Co-encadrant :	Kim Nguyen	-	Maître de conférences (LMF, Gif-sur-Yvette)
Président :	Didier Remy	-	Directeur de recherche (INRIA Paris)
Rapporteurs :	Alan Schmitt	-	Directeur de recherche (INRIA Rennes)
	Jan Vitek	-	Professeur (Charles University, Prague)
Examinatrices :	Amal Amhed	-	Professeure (Northeastern University, Boston)
	Delia Kesner	-	Professeure (IRIF, Paris)

Acknowledgements - Remerciements

First of all, I would like to thank my reviewers, Alan Schmitt and Jan Vitek, for their careful reading of my manuscript and their detailed insights and comments. I am very grateful for the time you spent reviewing my thesis. I would also like to thank my examiners, Amal Amhed, Delia Kesner, and Didier Remy, for accepting to be part of the committee. Thank you all for your attention to my work.

J'aimerais maintenant remercier mes deux encadrants, Giuseppe et Kim. Merci du fond du cœur pour ces quatre années formidables. Merci d'avoir été disponible à tout moment, merci pour votre bienveillance et vos conseils. Pour avoir écouté mes idées et mes difficultés (pourtant pas toujours exprimées très clairement), pour les réflexions et les discussions que nous avons eues ensemble, vous avez fait de ces quatre années une expérience épanouissante autant scientifiquement qu'humainement. Merci pour ces moments précieux, à Philadelphie, à Londres, et ici même, à l'IRIF et au LMF. Merci également à Alan Schmitt et Didier Remy pour avoir accepté d'être membre de mon CSI. Je suis honoré de l'attention que vous avez portée à mon travail et du temps que vous y avez consacré.

Il me tient à cœur de remercier également mes professeurs qui, en plus de compétences académiques, m'ont appris à apprécier les mathématiques, l'informatique théorique et la recherche. Vos enseignements m'ont motivé à continuer dans cette voie, et à mon tour, à faire de la recherche et de l'enseignement. Tout particulièrement, je pense à Jérôme Mollaret, qui dès le lycée m'a transmis sa passion du raisonnement, à travers son énergie et sa rigueur enthousiaste. Je crois, en tout cas j'espère, qu'à mon tour je transmets un peu de cet enthousiasme quand j'enseigne à mes étudiants des notions qui me tiennent à cœur. Merci à mes professeurs de prépa et à ceux de l'ENS Paris-Saclay. Encore une fois, merci à Giuseppe. Par tes introductions didactiques et motivées, tu m'as fait apprécier l'étude des langages de programmation et des systèmes de types.

Un immense merci à mes collègues et amis. À Victor, pour ta gentillesse et ton aide pendant mon stage de M2 et ma première année de thèse. À tous mes amis du MPRI, avec qui j'ai beaucoup travaillé, débattu et rigolé. Merci Kostia pour ces super souvenirs : les Escape Game, *Nene Quest*, les parties de Smash... À mes collègues de bureau, pour votre très agréable compagnie. Aliaume, ces discussions autour de tes nombreuses idées de *tooling* me manquent déjà ! Il faut bien quatre années pour faire une thèse tout en plaisantant avec Alexandra et toi : merci pour ce sabotage de productivité ! Florian, Emily, Emmanuel, un grand merci à vous pour votre soutien, pour m'avoir remplacé sans hésiter, et pour ces moments où nous maudissions ensemble les étudiants. Enfin, merci à Philippe pour ta compagnie et tes relectures, et bonne chance dans ta noble quête de *side-effects* : puisse l'inquisition rétablir l'ordre dans ton λ-calcul profané par les impuretés. Et à tous ceux que je n'ai pas cité, avec qui j'ai partagé tant de repas, de discussions et de rires. C'était un plaisir de venir au labo pour discuter avec vous tous !

À mes amis qui m'ont accompagné pendant ce périple. À Mathis et Margaux, pour tout ce que nous avons partagé : les Escape Game, le développement d'applications (et autres *bidouilles*), et tous ces précieux moments passés ensemble. À Stanislas et Augustin, pour toutes nos sessions Discord passées à échanger et à jouer, à farmer le Dracaille, à perdre sur Fall Guys, ou à faire des donjons sur New World. Et enfin, à mon compère Sylvain et à Rita, pour ces longues soirées que nous passons ensemble à partager nos passions. À ces victoires sur Ravenswatch et ces échecs sur Deep Rock Galactic, à notre temple Minecraft, à nos nombreux projets informatiques, à ces nombreuses dégustations de Shawarma, et à tous ces moments inoubliables.

À mon grand frère Jonathan, je partageais déjà mes premiers projets de programmation avec toi il y a 15 ans, et encore aujourd'hui je partage ma passion de l'informatique, de la recherche et des raisonnements. C'est en m'inspirant de toi et grâce à tes conseils que je présente aujourd'hui ma thèse de doctorat. Merci !

À Papa, à Maman, vous m'avez toujours soutenu dans mes études et êtes toujours présents pour partager des moments avec moi, pour me cuisiner des plats que j'aime et m'aider quand j'en ai besoin. Merci de m'avoir toujours fait confiance !

Et enfin, à toute ma famille, à Adeline, Éden, Mélodie, à mes cousins, mes oncles, mes tantes et mes grands-parents. À vous tous qui m'apportez tant de réconfort et de bons moments, merci !

Polymorphic type inference for dynamic languages: reconstructing types for systems combining parametric, ad hoc, and subtyping polymorphism

Abstract: In this thesis, we present a type system based on set-theoretic types that aims to type dynamic languages, such as JavaScript, Python or Elixir. The use of set-theoretic types is motivated by their expressivity: they feature a subtyping relation that supports unrestricted intersections, unions, and negations. This allows capturing many idiomatic behaviors of dynamic languages: intersection types are used to capture overloaded behaviors, and union and negation types open the way to a precise typing of type-cases (dynamic tests of type) using advanced techniques of occurrence typing (or type narrowing). Set-theoretic types can also be extended with type variables in order to implement parametric polymorphism, which allows designing a modular type system where the definitions are typed sequentially. However, set-theoretic types are usually used in a language where functions need to be explicitly annotated with their type by the programmer. We get rid of this constraint, working on a λ-calculus with no type annotation. Instead, an inference algorithm is charged of reconstructing the type of functions.

In the first part of this manuscript, we go through the foundations of set-theoretic types and introduce our language, a call-by-value λ-calculus with pairs and type-cases. We discuss and illustrate the challenges of typing such a language.

The second part contains the core formalization (and proofs) of our type system. The first type system we define is purely declarative. It combines several rules inspired from natural deduction, in particular: the union-elimination, the intersection-introduction, the instantiation, and the generalization. We prove the type safety of this system: a typeable program always reduces to a value of the same type or diverges. We then define an algorithmic (deterministic) type system that is equivalent to the declarative one, but that takes as additional input an annotation tree that specifies, among others, the types of the parameters of λ-abstractions. Finally, we define a reconstruction algorithm that aims to reconstruct annotation trees.

In the last part of this manuscript, we focus on some practical aspects. We discuss some extensions and optimizations of the algorithmic type system and reconstruction algorithm, and we present a prototype implementation. We evaluate this prototype on several examples, highlighting its strengths and weaknesses, and comparing it to other related approaches.

Keywords: static typing, dynamic language, type inference, set-theoretic types, semantic subtyping, occurrence typing, type narrowing, polymorphism, intersection types, union types

Inférence de types polymorphes pour des langages dynamiques: reconstruction de types pour des systèmes combinant polymorphisme paramétrique, surcharge et sous-typage

Résumé : Cette thèse porte sur la conception d'un système de types basé sur les types ensemblistes et visant à typer les langages dynamiques, tels que JavaScript, Python ou Elixir. L'utilisation des types ensemblistes est motivée par leur expressivité : ils sont dotés d'une relation de sous-typage qui prend en charge intersections, unions et négations. Cela permet de capturer de nombreux comportements idiomatiques des langages dynamiques : les types intersection sont utilisés pour capturer les comportements surchargés, et les types union et négation ouvrent la voie vers un typage précis des *type-cases* (tests de type dynamiques) via l'utilisation de techniques de rétrécissement de types. Les types ensemblistes peuvent également être étendus avec des variables de type afin d'implémenter du polymorphisme paramétrique, permettant la conception d'un système de types modulaire dans lequel les définitions sont typées séquentiellement. Cependant, les types ensemblistes sont généralement utilisés dans un langage où les fonctions doivent être explicitement annotées par leur type. Nous nous débarrassons de cette contrainte en travaillant sur un λ-calcul sans annotation de type. À la place, un algorithme d'inférence est chargé de reconstruire le type des fonctions.

Dans la première partie de ce manuscrit, nous passons en revue les fondements des types ensemblistes et présentons notre langage, un λ-calcul par appel par valeur avec paires et *type-cases*, et discutons des défis posés par le typage d'un tel langage.

La deuxième partie contient la formalisation de base (et les preuves) du système de types. Le premier système de types que nous définissons est purement déclaratif. Il combine plusieurs règles inspirées de la déduction naturelle, en particulier : l'élimination de l'union, l'introduction de l'intersection, l'instanciation et la généralisation. Nous prouvons la sûreté du typage de ce système : un programme typeable se réduit toujours à une valeur de même type ou diverge. Nous définissons ensuite un système de types algorithmique (déterministe) équivalent au système déclaratif, mais qui prend comme entrée supplémentaire un arbre d'annotations qui spécifie, entre autres, les types des paramètres des λ-abstractions. Enfin, nous définissons un algorithme de reconstruction qui vise à reconstruire les arbres d'annotations.

Dans la dernière partie de ce manuscrit, nous nous concentrons sur certains aspects pratiques. Nous discutons de certaines extensions et optimizations du système de types algorithmique et de l'algorithme de reconstruction, et nous présentons un prototype d'implémentation. Nous évaluons ce prototype sur plusieurs exemples, en soulignant ses forces et ses faiblesses, et en le comparant à d'autres approches.

Mots clés : typage statique, langage dynamique, inférence de type, types ensemblistes, sous-typage sémantique, typage d'occurrence, rétrecissement de type, polymorphisme, types intersection, types union

Résumé long

Cette thèse porte sur la conception d'un système de types basé sur les types ensemblistes et visant à typer les langages dynamiques, tels que JavaScript, Python ou Elixir. L'utilisation des types ensemblistes est motivée par leur expressivité : ils sont dotés d'une relation de sous-typage qui prend en charge intersections, unions et négations. Cela permet de capturer de nombreux comportements idiomatiques des langages dynamiques : les types intersection sont utilisés pour capturer les comportements surchargés, et les types union et négation ouvrent la voie vers un typage précis des *type-cases* (tests de type dynamiques) via l'utilisation de techniques de rétrécissement de types. Les types ensemblistes peuvent également être étendus avec des variables de type afin d'implémenter du polymorphisme paramétrique, permettant la conception d'un système de types modulaire dans lequel les définitions sont typées séquentiellement. Cependant, les types ensemblistes sont généralement utilisés dans un langage où les fonctions doivent être explicitement annotées par leur type. Nous nous débarrassons de cette contrainte en travaillant sur un λ-calcul sans annotation de type. À la place, un algorithme d'inférence est chargé de reconstruire le type des fonctions.

Types ensemblistes

Les types ensemblistes, introduits et formalisés par Frisch (2004), ajoutent aux constructeurs de types habituels (flèche $\to$ et produit $\times$) la possibilité d'exprimer l'union $\vee$ de deux types, l'intersection $\wedge$ de deux types, et la négation $\neg$ d'un type.

Le premier défi est de définir une relation de sous-typage sur ces types. Pour ce faire, à chaque type t est associée une interprétation $[\![t]\!]$ comme un ensemble de valeurs : par exemple, l'interprétation de $\texttt{Int} \times \texttt{Bool}$ est l'ensemble de toutes les paires dont le premier composant est un entier et le second un booléen. Notez toutefois que cette interprétation est plus difficile à définir pour les types de flèches et qu'elle nécessite une attention particulière. En utilisant cette interprétation, le sous-typage est naturellement défini comme suit : un type t_1 est un sous-type d'un type t_2 si et seulement si l'ensemble $[\![t_1]\!]$ est un sous-ensemble de $[\![t_2]\!]$. Cette relation de sous-typage est souvent appelée *sous-typage sémantique*.

Cette définition formelle du sous-typage est très bien, mais elle implique des ensembles infinis qui ne peuvent pas être facilement manipulés dans un algorithme. Le deuxième défi consiste donc à trouver un moyen calculable de déterminer si un type est sous-type d'un autre. Au lieu de considérer les interprétations des types, l'algorithme de sous-typage travaille directement sur l'arbre syntaxique des types. Une représentation spéciale est définie pour les types : la Forme Normale Disjonctive (DNF). Chaque type peut être représenté par une Disjunctive Normal Form (DNF), et cette forme simplifie la décision du sous-typage et, plus généralement, la mise en place d'opérateurs sur les types.

La DNF peut également être utilisée pour la résolution de contraintes de sous-typage impliquant des variables de type. Étant donné un ensemble de contraintes de

sous-typage $\{s_1 \dot{\leqslant} t_1 \ ; \ ... \ ; \ s_n \dot{\leqslant} t_n\}$, nous voulons trouver toutes les substitutions de type ϕ telles que $\forall i \in 1 \mathinner{..} n.\ s_i\phi \leqslant t_i\phi$. Ce problème, appelé *tallying*, est décidable, et il s'agit d'une opération clé utilisée par notre inférence de type.

Langage et problématique

Notre langage est un λ-calcul en appel par valeur, avec des constantes et des paires, ainsi qu'une construction $(e{\in}\tau)\,?\,e_1 : e_2$ appelée *type-case* qui permet de tester le type de l'expression e à l'exécution et de brancher dynamiquement sur la branche e_1 ou e_2 selon le résultat du test. Ici, τ est un type qui ne contient aucun type flèche à l'exception de $\mathtt{Empty} \to \mathtt{Any}$ (le supertype de toutes les fonctions). En pratique, cela signifie que nous pouvons vérifier si une valeur est une λ-abstraction ou non, mais nous ne pouvons pas vérifier, par exemple, si cette λ-abstraction accepte des entiers en entrée ou non. Cette restriction est nécessaire pour donner une sémantique correcte aux type-cases : comme nos λ-abstractions ne sont pas explicitement annotées avec leur type, il n'est pas possible, au moment de l'exécution, de déterminer le type d'une λ-abstraction.

Cette construction de type-case est très simple, et pourtant elle est difficile à typer. En effet, le typage d'une expression $(e{\in}\tau)\,?\,e_1 : e_2$ peut nécessiter l'utilisation de typage d'occurrence (ou rétrécissement de type) : il est possible que la première branche e_1 (respectivement, la seconde branche e_2) ne puisse être typée que dans un environnement de type raffiné qui tient compte du fait que l'expression e se réduit à une valeur de type τ (respectivement, de type $\neg\tau$). En outre, les type-cases permettent d'écrire des fonctions avec un comportement surchargé. Ce comportement surchargé doit être capturé par notre système de types, ce qui nous oblige à dériver des types intersection pour les fonctions.

Enfin, notre langage et notre système de types doivent être modulaires. Lors du typage d'une base de code conséquente, constituée de plusieurs définitions successives, le type déduit pour une définition ne doit pas dépendre des définitions ultérieures. Par exemple, une bibliothèque doit pouvoir être typée indépendamment des projets qui l'utilisent. Pour capturer cette notion de modularité, nous introduisons la notion de *programmes*, consistant en une séquence de définitions qui doivent être typées l'une après l'autre.

Système de types déclaratif

D'un point de vue conceptuel, le système de types déclaratif est assez simple : il fusionne trois des systèmes de types les plus expressifs étudiés dans la littérature, à savoir les types polymorphes de Hindley-Milner (Hindley, 1969; Milner, 1978), les types intersection (Coppo et al., 1981), et les types union (MacQueen et al., 1986; Barbanera et al., 1995). Nous y parvenons simplement en rassemblant de manière contrôlée les règles de déduction caractéristiques de chacun de ces systèmes, détaillées ci-dessous, et en prouvant que le système résultant est sûre.

Les types intersection peuvent être utilisés pour capturer le comportement des

fonctions surchargées : par exemple, l'opérateur surchargé `+` qui peut effectuer l'addition d'entiers et la concaténation de chaînes de caractères peut être typé $(\texttt{Int} \to \texttt{Int} \to \texttt{Int}) \wedge (\texttt{String} \to \texttt{String} \to \texttt{String})$. Pour dériver des types intersection pour les fonctions, nous ajoutons à notre système de types la règle d'introduction de l'intersection : si nous pouvons dériver le type t_1 et le type t_2 pour une expression e, alors nous pouvons dériver pour cette dernière le type $t_1 \wedge t_2$.

Les types union sont utilisés pour implémenter le typage d'occurrence. La puissance des types union est exploitée par la règle d'élimination de l'union : étant donné une expression e que nous voulons typer et une sous-expression e' de e dont le type peut être décomposé en une union $t_1 \vee t_2$, nous pouvons choisir de diviser la dérivation de typage en deux branches indépendantes, l'une qui suppose que les occurrences de cette sous-expression sont de type t_1, et l'autre qui suppose qu'elles sont de type t_2. Cette mécanique, associée à des des règles simples pour les type-cases qui permettent de sauter une branche lorsqu'elle est inaccessible[1], permet de typer une branche d'un type-case en considérant uniquement les environnements de type pour lesquels elle est accessible. Ainsi, ce mécanisme capture pleinement l'essence du typage d'occurrences.

Enfin, le polymorphisme paramétrique de Hindley-Milner est nécessaire pour rendre notre système de types modulaire. Bien que les types intersection puissent être utilisés pour capturer le comportement des fonctions surchargées, ils ne peuvent pas capturer complètement le comportement des fonctions génériques, car cela nécessiterait un nombre infini d'intersections : par exemple, la fonction identité peut être approximée par un type intersection tel que $(\texttt{Int} \to \texttt{Int}) \wedge (\neg\texttt{Int} \to \neg\texttt{Int})$, mais ce type ne convient que si l'on sait que cette fonction ne sera appliquée qu'à des arguments de type `Int` et $\neg$`Int`. Dans un langage modulaire, cependant, nous ne savons pas à l'avance dans quels contextes cette fonction sera utilisée. Le polymorphisme paramétrique apporte une solution à ce problème, en nous permettant de typer la fonction identité $\alpha \to \alpha$ et d'instancier librement la variable de type α plus tard, chaque fois que cette fonction est utilisée.

Le système de types que nous obtenons est puissant, mais il n'est pas algorithmique : plusieurs dérivations de typage peuvent exister pour un même jugement, et elles peuvent avoir des formes très différentes, en grande partie à cause de la règle d'élimination de l'union. Pour prouver la sûreté de ce système de types, nous restreignons la forme des jugements de typage en définissant une notion de *dérivations canoniques*, et nous décrivons un processus de normalisation permettant de transformer n'importe quelle dérivation en dérivation canonique. En particulier, nous montrons que la règle d'élimination de l'union n'a besoin d'être appliquée qu'une seule fois sur chaque sous-expression, et nous limitons les endroits de la dérivation où elle peut être appliquée. Cette notion de dérivation canonique constitue un premier pas vers un système de types algorithmique.

[1] Trois règles de typage au total : une qui couvre le cas où l'expression testée a le type vide `Empty` et donc les deux branches sont inaccessibles, et les deux autres qui couvrent les cas où l'une des deux branches est inaccessible

Système de types algorithmique

La définition d'un système de types algorithmique, correct et complet par rapport au système déclaratif, n'est pas une tâche facile. Pour être algorithmique, un système de types doit satisfaire deux propriétés : (i) il doit être dirigé par la syntaxe, et (ii) toutes ses règles doivent être analytiques[2].

Un système de types est dirigé par la syntaxe lorsque la syntaxe de l'expression que nous typons identifie de manière unique la règle à appliquer. Ce n'est pas le cas du système de types déclaratif, notamment à cause de la règle d'élimination de l'union : cette règle peut être utilisée sur n'importe quelle expression e pour décomposer le type de n'importe quelle sous-expression e' de e. Afin de rendre le système de types dirigé par la syntaxe, nous restreignons l'utilisation de la règle d'élimination de l'union en se basant sur la forme des dérivations canoniques. Sans entrer dans les détails, lors de la saisie d'une expression e, nous appliquons la règle d'élimination de l'union une seule fois sur chaque sous-expression e' de e, et ce dès que toutes les variables libres de e' sont dans l'environnement de typage. Pour implémenter ce comportement dans le système de types algorithmique, nous transformons d'abord l'expression e en une forme canonique κ, qui consiste en une séquence de *bindings* associant des variables à des expressions dans lesquelles chaque sous-expression propre est une variable. Par exemple, une forme canonique pour l'expression $(\mathsf{f}\ \mathsf{x}, \mathsf{f}\ \mathsf{x})$ est $\mathtt{bind\,y = f\,x\,in\,bind\,z = (y, y)\,in\,z}$. L'avantage de cette forme est que chaque sous-expression de e est maintenant liée à une variable : le système de types algorithmique peut donc simplement appliquer la règle d'élimination de l'union une fois sur chaque définition `bind`. Cependant, afin de préserver la typabilité, les formes canoniques doivent satisfaire certaines propriétés : en particulier, deux sous-expressions syntaxiquement équivalentes doivent être liées à la même variable (cette contrainte est nécessaire pour préserver la corrélation entre les différentes occurrences d'une même sous-expression). Une forme canonique qui satisfait cette propriété—et deux autres—est appelée forme MSC (Maximal Sharing Canonical). Nous prouvons que toute expression e possède une forme MSC unique, notée $\mathsf{MSC}(e)$, et nous fournissons un moyen de la calculer.

Pour avoir un système de types algorithmique, nous avons également besoin que toutes les règles soient analytiques. Une fois de plus, la règle d'élimination de l'union est problématique : la décomposition de type $t_1 \vee ... \vee t_n$ ne peut être déduite de la conclusion de la règle. Une autre règle non analytique est la règle de typage des λ-abstractions, car elle doit deviner un type pour le paramètre. Pour rendre ces règles analytiques, les jugements dérivés par le système de types algorithmique sont modifiés : en plus d'un environnement de type Γ et d'une forme canonique κ, ils prennent comme entrée supplémentaire un arbre d'annotations qui spécifie ces éléments. Ce triplet formé d'un environnement de type, d'une forme canonique et d'un arbre d'annotations, encode de manière unique une dérivation canonique

[2]Une règle est analytique (par opposition à synthétique) lorsque les entrées de la conclusion du jugement (c'est-à-dire l'environnement de type et l'expression) sont suffisantes pour déterminer les entrées des prémisses du jugement (cf. Martin-Löf (1994); Types (2019)).

du système de types déclaratif. Ainsi, nous avons réduit le problème consistant à trouver une dérivation pour une expression e avec le système de types déclaratif au problème consistant à rechercher un arbre d'annotations qui rend $\mathsf{MSC}(e)$ typable avec le système de types algorithmique.

Reconstruction de l'arbre d'annotations

Le problème suivant que nous abordons est la reconstruction (inférence) de l'arbre d'annotations utilisé par le système de types algorithmique. À cette fin, nous définissons un algorithme, décrit par un système de règles de déduction, qui raffine incrémentalement (en utilisant du backtracking) un arbre d'annotations, initialement composé d'un seul noeud "`infer`". Il combine deux mécanismes : (*i*) un qui infère le(s) domaine(s) des λ-abstractions et qui s'inspire de l'algorithme $\mathcal{W}$ de Damas and Milner (1982), et (*ii*) l'autre qui déduit les décompositions de type à utiliser par les applications de la règle d'élimination de l'union sur chaque définition `bind`.

Le premier mécanisme commence par typer chaque paramètre d'une λ-abstraction avec une nouvelle variable de type, utilisée comme marqueur symbolique qui est ensuite remplacé au fur et à mesure que de nouvelles contraintes sont découvertes. La principale différence avec $\mathcal{W}$ est que, alors que $\mathcal{W}$ utilise l'unification pour résoudre des contraintes syntaxiques, nous devons résoudre des contraintes de sous-typage sémantique : pour cela, nous utilisons l'algorithme de tallying. Les solutions à une instance de tallying sont caractérisées par un ensemble fini de substitutions, obligeant notre algorithme de reconstruction à se ramifier afin de considérer chaque substitution séparément.

Le second mécanisme, pour déduire les décompositions de type à utiliser par la règle d'élimination de l'union, est déclenché par les type-cases. Chaque fois qu'un type-case $\mathsf{x}\in\tau\,?\,\mathsf{y}:\mathsf{z}$ est rencontré, la décomposition de type associée à x est raffinée : le type de x est divisé en τ et $\neg\tau$. Plus précisément, si l'environnement de type courant est Γ, l'algorithme de reconstruction considère deux branches : l'une dans laquelle x est de type $\Gamma(\mathsf{x}) \wedge \tau$, et une autre dans laquelle x est de type $\Gamma(\mathsf{x}) \wedge \neg\tau$. Cette décomposition est ensuite rétropropagée sur les autres variables apparaissant dans la définition de x.

Aspects pratiques et implémentation

L'algorithme que nous utilisons pour reconstruire les arbres d'annotations repose fortement sur du backtracking et des ramifications. Par conséquent, plusieurs optimisations et heuristiques sont nécessaires afin de réduire l'explosion combinatoire des cas à explorer, et ainsi obtenir des performances raisonnables. En particulier, nous présentons un système qui élimine les branches redondantes de l'arbre d'annotations, des heuristiques pour simplifier les types et les substitutions, ainsi qu'un système de cache basé sur de la mémoïsation.

Ces optimisations sont mises en œuvre dans une implémentation prototype du système de types algorithmique et de l'algorithme de reconstruction. Ce proto-

type est écrit en OCaml, et il comporte plusieurs extensions: la prise en charge des *enregistrements*, des *annotations de type utilisateur*, des *let-bindings* et du *pattern matching*. Nous évaluons ce prototype sur plusieurs exemples et étudions l'impact des différentes optimisations sur les performances.

Cela nous permet de mettre en évidence les forces et les faiblesses de notre approche. Notre système de types est plus expressif que les approches concurrentes telles que Typed Racket et TypeScript, et il propose une inférence de type. Cependant, l'algorithme de reconstruction utilisé pour l'inférence de type n'est pas complet et souffre de performances inégales. Ces problèmes peuvent être atténués par l'introduction d'annotations de type explicites, écrites par le programmeur : non seulement ces annotations peuvent être utilisées pour guider la reconstruction et ainsi repousser les limites découlant de son incomplétude, mais elles améliorent aussi considérablement les performances, car les types annotés par l'utilisateur n'ont pas besoin d'être raffinés ultérieurement à mesure que de nouvelles contraintes sont découvertes, évitant ainsi d'avoir recours à du backtracking.

De futurs travaux sont encore nécessaires si nous voulons intégrer ce système de types à un langage existant. Outre l'amélioration des performances, certaines extensions doivent encore être étudiées et implémentées, telles que le typage graduel (afin que le système de types puisse être déployé progressivement sans avoir à typer toute la base de code déjà existante), la génération de messages d'erreur simples et pertinents, et la prise en charge des effets de bord (si le langage cible n'est pas pur).

Contents

Symbols and notations

Background

$t, s \in \mathcal{T}$	Set-theoretic types	17
$\alpha, \beta \in \mathcal{V}$	Type variables	17
ϕ (resp. Φ)	Substitution (resp. set of substitutions) from $\mathcal{V}$ to $\mathcal{T}$	18
$\mathsf{dom}(\phi)$	Domain of the substitution ϕ	19
$\phi\big\vert_V$	Restriction of ϕ to the domain V	19
$t\#V$	Disjointness of $\mathsf{vars}(t)$ with V	19
$\phi\#V$	Disjointness of $\mathsf{dom}(\phi)$ with V	19
$\mathsf{vars}(\phi)$	Type variables introduced by ϕ	19
$[\![t]\!]$	Interpretation of the type t as a set of values	21
$\mathsf{vars}(t)$	Meaningful type variables in type t	22
$\mathsf{dom}(t)$	Domain of the function type t	23
$\circ, \boldsymbol{\pi}_1, \boldsymbol{\pi}_2$	Type operators	23
$\phi \Vdash_\Delta C$	The substitution ϕ is a solution to the tallying problem (C, Δ)	25

Language

τ	Test type	27
$\equiv$	Syntactic equality	29
$\equiv_\alpha$	Syntactic equivalence modulo α-renaming	29
$\sqsubseteq_\alpha$	Subterm order modulo α-renaming	29
$\rightsquigarrow, \rightsquigarrow_{\mathsf{Pr}}$	Reduction step for expressions, programs	30
$\mathsf{fv}(e)$	Free variables in e	29
$e\{e'/x\}$	Capture-avoiding substitution of e' for x in e	29

Declarative type system

$\boldsymbol{\alpha}, \boldsymbol{\beta} \in \mathcal{V}_M$	Monomorphic type variables $(\mathcal{V}_M \cup \mathcal{V}_P = \mathcal{V})$	38
$\alpha, \beta \in \mathcal{V}_P$	Polymorphic type variables $(\mathcal{V}_M \cup \mathcal{V}_P = \mathcal{V})$	38

$\mathbf{u}, \mathbf{v} \in \mathcal{T}_M$	Monomorphic set-theoretic types	38
σ (resp. Σ)	Substitution (resp. set of substitutions) from $\mathcal{V}_P$ to $\mathcal{T}$	39
ψ (resp. Ψ)	Substitution (resp. set of substitutions) from $\mathcal{V}_M$ to $\mathcal{T}_M$	39
ρ	Renaming (i.e., injective substitution) from $\mathcal{V}_P$ to $\mathcal{V}_P$	39
$\mathsf{dom}(\Gamma)$	Domain of the type environment Γ	40
$\mathsf{vars}(\Gamma)$	Type variables in the type environment Γ	40
$\phi \# \Gamma$	Disjointness of $\mathsf{dom}(\phi)$ with $\mathsf{vars}(\Gamma)$	40
$\vdash, \vdash_{\mathsf{Pr}}$	Declarative type system for expressions, programs	40, 42
$\vDash, \vDash_{\mathsf{Pr}}$	Alternative type system for expressions, programs	46, 67
$\vDash_{\mathcal{N}}, \vDash_{\mathcal{N},\mathsf{Pr}}$	Type system with arrow negations for expressions, programs	71, 73
$\mathsf{Part}(t)$	Set of partitions of the type t	44
$x, y \in \mathsf{Vars}_\lambda$	Lambda variable	45
$\mathsf{x}, \mathsf{y} \in \mathsf{Vars}_\mathsf{B}$	Binding variable	45
$\boldsymbol{x}, \boldsymbol{y} \in \mathsf{Vars}$	Lambda or binding variable	45
$\lhd$	"Is better type" relation	47
$\rightsquigarrow_{\mathcal{P}}, \rightsquigarrow_{\mathcal{P},\mathsf{Pr}}$	Parallel reduction step for expressions, programs	65

Algorithmic type system

κ, a, η	Canonical form, atom, canonical form/atom	90
$\lceil \kappa \rceil$	Unwinding of the canonical form κ	92
$[\![e]\!]$	Transformation of expression e into a canonical form	93
$\dashrightarrow$	Canonical to MSC reduction step	98
$\mathbb{k}, \mathbb{a}, \mathbb{h}$	Form, atom, form/atom annotation	101
$\vdash_{\mathcal{A}}$	Algorithmic type system	103

Reconstruction

$\mathsf{tally}(.)$	Tallying solutions for polymorphic type variables	120
$\mathsf{tally\_infer}(.)$	Tallying solutions for monomorphic type variables	121
$\vdash_{\mathcal{R}}, \vdash^*_{\mathcal{R}}$	Main reconstruction system	121

$\mathcal{K}, \mathcal{A}, \mathcal{H}$	Form, atom, form/atom intermediate annotation	122
$\vdash_{\mathcal{S}}$	Substitution inference system	134
$\vdash_{\mathcal{B}}$	Split backpropagation system	137

Extensions

ξ	Intermediate expression	156
$(\!\|e\|\!)$	Transformation of source expr. e into an intermediate expr.	157

Practical aspects

$\lhd_{\mathsf{T}}$	Approximation of "is better type" relation	173
$\rhd_{\mathsf{T}}$	"Has better coverage" relation	175

Acronyms

Part I

Context and Motivations

Chapter 1

Introduction

Contents

This manuscript aims to design a type system for dynamic languages such as JavaScript or Python, using an approach based on *set-theoretic types* (Frisch et al., 2008). Initially, this thesis was focused on the typing of *dynamic type-cases* (in JavaScript, expressions of the form: `if (typeof(expr) === "type") { ... } else { ... }`). In particular, it aimed at integrating a typing technique usually referred as *occurrence typing* or *type narrowing* into a type system based on set-theoretic types. It has naturally grown into the study of a more general problem: the inference of types for *polymorphic* and *overloaded* functions.

1.1 Dynamic languages

Historically, programming languages allow programs to be executed through a phase of compilation that transforms the source code of the program into machine code, which can then be executed natively. The operations provided by a programming language can operate on different type and size of data (Boolean value, unsigned integer, signed integer, array, function, etc.), and thus a phase of type-checking may be added before compilation in order to verify that operations are applied to expressions of a compatible type, or to determine which operation to perform (for instance, at the level of machine code, dividing two unsigned integers is a different operation than dividing two signed integers). The native machine code produced this way is very efficient, as all operations have been statically resolved: a compiled program does not have to check at run-time whether the division must be performed on unsigned integers or on signed integers. However, another kind of programming language was developed, which we call *dynamic programming languages* (as opposed to *static programming languages*).

Dynamic programming languages execute at run-time many common programming behaviors that static programming languages perform during compilation.

Usually, dynamic languages do not compile programs to native machine code, but their source code (or a transformed version of it) is executed by an *interpreter*. This usually results in poorer performance because an intermediary is required to execute the code, but allows the expression of behaviors that would be difficult to emulate in a static programming language. Some common characteristics of dynamic languages are: (*i*) they offer limited static guarantees, (*ii*) the type of data manipulated can be tested at run-time, so functions can have overloaded behaviors that are resolved at run-time (dynamic dispatch), and (*iii*) the code is part of the state and is mutable (functions can be modified or added at run-time, which is sometimes referred as "reflection"). This characterization includes languages such as JavaScript, Python, Lua, PHP, Racket, R, or the more recent language Julia. Some other languages, such as Erlang and Elixir, can also be considered to be dynamic even though they do not satisfy the point (*iii*): the functional fragment of these two languages is pure (the state is immutable), thus there cannot be side effects.

As an illustration of the flexibility and concision that is made possible by dynamic languages, consider the following Python code:

```
1 def get_population(data, city):
2     df = pandas.read_csv(StringIO(data))
3     return df[df['city'] == city].loc[0,'population']
```

This function `get_population` takes two parameters, the first one is a string containing some CSV data, and the second one a string containing the name of a city. The CSV data must contain at least two columns, "city" and "population". Calling `get_population(data, city)` returns the population associated to `city` according to the CSV string `data`. Line 2 uses the Pandas library to parse the CSV data into a data-frame (a data structure for representing two-dimensional heterogeneous tables). Then, Line 3 filters the table to only keep rows for which the field "city" is equal to the variable `city`, and it returns the "population" field of the first row of the result.

The expression `df[df['city'] == city]` is interesting: it filters the data-frame `df` according to the condition written inside the square brackets. Usually, the `d[k]` notation is used to extract the `k`-th element of the list `d`, or to extract the value associated to the key `k` in the dictionary `d`. However, here, the `[]` operator is overloaded by the Pandas library (and so is the `==` operator):

- The `[]` operator is overloaded a first time so that it can apply to a data-frame and a string. The result of `df['city']` is a series (basically, a one-dimensional indexed array) that represents the "city" column of `df` (each element being indexed by the corresponding row number).

- The `==` operator is overloaded so that, when its first argument `s` is a series and its second argument `v` is a string, it returns a series of Boolean values which can be seen as a characteristic function representing, among the indexes of `s`, those that are associated to a value that is equal to `v`.

- Lastly, the `[]` operator is overloaded again so that, when applied to a data-frame `d` and a series `s`, it filters the rows of `d` according to the characteristic function represented by `s`.

Together, these overloaded definitions make it possible to filter a data-frame by writing `df[df['city'] == city]` without requiring any modification to the Python language itself.

Another thing to note about the `get_population` function is the absence of any type annotation. Indeed, variables do not have static types in Python[1]: types only exist at run-time, and are used for the dynamic dispatch.

This example illustrates how overloading with dynamic dispatch can make dynamic languages very flexible and concise, and thus, convenient for prototyping. However, the lack of static types makes them more error-prone, as invalid operations in the code are detected late at run-time, or worse, remain hidden by being automatically converted to another type, and eventually result in an incorrect computation. Moreover, large codebases may be hard to maintain due to the lack of static type information for the programmer and the toolchain (linter, auto-completion, etc.).

In this manuscript, we aim to design a static type system that could be integrated into a dynamic language (in the same way as TypeScript adds static types to JavaScript), that would provide static safety properties (in particular, if a program type-checks, then no type error should occur at run-time), and that would be doing so without restraining the features of the language. Of course, a compromise has to be found as some features of dynamic languages are notoriously difficult to type, such as the `eval` function available in many dynamic languages (Python, JavaScript, PHP, R, etc.). This thesis focuses on the "dynamic dispatch" aspect of dynamic languages: we will be trying to infer types for overloaded functions that perform dynamic tests of types. The type system we design works on a pure language, and thus cannot be directly applied to a dynamic language with a mutable state. Future work is required in order to support side effects, and to integrate gradual typing into our type system. Gradual typing allows typed and untyped code to coexist, which is essential for introducing a type system into an existing language, since it allows libraries to be typed incrementally. Gradual typing may also be useful in the presence of reflection, such as the `eval` function[2], since we cannot expect such operations to be typeable in the general case. This future work will be discussed in Chapter 10.

[1] Still, it is possible to write static type annotations, but those are ignored by the Python interpreter. These static type annotations may however be used by external tools such as Mypy (Jukka Lehtosalo).

[2] Although it is often discouraged, the use of `eval` is pervasive in some languages: it is often used for meta-programming purposes in languages that do not feature macros, as observed by Goel et al. (2021) for the language R.

1.2 Motivations

Typing dynamic languages is a challenging endeavor even for very simple pieces of code. For instance, JavaScript's logical or operator "`||`" behaves like the following function[3] (also in JavaScript):

```
4  function lOr (x, y) {
5      if (x) { return x; } else { return y; }
6  }
```

A naive type for this function is $(\texttt{Bool}, \texttt{Bool}) \to \texttt{Bool}$, which states that `lOr` is a function that takes two Boolean arguments and returns a Boolean result. This however is an overly restrictive type, that does not account for the fact that in JavaScript logical operators such as `lOr` can be applied to any pairs of arguments, not just to Boolean ones. JavaScript distinguishes two kinds of values: eight "falsy" values (i.e., `false`, `""`, `0`, `-0`, `0n`, `undefined`, `null`, and `NaN`) and the "truthy" values (all the others). The expression `if` executes the `else` code if and only if the tested value is falsy. If we want to change the previous type to account for this fact, then we should give `lOr` the type $(\texttt{Any}, \texttt{Any}) \to \texttt{Any}$ (where `Any` is the type of all values), which is a rather useless type since it essentially states that `lOr` is a binary function. To give `lOr` a more informative type, we need union and intersection types (which are already integrated in typed versions of JavaScript such as TypeScript (Microsoft) and Flow (Facebook)): we define the type `Falsy` as the following union type $\texttt{false} \vee \texttt{""} \vee \texttt{0} \vee \texttt{-0} \vee \texttt{0n} \vee \texttt{undefined} \vee \texttt{null} \vee \texttt{NaN}$, where each value denotes here the *singleton type* containing that value, and the type `Truthy` to be its complement, $\neg\texttt{Falsy}$, that is, the type of all values that are not of type `Falsy`. Then we can deduce for `lOr` the following more precise type:

$$\begin{aligned} & ((\texttt{Truthy}, \texttt{Any}) \to \texttt{Truthy}) \\ \wedge\ & ((\texttt{Falsy}, \texttt{Truthy}) \to \texttt{Truthy}) \\ \wedge\ & ((\texttt{Falsy}, \texttt{Falsy}) \to \texttt{Falsy}) \end{aligned} \tag{1.1}$$

In this type, $\wedge$ is a type combinator denoting intersection and meaning that the function has all the types given in the intersection: that is, in words, if the first argument of a function of this type is a `Truthy`, then the function returns a `Truthy` regardless of the second argument (first arrow type), while if the first argument is a `Falsy`, then the result is of the same type as the second argument's type (second and third arrow type). Notice how the use of an intersection of arrow types corresponds to the typing of an "overloading" behavior (also known as *ad hoc* polymorphism, Strachey (1967)), insofar as the type of the result of an application depends on the type of the input.

In order to derive such a type, the type system must deduce that whenever the condition tested by the `if` holds, then `x` is of type `Truthy` and, therefore, (*i*) that all occurrences of `x` in the "then" branch (here just one) have type `Truthy` and (*ii*)

[3] This definition does not capture the short-circuit evaluation of "`||`".

that all the occurrences of the same variable `x` in the branch "else" (here none) have thus type `Falsy`. This kind of deduction is usually referred as *type narrowing* or *occurrence typing* since it requires to "narrow" the type of a variable `x` differently for its different occurrences. A type system such as the one for Typed Racket—defined in (Tobin-Hochstadt and Felleisen, 2010) where the term *occurrence typing* was first introduced—is able to *check* that `l0r` has the type in (1.1), meaning that the deduction requires the programmer to explicitly specify the type in the code (though, Typed Racket has no negation types).

We can go a step further and type `l0r` using an intersection of *polymorphic* function types, yielding the following type (where α and β are type variables):

$$\forall\alpha,\beta\,.\;((\alpha\wedge\mathtt{Truthy},\mathtt{Any})\to\alpha\wedge\mathtt{Truthy})\wedge((\mathtt{Falsy},\beta)\to\beta) \tag{1.2}$$

This type can be considered as an encoding of the following type, expressed in so-called bounded polymorphism[4]:

$$\forall(\alpha\leqslant\mathtt{Truthy}).\forall(\beta)\,.\;((\alpha,\mathtt{Any})\to\alpha)\wedge((\mathtt{Falsy},\beta)\to\beta)$$

It completely specifies the semantics of the function `l0r`: it states that if the first argument is a `Truthy`, then the application of the function returns the first argument[5], otherwise it returns the second argument. This type is more precise than the one in (1.1), since it allows the system to deduce that, say, if the first argument of `l0r` is an object, then the result will be an object of the same type (rather than just a truthy value).

The type system described in this manuscript is not only capable of checking such a type, but also to infer it (i.e., to reconstruct it, without requiring the programmer to write explicit type annotations). Still, it does not seem too hard a feat to deduce that if we are testing whether `x` is a truthy value, then when the test succeeds we can assume that `x` is of type `Truthy`. To show the more advanced capabilities of our system let us have a look at how ECMAScript specifies the semantics of JavaScript logical operators, as defined in the 2021 version of the specification (Ecma, 2021, Section 13.13.1). Since in JavaScript there are no union or intersection types, then the falsy and truthy values are defined via an (abstract) function `ToBoolean` which simply checks whether its argument is one of the 8 falsy values and returns `false`, otherwise it returns `true`. In our system, `ToBoolean` has type (`Truthy` → `true`) ∧ (`Falsy` → `false`). All logical operators are then defined by ECMAScript in terms of this function: this has the advantage that any change to the specification of falsy (e.g., the addition of a new falsy value, like the addition of the built-in `bigint` type and its constant `0n` in ES2020) requires only the modification of this function, and

[4] A type $\forall(s_1\leqslant\alpha\leqslant s_2).\,t$ can be encoded by the type $\forall\alpha.t'$ where t' is obtained by replacing every occurrence of α in t by $(\alpha\vee s_1)\wedge s_2$. See Castagna (2024, Section 2) for more details.

[5] Strictly speaking, the type states that the function returns a result of the same type as the first argument, but by parametricity we can deduce that the result will be the first argument. Likewise for the second argument. A simple way to understand it is by instantiating both type variables in (1.2) with the singleton type of the (value result of the) argument.

is automatically propagated to all operators. So the actual definition of `lOr` for ECMAScript is the following one:

```
7  function lOr (x, y) {
8      if (ToBoolean(x)) { return x; } else { return y; }
9  }
```

If we feed this function to our system, then it infers for it the type in (1.2), that is, the same type it already deduced for the simpler version of `lOr` defined in lines 4-6. But here the deduction needed to perform occurrence typing is more challenging, since the system must deduce from the type $(\mathsf{Truthy} \to \texttt{true}) \wedge (\mathsf{Falsy} \to \texttt{false})$ of `ToBoolean` that when the *application* in line 8 returns a truthy value, then the *argument* of `ToBoolean` is of type `Truthy`, and it is of type `Falsy` otherwise. More generally, we need a system which, when a test is performed on an arbitrarily complex application, can narrow the type of all the variables occurring in the application by exploiting the information provided by the overloaded behavior of the functions therein.

Achieving such a degree of precision is a hard feat but, we argue, it is necessary if we want to reconstruct types for dynamic languages. Indeed, the core operators of these languages (e.g., JavaScript's "`||`", "`&&`", "`typeof`", ...) are characterized by an "overloaded" behavior, which is then passed over to the functions that use them. So for instance a simple use of JavaScript logical or "`||`" such as in the anonymous function `((x) => x || 42)` results in a function whose precise type, as reconstructed by our system, is $(\mathsf{Falsy} \to 42) \wedge (\mathsf{Truthy} \wedge \alpha \to \mathsf{Truthy} \wedge \alpha)$. JavaScript functions also routinely perform dynamic checks against constants (notably `null` and `undefined`), which our system also handles as part of its more general approach to type narrowing of arbitrary expressions.

1.3 Contributions

This manuscript aims to apply the power of set-theoretic types to dynamic languages featuring type-cases, first-order functions and polymorphism. Consequently, it makes an extensive use of prior work on set-theoretic types, such as Frisch (2004); Frisch et al. (2008); Castagna and Xu (2011).

Polymorphic type systems based on set-theoretic types have already been studied by Castagna et al. (2014, 2015), but for a language where λ-abstractions are explicitly typed. This manuscript proposes a new approach where λ-abstractions are not explicitly typed, and where our type system is able to infer the type of polymorphic and overloaded functions, while also fully capturing the essence of occurrence typing.

The general contribution of this work is twofold. First, it proposes a way to mix parametric, intersection/ad hoc polymorphism, and occurrence typing inside a formal system of deduction rules (the declarative type system). While the rules composing this formal system are not novel (intersection-introduction, union-elimination, instantiation, etc.), combining them all together in a type system requires some

care, in particular for the union-elimination rule, and is an original work.

Second, it proposes an effective way to implement this type discipline by defining a reconstruction algorithm; with respect to that, a fundamental role is played by the analysis of the type tests performed by the expressions, since they drive the way in which types are split: externally, to split the domain of functions yielding intersection of arrows (intersection-introduction); internally, to split the type of tested expressions, yielding a precise typing of branching (union-elimination). In doing so, it provides the first system that reconstructs types and that combines parametric and *ad hoc* polymorphism.

The technical contributions of this work can be summarized as follows:

- We define a declarative type system that combines parametric polymorphism with union and intersection types for a functional calculus with type-cases, and we prove its soundness.
- We define an algorithmic system that we prove sound and complete with respect to the previous system.
- We define an algorithm to reconstruct the type annotations of the previous algorithmic system, allowing the inference of overloaded and polymorphic types for functions.
- We provide several extensions, enriching our language with records, let-bindings, pattern-matching expressions, and user type annotations.
- We provide a full implementation of the algorithmic system and reconstruction algorithm with the extensions, available online (`https://www.cduce.org/dynlang/`, Castagna et al. (2024b)).

1.4 Timeline and scientific production

The approach presented in this manuscript is the successor of two prior attempts. These two prior attempts are not detailed in this manuscript as we consider the present approach to be a strict improvement, both in terms of expressivity and presentation. Still, they are briefly described and compared to the present approach in this section.

The first approach that we explored aimed at integrating occurrence typing (Tobin-Hochstadt and Felleisen (2008)) into a type system based on set-theoretic types, by refining the environment using an auxiliary deduction system before typing the branches of a type-case. In order to remember type information about complex expressions, the type environment was extended: in addition to storing type information about variables, it could hold information about the type of arbitrary expressions. For instance, when typing the first branch of an expression `if (f(x)) { ... } else { ... }`, the environment would map the expression `f(x)` to the type `Truthy`, and when typing the `else` branch it would map `f(x)` to the type `Falsy`. This capability of the environment to contain type information about arbitrary ex-

pressions adds some complexity to the declarative type system. To that complexity one also needs to add the complexity of the auxiliary deduction system used for refining the type environment (from the information that `f(x)` is `Truthy`, we may also learn type information about `f` and `x`). In the end, the type system proposed was complex, and we did not manage to propose an algorithmic version of this system that is both sound and complete (we proposed one that was sound, and that was complete for a restricted set of derivations). Nevertheless, this approach brought some novel ideas to set-theoretic type systems and has been published in Castagna et al. (2022a).

The second approach we explored is very different, though it follows the same goal, namely, integrating occurrence typing into a set-theoretic type system. Instead of using complex rules for type-cases, it features two very simple rules. The first one only applies when the type of the tested expression e is a subtype of the tested type t, in which case we only need to type the first branch. Conversely, the second rule applies when the type of e is disjoint from t, in which case only the second branch is typed. These two rules alone do not cover the general case, but the idea is to combine them with the union-elimination rule (from Barbanera et al. (1995)), a rule allowing to split the type of a subexpression into several smaller types and to consider each case separately. This combination of rules captures the essence of occurrence typing, yet still keeping a very simple declarative type system. Finding derivations for this type system, however, is challenging: we have to decide when to apply this union-elimination rule, on which subexpression, and how to decompose the type of this subexpression. The main development of this work thus consists in restricting the use of the union-elimination rule in a controlled way, and to use it to define an algorithmic type system that is both sound and complete with respect to the declarative one. This algorithmic type system, however, requires type annotations to be inserted in the expression we want to type. The last part of this work aims to define an inference algorithm whose role is to reconstruct those annotations. It is unable, however, to infer higher-order types for function arguments. This approach has led to a publication and presentation at POPL 2022 (Castagna et al., 2022b).

The present work can be seen as a polymorphic extension of this second approach, though its also has other benefits (in particular, a better type inference, made possible by the introduction of type variables). It borrows some key notions from it, such as (*i*) the combination of the union elimination rule with two simple rules for type-cases in order to capture the essence of occurrence typing, and (*ii*) the way to restrict the use of the union-elimination rule in order to obtain an algorithmic type system. However, the introduction of polymorphic types greatly modifies the meta-theory. Besides its influence on the union-elimination rule, the presence of type variables suggests a new approach for type inference, inspired by $\mathcal{W}$ from Damas and Milner (1982): parameters of λ-abstractions are first typed using a fresh type variable α, acting as a symbolic marker which is then substituted as new constraints are discovered. This yields a clear improvement, with the capability to infer higher-order types and types of recursive functions. This work has been published and presented at POPL 2024 (Castagna et al., 2024a). In this manuscript,

we give a more detailed presentation of this approach, including several practical considerations and discussions.

1.5 Outline

Part I: Context and Motivations This part introduces the motivations and the challenges of our approach, and explains some fundamental notions about set-theoretic types.

Chapter 2: Background This chapter introduces the notions of set-theoretic types that will be used throughout this manuscript. All the formalizations and results presented in this chapter come from previous work (Frisch, 2004; Castagna et al., 2014, 2015).

Set-theoretic types add to the usual type constructors (arrow $\rightarrow$ and product $\times$) the possibility to express the union $\vee$ of two types, the intersection $\wedge$ of two types, and the negation $\neg$ of a type.

The first challenge is to define a subtyping relation over these types. To this purpose, we associate to each type t an interpretation $[\![t]\!]$ as a set of values: for instance, the interpretation of $\texttt{Int} \times \texttt{Bool}$ is the set of all pairs whose first component is an integer, and whose second component is a Boolean. Note, however, that this interpretation is more difficult to define for function types, and requires special care. Using this interpretation, subtyping is naturally defined as follows: a type t_1 is a subtype of a type t_2 if and only if the set $[\![t_1]\!]$ is a subset of $[\![t_2]\!]$. This subtyping relation is often referred to as *semantic subtyping*.

This formal definition of subtyping is all well and good, but it involves infinite sets that cannot easily be manipulated in an algorithm. Thus, the second challenge consists in finding a way to decide this subtyping relation. Instead of dealing with interpretations, the subtyping algorithm directly works on the syntactic tree of types. A special representation is defined for types: the Disjunctive Normal Form (DNF). Each type can be represented by a DNF, and this form makes it simpler to decide subtyping, and more generally, to implement operators over types.

Finally, the last challenge tackled in this chapter consists in the resolution of subtyping constraints involving type variables. Given a set of subtyping constraints $\{s_1 \mathrel{\dot{\leqslant}} t_1 \ ; \ \ldots \ ; \ s_n \mathrel{\dot{\leqslant}} t_n\}$, we want to find all type substitutions ϕ such that $\forall i \in 1 \,..\, n.\ s_i\phi \leqslant t_i\phi$. This problem, called *tallying*, is decidable, and it will be a key operation of our type inference.

Chapter 3: Core Language This chapter formalizes the syntax and the semantics of our language. We start with the usual call-by-value λ-calculus,

with constants and pairs, and extend it with a type-case construct $(e{\in}\tau)\,?\,e_1 : e_2$, where τ is a type that does not contain any arrow type except $\texttt{Empty} \to \texttt{Any}$ (the supertype of all functions). In practice, this means that we can check whether a value is a λ-abstraction or not, but we cannot check, for instance, whether this λ-abstraction accepts integers as input or not. This restriction is necessary to give a proper semantics to type-cases: as our λ-abstractions are not explicitly annotated with their type, it is not possible, at run-time, to determine the type of a λ-abstraction.

This type-case construct is very simple, and yet challenging to type. As illustrated by our introductory example, typing an expression $(e{\in}\tau)\,?\,e_1 : e_2$ may require performing occurrence typing: the first branch e_1 (respectively, the second branch e_2) may only be typeable under a refined type environment that accounts for the fact that the expression e reduces to a value of type τ (respectively, of type $\neg\tau$). Additionally, type-cases make it possible to write functions with an overloaded behavior. This overloaded behavior should be captured by our type system, forcing us to derive intersection types for functions.

Another challenge that is discussed in this chapter is the need for modularity. When typing a consequent codebase, consisting in several successive definitions, the type inferred for a definition should not depend on later definitions. For instance, a library should be typeable independently of the projects that use it. To capture this notion of modularity, we introduce the notion of *programs*, consisting in a sequence of top-level definitions that must be typed one after the other.

Part II: Core Formalization This section is dedicated to the formalization of a system that addresses the issues raised in the previous part. To this purpose, it defines: (*i*) a declarative type system, composed of a set of simple yet powerful deduction rules, (*ii*) an algorithmic type system, equivalent to the declarative one, but whose rules are syntax-directed and analytic, and (*iii*) a reconstruction algorithm, that aims to reconstruct the annotations required by the algorithmic type system.

Chapter 4: Declarative Type System This chapter formalizes the declarative type system. Conceptually, it is quite simple: it just merges together three of the most expressive type systems studied in the literature, namely the Hindley-Milner polymorphic types (Hindley, 1969; Milner, 1978), intersection types (Coppo et al., 1981), and union types (MacQueen et al., 1986; Barbanera et al., 1995). We achieve it simply by putting together in a controlled way the deduction rules characteristic of each of these systems, detailed below, and proving that the resulting system is sound.

As seen with our introductory example, intersection types can be used to capture the behavior of overloaded functions. To derive intersection types for

functions, we add to our type system the intersection-introduction rule: if we can derive the type t_1 and the type t_2 for an expression e, then we can derive for it the type $t_1 \wedge t_2$.

Union types are used to implement occurrence typing. The power of union types is exploited by the union-elimination rule: given an expression e that we want to type, and a subexpression e' of e whose type can be decomposed into a union $t_1 \vee t_2$, we can choose to split the typing derivation in two independent branches, one that assumes that occurrences of this subexpression have type t_1, and the other that assumes that they have type t_2. This mechanics, coupled with simple rules for type-cases that allow skipping a branch when it is unreachable[6], makes it possible to type a branch of a type-case only under the type environments for which it is reachable. Thus, this mechanism fully captures the essence of occurrence typing.

Lastly, Hindley-Milner parametric polymorphism is required to make our type system modular. While intersection types can be used to capture the behavior of overloaded functions, they cannot fully capture the behavior of generic functions, as it would require infinitely-many intersections: for instance, the identity function can be approximated by an intersection type such as $(\texttt{Int} \to \texttt{Int}) \wedge (\neg\texttt{Int} \to \neg\texttt{Int})$, but this type is only suitable if we know that this function will only be applied to arguments of type `Int` and $\neg$`Int`. In a modular language, however, we do not know in advance in which contexts this function will be used. Parametric polymorphism brings a solution to this problem, allowing us to type the identity function $\alpha \to \alpha$ and to freely instantiate the type variable α later, whenever this function is used.

The type system we obtain is powerful, but it is non-algorithmic: derivations for the same typing judgment are not unique, and can have very different shapes, largely due to the union-elimination rule. Along the way to proving the type safety of this system, we restrict the shape of typing derivations by defining a notion of *canonical derivations*, and we describe a normalization process to turn any derivation into a canonical derivation. In particular, we show that the union-elimination rule only needs to be applied once on each subexpression, and we further restrict the locations in the derivation where it can be applied. This notion of canonical derivation is a first step towards an algorithmic type system.

Chapter 5: Algorithmic Type System This chapter defines an algorithmic type system, sound and complete with respect to the declarative one of Chapter 4. To be algorithmic, a type system must satisfy two properties: (*i*)

[6] Three typing rules in total: one that covers the case where the tested expression has the empty type `Empty` and thus both branches are unreachable, and the two others that cover the cases where one of the branches is unreachable.

it must be syntax-directed, and (*ii*) all its rules must be analytic[7].

A type system is syntax-directed when the syntax of the expression we are typing uniquely identifies which rule to apply. This is not the case of the declarative type system, in particular because of the union-elimination rule: this rule can be used on any expression e to decompose the type of any subexpression e' of e. In order to make the type system syntax-directed, we restrict the use of the union-elimination rule according to the shape of the canonical derivations characterized in Chapter 4. Roughly, when typing an expression e, we apply the union-elimination rule only once on every subexpression e' of e, and we do so as soon as all the free variables of e' are in the typing environment. To implement this behavior in the algorithmic type system, we first transform the expression e into a canonical form κ, which consists in a sequence of bindings associating variables to expressions in which every proper subexpression is a variable. For instance, a canonical form for the expression $(\mathsf{f}\ \mathsf{x}, \mathsf{f}\ \mathsf{x})$ is $\mathtt{bind}\,\mathsf{y} = \mathsf{f}\ \mathsf{x}\ \mathtt{in}\,\mathtt{bind}\,\mathsf{z} = (\mathsf{y}, \mathsf{y})\ \mathtt{in}\,\mathsf{z}$. The advantage of this form is that each subexpression of e is now bound to a variable: the algorithmic type system can thus simply apply the union-elimination rule once on each $\mathtt{bind}$ definition. However, in order to preserve typeability, canonical forms must satisfy some properties: in particular, two syntactically equivalent subexpressions must be bound to the same variable (this constraint is necessary to preserve the correlation between the different occurrences of a same subexpression). A canonical form that satisfies this property—and two others—is called Maximal Sharing Canonical (MSC) form. We prove that any expression e has a unique MSC form, noted $\mathsf{MSC}(e)$, and we provide a way to compute it.

To have an algorithmic type system, we also need all rules to be analytic. Again, the union-elimination rule is problematic: the type decomposition $t_1 \vee \ldots \vee t_n$ cannot be deduced from the conclusion of the rule. Another non-analytic rule is the rule for typing λ-abstractions, as it guesses a type for the parameter. To make these rules analytic, the judgments derived by the algorithmic type system are modified: in addition to a type environment Γ and a canonical form κ, they take as additional input an annotation tree that specifies these elements. This triplet formed of a type environment, a canonical form, and an annotation tree, uniquely encodes a canonical derivation of the declarative type system. Thus, we have reduced the problem of finding a derivation for an expression e with the declarative type system to the problem of finding an annotation tree that makes $\mathsf{MSC}(e)$ typeable with the algorithmic type system.

Chapter 6: Reconstruction Algorithm This chapter describes an algorithm to reconstruct the annotation tree used by the algorithmic type system

[7] A rule is analytic (as opposed to synthetic) when the input of the judgment at the conclusion (i.e., the type environment and the expression) is sufficient to determine the inputs of the judgments at the premises (cf. Martin-Löf (1994); Types (2019)).

defined in Chapter 5. This algorithm is described by a system of deduction rules that incrementally refines (using backtracking) an annotation tree, initially composed of a single node "`infer`". It mixes two mechanisms: (*i*) one that infers the domain(s) of λ-abstractions and that is inspired by algorithm $\mathcal{W}$ by Damas and Milner (1982), and (*ii*) the other that infers the type decompositions to be used by the applications of the union-elimination rule on each `bind` definition.

The first mechanism starts by typing each parameter of a λ-abstraction with a fresh type variable, used as a symbolic marker that is then substituted as new constraints are discovered. The main difference with $\mathcal{W}$ is that, while $\mathcal{W}$ uses unification to solve syntactic constraints, we have to solve constraints of semantic subtyping: for that, we rely on the tallying algorithm defined in Chapter 2. The solutions to a tallying instance are characterized by a principal finite set of substitutions, forcing our reconstruction algorithm to branch in order to consider each substitution separately.

The second mechanism, for inferring the type decompositions to be used by the union-elimination rule, is triggered by type-cases. Whenever a type-case $\mathtt{(x}\in\tau\mathtt{)\,?y:z}$ is encountered, the type decomposition associated to `x` is refined: the type of `x` is split into τ and $\neg\tau$. More precisely, if the current type environment is Γ, the reconstruction algorithm considers two branches: one in which `x` has type $\Gamma(\mathtt{x}) \wedge \tau$, and another in which `x` has type $\Gamma(\mathtt{x}) \wedge \neg\tau$. This decomposition is then backpropagated to other variables appearing in the definition of `x`.

The reconstruction algorithm we obtain is not complete: it may fail to reconstruct an annotation tree for an expression e even if e is typeable with the declarative type system. Still, it yields a type inference that is evaluated in the next part.

Part III: Towards a Practical Language In this part, we start from the core formalization defined in Part II and transform it into a practical implementation. The language is extended with new constructs (such as records, let-bindings, and pattern-matching), and some optimizations for the reconstruction algorithm are discussed. In addition, a prototype implementation is presented and evaluated on several examples.

Chapter 7: Extensions This chapter extends the source language with several new constructs: (*i*) records, which are implemented in many languages and can also be used to encode objects of object-oriented languages such as JavaScript, (*ii*) user type annotations, which may be used to compensate for the incompleteness of the reconstruction, (*iii*) let-bindings, and (*iv*) pattern-matching, which is encoded in our language using let-bindings and type-cases.

Chapter 8: Practical Aspects This chapter presents several optimizations and heuristics for the reconstruction algorithm: a system that trims redundant branches of the annotation tree, some heuristics for simplifying types and substitutions, and a caching system based on memoization. Because of the branching nature of the reconstruction algorithm, these optimizations are necessary to reduce the combinatorial explosion of cases.

Chapter 9: Prototype Implementation This chapter presents a prototype implementation for the algorithmic type system and reconstruction algorithm. Although it is a prototype, favoring proximity to the formalization rather than heavy optimization, it fully implements the extensions and optimizations introduced in Chapter 8. This prototype is then evaluated on several examples, highlighting its strengths and weaknesses, and the impact of the different optimizations on performance.

Chapter 10: Discussion and Conclusion This chapter discusses the limitations of our approach, and shows how user type annotations can be used to compensate for the incompleteness of the reconstruction algorithm and improve performance. It concludes with a related work section and a discussion of future work: in addition to improving performance, some extensions must be studied and implemented if we want to integrate this type system in a real-world language, such as gradual typing (so that the type system can be deployed progressively without having to type the entire codebase), the generation of simple and relevant error messages, and the support of side effects (if the target language is not pure).

Chapter 2

Background

Contents

This thesis builds on pre-existing work. We recall in this chapter the basic definitions and theorem that we reuse and refer to the relevant publications for details and proofs.

2.1 Set-theoretic types

Most of this work is built on the set-theoretic type theory, introduced by Frisch (2004) and extended with type variables by Castagna and Xu (2011).

Definition 1 (Set-theoretic types)**.** *The set $\mathcal{T}$ of* set-theoretic types *is the set of regular and contractive terms coinductively defined by the following grammar:*

$$\textbf{\textit{Types}} \quad t \quad ::= \quad b \mid \alpha \mid t \to t \mid t \times t \mid t \vee t \mid t \wedge t \mid \neg t \mid \mathbb{0} \mid \mathbb{1}$$

where $b \in \mathcal{B}$ is a base type and $\alpha \in \mathcal{V}$ a type variable. When writing a term, we use the following precedence (by decreasing priority): $\neg$, $\wedge$, $\vee$, $\times$, $\to$.
The notation $t_1 \backslash t_2$ is a syntactic sugar for $t_1 \wedge \neg t_2$.

The set $\mathcal{B}$ of base types and the set $\mathcal{V}$ of type variables are fixed, and we note $\mathcal{T}$ the set of types. Types are ranged over by meta-variables t and s.

As they are defined coinductively, types can be infinite trees, provided that they satisfy the constraints of regularity and contractivity explained below. This yields a definition of equirecursive types that does not require explicit binders for recursion.

A term is said *regular* if it only has a finite number of distinct subterms, and *contractive* if every infinite branch goes through an infinite number of arrows and products ($\to$ and $\times$). The contractivity constraint ensures that every type has a

meaningful interpretation: for instance, it prevents from expressing types such as the one satisfying the equation $t = \neg t$. The regularity constraint ensures decidability of the subtyping relation that we will define in the next sections.

The set of base types should be chosen according to the constants of the language. For each constant of the language, we have the associated base type (also called *singleton type*): the base type `True` for the constant `true`, the base type `False` for the constant `false`, the base type 42 for the constant 42, and so on. Constants are written lowercased, while types are written capitalized. In the examples below, we assume that our language features the usual constants (`true`, `false`, integers, etc.), and the associated singleton types. We also define the base type `Bool` (it is convenient to have it as a base type even though we can construct it as the union `True` $\vee$ `False`), as well as the base type `Int`. Note that `Bool` and `Int` are base types, but not singleton types, because they have more than one inhabitant.

The type $\mathbb{0}$ is a special type that is not inhabited by any value, and is the subtype of all types. Conversely, the type $\mathbb{1}$ is the supertype of all types.

The $\times$ constructor is used to type pairs of our language. Intuitively, it corresponds to the Cartesian product of two types. In particular, the product $\mathbb{1} \times \mathbb{1}$ is the supertype of all pairs: any well-typed pair can be typed with $\mathbb{1} \times \mathbb{1}$.

The $\rightarrow$ constructor is used to type functions (i.e., λ-abstractions). Intuitively, a λ-abstraction has type $t_1 \rightarrow t_2$ if and only if it accepts as argument a value of type t_1, in which case either it yields a value of type t_2 or it diverges. The type $\mathbb{0} \rightarrow \mathbb{1}$ is the supertype of all functions. Using the intersection $\wedge$, it is possible to express the type of overloaded functions. For instance, a function mapping integers to Boolean and Boolean to integers can be given the type (`Int` $\rightarrow$ `Bool`) $\wedge$ (`Bool` $\rightarrow$ `Int`). This capability of types to capture the behavior of overloaded functions is sometimes called *ad hoc polymorphism*.

Type variables α can be used by the type system to handle *parametric polymorphism*. However, at the level of the type algebra, type variables are not quantified: this will be handled by the type system.

2.2 Type substitutions

For any type t, we note $\mathsf{vars}(t)$ the set of type variables occurring in t. A formal definition of $\mathsf{vars}(t)$ will be given later.

Definition 2 (Ground type)**.** *A type t is a* ground type *if $\mathsf{vars}(t) = \varnothing$.*

Definition 3 (Type substitution)**.** *A* type substitution *is a function $\phi : \mathcal{V} \rightarrow \mathcal{T}$ from type variables to types which is the identity everywhere except for a finite set of type variables, called its domain and denoted by $\mathsf{dom}(\phi)$.*

Definition 4 (Application of a type substitution)**.** *The result of the application of a type substitution ϕ to a type t, noted $t\phi$, is the type satisfying these equations:*

$$\alpha\phi = \phi(\alpha) \qquad (t_1 \times t_2)\phi = (t_1\phi) \times (t_2\phi) \qquad (t_1 \vee t_2)\phi = (t_1\phi) \vee (t_2\phi)$$
$$b\phi = b \qquad (t_1 \to t_2)\phi = (t_1\phi) \to (t_2\phi) \qquad (\neg t)\phi = \neg(t\phi)$$
$$\mathbb{0}\phi = \mathbb{0}$$

Note that this system of equations has a unique solution, and this solution is a type as defined by Definition 1 (in particular, it is contractive and regular).

We use Φ to range over sets of type substitutions.

Definition 5 (Application of a set of type substitutions)**.** *The application of a finite set of type substitutions Φ to a type t, noted $t\Phi$, is defined as follows:*

$$t\Phi = \bigwedge_{\phi\in\Phi} t\phi$$

Applying a set of substitutions Φ to a type t amounts to applying all the substitutions in Φ to t independently, and then taking the intersection of the resulting types. For instance, applying the set of substitutions $\{\{\alpha \rightsquigarrow \texttt{Int}\}, \{\alpha \rightsquigarrow \texttt{Bool}\}\}$ to the type $\alpha \to \alpha$ yields the type $(\texttt{Int} \to \texttt{Int}) \wedge (\texttt{Bool} \to \texttt{Bool})$.

Notation.

$\mathsf{dom}(\phi)$ ..... $\stackrel{\text{def}}{=} \{\alpha \in \mathcal{V} \mid \phi(\alpha) \neq \alpha\}$
$t\#V$ ..... $\Leftrightarrow\ \mathsf{vars}(t) \cap V = \varnothing$
$\varnothing$ ..... *The identity type substitution*
$\{\alpha_1 \rightsquigarrow t_1\ ;\ \ldots\ ;\ \alpha_n \rightsquigarrow t_n\}$ ..... *Type substitution mapping α_i to t_i for $i \in 1\,..\,n$*
$\phi_1 \circ \phi_2$ ..... *Composition of substitutions ϕ_1 and ϕ_2*
$\phi\big|_V$ ..... *Restriction of ϕ to the domain V*
$\phi\#V$ ..... $\Leftrightarrow\ \mathsf{dom}(\phi) \cap V = \varnothing$
$\mathsf{vars}(\phi)$ ..... $\stackrel{\text{def}}{=} \{\mathsf{vars}(\phi(\alpha))\backslash\{\alpha\} \mid \alpha \in \mathsf{dom}(\phi)\}$

2.3 Type interpretation

In order to define subtyping over these types, the idea is to interpret each ground type (i.e., a type that does not contain type variables) as a set of values of our language. Then, subtyping can be defined as set containment over the interpretation of types. Intuitively, each ground type is associated to the set of values having this type: for instance, the base type `True` is interpreted as the singleton containing the constant `true`, while the type `Bool` = `True` $\vee$ `False` is interpreted as the set {`true`, `false`}.

However, this idea becomes subtler when dealing with arrow types. Although an arrow type intuitively corresponds to a function (i.e., a λ-abstraction), interpreting an arrow type as a set of λ-abstractions is problematic as it yields a circular reasoning: determining if a λ-abstraction is in the interpretation of a type requires to define a type system, which in turns needs the subtyping relation that we are trying to build. In order to break this circularity, the interpretation of types is not defined over values of our language but over a domain $\mathcal{D}$ defined below. Note that this does not necessarily invalidate the "types as set of values" intuition, as it is discussed in Castagna and Frisch (2005, Section 2.7).

Regarding polymorphic types, one may think that it is enough to define an interpretation only for ground types, and then define subtyping by stating that a type s is a subtype of a type t if and only if, for every instance of s and t, respectively $s\phi$ and $t\phi$, if $s\phi$ and $t\phi$ are ground types then the interpretation of $s\phi$ is contained in the interpretation of $t\phi$. However, Hosoya et al. (2005) show that this definition of subtyping for non-ground types is hard to decide and has counterintuitive consequences.

Consequently, we need to define an interpretation for all types, and not only ground ones (the interpretation domain $\mathcal{D}$ should account for type variables). A simple model was proposed by Gesbert et al. (2015). We succinctly present it in this section. The reader may refer to (Castagna, 2024, Section 3.3) for more details.

Definition 6 (Interpretation domain Gesbert et al. (2015))**.** *The* interpretation domain $\mathcal{D}$ *is the set of finite terms* d *produced inductively by the following grammar*

$$d ::= c^L \mid (d, d)^L \mid \{(d, \partial), \ldots, (d, \partial)\}^L$$
$$\partial ::= d \mid \Omega$$

where c *ranges over the set* $\mathcal{C}$ *of constants,* L *ranges over finite sets of type variables, and where* Ω *is such that* $\Omega \notin \mathcal{D}$*.*

The elements of $\mathcal{D}$ correspond, intuitively, to (denotations of) the results of the evaluation of expressions, labeled by finite sets of type variables. In particular, in a higher-order language, the results of computations can be functions which, in this model, are represented by sets of finite relations of the form $\{(d_1, \partial_1), \ldots, (d_n, \partial_n)\}^L$, where Ω (which is not in $\mathcal{D}$) can appear in second components to signify that the function fails (i.e., evaluation is stuck) on the corresponding input. This is implemented by using in the second projection the meta-variable ∂ which ranges over $\mathcal{D}_\Omega = \mathcal{D} \cup \{\Omega\}$ (we reserve d to range over $\mathcal{D}$, thus excluding Ω). This constant Ω is used to ensure that $\mathbb{1} \to \mathbb{1}$ is not a supertype of all function types: if we used d instead of ∂, then every well-typed function could be subsumed to $\mathbb{1} \to \mathbb{1}$ and, therefore, every application could be given the type $\mathbb{1}$, independently of its argument as long as this argument is typeable (see Section 4.2 of Frisch et al. (2008)

for details). The restriction to *finite* relations corresponds to the intuition that the denotational semantics of a function is given by the set of its finite approximations, where finiteness is a restriction necessary (for cardinality reasons) to give the semantics to higher-order functions. Finally, the sets of type variables that label the elements of the domain are used to interpret type variables: we interpret a type variable α by the set of all elements that are labeled by α, that is $[\![\alpha]\!] = \{d \mid \alpha \in \mathsf{tags}(d)\}$ (where we define $\mathsf{tags}(c^L) = \mathsf{tags}((d,d')^L) = \mathsf{tags}(\{(d_1,\partial_1),\dots,(d_n,\partial_n)\}^L) = L$).

We define the interpretation $[\![t]\!]$ of a type t so that it satisfies the following equalities, where $\mathcal{P}_{\text{fin}}$ denotes the restriction of the powerset to finite subsets and $\mathbb{B}$ denotes the function that assigns to each base type the set of constants of that type, so that for every constant c we have $c \in \mathbb{B}(\boldsymbol{b}_c)$ (we use $\boldsymbol{b}_c$ to denote the base type of the constant c):

$$\begin{array}{rrr} [\![\mathbb{0}]\!] = \varnothing & [\![\alpha]\!] = \{d \mid \alpha \in \mathsf{tags}(d)\} & [\![t_1 \vee t_2]\!] = [\![t_1]\!] \cup [\![t_2]\!] \\ [\![b]\!] = \mathbb{B}(b) \qquad [\![\neg t]\!] = \mathcal{D}\backslash[\![t]\!] & & [\![t_1 \times t_2]\!] = [\![t_1]\!] \times [\![t_2]\!] \end{array}$$
$$[\![t_1 \to t_2]\!] = \{R \in \mathcal{P}_{\text{fin}}(\mathcal{D} \times \mathcal{D}_\Omega) \mid \forall (d,\partial) \in R.\, d \in [\![t_1]\!] \implies \partial \in [\![t_2]\!]\}$$

Note that, even though we included $\mathbb{1}$ and the intersection $\wedge$ in the syntax of our types (Definition 1), those two can be defined from the other constructors: $\mathbb{1} = \neg\mathbb{0}$ and $t_1 \wedge t_2 = \neg(\neg t_1 \vee \neg t_2)$ (De Morgan's law). It is easy to see that, with these definitions, we have $[\![\mathbb{1}]\!] = \mathcal{D}$ and $[\![t_1 \wedge t_2]\!] = [\![t_1]\!] \cap [\![t_2]\!]$. Thus, it is not necessary to define an interpretation for them.

We cannot take the equations above directly as an inductive definition of $[\![]\!]$ because types are not defined inductively but coinductively. Notice however that the contractivity condition of Definition 1 ensures that the binary relation $\rhd \subseteq \mathcal{T} \times \mathcal{T}$ defined by $t_1 \vee t_2 \rhd t_i$, $t_1 \wedge t_2 \rhd t_i$, $\neg t \rhd t$ is Noetherian. This gives an induction principle[1] on $\mathcal{T}$ that we use combined with structural induction on $\mathcal{D}$ to give the following definition, which validates the equalities above.

Definition 7 (Set-theoretic interpretation of types)**.** *We define a binary predicate $(d : t)$ ("the element d belongs to the type t"), where $d \in \mathcal{D}$ and $t \in \mathcal{T}$, by induction on the pair (d,t) ordered lexicographically. The predicate is defined as follows:*

$$\begin{aligned} (c : b) &= c \in \mathbb{B}(b) \\ (d : \alpha) &= \alpha \in \mathsf{tags}(d) \\ ((d_1,d_2) : t_1 \times t_2) &= (d_1 : t_1) \text{ and } (d_2 : t_2) \\ (\{(d_1,\partial_1),\dots,(d_n,\partial_n)\} : t_1 \to t_2) &= \forall i \in [1..n].\ \text{if } (d_i : t_1) \text{ then } (\partial_i : t_2) \\ (d : t_1 \vee t_2) &= (d : t_1) \text{ or } (d : t_2) \\ (d : \neg t) &= \text{not } (d : t) \\ (\partial : t) &= \text{false} \qquad \textit{otherwise} \end{aligned}$$

[1] In a nutshell, we can do proofs and give definitions by induction on the structure of unions and negations—and, thus, intersections—but arrows, products, and base types are the base cases for the induction.

We define the set-theoretic interpretation $[\![.]\!] : \mathcal{T} \to \mathcal{P}(\mathcal{D})$ *as* $[\![t]\!] = \{d \in \mathcal{D} \mid (d : t)\}$.

2.4 Semantic subtyping

Now that we have a set-theoretic interpretation of types, we can define the subtyping preorder and its associated equivalence relation as follows.

Definition 8 (Subtyping relation)**.** *We define the* subtyping *relation* $\leqslant$ *and the* subtyping equivalence *relation* $\simeq$ *as* $t_1 \leqslant t_2 \stackrel{\text{def}}{\iff} [\![t_1]\!] \subseteq [\![t_2]\!]$ *and* $t_1 \simeq t_2 \stackrel{\text{def}}{\iff} (t_1 \leqslant t_2)$ and $(t_2 \leqslant t_1)$.

This subtyping relation is sometimes referred to as *semantic subtyping*, as it is not defined on the syntax of the type but on its interpretation.

This subtyping relation is decidable, as shown by Frisch (2004) for ground types and extended to support type variables by Castagna and Xu (2011). An explanation of the subtyping algorithm can be found in Castagna (2020).

With this set-theoretic definition of subtyping, usual properties of sets are inherited by subtyping, for instance:

$$t_1 \vee t_2 \simeq t_2 \vee t_1 \qquad t_1 \wedge t_2 \simeq t_2 \wedge t_1 \qquad \text{(commutativity)}$$
$$t \vee t \simeq t \qquad t \wedge t \simeq t \qquad \text{(idempotence)}$$
$$\neg(\neg t) \simeq t \qquad \text{(double complement)}$$
$$t \vee (s_1 \wedge s_2) \simeq (t \vee s_1) \wedge (t \vee s_2) \qquad t \wedge (s_1 \vee s_2) \simeq (t \wedge s_1) \vee (t \wedge s_2) \qquad \text{(distributivity)}$$

For any two type substitutions ϕ_1 and ϕ_2, we write $\phi_1 \simeq \phi_2$ the pointwise subtyping equivalence of ϕ_1 and ϕ_2. An important property of the interpretation above is that subtyping is preserved by type substitutions:

$$\forall t_1, t_2, \phi.\ t_1 \leqslant t_2 \Rightarrow t_1\phi \leqslant t_2\phi$$

However, a naive definition of $\mathsf{vars}(t)$ is not preserved by subtyping equivalence: for instance, we have $\mathbb{1} \simeq \alpha \vee \neg\alpha$, while a purely syntactic definition of $\mathsf{vars}(t)$ would yield $\mathsf{vars}(\mathbb{1}) = \varnothing$ and $\mathsf{vars}(\alpha \vee \neg\alpha) = \{\alpha\}$. In order to avoid this, we define $\mathsf{vars}(t)$ as being the set of *meaningful type variables* in t. This notion has been introduced by Castagna et al. (2016a), where it was noted as $\mathtt{mvar}(t)$, and is defined below.

Definition 9 (Type variables)**.** *The set of type variables of a type* t*, noted* $\mathsf{vars}(t)$*, is the following set of type variables:*

$$\mathsf{vars}(t) \stackrel{\text{def}}{=} \{\alpha \in \mathcal{V} \mid t\{\alpha \rightsquigarrow \mathbb{0}\} \not\simeq t\}$$

With this definition, the set of variables of a type is preserved by subtyping equivalence: $\forall t_1, t_2.\ t_1 \simeq t_2 \Rightarrow \mathsf{vars}(t_1) = \mathsf{vars}(t_2)$.

2.5 Disjunctive Normal Form and type operators

In order to design an algorithmic type system, we need not only to decide subtyping, but we also need a way to compute the domain of a function type, the type resulting from an application, or the type resulting from a projection.

More formally, we want to compute the three type operators below.

Definition 10 (Type operators)**.** *Let $t \leqslant \mathbb{0} \to \mathbb{1}$ and $t' \leqslant \mathbb{1} \times \mathbb{1}$.*

$$\begin{array}{rcll} \mathsf{dom}(t) & \stackrel{\text{def}}{=} & \max\{u \mid t \leqslant u \to \mathbb{1}\} & \\ t \circ s & \stackrel{\text{def}}{=} & \min\{u \mid t \leqslant s \to u\} & \text{where } s \leqslant \mathsf{dom}(t) \\ \boldsymbol{\pi}_1(t') & \stackrel{\text{def}}{=} & \min\{u \mid t' \leqslant u \times \mathbb{1}\} & \\ \boldsymbol{\pi}_2(t') & \stackrel{\text{def}}{=} & \min\{u \mid t' \leqslant \mathbb{1} \times u\} & \end{array}$$

The type $\mathsf{dom}(t)$ is the largest type that can be accepted as argument by a function of type t: it corresponds intuitively to the domain of t. For instance, the domain of the type $(\texttt{Int} \to \texttt{Int}) \wedge (\texttt{Bool} \to \texttt{Bool})$ is $\texttt{Int} \vee \texttt{Bool}$.

The type $t \circ s$ is, intuitively, the result type of t when applied to an argument of type s. This should not be confused with the codomain of t, as the codomain does not depend on the type of the argument: for instance, the type $(\texttt{Int} \to \texttt{Int}) \wedge (\texttt{Bool} \to \texttt{Bool})$ has the codomain $\texttt{Int} \vee \texttt{Bool}$, but when applied to an argument of type $\texttt{Int}$, the result type is $\texttt{Int}$.

Similarly, the types $\boldsymbol{\pi}_1(t)$ and $\boldsymbol{\pi}_2(t)$ respectively correspond to the result type of the left projection and right projection of t.

In order to compute those type operators, we introduce a normal form for types, called *Disjunctive Normal Form (DNF)*. For that, we first define the set $\mathcal{A}$ of atoms as follows:

$$\mathcal{A} = \mathcal{B} \cup \mathcal{V} \cup \{t_1 \times t_2 \mid t_1, t_2 \in \mathcal{T}\} \cup \{t_1 \to t_2 \mid t_1, t_2 \in \mathcal{T}\}$$

Definition 11 (Disjunctive Normal Form)**.** *A* Disjunctive Normal Form (DNF) *is a type $t \in \mathcal{T}$ such that:*

$$t \equiv \bigvee_{i \in I} \left(\bigwedge_{a \in P_i} a \wedge \bigwedge_{a \in N_i} \neg a \right) \qquad \text{with} \qquad \forall i \in I.\ \left(\bigwedge_{a \in P_i} a \wedge \bigwedge_{a \in N_i} \neg a \right) \not\simeq \mathbb{0}$$

where $\equiv$ is the syntactic equality, I is a finite set of indices, and for every $i \in I$, P_i and N_i are finite sets of atoms.

Proposition 1 (Existence of a DNF)**.** *Any type $t \in \mathcal{T}$ is equivalent to a type t' such that t' is a DNF. We say that t' is a DNF of t, and we note it $t \stackrel{\text{DNF}}{\simeq} t'$ (where t' is necessarily a DNF).*

We can compute a DNF of any type t by using the properties of distributivity. Note that DNFs are not unique: a type may be equivalent to several distinct DNFs. For instance, the two DNFs $(\mathtt{Bool} \vee \mathtt{Int}) \times \mathbb{1}$ and $(\mathtt{Bool} \times \mathbb{1}) \vee (\mathtt{Int} \times \mathbb{1})$ are equivalent.

Using DNFs, the operators defined above can be computed as follows.

For a function type (i.e., a type that is a subtype of $\mathbb{0} \to \mathbb{1}$) t such that

$$t \overset{\text{DNF}}{\cong} \bigvee_{i \in I} \left(\bigwedge_{p' \in P'_i} \alpha_{p'} \wedge \bigwedge_{n' \in N'_i} \neg \alpha'_{n'} \wedge \bigwedge_{p \in P_i} (s_p \to t_p) \wedge \bigwedge_{n \in N_i} \neg (s'_n \to t'_n) \right)$$

the first two operators are computed by:

$$\begin{aligned} \mathsf{dom}(t) &= \bigwedge_{i \in I} \bigvee_{p \in P_i} s_p \\ t \circ s &= \bigvee_{i \in I} \left(\bigvee_{\{Q \subsetneq P_i \mid s \not\leqslant \bigvee_{q \in Q} s_q\}} \left(\bigwedge_{p \in P_i \setminus Q} t_p \right) \right) \qquad (\text{for } s \leqslant \mathsf{dom}(t)) \end{aligned}$$

For a pair type (i.e., a type that is a subtype of $\mathbb{1} \times \mathbb{1}$) t such that

$$t \overset{\text{DNF}}{\cong} \bigvee_{i \in I} \left(\bigwedge_{p' \in P'_i} \alpha_{p'} \wedge \bigwedge_{n' \in N'_i} \neg \alpha'_{n'} \wedge \bigwedge_{p \in P_i} (s_p \times t_p) \wedge \bigwedge_{n \in N_i} \neg (s'_n \times t'_n) \right)$$

the last two operators are computed by:

$$\begin{aligned} \boldsymbol{\pi}_1(t) &= \bigvee_{i \in I} \bigvee_{N' \subseteq N_i} \left(\bigwedge_{p \in P_i} s_p \wedge \bigwedge_{n \in N'} \neg s'_n \right) \\ \boldsymbol{\pi}_2(t) &= \bigvee_{i \in I} \bigvee_{N' \subseteq N_i} \left(\bigwedge_{p \in P_i} t_p \wedge \bigwedge_{n \in N'} \neg t'_n \right) \end{aligned}$$

For more details and proofs, the reader may refer to Frisch et al. (2008); Castagna et al. (2022a).

2.6 The tallying problem

One very powerful tool when dealing with set-theoretic types is the tallying algorithm. Intuitively, it is the equivalent of the unification (used in algorithm $\mathcal{W}$, Damas and Milner (1982)), but for a system with subtyping. It has been introduced by Castagna et al. (2015).

Definition 12 (Tallying problem)**.** *Let C be a constraint set, that is, a finite set of pairs of types, and Δ a set of type variables. A type substitution ϕ is a solution to the* tallying problem *(C, Δ), noted $\phi \Vdash_\Delta C$, if $\phi \# \Delta$ and for all $(s,t) \in C$, $s\phi \leqslant t\phi$ holds.*

A constraint set $\{(s_i, t_i)\}_{i \in I}$ may also be noted $\{s_i \mathrel{\dot{\leqslant}} t_i\}_{i \in I}$.

While unification consists, for two types s and t, in finding every type substitution ϕ such that $s\phi$ and $t\phi$ are syntactically equivalent, tallying consists in finding every type substitution ϕ such that $s\phi \leqslant t\phi$. In the definition above, Δ corresponds to the type variables that cannot be substituted ($\phi \# \Delta$).

Whereas a unification problem has either no solution or an infinity of solutions characterized by one principal type substitution, solutions to a tallying problem are characterized by a principal finite set of type substitutions.

Proposition 2 (Principality)**.** *For every tallying problem (C, Δ), the set of all solutions can be characterized by a finite set of type substitutions Φ such that:*

$$\forall \phi \in \Phi.\ \phi \Vdash_\Delta C \qquad \textit{(soundness)}$$
$$\forall \phi''.\ \phi'' \Vdash_\Delta C \Rightarrow \exists \phi \in \Phi.\ \exists \phi'.\ \phi' \# \Delta \text{ and } \phi'' \simeq \phi' \circ \phi \qquad \textit{(completeness)}$$

The problem of characterizing all solutions to a tallying instance is decidable for the system defined in this chapter. An algorithm to characterize all solutions to a tallying instance is proposed in Castagna et al. (2015).

Tallying can be used to determine the type resulting from an application or projection involving polymorphic types. For instance, consider the application $f\ \mathsf{true}$ where f is the polymorphic identity function of type $\alpha \to \alpha$. We cannot use the $\circ$ type operator alone to type this application, as $(\alpha \to \alpha) \circ \mathsf{True}$ is not even defined ($\mathsf{True} \not\leqslant \mathsf{dom}(\alpha \to \alpha)$). Instead, we can consider the constraint $(\alpha \to \alpha,\ \mathsf{True} \to \beta)$ (where β is a fresh type variable that captures the type of the result of the application), and we solve this constraint using tallying (with $\Delta = \varnothing$). The solutions to this tallying instance can be characterized by one principal type substitution $\phi = \{\alpha \rightsquigarrow \mathsf{True} \vee \alpha\ ;\ \beta \rightsquigarrow \mathsf{True} \vee \alpha \vee \beta\}$. By applying ϕ to the type of f, we get the type $\mathsf{True} \vee \alpha \to \mathsf{True} \vee \alpha$, which can now be used to compute the type resulting from the application: $(\mathsf{True} \vee \alpha \to \mathsf{True} \vee \alpha) \circ \mathsf{True} = \mathsf{True} \vee \alpha$ (which can be simplified into the type True by substituting α by $\mathbb{0}$ [2]). Note that the tallying algorithm may return a set Φ of several substitutions, as characterizing all the solutions to a tallying instance may require more than one type substitution, in which

[2] More generally, when a polymorphic type variable α only occurs in covariant positions in a type t, this type t can be simplified by substituting α by $\mathbb{0}$. Likewise, a type variable that only occurs in contravariant positions can be substituted by $\mathbb{1}$. The type we get is an instance of t that is smaller than t.

case we would apply the set of type substitutions Φ to the type of f, as defined in Definition 5.

Chapter 3

Core Language

Contents

This chapter formalizes the language that will be used throughout this manuscript, and presents the challenges to be met in order to type it.

3.1 Syntax

Definition 13 (Syntax of the core language)**.** *The* expressions, values *and* programs *of our core language are the finite terms produced by the following grammar:*

$$\begin{array}{lrcl} \textbf{\textit{Expression}} & e & ::= & c \mid x \mid \lambda x.e \mid e\ e \mid (e,e) \mid \pi_i e \mid (e \in \tau)\,?\,e : e \\ \textbf{\textit{Value}} & v & ::= & c \mid \lambda x.e \mid (v,v) \\ \textbf{\textit{Program}} & p & ::= & \texttt{let}\ x = e\ ;\ p \mid e \end{array}$$

where τ is a test type.

A test type is a ground type that does not feature any arrow except $\mathbb{0} \to \mathbb{1}$.

Definition 14 (Test type)**.** *A* test type *is a type produced by the following grammar:*

$$\textbf{\textit{Test Type}} \quad \tau ::= b \mid \mathbb{0} \to \mathbb{1} \mid \tau \times \tau \mid \tau \vee \tau \mid \tau \wedge \tau \mid \neg\tau \mid \mathbb{0} \mid \mathbb{1}$$

Expressions of our language are λ-expressions with constants $c \in \mathsf{Const}$, variables $x \in \mathsf{Vars}$, applications $e\ e$, λ-abstractions $\lambda x.e$, pairs (e,e), pair projections $\pi_i e$, and type-cases $(e \in \tau)\,?\,e : e$.

A type-case $(e_0 \in \tau)\,?\,e_1 : e_2$ is a dynamic type test that first evaluates e_0 and, then, if e_0 reduces to a value v, evaluates e_1 if v has type τ or e_2 otherwise. Type-cases cannot test arbitrary types but just ground types (i.e., types without type variables occurring in them) where the only arrow type that can occur in them is $\mathbb{0} \to \mathbb{1}$, the type of all functions. This means that type-cases can distinguish functions from other values, but they cannot distinguish, say, functions that have type $\texttt{Int} \to \texttt{Int}$ from those that do not. This restriction is necessary in order to give our language a proper semantics: having full type tests of the form $v \in t$ would entail that we must be able to check at run-time the types of λ-abstractions. It is possible in languages such as CDuce, where λ-abstractions are decorated with their static type, and testing $v \in t$ amounts to checking that t is a supertype of the function's annotation. For us, the problem is more complex as we consider unannotated λ-abstractions: allowing run-time tests of arbitrary arrow types would make the definition of the dynamic semantics to depend on the type inference algorithm. We take the approach of restricting the run-time type tests to whether a value is a function or not. We believe this restriction to be acceptable since in practice, the dynamic languages we want to model can test at run-time whether a value is a function but cannot have more precise information (for the same reason as in our language: functions are not systematically annotated with their type, making it ambiguous to determine at run-time whether a function has a given arrow type or not). For instance, in JavaScript, one can write the following conditional statement:

```
10 if (typeof(f) === "function") {
11   return 42;
12 }
```

As we will see in Chapter 9, not only the type-case expression but also the `typeof` function can be encoded in our language and typed. Also note that the languages Erlang and Elixir allow testing the arity of a function. Adding this feature to our language and type system is possible, but requires a modification of the set-theoretic interpretation of functions (Castagna et al., 2023). Lastly, in some object-oriented languages, functions can be "boxed" in a class or interface which has a nominal type. This is the case, for instance, of Java where lambda expressions are translated to classes[1]. The type system defined in this manuscript does not include nominal subtyping (in particular, the subtyping we define for records in Chapter 7 is structural), but such an extension is discussed in the future work section (Chapter 10).

Programs are sequences of top-level definitions, ending with an expression that can be seen as the main entry point. This notion of program is useful to capture the modularity of our type system. Indeed, top-level definitions are typed sequentially: the type we obtain for a top-level definition is considered definitive and will not be challenged by a later definition. This is where parametric polymorphism becomes useful: for instance, if the identity function $\lambda x.x$ is defined at top-level, it will be

[1] More precisely, a lambda expression is translated to the class that is expected in the current context. Consequently, lambda expressions can only appear in contexts where the target type is unambiguous.

typed $\forall\alpha.\ \alpha \to \alpha$. Then, whenever this identity function is used in a later definition, it will be instantiated as required without having to re-type it. This would not be possible without parametric polymorphism: while the identity can be typed with several intersections, e.g. $(\mathbb{1} \to \mathbb{1}) \wedge (\mathtt{Int} \to \mathtt{Int})$, we cannot know in advance the types of the arguments that will be passed to it in later definitions. For instance, if in a later top-level definition this identity function is applied to a Boolean, it will yield the type $((\mathbb{1} \to \mathbb{1}) \wedge (\mathtt{Int} \to \mathtt{Int})) \circ \mathtt{Bool} = \mathbb{1}$, which is not satisfying.

Definition 15 (Free variables)**.** *The set of* free variables *of an expression e, noted $\mathsf{fv}(e)$, is inductively defined as follows:*

$$
\begin{aligned}
\mathsf{fv}(c) &= \varnothing \\
\mathsf{fv}(x) &= \{x\} \\
\mathsf{fv}(\lambda x.e) &= \mathsf{fv}(e)\backslash\{x\} \\
\mathsf{fv}(e_1 e_2) &= \mathsf{fv}(e_1) \cup \mathsf{fv}(e_2) \\
\mathsf{fv}((e_1, e_2)) &= \mathsf{fv}(e_1) \cup \mathsf{fv}(e_2) \\
\mathsf{fv}(\pi_i e) &= \mathsf{fv}(e) && i = 1, 2 \\
\mathsf{fv}((e_1 \in \tau)\ ?\ e_2 : e_3) &= \mathsf{fv}(e_1) \cup \mathsf{fv}(e_2) \cup \mathsf{fv}(e_3)
\end{aligned}
$$

Notation.
$e_1 \equiv e_2$ ..... *Syntactic equality*
$e_1 \equiv_\alpha e_2$ ..... *Syntactic equivalence modulo α-renaming*
$e_1 \sqsubseteq_\alpha e_2$ ..... *Subterm order modulo α-renaming*

Note that, for the subterm order $\sqsubseteq_\alpha$, a free variable is distinct from a bound variable: we do not have $x \sqsubseteq_\alpha \lambda x.x$ as x is free in the left-hand side expression and bound to a λ-abstraction (and thus subject to α-renaming) in the right-hand side expression.

3.2 Semantics

The reduction semantics for expressions is the one of call-by-value pure λ-calculus with products and with a type-case expression, together with the context rules that implement a leftmost outermost reduction strategy. It is formalized in Figure 3.1, where $\rightsquigarrow$ is a reduction step for expressions and $\rightsquigarrow_{\mathsf{Pr}}$ is a reduction step for programs.

Definition 16 (Capture-avoiding substitution)**.** *The* capture-avoiding substitution *(or just* substitution*) of e' for x in e, noted $e\{e'/x\}$, is defined inductively*

on e as follows:

$$
\begin{aligned}
c\{e'/x\} &= c \\
x\{e'/x\} &= e' \\
y\{e'/x\} &= y && x \neq y \\
(\lambda x.e)\{e'/x\} &= \lambda x.e \\
(\lambda y.e)\{e'/x\} &= \lambda y.(e\{e'/x\}) && x \neq y,\ y \notin \mathsf{fv}(e') \\
(e_1 e_2)\{e'/x\} &= (e_1\{e'/x\})(e_2\{e'/x\}) \\
(e_1, e_2)\{e'/x\} &= (e_1\{e'/x\}, e_2\{e'/x\}) \\
(\pi_i e)\{e'/x\} &= \pi_i(e\{e'/x\}) \\
((e_1 \in \tau)\,?\,e_2 : e_3)\{e'/x\} &= (e_1\{e'/x\} \in \tau)\,?\,e_2\{e'/x\} : e_3\{e'/x\}
\end{aligned}
$$

Capture-avoiding substitutions are up to α-renaming: the condition $y \notin \mathsf{fv}(e')$ in the case of λ-abstractions can be ensured by first performing an α-renaming on $\lambda y.e$.

Reduction rules

$$
\begin{array}{rcl@{\qquad}rcll}
(\lambda x.e)v & \rightsquigarrow & e\{v/x\} & (v \in \tau)\,?\,e_1 : e_2 & \rightsquigarrow & e_1 & \text{if } v \in \tau \\
\pi_1(v_1, v_2) & \rightsquigarrow & v_1 & (v \in \tau)\,?\,e_1 : e_2 & \rightsquigarrow & e_2 & \text{if } v \in \neg\tau \\
\pi_2(v_1, v_2) & \rightsquigarrow & v_2 & \mathtt{let}\, x = v\, ;\, p & \rightsquigarrow_{\mathsf{Pr}} & p\{v/x\} &
\end{array}
$$

Dynamic type test

$$
v \in \tau \Leftrightarrow \mathsf{typeof}(v) \leqslant \tau, \text{ where } \begin{cases} \mathsf{typeof}(c) & = \boldsymbol{b}_c \\ \mathsf{typeof}((v_1, v_2)) & = \mathsf{typeof}(v_1) \times \mathsf{typeof}(v_2) \\ \mathsf{typeof}(\lambda x.e) & = \mathbb{0} \to \mathbb{1} \end{cases}
$$

Evaluation Contexts

$$
\begin{aligned}
E &::= [\,] \mid v\ E \mid E\ e \mid (v, E) \mid (E, e) \mid \pi_i E \mid (E \in \tau)\,?\,e : e \\
P &::= [\,] \mid \mathtt{let}\, x = [\,]\, ;\, p
\end{aligned}
$$

$$
\frac{e \rightsquigarrow e'}{E[e] \rightsquigarrow E[e']} \qquad \frac{e \rightsquigarrow e'}{P[e] \rightsquigarrow_{\mathsf{Pr}} P[e']}
$$

Figure 3.1: Semantics of the source language

The relation $v \in \tau$ determines whether a *value* is of a given type or not and holds true if and only if $\mathsf{typeof}(v) \leqslant \tau$. Note that $\mathsf{typeof}(v)$ maps every λ-abstraction to $\mathbb{0} \to \mathbb{1}$ and, thus, dynamic type tests do not depend on static type inference. This approximation is allowed by the restriction on arrow types in the types used in type-cases. For every value v, we cannot have both $v \in \tau$ and $v \in \neg\tau$, thus the reduction semantics is deterministic. Finally, the reduction semantics for programs

sequentially reduces top-level definitions, together with a context rule that allows reducing the expression of the first definition.

Given a reduction step relation $\rightsquigarrow$, we write $e \rightsquigarrow^* e'$ when there exists a sequence of $\rightsquigarrow$ steps of any length from e to e'. Finally, we write $e \rightsquigarrow^\infty$ when there exists a sequence of $\rightsquigarrow$ steps starting from e and that can be prolonged indefinitely (in this case, we also say that e diverges).

3.3 Challenges

The main specificities of our language are (i) the presence of type-cases, (ii) the fact that λ-abstractions are not explicitly annotated with their type (which is quite uncommon for type systems using set-theoretic types), and (iii) the notion of programs as a list of top-level `let` definitions.

The presence of type-cases has two consequences for the type system: first, if we want them to be typed with precision, our type system should implement *occurrence typing*, and secondly, it enables the expression of overloaded behaviors, which should be captured by our type system.

The fact that λ-abstractions are not explicitly typed adds another challenge, as it requires our type system to be able to infer the type of the parameters of λ-abstractions. It also has significant consequences for the proof of type safety, as it will be discussed in Chapter 4.

Finally, the notion of programs emphasizes the need for modularity: we want our type system to be able to type later definitions without the need to re-type previous definitions.

3.3.1 Occurrence typing

When statically typing the branches of a type-case, we may narrow the type of some variables according to the result of the test. This is sometimes called *occurrence typing* as different occurrences of the same variable may be typed differently.

For instance, for typing the expression $(x \in \texttt{Int})\,?\,x + 1 : \texttt{not}\ x$ where $x : \texttt{Int} \vee \texttt{Bool}$, $+ : \texttt{Int} \to \texttt{Int} \to \texttt{Int}$ and $\texttt{not} : \texttt{Bool} \to \texttt{Bool}$, the type of x must be narrowed to `Int` in the first branch and to `Bool` in the second branch.

Occurrence typing becomes more subtle when the tested expression is more complex, for instance, when it is an application. Consider the following type-case, inspired from the introductory example of Section 1.2: $(\texttt{ToBoolean}\ x \in \texttt{True})\,?\,e_1 : e_2$ where $x : \mathbb{1}$ and $\texttt{ToBoolean} : (\texttt{Truthy} \to \texttt{True}) \wedge (\texttt{Falsy} \to \texttt{False})$. When typing e_1, we can assume that x has type `Truthy`: any value of x not in `Truthy` would have made the application `ToBoolean` x to return a value in `False`, thus taking the second branch. Likewise, we can assume that x has type `Falsy` when typing e_2.

We want to go a step further by not only narrowing the type of variables, but also the type of arbitrary expressions. For example, consider the expression $(fx \in \texttt{Int})\,?\,(fx) + 1 : x$ with $x : \mathbb{1}$ and $f : \mathbb{1} \to \mathbb{1}$. This time, we cannot narrow the type of x when typing the branches of the type-case: whatever type we give

to x, both branches can possibly be selected as the type of the application fx will remain $\mathbb{1}$. However, we can narrow the type of the expression fx: we can assume it has the type `Int` when typing the first branch, and the type $\neg$`Int` when typing the second branch[2].

Our type system must feature this capability to narrow the type of an expression appearing in the branch of a type-case. This will be achieved, in the declarative type system presented in Chapter 4, by adding a typing rule usually referred as the *union-elimination rule* that allows decomposing the type of any expression into several types and to treat each case separately.

3.3.2 Overloaded functions

Type-cases make it possible to define functions with an overloaded behavior. It is the case of the λ-abstraction $\lambda x.$ `(`x`∈Int)?`$x + 1$`:not`$\ x$, which performs a different operation depending on the type of its argument: an addition if it is an integer, and a logical not if it is a Boolean value.

Our type system must be able to capture this overloaded behavior. It does so by using type intersections: the λ-asbtraction above can be typed (`Int` $\to$ `Int`) $\wedge$ (`Bool` $\to$ `Bool`). To derive such an intersection type, the declarative type system (Chapter 4) needs to feature an *intersection-introduction rule*: if, for an expression e (here, our λ-abstraction), we can derive a type t_1 (here `Int` $\to$ `Int`) and a type t_2 (here `Bool` $\to$ `Bool`), then we can derive the type $t_1 \wedge t_2$ ((`Int` $\to$ `Int`) $\wedge$ (`Bool` $\to$ `Bool`)).

3.3.3 Modularity

Modularity is essential for large-scale programming. For instance, when using a library, the type-checker should only rely on the signature of functions and not their implementation: imported functions, whose implementation may not even be available, should not be re-typed. In our language, the need for modularity can be illustrated with the notion of programs. Consider for instance the following program excerpt:

$$\texttt{let}\ x = \lambda x.x\ ;\quad \ldots\quad ;\ \texttt{let}\ z = (x\ 42,\ x\ 24)\ ;\quad \ldots$$

The top-level definition x, which corresponds to the identity function, is used in a later definition z. This top-level definition z may be very distant from the definition of x (we may consider for instance that x is defined in an external library, together with many other functions). At the moment where the definition of x is typed, we do not have access to the future uses that will be made of it. Consequently, even if z could be typed precisely by giving x the type $(42 \to 42) \wedge (24 \to 24)$, there is no way to infer such a type for x as at this moment, it is not known that it will be applied to 42 and 24.

[2]Our language is pure, so if the occurrence of fx in the test reduced to a value v, any other occurrence of fx will reduce to the same value v.

For this reason, our type system needs to feature parametric polymorphism. Giving the top-level definition x the polymorphic type $\alpha \to \alpha$ enables the possibility to instantiate it in different ways later: it will be instantiated with the substitution $\{\alpha \rightsquigarrow 42\}$ when typing the application $x\ 42$, and $\{\alpha \rightsquigarrow 24\}$ when typing the application $x\ 24$.

3.3.4 Type inference

Languages using set-theoretic types usually have explicitly-typed λ-abstractions: this is the case for instance of the initial CDuce type system (Frisch (2004)) and its polymorphic extension (Castagna et al. (2014, 2015)). The λ-abstractions of our language, however, are not explicitly typed, and thus their type must be inferred by the type system (and not just checked).

From the perspective of the declarative type system, the types of the parameters of λ-abstractions do not need to be annotated, they can simply be guessed. Still, this implies significant changes in the proof of type safety. From an algorithmic perspective, however, the types of the parameters must be inferred from the context. This will be done in Chapter 6, with the formalization of an algorithm for reconstructing some type annotations for the algorithmic type system.

A set-theoretic type system for a language where λ-abstractions are not explicitly typed has already been formalized and implemented by Petrucciani (2019). However, his work is different from ours on two aspects: (*i*) though his language features typecases, those are typed naively (no occurrence typing), and (*ii*) only single arrows can be inferred for functions (no overloaded function types).

Part II

Core Formalization

Chapter 4

Declarative Type System

Contents

Section 4.1 formalizes a declarative type system for the core language defined in the previous chapter. Despite its apparent simplicity, this type system is quite expressive, featuring parametric polymorphism, occurrence typing (through a union-elimination rule) and ad hoc polymorphism (through an intersection-introduction rule). Derivations for this type system can take very different shapes, but we can restrict the way to combine them without losing expressivity (these restrictions should not affect which judgments are derivable): this is done in Section 4.2, where we introduce the notion of *canonical derivation*. Finally, Section 4.3 focuses on proving a type safety theorem for this type system.

4.1 Formalization

4.1.1 Polymorphic and monomorphic types

The set-theoretic types presented in Chapter 2 do not differentiate polymorphic type variables (i.e., those that can be instantiated) and monomorphic ones (i.e., those that cannot be instantiated). However, this distinction now becomes important, as our type system must feature parametric polymorphism and, thus, is in charge of performing instantiations.

The classical algorithm of Hindley-Milner infers for a closed expression (i.e., an expression with no free variable) a type scheme $\forall\vec{\alpha}.t$ (with $\vec{\alpha} \subseteq \mathsf{vars}(t)$). This type scheme is a syntactic object that denotes an infinite set of types $t\phi$ for all type substitutions ϕ such that $\mathsf{dom}(\phi) \subseteq \vec{\alpha}$. The algorithm is allowed to specialize (instantiate) the type of an expression to make the typing succeed. For instance, if $f : \forall\alpha.\alpha \to \alpha$, then $f\ 42$ is well typed and has type `Int` since $42 : \texttt{Int}$ and $(\alpha \to \alpha)\{\alpha \rightsquigarrow \texttt{Int}\} = \texttt{Int} \to \texttt{Int}$. Notice however that during type inference, "partially generalized" types such as $\forall\alpha.\ \beta \to \alpha$ may occur. In such a type scheme, it is not allowed to substitute β by an arbitrary type.

In this manuscript, we will use another approach, by partitioning the countably infinite set $\mathcal{V}$ of type variables into two countably infinite sets: the set $\mathcal{V}_P$ of polymorphic type variables and the set $\mathcal{V}_M$ of monomorphic type variables.

Notation.
$\mathcal{V}$ ..... *Set of type variables* ($\mathcal{V} = \mathcal{V}_P \cup \mathcal{V}_M$)
$\mathcal{V}_P$ ..... *Set of polymorphic type variables, ranged over by* α, β, *or* γ
$\mathcal{V}_M$ ..... *Set of monomorphic type variables, ranged over by* $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$, *or* $\boldsymbol{\gamma}$

Note that, from this point, α no longer ranges over $\mathcal{V}$, but only over $\mathcal{V}_P$. As we will see in the next section, a polymorphic type variable α can be freely substituted by the type system, while a monomorphic type variable $\boldsymbol{\alpha}$ (with bold font) cannot.

Definition 17 (Monomorphic types)**.** *The set of* monomorphic types $\mathcal{T}_M$ *is the set of types that do not contain polymorphic type variables, that is:*

$$\mathcal{T}_M \stackrel{\text{def}}{=} \{t \in \mathcal{T} \mid \mathsf{vars}(t) \subseteq \mathcal{V}_M\}$$

As before, the metavariables t and s are used to range over the set of types. When ranging over the set of monomorphic types specifically, we use the metavariables $\mathbf{u}$ and $\mathbf{v}$.

Note that the terms *polytypes* and *monotypes* can be found (albeit inconsistently) in the literature: in particular, Milner (1978) uses the latter to denote types with no type variables and the former when he wishes to imply that a type may, or does, contain a variable. We avoided using them to prevent any confusion with our *monomorphic types* we defined just above. While our *types* are indeed polytypes, our *monomorphic types* are not monotypes: monotypes do not have type variables while monomorphic types may have type variables, though only monomorphic ones. So we used instead *types* (which may have type variables), *ground types* (which cannot have any type variable), and *monomorphic types* (which may have monomorphic type variables, only).

For what concerns subtyping and type operators, there is no difference between monomorphic and polymorphic type variables: both $\boldsymbol{\alpha}$ and α are type variables in

$\mathcal{V}$ and thus have the same interpretation in the set-theoretic type theory presented in Chapter 2. They thus play the same role for subtyping and in every type operator defined in Chapter 2: for instance, $\mathsf{vars}(t)$ can contain both monomorphic and polymorphic type variables.

Our choice of using two disjoint sets for polymorphic and monomorphic type variables, instead of the classical approach of using type schemes $\forall\alpha_1...\alpha_n.t$, is justified by two reasons. First, type schemes are expected to be equivalent modulo renaming of their bounded type variables $\alpha_1, \ldots, \alpha_n$. In our case however, we do not want polymorphic type variables to be freely renamed because of the use, in the algorithmic type system of Chapter 5, of external annotations containing explicit substitutions over some polymorphic type variables of the context. Secondly, introducing type schemes would require redefining many of the usual set-theoretic type-related definitions, such as the subtyping relation $\leqslant$, and the type operators for application $\circ$ and projections π_i. Instead, we obtain a more streamlined theory by making subtyping and these operators ignore whether a type variable is polymorphic or monomorphic in the current context and by explicitly performing instantiations in the type system when required.

With this new distinction between monomorphic and polymorphic type variables, we introduce different symbols for ranging over substitutions.

Notation.

$\phi \in \mathcal{V} \to \mathcal{T}$	*General substitution, from type variables to types*
$\psi \in \mathcal{V}_M \to \mathcal{T}_M$	*Substitution from monomorphic variables to monomorphic types*
$\sigma \in \mathcal{V}_P \to \mathcal{T}$	*Substitution from polymorphic type variables to types*
$\rho \in \mathcal{V}_P \to \mathcal{V}_P$	*Renaming (injective substitution) of polymorphic type variables*

Moreover, we use Φ, Ψ and Σ respectively to range over sets of substitutions $\mathcal{V} \to \mathcal{T}$, sets of substitutions $\mathcal{V}_M \to \mathcal{T}_M$ and sets of substitutions $\mathcal{V}_P \to \mathcal{T}$.

4.1.2 Type system

The core of our type system is a classic Hindley-Milner system with first order polymorphism: a program is a list of let-bindings that define polymorphic functions; these are typed by inferring a type for the expressions that define them, this type is then generalized, yielding a prenex polymorphic type for the function. As usual, the deduction of the type of each of these expressions is performed in a type environment that records the generic types for the previously-defined polymorphic functions, and the type system can instantiate these types differently for each use of the polymorphic functions in the expression. The novelty of our system is that when deducing the types of the expressions that define and use polymorphic functions, the type system can use not only instantiations of polymorphic types (rule [Inst] in Figure 4.1), but also intersection and union types. More precisely, the type system can decide to use the classic rules of intersection-introduction (rule $[\wedge]$) and union-elimination

$$[\textsc{Const}]\ \frac{}{\Gamma \vdash c : \boldsymbol{b}_c} \qquad [\textsc{Var}]\ \frac{}{\Gamma \vdash x : \Gamma(x)}$$

$$[{\to}\mathrm{I}]\ \frac{\Gamma, x : \mathbf{u} \vdash e : t}{\Gamma \vdash \lambda x.e : \mathbf{u} \to t} \qquad [{\to}\mathrm{E}]\ \frac{\Gamma \vdash e_1 : t_1 \to t_2 \qquad \Gamma \vdash e_2 : t_1}{\Gamma \vdash e_1 e_2 : t_2}$$

$$[{\times}\mathrm{I}]\ \frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e_1, e_2) : t_1 \times t_2} \qquad [{\times}\mathrm{E}_1]\ \frac{\Gamma \vdash e : t_1 \times t_2}{\Gamma \vdash \pi_1 e : t_1} \qquad [{\times}\mathrm{E}_2]\ \frac{\Gamma \vdash e : t_1 \times t_2}{\Gamma \vdash \pi_2 e : t_2}$$

$$[\mathbb{0}]\ \frac{\Gamma \vdash e : \mathbb{0}}{\Gamma \vdash (e{\in}\tau)\,?\,e_1 : e_2 : \mathbb{0}} \qquad [{\in}_1]\ \frac{\Gamma \vdash e : \tau \qquad \Gamma \vdash e_1 : t_1}{\Gamma \vdash (e{\in}\tau)\,?\,e_1 : e_2 : t_1} \qquad [{\in}_2]\ \frac{\Gamma \vdash e : \neg\tau \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e{\in}\tau)\,?\,e_1 : e_2 : t_2}$$

$$[\vee]\ \frac{\Gamma \vdash e' : s \qquad \Gamma, x : s \wedge \mathbf{u} \vdash e : t \qquad \Gamma, x : s \wedge \neg\mathbf{u} \vdash e : t}{\Gamma \vdash e\{e'/x\} : t}$$

$$[\wedge]\ \frac{\Gamma \vdash e : t_1 \qquad \Gamma \vdash e : t_2}{\Gamma \vdash e : t_1 \wedge t_2} \qquad [\textsc{Inst}]\ \frac{\Gamma \vdash e : t}{\Gamma \vdash e : t\sigma} \qquad [\leqslant]\ \frac{\Gamma \vdash e : t}{\Gamma \vdash e : t'}\ t \leqslant t'$$

Figure 4.1: Declarative Type System

(rule $[\vee]$). As we will see, the combination of the union-elimination with the rules of type-cases given in Figure 4.1 constitutes the essence of narrowing and occurrence typing.

Definition 18 (Type environment)**.** *A* type environment Γ *is a finite mapping from* Vars *to* $\mathcal{T}$*. We note* $(\Gamma, x : t)$ *the extension of* Γ *that maps* x *to* t*, with the condition that* x *is not already in the domain of* Γ*.*

Notation.

$\mathsf{dom}(\Gamma)$ ..... *Domain of* Γ:	$\mathsf{dom}(\Gamma) \stackrel{\text{def}}{=} \{x \mid (x : t) \in \Gamma\}$
$\mathsf{vars}(\Gamma)$	$\stackrel{\text{def}}{=} \bigcup_{x \in \mathsf{dom}(\Gamma)} \mathsf{vars}(\Gamma(x))$
$\phi \# \Gamma$	$\Leftrightarrow\ \mathsf{dom}(\phi) \cap \mathsf{vars}(\Gamma) = \varnothing$
$\Gamma\big\vert_S$ ..... *Restriction of* Γ *to the domain* S	
$\Gamma\phi$	$\stackrel{\text{def}}{=} \{(x : t\phi) \mid (x : t) \in \Gamma\}$
$\Gamma\Phi$	$\stackrel{\text{def}}{=} \{(x : t\Phi) \mid (x : t) \in \Gamma\}$

Our type system for expressions is given in full in Figure 4.1. Constants and variables are typed by the corresponding axioms [Const] and [Var]. The arrow and product constructor have introduction and elimination rules. Notably, in the case of rule $[{\to}\mathrm{I}]$ the type of the argument is monomorphic. Also, the variable x must not already be in $\mathsf{dom}(\Gamma)$ in order for $\Gamma, x : \mathbf{u}$ to be defined, but α-renaming can be applied implicitly on expressions whenever needed. The rules for intersection

([$\wedge$]) and subtyping ([$\leqslant$]) are the classical ones, and so is the rule for instantiation ([Inst]) where σ denotes a substitution from polymorphic variables to types. The type-case construction is handled by three rules: [$\mathbb{0}$]; [$\in_1$]; [$\in_2$] . Rule [$\mathbb{0}$] handles the case where the tested expression is known to have the empty type. The other two are symmetric and handle the case when the tested expression is known to have either the type τ or its negation, in which case the corresponding branch is typed. These rules work together with Rule [$\vee$], which we now describe in detail.

At first sight, the formulation of rule [$\vee$] seems odd, since the $\vee$ connector does not appear in it. To understand it, consider the classic union-elimination rule by MacQueen et al. (1986):

$$[\vee\text{E}]\ \frac{\Gamma \vdash e' : s_1 \vee s_2 \quad \Gamma, x : s_1 \vdash e : t \quad \Gamma, x : s_2 \vdash e : t}{\Gamma \vdash e\{e'/x\} : t}$$

Rule [$\vee$E] types an expression that contains occurrences of an expression e' that has a union type $s_1 \vee s_2$; the rule substitutes in this expression *some* occurrences of e' by the variable x yielding an expression e, and then types e first under the hypothesis that x has type s_1 and then under the hypothesis that x has type s_2. If both succeed, then the common type is returned for the expression at issue. This rule, together with the rules for type-cases, allows the system to perform occurrence typing. For instance, consider the expression $\texttt{(}fy\texttt{$\in$Int)?(}fy\texttt{)+1:false}$, in the context where f has type $\mathbb{1} \to \mathbb{1}$ and y is of type $\mathbb{1}$. This expression can be typed thanks to the [$\vee$E] rule, by considering the subexpression fy. This subexpression has type $\mathbb{1}$ which can be seen as the union type $\mathbb{1} \simeq \texttt{Int} \vee \neg\texttt{Int}$. We can then replace x for fy and type, using [$\in_1$], the expression $\texttt{(}x\texttt{$\in$Int)?}x\texttt{+1:false}$, with $x : \texttt{Int}$. This yields a type Int (Rule [$\in_1$] ignores the second branch) and by subtyping, the expression has type $\texttt{Int} \vee \texttt{False}$. Likewise for the choice $x : \neg\texttt{Int}$, using Rule [$\in_2$] the second branch has type False and therefore $\texttt{Int} \vee \texttt{False}$ (again via subtyping). The whole expression has thus the desired type $\texttt{Int} \vee \texttt{False}$.

A key element is that the [$\vee$E] rule guessed how to split the type $\mathbb{1}$ of fy into $\texttt{Int} \vee \neg\texttt{Int}$. In a non-polymorphic setting, this is perfectly fine. But in a type system featuring polymorphism, particular care must be taken when introducing (fresh) type variables. As it is stated, Rule [$\vee$E] could choose to split, say, $\mathbb{1}$ into a union $\alpha \vee \neg\alpha$, with α a polymorphic type variable. If so, then the rule becomes unsound. As a matter of fact, the premises of the [$\vee$E] behave as in rule [$\to$I], in that they introduce in the typing environment a fresh type whose variables must *not* be instantiated. In our example, however, in one premise, the rule introduces $x : \alpha$ in the typing environment which can, for instance, be instantiated by the [Inst] rule. In the second premise, it introduces $x : \neg\alpha$ which can also be instantiated in a *completely different way*. In other words, the correlation between the two occurrences of the same variable α is lost, which amounts to commuting the (implicit) universal quantification with the $\vee$ type connective, yielding a non-prenex polymorphic type $(\forall\alpha.\alpha) \vee (\forall\alpha.\neg\alpha)$. To avoid this unsound situation, we need to ensure that when a type is split between two components of a union, no polymorphic

variable is introduced. This is achieved by the $[\vee]$ rule which requires the type s of e' to be split as $s \equiv (s \wedge \mathbf{u}) \vee (s \wedge \neg\mathbf{u})$ (here is our hidden union).

Finally, note that this $[\vee]$ rule is only sound because our language is pure. In the presence of side effects, two different occurrences of the same subexpression could yield two different results. The support for side effects is not treated in this manuscript: it is a future work that will be briefly discussed in Chapter 10.

The top-level definitions of a program are typed sequentially by two specific rules:

$$[\textsc{TopLevel-Expr}]\ \frac{\Gamma \vdash e : t}{\Gamma \vdash_{\mathsf{Pr}} e : t\phi}\ \phi\#\Gamma \qquad [\textsc{TopLevel-Let}]\ \frac{\Gamma \vdash_{\mathsf{Pr}} e : t \qquad \Gamma, x : t \vdash_{\mathsf{Pr}} p : t'}{\Gamma \vdash_{\mathsf{Pr}} \mathtt{let}\, x = e\, ;\, p\ : t'}$$

where, intuitively, ϕ substitutes monomorphic type variables in $\mathsf{vars}(t)$ by fresh polymorphic ones. The typing rule [TopLevel-Expr] generalizes the type t of expression e, converting its monomorphic type variables into polymorphic ones. Intuitively, this corresponds to the [Gen] rule of Hindley-Milner type systems, except it is only applicable on top-level definitions. The generalized type t is integrated in the environment by the [TopLevel-Let] rule, which then proceeds to type the rest of the program.

Note that the substitution ϕ in [TopLevel-Expr] can be any substitution: it is not required for it to map monomorphic type variables to fresh polymorphic ones. It can map any type variable (monomorphic or polymorphic) to an arbitrary type, as long as the guard condition $\phi\#\Gamma$ is satisfied. In a Hindley-Milner system, this would amount to successively apply a generalization and an instantiation:

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash_{\mathsf{Pr}} e : t\phi}\ \phi\#\Gamma \quad \leftrightarrow \quad [\textsc{Inst-HM}]\ \frac{[\textsc{Gen-HM}]\ \dfrac{\Gamma \vdash_{\mathtt{HM}} e : t}{\Gamma \vdash_{\mathtt{HM}} e : \forall\vec{\alpha}.t}\ \vec{\alpha}\#\Gamma}{\Gamma \vdash_{\mathtt{HM}} e : t\phi}\ (\forall\vec{\alpha}.t) \sqsubseteq t\phi$$

where $t_1 \sqsubseteq t_2$ means that t_1 is more general than t_2, that is, t_1 can be transformed into t_2 by substituting universally-quantified type variables.

Note that, in our type system, generalization only takes place in "TopLevel" rules: no rule in the type system for expressions (Figure 4.1) allows the generalization of a type variable. However, restricting generalization to occur only at top-level is not a limitation since intersection types are more powerful than Hindley-Milner polymorphism. For instance, let us consider a local definition of the identity function $\lambda x.x$, and assume that it is given a generalized type $\alpha \to \alpha$. This polymorphic type allows the identity function to be instantiated later. However, since it is a local definition, all these instantiations are known. Thus, we can consider the set of substitutions $\{\sigma_i\}_{i\in I}$ applied to this identity function, where each σ_i is of the form $\{\alpha \rightsquigarrow t_i\}$. These instantiations can then be eliminated from the derivation by typing the identity function with the type $\bigwedge_{i\in I}(t_i \to t_i)$ using an intersection rule. This type being a subtype of every instantiation $(\alpha \to \alpha)\sigma_i$, instantiations can thus be replaced by a subsumption rule.

Still, as discussed in Chapter 3 (Section 3.1), generalization is of practical importance since it is necessary to the modularity of type-checking. However, for

this purpose, it is enough to generalize at top-level. The reason why we restrict generalization at top-level will be explained in Section 4.2.

Our type system is safe: if $\varnothing \vdash_{\mathsf{Pr}} p : \tau$, then either p diverges or $p \rightsquigarrow_{\mathsf{Pr}} v$ with $v \in \tau$. This will be proved in Section 4.3.

4.2 Canonical typing derivations

Derivations for the declarative type system can have many shapes. In particular, the union-elimination rule $[\vee]$ can be used anywhere in the derivation and changes the expression to type by performing a substitution on it. Other non-structural rules such as $[\wedge]$, $[\leqslant]$ and [Inst] can also be applied anywhere in the derivation (note that we cannot even say that the rule $[\wedge]$ is driven by the syntax of the output type, since our types are considered modulo semantic equivalence). In this section, we define canonical derivations that restrict the use of those rules. In addition to being a first step towards an algorithmic type system, some of these restrictions will be used in Section 4.3, whose goal is to establish a type safety theorem.

Terminology (Derivation trees)**.** *For a derivation tree D, we use the following terminology:*

Node *A node in D corresponds to an application of a typing rule in D.*

Conclusion *The conclusion of a node N is the judgment it derives.*

Premise *An hypothesis of a node N is called a premise of N.*

Path *A path π is a sequence of non-negative natural numbers $n_1; n_2; ...; n_k$ that describes the position of a node relatively to the root of D. For instance, the path $0; 1$ denotes the second premise of the first premise of the root of D. The empty path is noted ε. The node in D at path π is denoted by $D(\pi)$. The set of valid paths in D is denoted by $\mathsf{dom}(D)$.*

Segment *A segment of D denotes a sequence of nodes of D such that the node at position $i + 1$ is a premise of the node at position i.*

Definition premise *Given a $[\vee]$ node N of D, the definition premise of N denotes the first premise of N.*

Body premise *Given a $[\vee]$ node N of D, a body premise of N is a premise of N that is not its first premise.*

Notation.

$|D|$ ..... *Number of nodes in the derivation tree D*
$|D|_{[\mathrm{R}]}$ ..... *Number of* [R] *nodes in the derivation tree D*
$(\mathcal{O}_1, ..., \mathcal{O}_n)_{\mathsf{lex}}$ ..... *Lexicographic order based on orders $\mathcal{O}_1$, ..., $\mathcal{O}_n$*

4.2.1 Alternative form of the declarative type system

To define our canonical typing derivations, we first need to slightly modify some rules of the declarative type system. In order to avoid confusions, this modified declarative type system will produce judgments of the form $\Gamma \models e : t$ (notice the $\models$ turnstile).

Definition 19 (Partition of a type)**.** *Let t be a type. The set of* partitions *of t, noted $\mathsf{Part}(t)$, is the set of all sets $\{t_i\}_{i\in I}$ such that: (i) $\bigvee_{i\in I} t_i \simeq t$, (ii) $\forall i \in I.\ t_i \not\simeq \mathbb{0}$, and (iii) $\forall i, j \in I.\ i \neq j \Rightarrow t_i \wedge t_j \simeq \mathbb{0}$.*

First, we modify the [Var] rule so that it can perform a renaming of the polymorphic type variables in $\Gamma(x)$:

$$[\textsc{Var}]\ \frac{}{\Gamma \models x : \Gamma(x)\rho}$$

This new [Var] rule is derivable in the initial declarative type system by composing a [Var] rule and an [Inst] rule. Still, allowing the [Var] rule to perform a renaming of polymorphic type variables is useful, as it allows decorrelating types without resorting to the [Inst] rule. For instance, consider the pair (x, x) with x having the type $\alpha \to \alpha$. While this pair could be typed $(\alpha \to \alpha) \times (\alpha \to \alpha)$, this type does not allow instantiating the left-hand side and right-hand side of the product independently. A better type would be $(\alpha \to \alpha) \times (\beta \to \beta)$, and with this new [Var] rule, it can be derived without having to use an [Inst] rule. In this way, the [Inst] rule can be reserved to cases that require non-trivial instantiations (i.e., not just renamings). Note that the necessity of performing this renaming comes from the fact that we do not use type schemes $\forall\vec{\alpha}.\ t$, where renaming of the type variables in $\vec{\alpha}$ can be performed implicitly anywhere.

Secondly, we use a $[\wedge]$ rule of multiple arity instead of a binary one:

$$[\wedge]\ \frac{(\forall i \in I) \quad \Gamma \models e : t_i}{\Gamma \models e : \bigwedge_{i\in I} t_i}\ I \neq \varnothing$$

This allows combining successive $[\wedge]$ rule applications into one $[\wedge]$ rule, making the notion of canonical derivation easier to formulate. This new $[\wedge]$ rule is admissible in the $\vdash$ system: it can be replaced by several consecutive $[\wedge]$ nodes.

Similarly, we use a $[\vee]$ rule of multiple arity:

$$[\vee]\ \frac{\Gamma \models e' : s \qquad (\forall i \in I) \quad \Gamma, x : s \wedge \mathbf{u}_i \models e : t}{\Gamma \models e\{e'/x\} : t}\ \{\mathbf{u}_i\}_{i\in I} \in \mathsf{Part}(\mathbb{1})$$

This allows combining successive $[\vee]$ rule applications substituting the same subexpression into one $[\vee]$ rule, making the notion of canonical derivation easier

to formulate. Again, this new $[\vee]$ rule is admissible. For instance, the following derivation:

$$[\vee]\ \frac{\dfrac{A}{\Gamma \vDash e' : s} \quad \dfrac{B}{\Gamma, y : s \wedge \mathbf{u}_1 \vDash e : t} \quad \dfrac{C}{\Gamma, y : s \wedge \mathbf{u}_2 \vDash e : t} \quad \dfrac{D}{\Gamma, y : s \wedge \mathbf{u}_3 \vDash e : t}}{\Gamma \vDash e\{e'/x\} : t}$$

can be transformed to apply the binary $[\vee]$ rule twice, first performing the decomposition $\{\mathbf{u}_1, \neg\mathbf{u}_1\}$, and then performing the decomposition $\{\mathbf{u}_2, \neg\mathbf{u}_2\}$ (since $\mathbf{u}_3$ is equivalent to $\neg\mathbf{u}_1 \wedge \neg\mathbf{u}_2$):

$$[\vee]\ \frac{\dfrac{A}{\Gamma \vDash e' : s} \quad \dfrac{B}{\Gamma, x : s \wedge \mathbf{u}_1 \vDash e : t} \quad \dfrac{X}{\Gamma, x : s \wedge \neg\mathbf{u}_1 \vDash e : t}}{\Gamma \vDash e\{e'/x\} : t}$$

with X being the following derivation:

$$[\vee]\ \frac{[\text{Var}] \quad \dfrac{C\{y/x\}}{\Gamma, x : s \wedge \neg\mathbf{u}_1, y : s \wedge \mathbf{u}_2 \vDash e\{y/x\} : t} \quad \dfrac{D\{y/x\}}{\Gamma, x : s \wedge \neg\mathbf{u}_1, y : s \wedge \mathbf{u}_3 \vDash e\{y/x\} : t}}{\Gamma, x : s \wedge \neg\mathbf{u}_1 \vDash (e\{y/x\})\{x/y\} : t}$$

This construction can be generalized for a partition of $\mathbb{1}$ of any finite cardinality.

Lastly, we distinguish variables that are introduced by a $[\to\text{I}]$ node from variables introduced by a $[\vee]$ node:

Terminology.

Lambda variable *A lambda variable is a variable introduced by a λ-abstraction. The set of lambda variables is denoted by Vars_λ, and ranged over by x, y, and z.*

Binding variable *A binding variable is a variable introduced by a $[\vee]$ rule. The set of binding variables is denoted by Vars_B, and ranged over by x, y, and z.*

Variable *When not specified, a variable can be either a binding variable or a lambda variable. The set of variables is denoted by Vars, and ranged over by $\boldsymbol{x}$, $\boldsymbol{y}$, and $\boldsymbol{z}$.*

The sets Vars_λ and Vars_B form a partition of the set of variables Vars. Notice that from this point, the symbol x no longer ranges over Vars, but only over Vars_λ. When needed, we use the notation $\boldsymbol{x}$ to range over both binding variables and lambda variables (for instance we can write $\forall \boldsymbol{x} \in \mathsf{dom}(\Gamma).\ \Gamma(\boldsymbol{x}) \not\simeq \mathbb{0}$).

The syntax of expressions and the rules of the type system are changed accordingly, as defined in Figure 4.2 which presents the full alternative declarative type system. This new system is equivalent to the initial type system: the combination of both $[\text{VAR}_\lambda]$ and $[\text{VAR}_\vee]$ gives the previous [VAR] rule.

$$\begin{array}{lrcl} \textbf{Expression} & e & ::= & c \mid x \mid \mathsf{x} \mid \lambda x.e \mid e\ e \mid (e,e) \mid \pi_i e \mid (e{\in}\tau)\,?\,e : e \\ \textbf{Value} & v & ::= & c \mid \lambda x.e \mid (v,v) \end{array}$$

$$[\textsc{Const}]\ \frac{}{\Gamma \vDash c : \boldsymbol{b}_c} \qquad [\textsc{Var}_\lambda]\ \frac{}{\Gamma \vDash x : \Gamma(x)\rho} \qquad [\textsc{Var}_\vee]\ \frac{}{\Gamma \vDash \mathsf{x} : \Gamma(\mathsf{x})\rho}$$

$$[{\to}\mathrm{I}]\ \frac{\Gamma, x : \mathbf{u} \vDash e : t}{\Gamma \vDash \lambda x.e : \mathbf{u} \to t} \qquad [{\to}\mathrm{E}]\ \frac{\Gamma \vDash e_1 : t_1 \to t_2 \qquad \Gamma \vDash e_2 : t_1}{\Gamma \vDash e_1 e_2 : t_2}$$

$$[{\times}\mathrm{I}]\ \frac{\Gamma \vDash e_1 : t_1 \qquad \Gamma \vDash e_2 : t_2}{\Gamma \vDash (e_1, e_2) : t_1 \times t_2} \qquad [{\times}\mathrm{E}_1]\ \frac{\Gamma \vDash e : t_1 \times t_2}{\Gamma \vDash \pi_1 e : t_1} \qquad [{\times}\mathrm{E}_2]\ \frac{\Gamma \vDash e : t_1 \times t_2}{\Gamma \vDash \pi_2 e : t_2}$$

$$[\mathbb{0}]\ \frac{\Gamma \vDash e : \mathbb{0}}{\Gamma \vDash (e{\in}\tau)\,?\,e_1 : e_2 : \mathbb{0}} \qquad [\in_1]\ \frac{\Gamma \vDash e : \tau \qquad \Gamma \vDash e_1 : t_1}{\Gamma \vDash (e{\in}\tau)\,?\,e_1 : e_2 : t_1} \qquad [\in_2]\ \frac{\Gamma \vDash e : \neg\tau \qquad \Gamma \vDash e_2 : t_2}{\Gamma \vDash (e{\in}\tau)\,?\,e_1 : e_2 : t_2}$$

$$[\vee]\ \frac{\Gamma \vDash e' : s \qquad (\forall i \in I) \quad \Gamma, \mathsf{x} : s \wedge \mathbf{u}_i \vDash e : t}{\Gamma \vDash e\{e'/\mathsf{x}\} : t}\ \{\mathbf{u}_i\}_{i\in I} \in \mathsf{Part}(\mathbb{1})$$

$$[\wedge]\ \frac{(\forall i \in I) \quad \Gamma \vDash e : t_i}{\Gamma \vDash e : \bigwedge_{i\in I} t_i}\ I \neq \varnothing \qquad [\textsc{Inst}]\ \frac{\Gamma \vDash e : t}{\Gamma \vDash e : t\sigma} \qquad [\leqslant]\ \frac{\Gamma \vDash e : t}{\Gamma \vDash e : t'}\ t \leqslant t'$$

Figure 4.2: Alternative Declarative Type System

Terminology.

Structural rules [Const] [Var_λ] [$\to$I] [$\to$E] [$\times$I] [$\times\mathrm{E}_1$] [$\times\mathrm{E}_2$] [$\mathbb{0}$] [$\in_1$] [$\in_2$]

Non-structural rules [$\mathrm{Var}_\vee$] [$\vee$] [$\wedge$] [Inst] [$\leqslant$]

The rules [Const], [Var_λ], [$\to$I], [$\to$E], [$\times$I], [$\times\mathrm{E}_1$], [$\times\mathrm{E}_2$], [$\mathbb{0}$], [$\in_1$] and [$\in_2$] are called *structural rules* as their use is guided by the structure of the expression to type, each of them allowing to type a specific syntactic construction. In particular, note that the rule [$\mathrm{Var}_\vee$] is not considered structural as binding variables do not appear in the initial expression: they are only introduced in the derivation by [$\vee$] rules.

Definition 20 (Ground expression)**.** *An expression e is a* ground expression *if and only if e does not contain any binding variable:* $\mathsf{fv}(e) \cap \mathit{Vars}_B = \varnothing$.

All the proofs in the next sections and chapters will use the $\vDash$ declarative type system, which is equivalent to the $\vdash$ type system:

Proposition 3. *For every ground expression e, type environment Γ and type t:*

$$\Gamma \vdash e : t \Leftrightarrow \Gamma \models e : t$$

Proof. Both directions are proved by structural induction on the derivation. The $\Rightarrow$ direction is trivial. The $\Leftarrow$ direction follows from the rules' admissibility outlined at the beginning of this section. □

We introduce a new binary relation $\lhd$ over types. Intuitively, $t_1 \lhd t_2$ means that the type t_1 is *better* than the type t_2. More precisely, it means that there exists instances of t_1 whose conjunction is a subtype of t_2. Another way to characterize the relation $\lhd$ is by saying that $t_1 \lhd t_2$ if and only if, for every Γ and e, a derivation of $\Gamma \models e : t_1$ can be transformed into a derivation of $\Gamma \models e : t_2$ by applying some [Inst], $[\wedge]$ and $[\leqslant]$ rules.

Definition 21. *We define the binary relation $\lhd$ over types as follows:*

$$\forall t_1, t_2.\ t_1 \lhd t_2 \Leftrightarrow \exists \Sigma.\ t_1\Sigma \leqslant t_2$$

Proposition 4. *The relation $\lhd$ is a preorder (i.e., it is reflexive and transitive).*

Proof. Reflexivity is trivial. Transitivity is proved below.

Let t_1, t_2, and t_3 be three types such that $t_1 \lhd t_2$ and $t_2 \lhd t_3$. Let Σ_1 and Σ_2 be two sets of substitutions such that $t_1\Sigma_1 \leqslant t_2$ and $t_2\Sigma_2 \leqslant t_3$. By posing $\Sigma = \{\sigma_2 \circ \sigma_1 \mid \sigma_1 \in \Sigma_1, \sigma_2 \in \Sigma_2\}$, we get $t_1\Sigma \leqslant \bigwedge_{\sigma_1\in\Sigma_1,\sigma_2\in\Sigma_2}(t_1\sigma_1)\sigma_2 \leqslant \bigwedge_{\sigma_2\in\Sigma_2}(\bigwedge_{\sigma_1\in\Sigma_1} t_1\sigma_1)\sigma_2 \leqslant (t_1\Sigma_1)\Sigma_2 \leqslant t_3$. Thus, $t_1 \lhd t_3$. □

Definition 22. *For every preorder $\mathcal{O}$ over types, we define the preorder $\mathcal{O}$ over type environments as follows:*

$$\forall \Gamma_1, \Gamma_2.\ (\Gamma_1)\mathcal{O}(\Gamma_2) \Leftrightarrow \forall \boldsymbol{x} \in \mathit{dom}(\Gamma_2).\ \boldsymbol{x} \in \mathit{dom}(\Gamma_1) \text{ and } (\Gamma_1(x))\mathcal{O}(\Gamma_2(x))$$

For convenience, we introduce a new rule $[\lhd]$ that can perform those instantiations and subsumptions all at once:

Notation.

The typing rule $[\lhd]$ *is a shorthand for this specific pattern in a derivation tree:*

$$[\lhd]\ \frac{\dfrac{A}{\Gamma \models e : t}}{\Gamma \models e : t'}\ t \lhd t'$$

(with Σ *such that* $t\Sigma \leqslant t'$*)*

$$\updownarrow$$

$$[\leqslant]\ \frac{[\wedge]\ \dfrac{(\forall \sigma \in \Sigma) \quad [\textsc{Inst}]\ \dfrac{\dfrac{A}{\Gamma \models e : t}}{\Gamma \models e : t\sigma}}{\Gamma \models e : t\Sigma}}{\Gamma \models e : t'}\ t\Sigma \leqslant t'$$

Lemma 1 (Monotonicity)**.** *Let* Γ *be an environment,* e *an expression, and* t *a type such that* $\Gamma \models e : t$ *is derivable. Let* Γ' *be an environment such that* $\Gamma' \lhd \Gamma$*. Then,* $\Gamma' \models e : t$ *is derivable.*

Proof. We consider a derivation D of $\Gamma \models e : t$ and show that we can build a derivation D' of $\Gamma' \models e : t$. We show this result by structural induction on the proof tree D. When the root is a $[\textsc{Var}_\vee]$ or $[\textsc{Var}_\lambda]$ node, we conclude by applying $[\lhd]$. All the other cases are just straightforward applications of the induction hypothesis. □

4.2.2 Normalization of typing derivations

Derivations for the declarative type system of Figure 4.2 can still take very different shapes. In this section, we define three notions of canonical derivation, each restricting the use of a non-structural rule: one for the $[\vee]$ rule, one for the [Inst] rule, one for the $[\leqslant]$ rule, and one for the $[\wedge]$ rule. Each time, we prove a lemma that shows how every typing derivation can be normalized into a canonical derivation.

4.2.2.1 Normalization of $[\vee]$ nodes

Our objective is to constrain the shape of derivations without losing expressivity, that is, such that a judgment is derivable in the full system if and only if it is derivable by a derivation in the constrained shape. In particular, in this section, we are focusing on restricting the use of the $[\vee]$ rule. In order to do so, we first prove two lemmas that allow manipulating $[\vee]$ nodes in a derivation.

Lemma 2 allows us to remove from a derivation D all the $[\vee]$ nodes that perform aliasing, that is, the $[\vee]$ nodes that apply a substitution of the form $\{\mathsf{y}/\mathsf{x}\}$. The idea

is that the introduction of this new variable y is useless, as it is just an alias for x: the type decomposition performed on the variable y can instead be performed directly on x.

This lemma is then used in Lemma 3, which gives us the ability to insert a $[\vee]$ node performing an arbitrary substitution $\{e'/\mathsf{x}\}$ at the root of a derivation D, provided that e' is typeable and that there is no strict subexpression in e' that is the object of another $[\vee]$ node in D. The resulting derivation is guaranteed not to contain $[\vee]$ nodes performing aliasing on x. This manipulation will allow us to order $[\vee]$ nodes arbitrarily while getting rid of aliasing: this is the point of Lemma 4.

Definition 23. *Let D be a derivation, and N be a $[\vee]$ node of D. We say that N* performs aliasing *for x if it applies a substitution of the form $\{\mathsf{x}/\mathsf{y}\}$ for some binding variables y.*

Definition 24 (Aliasing-free derivation)**.** *Let D be a derivation. We say that D is* x-aliasing-free *if and only if D does not contain any $[\vee]$ node that performs aliasing for x.*

Lemma 2 (Elimination of aliasing)**.** *Let Γ be a type environment, e an expression, and t a type. Let x be a binding variable in $\mathsf{fv}(e)$. Let D be a derivation of $\Gamma, \mathsf{x} : s \vdash e : t$. Then, there exists a partition $\{\boldsymbol{u}_i\}_{i\in I}$ of $\mathbb{1}$ such that for every $i \in I$, there exists a x-aliasing-free derivation of $\Gamma, \mathsf{x} : s \wedge \boldsymbol{u}_i \vdash e : t$.*

Proof. We proceed by induction on $(|D|_{[\vee]}, |D|)$ for the lexicographic order.

- If the root is an axiom ([VAR$_\wedge$], [VAR$_\vee$], [CONST]), the property holds for the trivial partition $\{\mathbb{1}\}$.
- If the root is a $[\wedge]$ node deriving an intersection $\bigwedge_{i\in I} t_i$, then we apply the induction hypothesis on all its premises. It yields a set of partitions $\{\{\mathbf{u}_j\}_{j\in J_i}\}_{i\in I}$ of $\mathbb{1}$ and the associated x-aliasing-free derivations $\{\{D_j\}_{j\in J_i}\}_{i\in I}$. We consider a partition $\{\mathbf{v}_k\}_{k\in K}$ of $\mathbb{1}$ that satisfies this property: $\forall k \in K.\ \forall i \in I.\ \forall j \in J_i.\ \mathbf{v}_k \leqslant \mathbf{u}_j$ or $\mathbf{v}_k \wedge \mathbf{u}_j \simeq \mathbb{0}$ (such a partition can easily be built by induction on $|\{\mathbf{u}_j \mid i \in I,\ j \in J_i\}|$).

  For each $k \in K$ and $i \in I$, there exists $j \in J_i$ such that $\mathbf{v}_k \leqslant \mathbf{u}_j$, and thus we can derive $\Gamma, \mathsf{x} : s \wedge \mathbf{v}_k \vdash e : t_i$ from D_j by monotonicity (Lemma 1). Consequently, for each $k \in K$, we can build a x-aliasing-free derivation of $\Gamma, \mathsf{x} : s \wedge \mathbf{v}_k \vdash e : \bigwedge_{i\in I} t_i$ using a $[\wedge]$ node, which conclude this case.
- If the root is a $[\vee]$ node that does not perform aliasing for x, then we proceed similarly to the previous case.

- If the root is a $[\vee]$ node that performs a substitution $\{\mathsf{y}/\mathsf{x}\}$ for some y, and that uses a partition $\{\mathbf{u}_i\}_{i\in I}$, we have the following premises:

  Definition premise $\Gamma, \mathsf{x} : s \vDash \mathsf{x} : s'$ (with $s \leqslant s'$)

  Body premises $\forall i \in I.\ \Gamma, \mathsf{x} : s, \mathsf{y} : s' \wedge \mathbf{u}_i \vDash e : t$

  We build, for each $i \in I$, a derivation $\Gamma, \mathsf{x} : s \wedge \mathbf{u}_i \vDash e' : t$ where $e' \equiv e\{\mathsf{x}/\mathsf{y}\}$. We do that by applying the monotonicity lemma (Lemma 1) on the body premise $\Gamma, \mathsf{x} : s, \mathsf{y} : s' \wedge \mathbf{u}_i \vDash e : t$, yielding $\Gamma, \mathsf{x} : s \wedge \mathbf{u}_i, \mathsf{y} : s \wedge \mathbf{u}_i \vDash e : t$, and then by substituting every occurrence of y by x in the derivation.

  For each $i \in I$, we apply the induction hypothesis on our derivation $\Gamma, \mathsf{x} : s \wedge \mathbf{u}_i \vDash e' : t$. Each time, it yields a partition $\{\mathbf{v}_j\}_{J\in J_i}$ of $\mathbb{1}$, and the associated x-aliasing-free derivations. We consider the following partition of $\mathbb{1}$:

  $$\{\mathbf{v}'_k\}_{k\in K} \stackrel{\text{def}}{=} \{\mathbf{v}_j \wedge \mathbf{u}_i \mid i \in I, j \in J_i, \mathbf{v}_j \wedge \mathbf{u}_i \not\simeq \mathbb{0}\}$$

  For each $k \in K$, we thus have a x-asliasing-free derivation $\Gamma, \mathsf{x} : s \wedge \mathbf{v}'_k \vDash e' : t$, which concludes this case.

- The other cases are similar to the $[\wedge]$ case.

□

An interesting observation is that the proof above would not work if the declarative type system for expressions featured a generalization rule such as:

$$[\textsc{Gen}]\ \frac{\Gamma \vDash e : t}{\Gamma \vDash e : t\phi}\ \phi\#\Gamma$$

Indeed, the guard condition $\phi\#\Gamma$ would invalidate the monotonicity lemma, as having a new hypothesis in the environment could invalidate the application of a [Gen] rule. More precisely, in the $[\vee]$ case of the proof of Lemma 2, if the partition $\{\mathbf{u}_i\}_{i\in I}$ introduces a new monomorphic type variable, then the application of the monotonicity lemma to derive $\Gamma, \mathsf{x} : s \wedge \mathbf{u}_i, \mathsf{y} : s \wedge \mathbf{u}_i \vDash e : t$ is compromised. This would be a major issue, as our normalization process (Lemma 4) and the algorithmic type system we define in Chapter 5 both rely on the property that the union-elimination rule only needs to be applied once on every subexpression (i.e., $[\vee]$ nodes performing aliasing are redundant and can be avoided). This is the reason why we decided to restrict the use of the generalization rule on top-level definitions only.

Lemma 3 (Introduction of an arbitrary $[\vee]$ node)**.** *Let Γ be a type environment, x a binding variable, e, e_x two expressions, and t a type. Let D be a derivation of $\Gamma \vDash e\{e_x/x\} : t$ such that D does not contain any $[\vee]$ node performing a substitution $\{e_y/y\}$ for some y and e_y where e_y is a strict subexpression of e_x. If $\Gamma \vDash e_x : \mathbb{1}$ is derivable, then there exists some type s and partition $\{u_i\}_{i\in I}$*

of $\mathbb{1}$ *such that* $\Gamma \vDash e\{e_x/x\} : t$ *is derivable by a derivation whose root is a* $[\vee]$ *node of the following form, whose definition premise starts with an intersection of structural and/or* $[\text{VAR}_\vee]$ *nodes, and whose body premises are* x*-aliasing-free:*

$$[\vee]\ \frac{\dfrac{\cdots}{\Gamma \vDash e_x : s} \qquad \dfrac{\cdots}{\Gamma, x : s \wedge \boldsymbol{u}_i \vDash e : t}\ \forall i \in I}{\Gamma \vDash e\{e_x/x\} : t}$$

Proof. Let C be a derivation of $\Gamma \vDash e_\mathsf{x} : \mathbb{1}$. We collect in D and C the set $\{C_k\}_{k\in K}$ of all the subderivations of the form $\Gamma' \vDash e_\mathsf{x} : t'$, for some Γ' and t', and whose root is either a structural rule or a $[\text{VAR}_\vee]$ rule (if e_x is a binding variable). We have the guarantee that $k \neq \varnothing$ as C contains at least one such derivation. We know that, for every $k \in K$, the derivation $\{C_k\}_{k\in K}$ is still valid under the environment Γ: no variable in $\mathsf{fv}(e_\mathsf{x})$ can be introduced by a $[\to\text{I}]$ node in the segment from the root to of D to the root of C_k (the substitution $e\{e_\mathsf{x}/\mathsf{x}\}$ is capture-avoiding) nor by a $[\vee]$ node (the binding variable introduced by a $[\vee]$ node cannot appear in the final expression). For each $k \in K$, we note s_k the types derived by the derivations C_k. We pose $s = \bigwedge_{k\in K} s_k$.

Then, we build a derivation D' of $\Gamma, \mathsf{x} : s \vDash e : t$ by substituting in D every occurrence of e_x by x, and by using $[\text{VAR}_\vee]$ and $[\leqslant]$ rules to type occurrences of x in place of the subderivations $\{C_k\}_{k\in K}$. Note that this is only possible thanks to the fact that no $[\vee]$ node in D performs a substitution $\{e_\mathsf{y}/\mathsf{y}\}$ with e_y a strict subexpression of e_x: this ensures that this transformation does not prevent a $[\vee]$ node to apply.

Then, we apply Lemma 2 to D', yielding a partition $\{\mathbf{u}_i\}_{i\in I}$ and, for every $i \in I$, a x-aliasing-free derivation D''_i of $\Gamma, \mathsf{x} : s \wedge \mathbf{u}_i \vDash e : t$.

Now, we build the following derivation, which concludes this proof:

$$[\vee]\ \frac{[\wedge]\ \dfrac{\dfrac{C_k}{\Gamma \vDash e_\mathsf{x} : s_k}\ \forall k \in K}{\Gamma \vDash e_\mathsf{x} : s} \qquad \dfrac{D''_i}{\Gamma, \mathsf{x} : s \wedge \mathbf{u}_i \vDash e : t}\ \forall i \in I}{\Gamma \vDash e\{e_\mathsf{x}/\mathsf{x}\} : t}$$

□

We are now ready to define a notion of canonical derivation that restricts the use of the $[\vee]$ rule, and to prove a normalization lemma stating that, if a judgment is derivable in the full system, then it is derivable by such a canonical derivation.

Definition 25 (Acceptable $[\vee]$ node)**.** *In any derivation, a* $[\vee]$ *node* N *performing the substitution* $e\{e'/x\}$ *is said* acceptable *if it satisfies the following constraints:*

- e *contains* x *(no useless substitution), and*

- *e does not contain e' (maximal sharing)*

Definition 26 (Definition context)**.** *A* definition context Δ *is an ordered list of mappings from binding variables to expressions. Each mapping is written as a pair* (x, e)*. We note these lists extensionally by separating elements by a semicolon, that is,* $(\mathsf{x}_1, e_1); \ldots; (\mathsf{x}_n, e_n)$ *and use* ε *to denote the empty list.*

Definition 27 (Application of definition context to an expression)**.** *The application of a definition context* Δ *to an expression* e*, noted* $e\Delta$*, is the expression inductively defined as follows:*

$$e\varepsilon = e$$
$$e(\Delta; (\mathsf{x}, e')) = (e\{e'/\mathsf{x}\})\Delta$$

Note that the last mapping in Δ is the first to be applied to e (intuitively, we can interpret Δ as a sequence of let-definitions preceding the body e).

Definition 28 (Application of a definition context to an expression order)**.** *Let* $\sqsubseteq$ *be an expression order. Let* Δ *be a definition context. The relation* $\sqsubseteq_\Delta$ *is the expression order defined by* $e_1 \sqsubseteq_\Delta e_2 \Leftrightarrow e_1\Delta \sqsubseteq e_2\Delta$*.*

Definition 29 (Definition context of a node)**.** *Let* D *be a derivation of a judgment* $\Gamma \vDash e : t$*. Let* N *be a node at path* π *in* D*. Let* $\pi_1, \ldots, \pi_n$ *be the sequence of all prefixes of* π*, ordered by increasing length, that satisfy the following conditions: for a prefix* π' *of* π*,*
(i) $D(\pi')$ *is a* $[\vee]$ *node, and*
(ii) π' *is followed in* π *by a natural number* $k \geqslant 1$*.*

We call definition context *of* N *in* D *the definition context* $(\mathsf{x}_1, e_1); \ldots; (\mathsf{x}_n, e_n)$*, where* $\{e_1/\mathsf{x}_1\}, \ldots, \{e_n/\mathsf{x}_n\}$ *are the successive substitutions made by the nodes* $D(\pi_1), \ldots, D(\pi_n)$*.*

Intuitively, the definition context of a node N is the sequence of substitutions performed by the $[\vee]$ nodes in the path from the root to N. For instance, consider the following derivation:

$$\dfrac{\dfrac{\dfrac{[\textsc{Var}_\lambda]\ \dfrac{}{x : t \vDash x : t} \quad [\textsc{Var}_\vee]\ \dfrac{}{x : t, \mathsf{z} : t \vDash \mathsf{z} : t}}{x : t \vDash \mathsf{z}\{x/\mathsf{z}\} : t}\,[\vee] \qquad [\textsc{Var}_\vee]\ \dfrac{}{x : t, \mathsf{y} : t \vDash \mathsf{y} : t}}{x : t \vDash \mathsf{y}\{(\mathsf{z}\{x/\mathsf{z}\})/\mathsf{y}\} : t}\,[\vee]$$

The definition context of the $[\text{VAR}_\vee]$ node in red (the middle leaf) is (z, x): in particular, we do not consider the substitution made by the $[\vee]$ node at the root because this substitution only applies to the expression of the second premise, while our $[\text{VAR}_\vee]$ node is under the first premise.

In our definition of canonical derivations, we want to capture the fact that (*i*) the type of a subexpression only needs to be decomposed once (i.e., in any branch of our derivation, we only need one $[\vee]$ rule application per distinct subexpression), and (*ii*) the order of these different applications of the $[\vee]$ rule does not matter: when building a typing derivation for an expression e that contains two subexpressions e_1 and e_2 that are incomparable for the subterm order $\sqsubseteq_\alpha$, we can arbitrarily decide whether we first decompose the type of e_1 or the type of e_2.

In order to guarantee this second point, some definitions and lemmas that follow are parametrized by an arbitrary order over expressions. This *expression order* must satisfy the following properties:

Definition 30 (Expression order)**.** *A (possibly partial) order $\sqsubseteq$ over expressions is an* expression order *if it is compatible with α-equivalence and if it contains the subterm order $\sqsubseteq_\alpha$.*

Definition 31 (Well-positioned $[\vee]$ node)**.** *Let D be a derivation of a judgment $\Gamma \vDash e : t$. Let N be a $[\vee]$ node of D performing the substitution $\{e_1/x\}$. Let Δ be the definition context of N in D. N is* well-positioned *in $(D, \sqsubseteq)$ if for every node N' in the segment from the root (included) to N (excluded), either:*

- *There exists $\boldsymbol{x} \in \mathsf{fv}(e_1\Delta)$ such that $\boldsymbol{x} \notin \mathsf{dom}(\Gamma)$, with Γ being the type environment of the conclusion of N', or*
- *N' is a $[\vee]$ rule of definition context Δ' and performing a substitution $\{e_2/y\}$ such that $e_1\Delta \not\equiv e_2\Delta'$.*

Intuitively, a $[\vee]$ node that decomposes the type of the subexpression e_1 is well-positioned if and only if it happens as early as possible in the derivation (that is, as soon as all the free variables in e_1 are in the type environment). The only nodes that are allowed to be applied before are other $[\vee]$ nodes that decompose the type of a subexpression e_2 such that e_2 is incomparable to e_1 or strictly smaller than e_1 for the order $\sqsubseteq$. Note that, in particular, a $[\vee]$ node cannot be well-positioned if it is performing aliasing for a binding variable introduced by a previous $[\vee]$ node.

Definition 32 ($[\vee]$-canonical derivation)**.** *For a given expression order $\sqsubseteq$, a derivation D is* $[\vee]$-canonical *for the order $\sqsubseteq$ if every $[\vee]$ node it contains is acceptable and well-positioned in $(D, \sqsubseteq)$.*

We next characterize derivations with specific shapes through four new definitions: union-free derivations, non-structural derivations, and most importantly, form derivations and atomic derivations.

Definition 33 (Union-free derivation)**.** *A derivation D is said* union-free *if, for every $[\vee]$ node N of path π in D, π can be decomposed into $\pi_1;\pi_2$ such that $D(\pi_1)$ is a $[\to\text{I}]$ node.*

Roughly, a union-free derivation does not contain any $[\vee]$ node except in the subderivation of the premise of a $[\to\text{I}]$ node.

Definition 34 (Non-structural derivation)**.** *A derivation D is said* non-structural *if, for every structural node N of path π in D, π can be decomposed into $\pi_1;0;\pi_2$ such that $D(\pi_1)$ is a $[\vee]$ node.*

A non-structural derivation does not contain any structural node except in the subderivation of the definition premise of a $[\vee]$ node.

Definition 35 (Form derivations, atomic derivations)**.** *Let Γ be an environment, e an expression, and t a type. Let D be a derivation of $\Gamma \vdash e : t$.*

We say that D is a form derivation *if:*

- *D is non-structural, and*
- *For every $[\vee]$ node in D, its definition premise is an atomic derivation.*

We say that D is an atomic derivation *if:*

- *D is union-free, and*
- *Every segment from the root of D to a leaf contains at least one structural node, and*
- *For every $[\to\text{I}]$ node in D, its premise is a form derivation, and*
- *For every structural node in D that is not a $[\to\text{I}]$ node, its premises are non-structural.*

Roughly, atomic derivations are derivations that have exactly one structural node: they can type a single application, or a single projection, or a single lambda variable... Form derivations, on the other hand, are derivations that cannot apply any structural rule directly: when typing an expression e, they must apply several successive union-elimination rules on the different "atomic" subexpressions of e, and each such subexpression can then be typed by an atomic derivation. Thus, in a form derivation, structural rules only appear under the definition premises of $[\vee]$ nodes.

For instance, for typing the expression $(\mathsf{x}_1\mathsf{x}_2,\ y)$, a form derivation could first apply a $[\vee]$ rule performing the substitution $\{\mathsf{x}_1\mathsf{x}_2/\mathsf{x}\}$, and type $\mathsf{x}_1\mathsf{x}_2$ using the structural rule $[\to\text{E}]$. Then, in the body premise(s) of this $[\vee]$ node, it applies a $[\vee]$ rule

again, performing the substitution $\{y/\mathsf{y}\}$, and typing y with a $[\textsc{Var}_\lambda]$ rule. Lastly, it applies a $[\vee]$ rule $\{(\mathsf{x},\mathsf{y})/\mathsf{z}\}$ and types (x,y) using a $[\times\mathrm{I}]$ rule. It is illustrated by the derivation sketch below, where structural rules have their names written in red (note that, in this example, $[\vee]$ nodes only have one body premise as they use the trivial type decomposition $\{\mathbb{1}\}$):

$$[\vee]\ \frac{[\to\mathrm{E}]\ \dfrac{\cdots}{\Gamma \vDash \mathsf{x}_1\ \mathsf{x}_2 : t}\quad [\vee]\ \dfrac{[\textsc{Var}_\lambda]\ \dfrac{}{\Gamma,\mathsf{x}:t \vDash y : s}\quad [\vee]\ \dfrac{[\times\mathrm{I}]\quad [\textsc{Var}_\vee]}{\Gamma,\mathsf{x}:t,\mathsf{y}:s \vDash \mathsf{z}\{(\mathsf{x},\mathsf{y})/\mathsf{z}\} : t\times s}}{\Gamma,\mathsf{x}:t \vDash (\mathsf{x},\mathsf{y})\{y/\mathsf{y}\} : t\times s}}{\Gamma \vDash (\mathsf{x},y)\{\mathsf{x}_1\ \mathsf{x}_2/\mathsf{x}\} : t\times s}$$

A derivation that is both a $[\vee]$-canonical derivation and a form derivation is called $[\vee]$-canonical form derivation.

In summary, we just defined a notion of $[\vee]$-canonical form derivation that severely restricts the use of the $[\vee]$ rule. The next step is to prove that this does not change the expressivity of the deduction system, in the sense that a judgment is derivable in the full system if and only if it is derivable by a $[\vee]$-canonical form derivation.

Lemma 4 (Normalization of $[\vee]$)**.** *Let $\sqsubseteq$ be an expression order. Let Γ be a type environment, e an expression, and t a type. Let D be a derivation of $\Gamma \vDash e : t$. Then, there exists, for some type $t' \lhd t$, a $[\vee]$-canonical form derivation of $\Gamma \vDash e : t'$ for the order $\sqsubseteq$.*

Proof. From D, we can build a $[\vee]$-canonical form derivation of $\Gamma \vDash e : t'$.

First, we can trivially eliminate in D the $[\vee]$ nodes doing a substitution $e\{e_\mathsf{x}/\mathsf{x}\}$ where e does not contain x. We note V the set of free binding variables in e ($V = \mathsf{fv}(e) \cap \mathsf{Vars_B}$): these binding variables are the only ones that can be aliased by a $[\vee]$ node in the $[\vee]$-canonical derivation we are trying to build.

Then, we proceed by induction on $\mathsf{size}(e)$, with $\mathsf{size}(e)$ inductively defined as follows:

$$\begin{aligned}
\mathsf{size}(x) &= 1 & \mathsf{size}(\mathsf{x}) &= 1 \text{ if } \mathsf{x} \in V,\ 0 \text{ otherwise}\\
\mathsf{size}(c) &= 1 & \mathsf{size}(e_1 e_2) &= 1 + \mathsf{size}(e_1) + \mathsf{size}(e_2)\\
\mathsf{size}(\lambda x.e) &= 1 + \mathsf{size}(e) & \mathsf{size}((e_1,e_2)) &= 1 + \mathsf{size}(e_1) + \mathsf{size}(e_2)\\
\mathsf{size}(\pi_i e) &= 1 + \mathsf{size}(e) & \mathsf{size}((e \in \tau)\ ?\ e_1 : e_2) &= 1 + \mathsf{size}(e) + \mathsf{size}(e_1) + \mathsf{size}(e_2)
\end{aligned}$$

Note that the set V does not change during this induction: it corresponds to the free binding variables in the *initial* expression e.

The base case, $\mathsf{size}(e) = 0$, implies that e is a binding variable not in V, and thus it is a valid $[\vee]$-canonical form derivation.

For the inductive case $\mathsf{size}(e) > 0$, we proceed as follows. We consider all the subexpressions e' of e such that:

- e' is not a binding variable, or it is a binding variable in V, and
- $\forall \boldsymbol{x} \in \mathsf{fv}(e').\ \boldsymbol{x} \in \mathsf{dom}(\Gamma)$, and
- There exists in D a subderivation A whose judgment is $\Gamma' \vDash e' : t'$ for some Γ', e', and t'.

Among all these subexpressions, we choose one that is minimal for the expression order $\sqsubseteq$. We note e' this subexpression, and A a derivation of $\Gamma' \vDash e' : t'$. Note that we are sure that there exists at least one such subexpression e', otherwise it would mean that e is a binding variable not in V, contradicting $\mathsf{size}(e) > 0$.

From A, we can obtain a derivation $\Gamma \vDash e' : \mathbb{1}$ by using a $[\leqslant]$ node and by using the fact that $\forall \boldsymbol{x} \in \mathsf{fv}(e').\ \boldsymbol{x} \in \mathsf{dom}(\Gamma)$. Also note that there cannot be in D any $[\vee]$ node substituting a strict subexpression e'' of e': otherwise we would have $e'' \sqsubseteq e'$, contradicting the minimality of e'. Thus, we can apply Lemma 3 on D: it yields a derivation D' that starts with a $[\vee]$ node performing the substitution $\{e'/\mathsf{x}\}$ and whose body premises are x-aliasing-free.

The definition premise of D' cannot contain any $[\vee]$ node, except in the subderivation of a $[\to\mathrm{I}]$ node: this is ensured by the fact that it starts with an intersection of structural and/or $[\textsc{Var}_\vee]$ nodes, and by the minimality of e'. In the case where it contains one or several $[\to\mathrm{I}]$ nodes, we apply the induction hypothesis on their premises in order to turn them into $[\vee]$-canonical form derivations.

Lastly, we apply the induction hypothesis on each body premise of D', with respects to the new expression order $\sqsubseteq_{(\mathsf{x},e')}$ (cf. Definition 28). The resulting derivation satisfies all the properties of this lemma. □

4.2.2.2 Normalization of [Inst] nodes

Now, we focus on defining another notion of canonical derivation that constrains, this time, the use of the [Inst] rule. Roughly, we can restrict the use of the [Inst] rule so that it is only used as a premise of a $[\to\mathrm{E}]$, $[\times\mathrm{E}_1]$, $[\times\mathrm{E}_2]$, $[\mathbb{0}]$, $[\in_1]$, or $[\in_2]$ node. In particular, we get rid of premature instantiations: an instantiation should only happen when it is made necessary by an application, a projection, or a type test.

Lemma 5. *Let Γ be a type environment, e an expression, and t a type. If $\Gamma \vDash e : t$ is derivable, then for every renaming ρ, $\Gamma \vDash e : t\rho$ is derivable.*

Proof. We proceed by induction on the derivation of $\Gamma \vDash e : t$.

Note that any polymorphic type variable in t is introduced either by a $[\textsc{Var}_\lambda]$, $[\textsc{Var}_\vee]$, or [Inst] node. Thus, the interesting cases are the following:

- If the root is a $[\textsc{Var}_\lambda]$ or $[\textsc{Var}_\vee]$ node performing a renaming ρ', we perform the renaming $\rho \circ \rho'$ instead, and
- If the root is an [Inst] node performing a substitution σ, we perform the substitution $\rho \circ \sigma \circ \rho^{-1}$ instead.

- The other cases are straightforward applications of the induction hypothesis.

□

Proving a normalization lemma for [Inst] nodes requires manipulating sets of substitutions. Thus, we first prove some properties about substitutions.

Proposition 5. *Let $\{(t_i, t'_i)\}_{i\in I}$ be a set of pairs of types such that $\forall i \in I.\ t'_i \lhd t_i$. Then, the following relation holds: $\bigwedge_{i\in I} t'_i \lhd \bigwedge_{i\in I} t_i$.*

Proof. For each $i \in I$, let Σ_i be a set of substitutions such that $t'_i\Sigma_i \leqslant t_i$. We consider the set of substitutions $\Sigma = \bigcup_{i\in I} \Sigma_i$, and we show that $(\bigwedge_{i\in I} t'_i)\Sigma \leqslant \bigwedge_{i\in I} t_i$:

$$\begin{aligned}\textstyle\bigwedge_{\sigma\in\Sigma}(\bigwedge_{i\in I} t'_i)\sigma &\simeq \textstyle\bigwedge_{i\in I}\bigwedge_{\sigma\in\Sigma_i}(\bigwedge_{j\in I} t'_j)\sigma\\ &\leqslant \textstyle\bigwedge_{i\in I}\bigwedge_{\sigma\in\Sigma_i} t'_i\sigma\\ &\leqslant \textstyle\bigwedge_{i\in I} t'_i\Sigma_i \leqslant \bigwedge_{i\in I} t_i\end{aligned}$$

□

Proposition 6. *Let $\{(t_i, t'_i)\}_{i\in I}$ be a set of pairs of types such that $\forall i \in I.\ t'_i \lhd t_i$ and such that $\forall i, j \in I.\ \mathsf{vars}(t'_i) \cap \mathsf{vars}(t'_j) \cap \mathcal{V}_P = \varnothing$. Then, the following relation holds: $\bigvee_{i\in I} t'_i \lhd \bigvee_{i\in I} t_i$.*

Proof. For each $i \in I$, let Σ_i be a set of substitutions such that $t'_i\Sigma_i \leqslant t_i$. We consider the set of substitutions $\Sigma = \{\sigma_1 \cup \ldots \cup \sigma_n \mid \sigma_1 \in \Sigma_1, \ldots, \sigma_n \in \Sigma_n\}$ for $I = \{1, \ldots, n\}$, where $\cup$ denotes the union of two substitutions with disjoint domains (this is guaranteed by the hypothesis $\forall i, j \in I.\ \mathsf{vars}(t'_i) \cap \mathsf{vars}(t'_j) \cap \mathcal{V}_P = \varnothing$), and we show that $(\bigvee_{i\in I} t'_i)\Sigma \leqslant \bigvee_{i\in I} t_i$:

$$\begin{aligned}&\textstyle\bigwedge_{\sigma\in\Sigma}(\bigvee_{i\in 1..n} t'_i)\sigma\\ \simeq\ &\textstyle\bigwedge_{(\sigma_1,\ldots,\sigma_n)\in\Sigma_1\times\cdots\times\Sigma_n}(\bigvee_{i\in 1..n} t'_i)(\sigma_1 \cup \ldots \cup \sigma_n)\\ \simeq\ &\textstyle\bigwedge_{(\sigma_1,\ldots,\sigma_n)\in\Sigma_1\times\cdots\times\Sigma_n}\bigvee_{i\in 1..n} t'_i(\sigma_1 \cup \ldots \cup \sigma_n)\\ \simeq\ &\textstyle\bigwedge_{(\sigma_1,\ldots,\sigma_n)\in\Sigma_1\times\cdots\times\Sigma_n}\bigvee_{i\in 1..n} t'_i\sigma_i && \text{(disjointness of type variables)}\\ \simeq\ &\textstyle\bigvee_{i\in 1..n}\bigwedge_{\sigma_i\in\Sigma_i} t'_i\sigma_i && \text{(distributivity of } \vee \text{ over } \wedge)\\ \simeq\ &\textstyle\bigvee_{i\in 1..n} t'_i\Sigma_i \leqslant \bigvee_{i\in 1..n} t_i\end{aligned}$$

□

Definition 36 ([Inst]-canonical derivation)**.** *A derivation D is* [Inst]-canonical *if every* [Inst] *node it contains is part of a* $[\lhd]$ *pattern that is either:*

- *The first premise of a* $[\mathbb{0}]$, $[\in_1]$ *or* $[\in_2]$ *node, or*

- *The premise of a* $[\times \mathrm{E}_1]$ *or* $[\times \mathrm{E}_2]$ *node, or*
- *One of the premises of a* $[\to \mathrm{E}]$ *node*

The idea of the normalization lemma for the [Inst] rule is to push applications of [Inst] towards the root as much as possible.

Lemma 6 (Normalization of [Inst])**.** *Let* $\sqsubseteq$ *be an expression order. Let* Γ *be a type environment,* e *an expression, and* t *a type. Let* D *be a* $[\vee]$*-canonical form (resp. atomic) derivation of* $\Gamma \vdash e : t$ *for the order* $\sqsubseteq$*. Then there exists, for some type* $t' \lhd t$*, a* $[\vee][\text{Inst}]$*-canonical form (resp. atomic) derivation of* $\Gamma \vdash e : t'$ *for the order* $\sqsubseteq$*.*

Proof. We build a $[\vee][\text{Inst}]$-canonical form (resp. atomic) derivation by induction on D for the order $(\leqslant_\vee, \leqslant_*)_{\mathtt{lex}}$, where $D_1 \leqslant_\vee D_2 \Leftrightarrow |D_1|_{[\vee]} \leqslant |D_2|_{[\vee]}$ and $D_1 \leqslant_* D_2 \Leftrightarrow |D_1| \leqslant |D_2|$.

- If the root is an axiom ($[\text{Var}_\lambda]$,$[\text{Var}_\vee]$,[Const]), the derivation is already a $[\vee][\text{Inst}]$-canonical derivation.
- If the root is a [Inst] or $[\leqslant]$, we consider its unique premise as our new derivation and apply the induction hypothesis on it.
- If the root is a $[\wedge]$, we call the induction hypothesis on all its premises and use the resulting derivations as premises of a new $[\wedge]$ root, deriving a type t'. We know that t' satisfies $t' \lhd t$ according to Proposition 5.
- If the root is a $[\vee]$ of the following form:

$$[\vee]\ \frac{\dfrac{A}{\Gamma \vdash e' : s} \quad \dfrac{B_i}{\Gamma, \mathsf{x} : s \wedge \mathbf{u}_i \vdash e : t}\ \forall i \in I}{\Gamma \vdash e\{e'/\mathsf{x}\} : t}$$

  1. We first apply the induction hypothesis on A, which gives a derivation A'. We consider the following derivation, where $s' \lhd s$ (and thus $s' \wedge \mathbf{u}_i \lhd s \wedge \mathbf{u}_i$):

$$[\vee]\ \frac{\dfrac{A'}{\Gamma \vdash e' : s'} \quad \dfrac{B'_i}{\Gamma, \mathsf{x} : s' \wedge \mathbf{u}_i \vdash e : t}\ \forall i \in I}{\Gamma \vdash e\{e'/\mathsf{x}\} : t}$$

     with B'_i a derivation easily derived from B_i by monotonicity (Lemma 1). Note that the application of the monotonicity lemma might insert unwanted $[\lhd]$ patterns on $[\text{Var}_\lambda]$ and $[\text{Var}_\vee]$ nodes, but they will be eliminated with the next step.

2. The next step is to apply the induction hypothesis on the $\{B'_i\}_{i\in I}$ premises, yielding some derivations $\{B''_i\}_{i\in I}$ that derive some types $\{t_i\}_{i\in I}$ (with $\forall i \in I.\, t_i \lhd t$). We can suppose that all the $\{t_i\}_{i\in I}$ have disjoint polymorphic type variables: if it is not the case, it can be ensured by applying Lemma 5 to these premises. Then, we consider the following derivation:

$$[\vee]\ \frac{\dfrac{A'}{\Gamma \vdash e' : s'} \qquad [\leqslant]\ \dfrac{\dfrac{B''_i}{\Gamma, \mathsf{x} : s' \wedge \mathbf{u}_i \vdash e : t_i}}{\Gamma, \mathsf{x} : s' \wedge \mathbf{u}_i \vdash e : \bigvee_{i\in I} t_i}\ \forall i \in I}{\Gamma \vdash e\{e'/\mathsf{x}\} : \bigvee_{i\in I} t_i}$$

The result $\bigvee_{i\in I} t_i$ satisfies $\bigvee_{i\in I} t_i \lhd t$ according to Proposition 6.

3. The new $[\leqslant]$ nodes that appear as premise of the $[\vee]$ root could break the properties of Lemma 4 if the corresponding B''_i ends with a $[\vee]$ node. In this case, we move up the faulty $[\leqslant]$ nodes as needed using this transformation:

$$[\leqslant]\ \frac{[\vee]\ \dfrac{\dfrac{M}{\Gamma \vdash e' : s} \qquad \dfrac{N_i}{\Gamma, \mathsf{x} : s \wedge \mathbf{u}_i \vdash e : t'}\ \forall i \in I}{\Gamma \vdash e\{e'/\mathsf{x}\} : t'}}{\Gamma \vdash e\{e'/\mathsf{x}\} : t}$$

$$\downarrow$$

$$[\vee]\ \frac{\dfrac{M}{\Gamma \vdash e' : s} \qquad [\leqslant]\ \dfrac{\dfrac{N_i}{\Gamma, \mathsf{x} : s \wedge \mathbf{u}_i \vdash e : t'}}{\Gamma, \mathsf{x} : s \wedge \mathbf{u}_i \vdash e : t}\ \forall i \in I}{\Gamma \vdash e\{e'/\mathsf{x}\} : t}$$

- The other cases are straightforward applications of the induction hypothesis.

□

4.2.2.3 Normalization of $[\leqslant]$ nodes

Again, we define another notion of canonical derivation, this time to constrain the use of the $[\leqslant]$ rule. Similarly to the [Inst] rule, we want to avoid premature applications of $[\leqslant]$. The idea of the associated normalization lemma is to push applications of the $[\leqslant]$ rule towards the root as much as possible, making them appear only as premises of $[\to\mathrm{E}]$, $[\times\mathrm{E}_1]$, $[\times\mathrm{E}_2]$, $[\in_1]$, $[\in_2]$, and $[\vee]$ nodes.

Definition 37 ($[\leqslant]$-canonical derivation)**.** *A derivation D is* $[\leqslant]$-canonical *if every $[\leqslant]$ node it contains is either:*

- *The first premise of a $[\in_1]$ or $[\in_2]$ node, or*
- *One of the body premises of a $[\vee]$ node, or*

- *The premise of a* $[\times\mathrm{E}_1]$ *or* $[\times\mathrm{E}_2]$ *node, or*
- *The first premise of a* $[\to\mathrm{E}]$ *node*

Lemma 7 (Normalization of $[\leqslant]$)**.** *Let* $\sqsubseteq$ *be an expression order. Let* Γ *be a type environment,* e *an expression, and* t *a type. Let* D *be a* $[\vee][\textsc{Inst}]$*-canonical form (resp. atomic) derivation of* $\Gamma \vdash e : t$ *for the order* $\sqsubseteq$*. Then there exists, for some type* $t' \lhd t$*, a* $[\vee][\textsc{Inst}][\leqslant]$*-canonical form (resp. atomic) derivation of* $\Gamma \vdash e : t'$ *for the order* $\sqsubseteq$*.*

Proof. We build a $[\vee][\textsc{Inst}][\leqslant]$-canonical form (resp. atomic) derivation by induction on D for the order $(\leqslant_\vee, \leqslant_*)_{\mathtt{lex}}$, where $D_1 \leqslant_\vee D_2 \Leftrightarrow |D_1|_{[\vee]} \leqslant |D_2|_{[\vee]}$ and $D_1 \leqslant_* D_2 \Leftrightarrow |D_1| \leqslant |D_2|$.

- If the root is an axiom ($[\textsc{Var}_\lambda]$,$[\textsc{Var}_\vee]$,$[\textsc{Const}]$), the derivation is already a $[\vee][\textsc{Inst}][\leqslant]$-canonical derivation.
- If the root is a $[\textsc{Inst}]$ or $[\leqslant]$, we consider its unique premise as our new derivation and apply the induction hypothesis on it.
- If the root is a $[\wedge]$, we call the induction hypothesis on all its premises and use the resulting derivations as premises of a new $[\wedge]$ root, deriving a type t'. We know that t' satisfies $t' \lhd t$ (trivial).
- If the root is a $[\to\mathrm{I}]$, we call the induction hypothesis on its premise and use the resulting derivation as premise of a new $[\to\mathrm{I}]$ root, deriving a type t'. We know that t' satisfies $t' \lhd t$ (trivial).
- If the root is a $[\vee]$ of the following form:

$$[\vee]\ \frac{\dfrac{A}{\Gamma \vdash e' : s} \qquad \dfrac{B_i}{\Gamma, \mathsf{x} : s \wedge \mathbf{u}_i \vdash e : t}\ \forall i \in I}{\Gamma \vdash e\{e'/\mathsf{x}\} : t}$$

  1. We first apply the induction hypothesis on A, yielding a derivation A'. We then consider the following derivation, with $s' \lhd s$:

$$[\vee]\ \frac{\dfrac{A'}{\Gamma \vdash e' : s'} \qquad \dfrac{B'_i}{\Gamma, \mathsf{x} : s' \wedge \mathbf{u}_i \vdash e : t}\ \forall i \in I}{\Gamma \vdash e\{e'/\mathsf{x}\} : t}$$

     where each B'_i is derived from B_i by applying Lemma 1 (monotonicity), and then Lemma 6 in order to normalize [Inst] nodes that might have

been introduced by the monotonicity lemma. Note that this might add unwanted $[\leqslant]$ nodes, but they will be eliminated with the next step.

2. We apply the induction hypothesis on the $\{B'_i\}_{i\in I}$ premises, yielding some derivations $\{B''_i\}_{i\in I}$ that derive some types $\{t_i\}_{i\in I}$ (with $\forall i \in I.\ t_i \lhd t$). Then, we consider the following derivation:

$$[\vee]\ \frac{\dfrac{A'}{\Gamma \vdash e' : s'} \qquad [\leqslant]\ \dfrac{\dfrac{B''_i}{\Gamma, \mathsf{x} : s' \wedge \mathbf{u}_i \vdash e : t_i}}{\Gamma, \mathsf{x} : s' \wedge \mathbf{u}_i \vdash e : \bigvee_{i\in I} t_i}\ \forall i \in I}{\Gamma \vdash e\{e'/\mathsf{x}\} : \bigvee_{i\in I} t_i}$$

The result $\bigvee_{i\in I} t_i$ trivially satisfies $\bigvee_{i\in I} t_i \lhd t$.

3. The new $[\leqslant]$ nodes that appear as premise of the $[\vee]$ root could break the properties of Lemma 4 if the corresponding B''_i ends with a $[\vee]$ node. In this case, we move up the faulty $[\leqslant]$ nodes as needed using this transformation:

$$[\leqslant]\ \frac{[\vee]\ \dfrac{\dfrac{M}{\Gamma \vdash e' : s} \qquad \dfrac{N_i}{\Gamma, \mathsf{x} : s \wedge \mathbf{u}_i \vdash e : t'}\ \forall i \in I}{\Gamma \vdash e\{e'/\mathsf{x}\} : t'}}{\Gamma \vdash e\{e'/\mathsf{x}\} : t}$$

$$\downarrow$$

$$[\vee]\ \frac{\dfrac{M}{\Gamma \vdash e' : s} \qquad [\leqslant]\ \dfrac{\dfrac{N_i}{\Gamma, \mathsf{x} : s \wedge \mathbf{u}_i \vdash e : t'}}{\Gamma, \mathsf{x} : s \wedge \mathbf{u}_i \vdash e : t}\ \forall i \in I}{\Gamma \vdash e\{e'/\mathsf{x}\} : t}$$

- The other cases are straightforward applications of the induction hypothesis.

□

4.2.2.4 Normalization of $[\wedge]$ nodes

Lastly, we define a notion of canonical derivation that constrains the use of $[\wedge]$ nodes.

Definition 38 ($[\wedge]$-canonical derivation)**.** *A derivation D is* $[\wedge]$-canonical *if every $[\wedge]$ node it contains has either:*

- *Only* [Inst] *premises, or*
- *Only* $[\to\text{I}]$ *premises*

Lemma 8 (Normalization of $[\wedge]$)**.** *Let $\sqsubseteq$ be an expression order. Let Γ be a type environment, e an expression, and t a type. Let D be a $[\vee][\textsc{Inst}][\leqslant]$-canonical form (resp. atomic) derivation of $\Gamma \vDash e : t$ for the order $\sqsubseteq$. Then there exists a $[\vee][\textsc{Inst}][\leqslant][\wedge]$-canonical form (resp. atomic) derivation of $\Gamma \vDash e : t$ for the order $\sqsubseteq$.*

Proof. We build a $[\vee][\textsc{Inst}][\leqslant][\wedge]$-canonical form (resp. atomic) derivation by structural induction on D.

If D is a form derivation, then either the root is a $[\vee]$ node, in which case we apply the induction hypothesis on every premise of D, or D is a $[\textsc{Var}_\vee]$ node (or an intersection of multiple copies of this $[\textsc{Var}_\vee]$ node), in which case we can just return this $[\textsc{Var}_\vee]$ node.

Otherwise, D is an atomic derivation. We can suppose without loss of generality that its root is a $[\wedge]$ node. If some premises are also $[\wedge]$ nodes, we recursively merge them with the root. Since D is a $[\vee][\textsc{Inst}][\leqslant]$-canonical atomic derivation, we know that all the premises are structural nodes (of the same kind). There are several cases:

- If all premises are $[\to\text{I}]$ nodes, we apply the induction hypothesis on their premises.
- If all premises are $[\to\text{E}]$ nodes, we apply the following transformation:

$$[\wedge]\ \frac{(\forall i \in I)\quad [\to\text{E}]\ \dfrac{[\lhd]\ \dfrac{[\textsc{Var}_\vee]\ \dfrac{}{\Gamma \vDash \mathsf{x}_1 : \Gamma(\mathsf{x}_1)}}{\Gamma \vDash \mathsf{x}_1 : t_i \to s_i} \qquad [\lhd]\ \dfrac{[\textsc{Var}_\vee]\ \dfrac{}{\Gamma \vDash \mathsf{x}_2 : \Gamma(\mathsf{x}_2)}}{\Gamma \vDash \mathsf{x}_2 : t_i}}{\Gamma \vDash \mathsf{x}_1\mathsf{x}_2 : s_i}}{\Gamma \vDash \mathsf{x}_1\mathsf{x}_2 : \bigwedge_{i\in I} s_i}$$

$$\downarrow$$

$$[\to\text{E}]\ \frac{[\lhd]\ \dfrac{[\textsc{Var}_\vee]\ \dfrac{}{\Gamma \vDash \mathsf{x}_1 : \Gamma(\mathsf{x}_1)}}{\Gamma \vDash \mathsf{x}_1 : (\bigwedge_{i\in I} t_i) \to (\bigwedge_{i\in I} s_i)} \qquad [\lhd]\ \dfrac{[\textsc{Var}_\vee]\ \dfrac{}{\Gamma \vDash \mathsf{x}_2 : \Gamma(\mathsf{x}_2)}}{\Gamma \vDash \mathsf{x}_2 : \bigwedge_{i\in I} t_i}}{\Gamma \vDash \mathsf{x}_1\mathsf{x}_2 : \bigwedge_{i\in I} s_i}$$

  Note that $\Gamma(\mathsf{x}_1) \lhd (\bigwedge_{i\in I} t_i) \to (\bigwedge_{i\in I} s_i)$ can be deduced from $\forall i \in I.\ \Gamma(\mathsf{x}_1) \lhd t_i \to s_i$ using Proposition 5. Similarly, $\Gamma(\mathsf{x}_2) \lhd \bigwedge_{i\in I} t_i$ can be deduced from $\forall i \in I.\ \Gamma(\mathsf{x}_2) \lhd t_i$.
- The other cases are similar.

□

Now that we have defined four notions of canonical derivation constraining respectively the use of the $[\vee]$ rule, the [Inst] rule, the $[\leqslant]$ rule, and the $[\wedge]$ rule, we combine them together into a single definition.

Definition 39 (Canonical derivation)**.** *Let $\sqsubseteq$ be an expression order. A* canonical derivation *for the expression order $\sqsubseteq$ is a $[\vee][\textsc{Inst}][\leqslant][\wedge]$-canonical derivation for the expression order $\sqsubseteq$.*

A derivation that is both a canonical derivation and a form derivation is called canonical form derivation. Likewise, a derivation that is both a canonical derivation and an atomic derivation is called canonical atomic derivation.

When qualifying a derivation D of canonical, the expression order $\sqsubseteq$ may be omitted: in this case, it means that there exists an expression order $\sqsubseteq$ such that D is canonical for the order $\sqsubseteq$.

Theorem 1 (Normalization of derivations)**.** *Let $\sqsubseteq$ be an expression order. Let Γ be a type environment, e an expression, and t a type. If $\Gamma \vDash e : t$ is derivable, then there exists, for some type $t' \lhd t$, a canonical form derivation of $\Gamma \vDash e : t'$ for the order $\sqsubseteq$.*

Proof. Successive application of Lemma 4, Lemma 6, Lemma 7, and Lemma 8. □

Canonical form derivations restrain the possible locations where $[\vee]$, [Inst], $[\leqslant]$, and $[\wedge]$ rules can be used, and are thus a first step towards an algorithmic type system. In addition, we can use the structure of canonical form derivations to establish a type safety theorem for the declarative type system: this is the focus of the next section.

4.3 Type safety

As stated by Milner (1978), "well-typed programs cannot go wrong". In this section, we formalize and prove that this is true for our declarative type system. For that we use a standard methodology: we first prove *subject reduction* (Section 4.3.4), stating that the type of an expression is preserved by reduction. Then, we prove *progress* (Section 4.3.5), stating that a closed well-typed expression that is not already a value can be reduced. However, some difficulties must be addressed first, for subject reduction to hold. We tackle those in Sections 4.3.1, 4.3.2, and 4.3.3.

4.3.1 The parallel semantics

Our plan is to prove type safety by proving subject reduction and progress. Unfortunately, subject reduction does not hold for the semantics presented in Figure 3.1, as performing a reduction step on an expression e might break the use of a $[\vee]$ rule. Indeed, if in the typing derivation of the reducendum a rule $[\vee]$ substitutes multiple occurrences of the subexpression e by a variable x, reducing one occurrence of e but not the others can make the application of this $[\vee]$ rule impossible for the reductum:

the correlation between the reduced occurrence of e and the other occurrences of e is thus lost.

For instance, consider the pair (e, e) where $e = (\lambda x.\mathtt{f}\ x)\ 42$, under the type environment $\Gamma = \{\mathtt{f} : \mathbb{1} \to \mathbb{1}\}$. This pair can be given the type $(\mathtt{Int} \times \mathtt{Int}) \vee (\neg\mathtt{Int} \times \neg\mathtt{Int})$ using the $[\vee]$ rule. Indeed, although e can only be typed $\mathbb{1}$, the $[\vee]$ rule allows correlating the type of both occurrences of e by considering two cases: a first case where both occurrences have the type $\mathtt{Int}$, and a second case where they both have the type $\neg\mathtt{Int}$.

Now, we apply a step of reduction to (e, e), yielding the expression $(\mathtt{f}\ 42, e)$. In this new expression, it is no longer possible to correlate the left and right parts of the pair: the $[\vee]$ rule does not apply anymore as $\mathtt{f}\ 42$ and $(\lambda x.\mathtt{f}\ x)\ 42$ are not syntactically equivalent. Thus, it is impossible to infer for this expression a type that is smaller or equivalent to $(\mathtt{Int} \times \mathtt{Int}) \vee (\neg\mathtt{Int} \times \neg\mathtt{Int})$. Actually, the smallest type we can infer for it is $42 \times \mathbb{1}$, which is not a subtype of $(\mathtt{Int} \times \mathtt{Int}) \vee (\neg\mathtt{Int} \times \neg\mathtt{Int})$ (in particular, the value $(42, \mathtt{true})$ is in the former type and not in the latter).

To circumvent this issue, we introduce a notion of parallel reduction which forces to reduce all occurrences of a subexpression at the same time. With this semantics, the expression (e, e) reduces to $(\mathtt{f}\ 42, \mathtt{f}\ 42)$ after one step of reduction, preserving the syntactic equivalence between the left and right parts of the pair.

The idea is to first define reduction rules that only apply at top-level, and then define a context rule (cf. rule $[\kappa]$ below) that allows reducing under an evaluation context, but that also applies this reduction everywhere in the term. Since our calculus is pure and deterministic, proving a type safety theorem for the parallel semantics will allow us to deduce another (weaker) type safety theorem for the initial semantics (Figure 3.1): this weaker version guarantees that a well-typed term either diverges or reduces to a value of the same test type τ (type preservation is not guaranteed for an arbitrary type t, but for a test type τ).

The parallel semantics is formalized in Figure 4.3. A step of reduction happening at top-level is noted $\rightsquigarrow_\top$, and a step of reduction of the parallel semantics under any evaluation context is noted $\rightsquigarrow_{\mathcal{P}}$. Notice that the rule $[\kappa]$ applies a substitution from an expression e' to an expression e, which is defined below.

Definition 40 (Expression substitution)**.** *The substitution in e'' of the expression e' by the expression e, noted $e''\{e/e'\}$, is defined as follows:*

- *If $e' \equiv_\alpha e''$, then $e''\{e/e'\} = e$.*
- *If $e' \not\equiv_\alpha e''$, then $e''\{e/e'\}$ is inductively defined as*

Top-level reductions:

$$
\begin{array}{rcll}
(\lambda x.e)v & \rightsquigarrow_\top & e\{v/x\} & (4.1)\\
\pi_1(v_1, v_2) & \rightsquigarrow_\top & v_1 & (4.2)\\
\pi_2(v_1, v_2) & \rightsquigarrow_\top & v_2 & (4.3)\\
(v{\in}\tau)\,?\,e_1 : e_2 & \rightsquigarrow_\top & e_1 \quad \text{if } v \in \tau & (4.4)\\
(v{\in}\tau)\,?\,e_1 : e_2 & \rightsquigarrow_\top & e_2 \quad \text{if } v \in \neg\tau & (4.5)
\end{array}
$$

Parallel reductions:

$$[\kappa]\ \frac{e \rightsquigarrow_\top e'}{E[e] \rightsquigarrow_{\mathcal{P}} (E[e])\{e'/e\}}$$

Evaluation Context $E ::= [\,] \mid v\ E \mid E\ e \mid (v, E) \mid (E, e) \mid \pi_i E \mid (E{\in}\tau)\,?\,e : e$

Figure 4.3: Parallel Semantics

$$
\begin{array}{rcll}
c\{e/e'\} &=& c & \\
x\{e/e'\} &=& x & \\
\mathsf{x}\{e/e'\} &=& \mathsf{x} & \\
(e_1 e_2)\{e/e'\} &=& (e_1\{e/e'\})(e_2\{e/e'\}) & \\
(\lambda x.e_\circ)\{e/e'\} &=& \lambda x.(e_\circ\{e/e'\}) & x \notin \mathsf{fv}(e'),\ x \notin \mathsf{fv}(e)\\
(\pi_i e_\circ)\{e/e'\} &=& \pi_i(e_\circ\{e/e'\}) & \\
(e_1, e_2)\{e/e'\} &=& (e_1\{e/e'\}, e_2\{e/e'\}) & \\
((e_1{\in}t)\,?\,e_2 : e_3)\{e/e'\} &=& (e_1\{e/e'\}{\in}t)\,?\,e_2\{e/e'\} : e_3\{e/e'\} &
\end{array}
$$

Expression substitutions are up to α-renaming: the conditions $x \notin \mathsf{fv}(e')$ and $x \notin \mathsf{fv}(e)$ in the case of λ-abstractions can be ensured by first performing an α-renaming on $\lambda x.e_\circ$.

Here is a reduction step on the previous example using the parallel semantics:

$$[\kappa]\ \frac{(\lambda x.\mathsf{f}\ x)\ 42 \rightsquigarrow_\top \mathsf{f}\ 42}{((\lambda x.\mathsf{f}\ x)\ 42, (\lambda x.\mathsf{f}\ x)\ 42) \rightsquigarrow_{\mathcal{P}} (\mathsf{f}\ 42, \mathsf{f}\ 42)}$$

The parallel semantics is extended to programs in a straightforward way:

$$
\begin{array}{ll}
\textbf{Reduction rule} & \mathtt{let}\, x = v\ ;\ p \ \rightsquigarrow_{\mathcal{P},\mathsf{Pr}}\ p\{v/x\} \qquad \dfrac{e \rightsquigarrow_{\mathcal{P}} e'}{P[e] \rightsquigarrow_{\mathcal{P},\mathsf{Pr}} P[e']}\\
\textbf{Evaluation Context} & P ::= [\,] \mid \mathtt{let}\, x = [\,]\ ;\ p
\end{array}
$$

4.3.2 Elimination of instantiations and generalizations

In this section, we show that we can eliminate, in the typing derivation of a program, every [Inst] node and every generalization of monomorphic type variables

happening in a [TopLevel-Expr] node. This is done by performing all necessary instantiations beforehand (that is, in the initial environment Γ). Note that this is only possible because of intersection types: they can replace parametric polymorphism locally by encoding all the instantiations we need for a type.

Definition 41 (Instantiation-free derivation)**.** *A derivation D of $\Gamma \vDash e : t$ is* instantiation-free *if it does not contain any* [Inst] *node.*

Lemma 9 (Weak monotonicity)**.** *Let D be an instantiation-free derivation of $\Gamma \vDash e : t$, and Γ' be an environment such that $\Gamma' \leqslant \Gamma$. Then, there exists an instantiation-free derivation of $\Gamma' \vDash e : t$.*

Proof. Straightforward structural induction on the derivation $\Gamma \vDash e : t$. If the root is a [Var$_\vee$-N] or [Var$_\lambda$-N] node, we use a [$\leqslant$] node. The other cases are straightforward applications of the induction hypothesis. □

Lemma 10 (Elimination of [Inst] nodes)**.** *Let Γ be an environment, e an expression, and t a type. Let D be a derivation of $\Gamma \vDash e : t$. Let Σ be a set of substitutions. Then, there exists an instantiation-free derivation of $\Gamma\Sigma' \vDash e : t\Sigma$ for some set of substitutions Σ'.*

Proof. We proceed by structural induction on the derivation D.

We consider the root of D:

[Const] The type t has no type variable, thus we can directly conclude by choosing $\Sigma' = \varnothing$.

[Var$_\lambda$] We pose $e \equiv x$, and we have $t \simeq \Gamma(x)$. We can derive $\Gamma\Sigma' \vDash x : t\Sigma$ using a [Var$_\lambda$] node by choosing $\Sigma' = \Sigma$.

[Var$_\vee$] Similar to the previous case.

[Inst] **or** [$\leqslant$] We have $t' \lhd t \lhd t\Sigma$, and thus by transitivity $t' \lhd t\Sigma$. Thus, there exists some Σ'' such that $t'\Sigma'' \leqslant t\Sigma$. We apply the induction hypothesis on the premise $\Gamma \vDash e : t'$ in order to derive $\Gamma\Sigma' \vDash e : t'\Sigma''$. We can then derive $\Gamma\Sigma' \vDash e : t\Sigma$ by inserting a [$\leqslant$] node at the root.

[$\vee$] The [$\vee$] node at the root is doing a substitution $e\{e'/\mathsf{x}\}$. We first apply the induction hypothesis on its body premises. It yields some derivations $\Gamma\Sigma'_i, \mathsf{x} : s\Sigma'_i \wedge \mathbf{u}_i \vDash e : t\Sigma$ for $i \in I$. We consider the set of substitutions $\Sigma' = \bigcup_{i \in I} \Sigma'_i$.

Now, we derive $\Gamma\Sigma'' \models e' : s\Sigma'$ for some Σ'' by applying the induction hypothesis on the definition premise. We pose $\Sigma''' = \Sigma'' \cup \Sigma'$. We transform, for each $i \in I$, the derivation $\Gamma\Sigma'_i, \mathsf{x} : s\Sigma'_i \wedge \mathbf{u}_i \models e : t\Sigma$ into a derivation $\Gamma\Sigma''', \mathsf{x} : s\Sigma' \wedge \mathbf{u}_i \models e : t\Sigma$ using Lemma 9. Similarly, we transform $\Gamma\Sigma'' \models e' : s\Sigma'$ into $\Gamma\Sigma''' \models e' : s\Sigma'$. By combining all those derivations into a $[\vee]$ node, we can derive $\Gamma\Sigma''' \models e\{e'/\mathsf{x}\} : t\Sigma$.

Other rules We apply the induction hypothesis to each premise $\Gamma \models e_i : s_i$ in order to derive, for $i \in I$, $\Gamma\Sigma'_i \models e_i : s_i\Sigma$. We then consider $\Sigma' = \bigcup_{i\in I} \Sigma'_i$, and derive, for each $i \in I$, $\Gamma\Sigma' \models e_i : s_i\Sigma$ using Lemma 9. We then use these derivations as premises of a node of the same kind as the initial root.

□

Lemma 11 (Monomorphization lemma)**.** *Let e be an expression, Γ an environment, and t a type. Let D be a derivation of $\Gamma \models e : t$. Let σ be an injective renaming from polymorphic type variables to monomorphic ones such that $\mathsf{vars}(\Gamma) \cap \mathcal{V}_P \subseteq \mathsf{dom}(\sigma)$. Then, there exists Ψ' such that $(\Gamma\sigma)\Psi' \models e : t\sigma$.*

Proof. Using Lemma 10, we transform D into an instantiation-free derivation D' of $\Gamma\Sigma' \models e : t$ for some Σ'. Then, we build a derivation D'' of $(\Gamma\Sigma')\sigma \models e : t\sigma$ by applying the renaming σ to every type and environment in D'. This is only possible because D' does not contain any [Inst] node (substituting a polymorphic type variable by a monomorphic one could invalidate an [Inst] node).

As σ is injective, it has an inverse substitution ϕ, which is a renaming from monomorphic type variables to polymorphic type variables. We pose $\Psi' = \{\sigma \circ \sigma' \circ \phi \mid \sigma' \in \Sigma'\}$ (it is a monomorphic substitution since $\mathsf{dom}(\Sigma') \subseteq \mathsf{vars}(\Gamma) \cap \mathcal{V}_P \subseteq \mathsf{dom}(\sigma)$). We have $(\Gamma\sigma)\Psi' \simeq (((\Gamma\sigma)\phi)\Sigma')\sigma \simeq (\Gamma\Sigma')\sigma$. Thus, D'' derives $(\Gamma\sigma)\Psi' \models e : t\sigma$, which concludes this proof. □

Lemma 12. *Let D be a derivation of $\Gamma \models e : t$, and ψ a substitution. Then, there exists a derivation of $\Gamma\psi \models e : t\psi$.*

Proof. Straightforward structural induction on D. The type substitution ψ can be applied to every type (and type environment) in D, and as $\mathsf{dom}(\psi) \subseteq \mathcal{V}_M$ it does not conflict with an [Inst] node. Subtyping relations are preserved as $t_1 \leqslant t_2 \Rightarrow t_1\psi \leqslant t_2\psi$. □

We define the $\models_{\mathsf{Pr}}$ deduction rules for typing programs:

$$[\textsc{TopLevel-Expr}]\ \frac{\Gamma \models e : t}{\Gamma \models_{\mathsf{Pr}} e : t\phi}\ \phi\#\Gamma \qquad [\textsc{TopLevel-Let}]\ \frac{\Gamma \models_{\mathsf{Pr}} e : t \qquad \Gamma, x : t \models_{\mathsf{Pr}} p : t'}{\Gamma \models_{\mathsf{Pr}} \mathtt{let}\, x = e\, ;\, p\ : t'}$$

The type safety properties will be proved for the $\models_{\mathsf{Pr}}$ type system, which is equivalent to the $\vdash_{\mathsf{Pr}}$ type system:

Proposition 7. *For every program p, type environment Γ and type t:*

$$\Gamma \vdash_{Pr} e : t \Leftrightarrow \Gamma \models_{Pr} e : t$$

Proof. Both directions are proved by structural induction on the derivation, using Proposition 3. □

Definition 42 (Generalization-free derivation)**.** *A derivation D of $\Gamma \models_{Pr} p : t$ is* generalization-free *if every* [TopLevel-Expr] *node it contains uses the substitution $\phi = \varnothing$.*

Lemma 13 (Monotonicity of programs)**.** *Let Γ, Γ' be two environments such that $\Gamma' \leqslant \Gamma$. Let p be a program, and t a type. Let D be a generalization-free derivation of $\Gamma \models_{Pr} p : t$. Then, there exists a generalization-free derivation of $\Gamma' \models_{Pr} p : t$.*

Proof. Straightforward induction on D, using monotonicity of $\models$ (Lemma 9). □

Lemma 14 (Elimination of generalization)**.** *Let p be a program, and t a type. Let D be a derivation of $\varnothing \models_{Pr} p : t$. Then, there exists a generalization-free derivation of $\varnothing \models_{Pr} p : t\sigma$ for some injective renaming σ from polymorphic type variables to fresh monomorphic ones.*

Proof. Let σ be an injective renaming that maps every polymorphic type variable that occurs in D to a fresh monomorphic type variable.

Now, we prove by structural induction on a derivation D of $\Gamma \models_{\mathsf{Pr}} p : t$ that, given a set of substitutions Ψ, there exists a generalization-free derivation of $(\Gamma\sigma)\Psi' \models_{\mathsf{Pr}} p : (t\sigma)\Psi$ for some set of substitutions Ψ'.

We consider the root of the derivation:

[TopLevel-Expr] The conclusion of the root is $\Gamma \models_{\mathsf{Pr}} e : t'\phi$, and the premise is $\Gamma \models e : t'$.

We pose $\psi = (\sigma \circ \phi)\big|_{\mathsf{dom}(\phi)}$. Note that $\psi \# \Gamma$. By applying Lemma 12 on $\Gamma \models e : t'$ and ψ, we get $\Gamma \models e : t'\psi$. Then, by applying Lemma 11, we get $(\Gamma\sigma)\Psi' \models e : (t'\psi)\sigma$ for some Ψ'. As $(t'\psi)\sigma \simeq ((t'\phi)\sigma)\sigma \simeq (t'\phi)\sigma$, this judgment can be rewritten $(\Gamma\sigma)\Psi' \models e : (t'\phi)\sigma$.

By applying Lemma 12 on this last judgment and Ψ, we derive $((\Gamma\sigma)\Psi')\Psi \models e : ((t'\phi)\sigma)\Psi$. By posing $\Psi'' = \{\psi \circ \psi' \mid \psi \in \Psi, \psi' \in \Psi'\}$, this judgment can be rewritten $(\Gamma\sigma)\Psi'' \models e : ((t'\phi)\sigma)\Psi$. We can conclude this case by building a generalization-free [TopLevel-Expr] node that uses this premise.

[TopLevel-Let] The conclusion of the root is $\Gamma \vDash_{\mathsf{Pr}} \texttt{let}\, x = e\,;\, p : t$, and the two premises are:

1. $\Gamma \vDash_{\mathsf{Pr}} e : t'$
2. $\Gamma, x : t' \vDash_{\mathsf{Pr}} p : t$

By applying the induction hypothesis on the premise (2), we get a generalization-free derivation B of $(\Gamma\sigma)\Psi', x : (t'\sigma)\Psi' \vDash_{\mathsf{Pr}} p : t\sigma$ for some Ψ'.

By applying the induction hypothesis on the premise (1) and Ψ', we get a generalization-free derivation A of $(\Gamma\sigma)\Psi'' \vDash_{\mathsf{Pr}} e : (t'\sigma)\Psi'$ for some Ψ''.

We now consider the set of substitutions $\Psi''' = \Psi'' \cup \Psi$. By monotonicity (Lemma 13), we transform A into a generalization-free derivation A' of $(\Gamma\sigma)\Psi''' \vDash_{\mathsf{Pr}} e : (t'\sigma)\Psi'$ and B into a generalization-free derivation B' of $(\Gamma\sigma)\Psi''', x : (t'\sigma)\Psi' \vDash_{\mathsf{Pr}} p : t\sigma$.

We can thus conclude by building a generalization-free derivation of $(\Gamma\sigma)\Psi''' \vDash_{\mathsf{Pr}} \texttt{let}\, x = e\,;\, p : t\sigma$ using a [TopLevel-Let] node.

□

4.3.3 Deriving negations of arrows

Before proving the type safety, we have to deal with one last difficulty: the subject reduction still does not hold for our type system $\vDash$, even using the parallel semantics. As a counterexample, consider this reduction step:

$$((\lambda x.x)\ \texttt{false}, \lambda x.x) \rightsquigarrow_{\mathcal{P}} (\texttt{false}, \lambda x.x)$$

Let us call e the expression before reduction. We can derive for e the type $\texttt{False} \times ((\texttt{False} \to \texttt{False}) \wedge \neg(\texttt{False} \to \texttt{True}))$ by using a $[\vee]$ node as follows:

$$[\vee]\ \dfrac{\dfrac{\dfrac{A}{\varnothing \vDash \lambda x.x : \texttt{False} \to \texttt{False}} \qquad \dfrac{B}{\texttt{y} : (\texttt{False} \to \texttt{False}) \wedge (\texttt{False} \to \texttt{True}) \vDash (\texttt{y}\ \texttt{false}, \texttt{y}) : \mathbb{0}}}{\texttt{y} : (\texttt{False} \to \texttt{False}) \backslash (\texttt{False} \to \texttt{True}) \vDash (\texttt{y}\ \texttt{false}, \texttt{y}) : \texttt{False} \times ((\texttt{False} \to \texttt{False}) \backslash (\texttt{False} \to \texttt{True}))}\ C}{\varnothing \vDash (\texttt{y}\ \texttt{false}, \texttt{y})\{(\lambda x.x)/\texttt{y}\} : \texttt{False} \times ((\texttt{False} \to \texttt{False}) \backslash (\texttt{False} \to \texttt{True}))}$$

This $[\vee]$ node is decomposing the type of both occurrences of $\lambda x.x$ into $\texttt{False} \to \texttt{True}$ and $\neg(\texttt{False} \to \texttt{True})$. The subderivations A and C are straightforward. The subderivation B uses the fact that $(\texttt{False} \to \texttt{False}) \wedge (\texttt{False} \to \texttt{True}) \simeq \texttt{False} \to \mathbb{0}$, and thus the application can be typed $\mathbb{0}$ (which makes the pair also have type $\mathbb{0}$).

However, after one step of reduction, we obtain the expression $(\texttt{false}, \lambda x.x)$ for which it is not possible to infer the type $\texttt{False} \times ((\texttt{False} \to \texttt{False}) \backslash (\texttt{False} \to \texttt{True}))$: applying a $[\vee]$ rule as above would be useless as there is only one occurrence of $\lambda x.x$, and there is no other way to derive a negative arrow type for the λ-abstraction $\lambda x.x$.

Another way to interpret this example is by seeing the term $(\lambda x.x)\ \texttt{false}$ as a witness of the absurdity to derive the type $\texttt{False} \to \texttt{True}$ for $\lambda x.x$, thus allowing to

derive its negation $\neg(\texttt{False} \to \texttt{True})$. After reduction, this witness disappears, and thus the type $\neg(\texttt{False} \to \texttt{True})$ is not derivable anymore.

More generally, the property that we need for subject reduction to hold is a property of *atomicity* of the type of values: for an environment Γ, a typeable value v, and a type t, we must be able to derive either $\Gamma \models v : t$ or $\Gamma \models v : \neg t$ (a more specific version of this property is proved in Lemma 19). Unfortunately, this property does not hold in our current type system because of the impossibility to derive, in general, a negative arrow type for a λ-abstraction (e.g., by choosing $t = (\texttt{False} \to \texttt{True})$, we can neither derive $\varnothing \models \lambda x.x : t$ nor $\varnothing \models \lambda x.x : \neg t$).

Usually, set-theoretic type systems (such as Frisch et al. (2008)) overcome this difficulty by adding this deduction rule (in a language where λ-abstractions are explicitly annotated with their type):

$$[\textsc{Abs-}]\ \frac{\Gamma \vdash \lambda^{\bigwedge_{i\in I} s_i \to t_i} x.\ e : \mathbb{1}}{\Gamma \models e\{e'/\mathtt{x}\} : \neg(s \to t)}\ \neg(s \to t) \wedge \left(\bigwedge_{i\in I} s_i \to t_i\right) \not\simeq \mathbb{0}$$

It allows any λ-abstraction to be typed with a negative arrow type, as long as this negative arrow type is compatible (i.e., not disjoint) with the type annotated for the λ-abstraction. Using this rule, the λ-abstraction $\lambda^{\texttt{False}\to\texttt{False}} x.x$ can be typed $\neg(\texttt{False} \to \texttt{True})$. Soundness is ensured by the side condition: it prevents a λ-abstraction from being given an arrow type $s \to t$ and its negation $\neg(s \to t)$, which could then yield the type $\mathbb{0}$ by using a $[\wedge]$ rule.

In our language, λ-abstractions are not annotated with (the positive part) of their type. Thus, we cannot add a rule such as the [Abs-] rule above. Instead, we define a new type system $\models_{\mathcal{N}}$, shown in Figure 4.4.

There are several things to note about this new type system. First, there is no intersection rule: instead, the $[\to$I-N$]$ rule can directly infer a conjunction of several arrow types (in other terms, the intersection rule has been embedded in the $[\to$I-N$]$ rule). As shown in Lemma 15, this can be done without loss of generality. The reason why the intersection rule has been eliminated is because it would make the type system unsafe in the presence of this new $[\to$I-N$]$ rule (this is explained later).

Second, the $[\to$I-N$]$ rule allows deriving any conjunction of negative arrow types for a λ-abstraction, as long as these negative arrow types are compatible with the positive arrow types inferred. This is ensured by the side condition $\bigwedge_{i\in I}(\mathbf{u}_i \to t_i) \wedge \bigwedge_{j\in J} \neg(\mathbf{v}_j \to \mathbf{v}'_j) \not\simeq \mathbb{0}$. This side condition prevents a $[\to$I-N$]$ node from deriving the type $\mathbb{0}$ for a λ-abstraction, which would be unsound. Note that this is also the reason why the intersection rule has been removed: it would make it possible to infer the type $\mathbb{0}$ for a λ-abstraction, as shown by the derivation below.

$$[\wedge]\ \frac{[\to\text{I-N}]\ \dfrac{[\textsc{Var}_\lambda\text{-N}]\ \dfrac{}{x : \mathbb{0} \models_{\mathcal{N}} x : \mathbb{0}}}{\varnothing \models_{\mathcal{N}} \lambda x.x : (\mathbb{0} \to \mathbb{0}) \wedge \neg(\texttt{False} \to \texttt{False})} \qquad [\to\text{I-N}]\ \dfrac{[\textsc{Var}_\lambda\text{-N}]\ \dfrac{}{x : \texttt{False} \models_{\mathcal{N}} x : \texttt{False}}}{\varnothing \models_{\mathcal{N}} \lambda x.x : \texttt{False} \to \texttt{False}}}{\varnothing \models_{\mathcal{N}} \lambda x.x : \mathbb{0}}$$

Although it is more subtle, the side condition $\forall i, j \in I.\ i \neq j \Rightarrow \mathbf{u}_i \wedge \mathbf{u}_j \simeq \mathbb{0}$ has the same purpose: it prevents a potentially unsound intersection from happening.

$$[\textsc{Const-N}]\ \frac{}{\Gamma \vdash_{\mathcal{N}} c : \boldsymbol{b}_c} \qquad [\textsc{Var}_\lambda\textsc{-N}]\ \frac{}{\Gamma \vdash_{\mathcal{N}} x : \Gamma(x)\rho} \qquad [\textsc{Var}_\vee\textsc{-N}]\ \frac{}{\Gamma \vdash_{\mathcal{N}} \mathsf{x} : \Gamma(\mathsf{x})\rho}$$

$$[\to\text{I-N}]\ \frac{(\forall i \in I) \quad \Gamma, x : \mathbf{u}_i \vdash_{\mathcal{N}} e : t_i}{\Gamma \vdash_{\mathcal{N}} \lambda x.e : \bigwedge_{i\in I}(\mathbf{u}_i \to t_i) \wedge \bigwedge_{j\in J} \neg(\mathbf{v}_j \to \mathbf{v}'_j)}\ I \neq \varnothing,\ \ \forall i,j \in I.\ i \neq j \Rightarrow \mathbf{u}_i \wedge \mathbf{u}_j \simeq \mathbb{0}\ \ (\textstyle\bigwedge_{i\in I}(\mathbf{u}_i \to t_i) \wedge \bigwedge_{j\in J} \neg(\mathbf{v}_j \to \mathbf{v}'_j)) \not\simeq \mathbb{0}$$

$$[\to\text{E-N}]\ \frac{\Gamma \vdash_{\mathcal{N}} e_1 : t_1 \to t_2 \qquad \Gamma \vdash_{\mathcal{N}} e_2 : t_1}{\Gamma \vdash_{\mathcal{N}} e_1 e_2 : t_2} \qquad [\times\text{I-N}]\ \frac{\Gamma \vdash_{\mathcal{N}} e_1 : t_1 \qquad \Gamma \vdash_{\mathcal{N}} e_2 : t_2}{\Gamma \vdash_{\mathcal{N}} (e_1, e_2) : t_1 \times t_2}$$

$$[\times\text{E}_1\text{-N}]\ \frac{\Gamma \vdash_{\mathcal{N}} e : t_1 \times t_2}{\Gamma \vdash_{\mathcal{N}} \pi_1 e : t_1} \qquad [\times\text{E}_2\text{-N}]\ \frac{\Gamma \vdash_{\mathcal{N}} e : t_1 \times t_2}{\Gamma \vdash_{\mathcal{N}} \pi_2 e : t_2}$$

$$[\in_1\text{-N}]\ \frac{\Gamma \vdash_{\mathcal{N}} e : \tau \qquad \Gamma \vdash_{\mathcal{N}} e_1 : t_1}{\Gamma \vdash_{\mathcal{N}} (e\in\tau)\,?\,e_1 : e_2 : t_1} \qquad [\in_2\text{-N}]\ \frac{\Gamma \vdash_{\mathcal{N}} e : \neg\tau \qquad \Gamma \vdash_{\mathcal{N}} e_2 : t_2}{\Gamma \vdash_{\mathcal{N}} (e\in\tau)\,?\,e_1 : e_2 : t_2}$$

$$[\mathbb{0}\text{-N}]\ \frac{\Gamma \vdash_{\mathcal{N}} e : \mathbb{0}}{\Gamma \vdash_{\mathcal{N}} (e\in\tau)\,?\,e_1 : e_2 : \mathbb{0}} \qquad [\leqslant\text{-N}]\ \frac{\Gamma \vdash_{\mathcal{N}} e : t}{\Gamma \vdash_{\mathcal{N}} e : t'}\ t \leqslant t'$$

$$[\vee\text{-N}]\ \frac{\Gamma \vdash_{\mathcal{N}} e' : s \qquad (\forall i \in I) \quad \Gamma, \mathsf{x} : s \wedge \mathbf{u}_i \vdash_{\mathcal{N}} e : t}{\Gamma \vdash_{\mathcal{N}} e\{e'/\mathsf{x}\} : t}\ \{\mathbf{u}_i\}_{i\in I} \in \mathsf{Part}(\mathbb{1})$$

Figure 4.4: Declarative Type System with Negations of Arrows

Consider the expression $(\lambda y.(\lambda x.x))\ 42$. As we have just seen, $\lambda x.x$ can be typed $\mathtt{False} \to \mathtt{False}$ using a [→I-N] node, and it can also be typed $\neg(\mathtt{False} \to \mathtt{False})$ using a different [→I-N] node. When typing the outer λ-abstraction $\lambda y.(\lambda x.x)$ using a [→I-N] node without this side condition, we may choose to type it twice for the same domain $\mathbb{1}$. This way, we may derive the type $(\mathbb{1} \to (\mathtt{False} \to \mathtt{False})) \wedge (\mathbb{1} \to (\neg(\mathtt{False} \to \mathtt{False})))$. The issue with this type is that it is equivalent to $\mathbb{1} \to ((\mathtt{False} \to \mathtt{False}) \wedge \neg(\mathtt{False} \to \mathtt{False})) \simeq \mathbb{1} \to \mathbb{0}$. Consequently, the type $\mathbb{0}$ can be derived for $(\lambda y.(\lambda x.x))\ 42$, which is unsound. This issue can be avoided by forcing the domains explored for a λ-abstraction to be disjoint. By doing so, the different codomains are never intersected, regardless of the value to which this λ-abstraction is applied. This side condition is used in the proof of subject reduction (Lemma 21, case [→E-N]).

Terminology.

Structural rules [Const-N] [Var_λ-N] [→I-N] [→E-N] [×I-N] [$\times\text{E}_1$-N] [$\times\text{E}_2$-N] [$\mathbb{0}$-N] [$\in_1$-N] [$\in_2$-N]

Non-structural rules [$\text{Var}_\vee$-N] [$\vee$-N] [$\leqslant$-N]

The proofs of subject reduction and progress require an additional constraint on

the shape of these derivations.

Definition 43 (Union-free derivation)**.** *A derivation D of $\Gamma \vDash_{\mathcal{N}} e : t$ is said* union-free *if, for every* [$\vee$-N] *node N of path π in D, π can be decomposed into $\pi_1;\pi_2$ such that $D(\pi_1)$ is a* [$\to$I-N] *node.*

Roughly, a union-free derivation does not contain any [$\vee$-N] node except in the subderivation of the premise of a [$\to$I-N] node.

We now introduce a weaker version of canonical derivations, namely *weakly-canonical derivations*, which will be preserved throughout the transformations defined in the following proofs.

Definition 44 (Weakly-canonical derivation)**.** *A derivation D of $\Gamma \vDash_{\mathcal{N}} e : t$ is said* weakly-canonical *if and only if:*

- *The premises of every structural node except* [$\to$I-N] *are union-free, and*
- *The definition premise of every* [$\vee$-N] *node is union-free.*

In the lemmas and theorems below, most of the derivations we manipulate will be weakly-canonical.

Lemma 15 (Inclusion of $\vDash$ in $\vDash_{\mathcal{N}}$)**.** *Let Γ be an environment such that $\mathsf{vars}(\Gamma) \subseteq \mathcal{V}_M$, e an expression, and t a type. Let D be a derivation of $\Gamma \vDash e : t$. Then, there exists a weakly-canonical derivation of $\Gamma \vDash_{\mathcal{N}} e : t$.*

Proof. Using Theorem 1, we transform D into an instantiation-free canonical derivation of $\Gamma \vDash e : s$ for $s \lhd t$ (the fact that it is instantiation-free is a direct consequence of $\mathsf{vars}(\Gamma) \subseteq \mathcal{V}_M$).

We then show the following result: for any instantiation-free canonical derivation D of $\Gamma \vDash e : s$, and for any type t such that $s \lhd t$, we can build a weakly-canonical derivation of $\Gamma \vDash_{\mathcal{N}} e : t$. We proceed by induction on $(\mathsf{s}(e), |D|)$ for the lexicographic order, where $\mathsf{s}(e)$ is the number of λ-abstractions in e.

We first build a weakly-canonical derivation of $\Gamma \vDash_{\mathcal{N}} e : s$:

- If the root of D is a [$\wedge$] node, then all its premises must be [$\to$I] nodes (because D is an instantiation-free canonical derivation). Thus, we know that $e \equiv \lambda x.e'$. Let $\{(\mathbf{u}_i, \mathbf{u}'_i)\}_{i\in I}$ be the pairs of domain and codomain used by those [$\to$I] nodes. We have $s \simeq \bigwedge_{i\in I}(\mathbf{u}_i \to \mathbf{u}'_i)$. We consider a partition $\{\mathbf{v}_j\}_{j\in J}$ of $\bigvee_{i\in I} \mathbf{u}_i$ such that $\forall i \in I.\ \forall j \in J.\ \mathbf{u}_i \wedge \mathbf{v}_j \simeq \mathbb{0}$ or $\mathbf{v}_j \leqslant \mathbf{u}_i$ (such a partition can easily be built by induction on $|\{\mathbf{u}_i\}_{i\in I}|$). If $\bigvee_{i\in I} \mathbf{u}_i \simeq \mathbb{0}$, we consider $\{\mathbf{v}_j\}_{j\in J} = \{\mathbb{0}\}$ instead.

  For each $j \in J$, we do the following:

  1. We pose $I_j = \{i \in I \mid \mathbf{v}_j \leqslant \mathbf{u}_i\}$. Note that $\forall i \in I \backslash I_j.\ \mathbf{u}_i \wedge \mathbf{v}_j \simeq \mathbb{0}$.

2. For each $i \in I_j$, we derive $\Gamma, x : \mathbf{v}_j \vDash e' : \mathbf{u}'_i$ by applying the monotonicity lemma (Lemma 1) on the derivation $\Gamma, x : \mathbf{u}_i \vDash e' : \mathbf{u}'_i$.
3. We intersect the resulting derivations with a $[\wedge]$ node, yielding a derivation of $\Gamma, x : \mathbf{v}_j \vDash e' : \bigwedge_{i \in I_j} \mathbf{u}'_i$.
4. We apply Theorem 1 on this new derivation in order to get a canonical form derivation of $\Gamma, x : \mathbf{v}_j \vDash e' : s_j$ with $s_j \lhd \bigwedge_{i \in I_j} \mathbf{u}'_i$.
5. We apply the induction hypothesis on it in order to build a weakly-canonical derivation of $\Gamma, x : \mathbf{v}_j \vDash_{\mathcal{N}} e' : \bigwedge_{i \in I_j} \mathbf{u}'_i$.

We regroup the resulting derivations (for each $j \in J$) in a $[\to$I-N$]$ node (it can be verified that $\bigwedge_{j \in J}(\mathbf{v}_j \to \bigwedge_{i \in I_j} \mathbf{u}'_i) \simeq \bigwedge_{i \in I}(\mathbf{u}_i \to \mathbf{u}'_i) \simeq s$).

- The other cases are just straightforward applications of the induction hypothesis.

We have a derivation of $\Gamma \vDash_{\mathcal{N}} e : s$ with $s \lhd t$. As $\mathsf{vars}(\Gamma) \subseteq \mathcal{V}_M$, we know that $\mathsf{vars}(s) \subseteq \mathcal{V}_M$ and thus $s \leqslant t$. Consequently, using a $[\leqslant$-N$]$ node, we can derive a weakly-canonical derivation of $\Gamma \vDash_{\mathcal{N}} e : t$, which concludes the proof. □

We now define the $\vDash_{\mathcal{N},\mathsf{Pr}}$ typing rules for programs:

$$[\textsc{TopLevel-Expr-N}]\ \frac{\Gamma \vDash_{\mathcal{N}} e : t}{\Gamma \vDash_{\mathcal{N},\mathsf{Pr}} e : t}$$

$$[\textsc{TopLevel-Let-N}]\ \frac{\Gamma \vDash_{\mathcal{N},\mathsf{Pr}} e : t \qquad \Gamma, x : t \vDash_{\mathcal{N},\mathsf{Pr}} p : t'}{\Gamma \vDash_{\mathcal{N},\mathsf{Pr}} \mathtt{let}\, x = e\, ;\, p : t'}$$

Definition 45 (Weakly-canonical derivation)**.** *A derivation D of $\Gamma \vDash_{\mathcal{N},\mathsf{Pr}} p : t$ is said* weakly-canonical *if and only if every subderivation $\Gamma' \vDash_{\mathcal{N}} e' : t'$ it contains is weakly-canonical.*

Lemma 16 (Inclusion of $\vDash_{\mathsf{Pr}}$ in $\vDash_{\mathcal{N},\mathsf{Pr}}$)**.** *Let p be a program, and t a type. Let D be a derivation of $\varnothing \vDash_{\mathsf{Pr}} p : t$. Then, there exists a weakly-canonical derivation of $\varnothing \vDash_{\mathcal{N},\mathsf{Pr}} p : t$.*

Proof. We first apply Lemma 14 on D in order to derive a generalization-free derivation D' of $\varnothing \vDash_{\mathsf{Pr}} p : t$.

Then, proceed with a straightforward structural induction on D'. For the case of a [TopLevel-Expr] root, we apply Lemma 15 on the premise. □

4.3.4 Subject reduction

In this section, we prove a subject reduction theorem for the type system $\vDash_{\mathcal{N}}$ and the parallel semantics $\rightsquigarrow_{\mathcal{P}}$.

Lemma 17 (Monotonicity)**.** *Let Γ be an environment, e an expression, and t a type. Let D be a weakly-canonical derivation of $\Gamma \vDash_{\mathcal{N}} e : t$. Let Γ' be an environment such that $\Gamma' \leqslant \Gamma$. Then, there exists a weakly-canonical derivation of $\Gamma' \vDash_{\mathcal{N}} e : t$.*

Proof. We show this result by structural induction on the proof tree D. When the root is a [VAR$_\vee$-N] or [VAR$_\lambda$-N] node, we conclude by applying [$\leqslant$-N]. All the other cases are just straightforward applications of the induction hypothesis. □

Lemma 18 (Substitution lemma)**.** *Let Γ be an environment, $\boldsymbol{x}$ a binding or lambda variable, e, e' two expressions, and s, t two types. Let D be a weakly-canonical derivation of $\Gamma, \boldsymbol{x} : s \vDash_{\mathcal{N}} e : t$, and A a weakly-canonical derivation of $\Gamma \vDash_{\mathcal{N}} e' : s$. Then, there exists a weakly-canonical derivation of $\Gamma \vDash_{\mathcal{N}} e\{e'/\boldsymbol{x}\} : t$.*

Proof. One naive way to construct a derivation of $\Gamma \vDash_{\mathcal{N}} e\{e'/\boldsymbol{x}\} : t$ is to replace in D the [VAR$_\lambda$-N] or [VAR$_\vee$-N] nodes applied on $\boldsymbol{x}$ by A. However, this would not necessarily yield a weakly-canonical derivation. In order to get a weakly-canonical derivation, we will move the [$\vee$-N] nodes in A at the root of D.

We proceed by structural induction on A.

Depending on the root of A:

Axiom node We replace in D every [VAR$_\lambda$-N] or [VAR$_\vee$-N] node applied on $\boldsymbol{x}$ by the axiom node, and return the resulting derivation.

Structural node We proceed similarly to the previous case: we replace in D every [VAR$_\lambda$-N] or [VAR$_\vee$-N] node applied on $\boldsymbol{x}$ by A, and return the resulting derivation. The derivation obtained is weakly-canonical: as A is weakly-canonical and its root is structural, all [$\vee$-N] nodes contained in A must be in the subderivation of the premise of a [$\to$I-N] node.

[$\leqslant$-N] Let us call A' the subderivation of the premise of this [$\leqslant$-N] node. Let t' be the type derived by A', and t the type derived by A. We replace in the derivation D every environment Γ by the environment $\Gamma \backslash \{(\boldsymbol{x}, t)\}, \boldsymbol{x} : t'$, and replace every [VAR$_\lambda$-N] or [VAR$_\vee$-N] node applied on $\boldsymbol{x}$ by the following subderivation:

$$[\leqslant\text{-N}]\ \frac{[\text{VAR}_\lambda\text{-N}/\text{VAR}_\vee\text{-N}]\ \dfrac{}{\Gamma' \vDash_{\mathcal{N}} \boldsymbol{x} : t'}}{\Gamma' \vDash_{\mathcal{N}} \boldsymbol{x} : t}$$

Let us call D' the resulting derivation. We conclude by applying the induction hypothesis on D' and A'.

[∨-N] We apply the following transformation to A (where $e' \equiv e'_1\{e'_2/\mathsf{y}\}$, and $e\{e'/\boldsymbol{x}\} \equiv e\{(e'_1\{e'_2/\mathsf{y}\})/\boldsymbol{x}\} \equiv e\{e'_1/\boldsymbol{x}\}\{e'_2/\mathsf{y}\}$):

$$[\vee\text{-N}]\ \frac{\dfrac{A'}{\Gamma \vDash_{\mathcal{N}} e'_2 : s} \qquad (\forall i \in I) \quad \dfrac{A'_i}{\Gamma, \mathsf{y} : s \wedge \mathbf{u}_i \vDash_{\mathcal{N}} e'_1 : t}}{\Gamma \vDash_{\mathcal{N}} e'_1\{e'_2/\mathsf{y}\} : t}$$

$$\downarrow$$

$$[\vee\text{-N}]\ \frac{\dfrac{A'}{\Gamma \vDash_{\mathcal{N}} e'_2 : s} \qquad (\forall i \in I) \quad \dfrac{D'_i}{\Gamma, \mathsf{y} : s \wedge \mathbf{u}_i \vDash_{\mathcal{N}} e\{e'_1/\boldsymbol{x}\} : t}}{\Gamma \vDash_{\mathcal{N}} e\{e'_1/\boldsymbol{x}\}\{e'_2/\mathsf{y}\} : t}$$

where D'_i is obtained by applying the induction hypothesis on D and A'_i

(the environment used by D can be extended with $\mathsf{y} : s \wedge \mathbf{u}_i$ using Lemma 17).

□

To prove subject reduction, we need to constrain the use of [∨-N] nodes furthermore: [∨-N] nodes should not decompose the type of a value. We prove in Lemma 20 that [∨-N] nodes decomposing the type of a value are not really useful and can be eliminated.

Definition 46 (Avoidable [∨-N] node)**.** *Let D be a derivation. Let N be a* [∨-N] *node of D performing the substitution $\{e'/x\}$. N is said* avoidable *if and only if e' is a value.*

Definition 47 (Minimal derivation)**.** *A derivation is said* minimal *if it does not contain any avoidable* [∨-N] *node.*

Lemma 19 (Atomicity of union-free value derivations)**.** *Let Γ be an environment, and v a value. Let D be a union-free minimal weakly-canonical derivation of $\Gamma \vDash_{\mathcal{N}} v : s$. Let $\{\boldsymbol{u}_i\}_{i\in I}$ be a partition of $\mathbb{1}$. Then, there exists for some $i \in I$ a union-free minimal weakly-canonical derivation of $\Gamma \vDash_{\mathcal{N}} v : s \wedge \boldsymbol{u}_i$.*

Proof. We proceed by structural induction on D.

- If the root of D is a [→I-N] node, then v is a λ-abstraction $\lambda x.e$, and s is a conjunction of positive and negative arrows. Let us show that we can find $i \in I$ such that $s \wedge \mathbf{u}_i \geqslant s \wedge \bigwedge_{j\in J} \neg(\mathbf{v}_j \to \mathbf{v}'_j)$ and $s \wedge \bigwedge_{j\in J} \neg(\mathbf{v}_j \to \mathbf{v}'_j) \not\simeq \mathbb{0}$ for some $\{(\mathbf{v}_j, \mathbf{v}'_j)\}_{j\in J}$.

  By contradiction, let us assume that it is not the case. Let us fix $i \in I$. We

consider a set of types $\{\mathbf{u}_{i,j}\}_{j\in J_i}$ such that $s \wedge \mathbf{u}_i \overset{\text{DNF}}{\simeq} \bigvee_{j\in J_i} s \wedge \mathbf{u}_{i,j}$, where every $\mathbf{u}_{i,j}$ is a conjunction of positive and negative arrows. We know that, for every $j \in J_i$, $s \wedge \mathbf{u}_{i,j} \not\geqslant s \wedge \mathbf{v}$ for every conjunction of negative arrows $\mathbf{v}$ such that $s \wedge \mathbf{v} \not\simeq \mathbb{0}$. This means that, for every $j \in J_i$, there exists a positive arrow $\mathbf{v}_{i,j} \to \mathbf{v}'_{i,j}$ such that $s \wedge \mathbf{u}_{i,j} \leqslant s \wedge (\mathbf{v}_{i,j} \to \mathbf{v}'_{i,j})$ and $(\mathbf{v}_{i,j} \to \mathbf{v}'_{i,j}) \not\leqslant s$. Thus, we have $s \wedge \mathbf{u}_i \leqslant s \wedge (\bigvee_{j\in J_i} \mathbf{v}_{i,j} \to \mathbf{v}'_{i,j})$ with $\bigvee_{j\in J_i} \mathbf{v}_{i,j} \to \mathbf{v}'_{i,j} \not\leqslant s$.

As this is true for every $i \in I$, we have $s \wedge \bigvee_{i\in I} \mathbf{u}_i \leqslant s \wedge \bigvee_{i\in I}(\bigvee_{j\in J_i} \mathbf{v}_{i,j} \to \mathbf{v}'_{i,j}) \not\leqslant s$, which contradicts the fact that $\{\mathbf{u}_i\}_{i\in I}$ is a partition of $\mathbb{1}$. Thus, there exists some $i \in I$ and $\{(\mathbf{v}_j, \mathbf{v}'_j)\}_{j\in J}$ such that $s \wedge \mathbf{u}_i \geqslant s \wedge \bigwedge_{j\in J} \neg(\mathbf{v}_j \to \mathbf{v}'_j) \not\simeq \mathbb{0}$.

We can thus derive $\Gamma \vDash_{\mathcal{N}} \lambda x.e : s \wedge \bigwedge_{j\in J} \neg(\mathbf{v}_j \to \mathbf{v}'_j)$ using a [→I-N] node, and conclude by inserting a [⩽-N] node at the root to derive $\Gamma \vDash_{\mathcal{N}} \lambda x.e : s \wedge \mathbf{u}_i$.

- The other cases are straightforward.

□

Lemma 20. *Let Γ be an environment, e an expression, and t a type. Let D be a weakly-canonical derivation of $\Gamma \vDash_{\mathcal{N}} e : t$. Then, there exists a minimal weakly-canonical derivation of $\Gamma \vDash_{\mathcal{N}} e : t$.*

Proof. We proceed by structural induction on D, with an additional inductive invariant stating that if D is union-free, then the minimal weakly-canonical derivation we build is also union-free.

- If the root is an axiom ([Const-N], [Var$_\lambda$-N] or [Var$_\vee$-N]), then D is already a minimal weakly-canonical derivation.
- If the root is an avoidable [∨-N] node, we know that it is doing a substitution $e\{v/\mathsf{x}\}$ with v a value. We note $\{\mathbf{u}_i\}_{i\in I}$ the partition of $\mathbb{1}$ used by this node.

  We apply the induction hypothesis on the definition premise $\Gamma \vDash_{\mathcal{N}} v : s$ in order to get a union-free minimal weakly-canonical derivation A of $\Gamma \vDash_{\mathcal{N}} v : s$. By applying Lemma 19 on A, we construct a (union-free) minimal weakly-canonical derivation A' of $\Gamma \vDash_{\mathcal{N}} v : s \wedge \mathbf{u}_i$ for some $i \in I$.

  We consider the corresponding body premise B, $\Gamma, \mathsf{x} : s \wedge \mathbf{u}_i \vDash_{\mathcal{N}} e : t$, and we apply the induction hypothesis on it in order to get a minimal weakly-canonical derivation B' of $\Gamma, \mathsf{x} : s \wedge \mathbf{u}_i \vDash_{\mathcal{N}} e : t$.

  We can then apply Lemma 18 on B', x and A' to construct a minimal weakly-canonical derivation of $\Gamma \vDash_{\mathcal{N}} e\{v/\mathsf{x}\} : t$ (note that, as both B' and A' are minimal, the derivation resulting from Lemma 18 is also minimal as it does not introduce any new [∨-N] node).
- The other cases are straightforward applications of the induction hypothesis.

□

Now, that we have restricted the shape of derivations, we can prove the subject reduction property.

Proposition 8. *Let Γ be an environment, v a value, and τ a test type. Let D be a derivation of $\Gamma \vDash_{\mathcal{N}} v : \tau$. Then, we have the relation $v \in \tau$ (see Figure 3.1 for the definition of $\in$).*

Proof. Straightforward structural induction on D. Note that the case of λ-abstractions is trivial as arrows in τ can only be $\mathbb{0} \to \mathbb{1}$. □

Proposition 8 will be used for proving the progress (Lemma 22), but we also need it for the subject reduction, for treating the case of type-cases.

The next lemma is, in a sense, a more general version of the subject reduction property, where the reduction can happen under any context and is replicated everywhere in the expression.

Lemma 21. *Let Γ be an environment, e an expression, and t a type. Let D be a minimal weakly-canonical derivation of $\Gamma \vDash_{\mathcal{N}} e : t$. Let $e_\circ$ and $e'_\circ$ be two expressions such that $e_\circ \rightsquigarrow_\top e'_\circ$. Then, there exists a weakly-canonical derivation of $\Gamma \vDash_{\mathcal{N}} e\{e'_\circ/e_\circ\} : t$.*

Proof. We proceed by structural induction on D.

If e contains no occurrence of $e_\circ$ (modulo α-renaming), the result is trivial. Thus, let us assume that e contains at least one occurrence of $e_\circ$.

We denote by e' the expression $e\{e'_\circ/e_\circ\}$, and consider the root of D:

[Const-N] Impossible case (e cannot contain any reducible expression).

[Var$_\lambda$-N] Impossible case (e cannot contain any reducible expression).

[Var$_\vee$-N] Impossible case (e cannot contain any reducible expression).

[$\leqslant$-N] By applying the induction hypothesis on the premise $\Gamma \vDash_{\mathcal{N}} e : t'$ (with $t' \leqslant t$), we get a derivation of $\Gamma \vDash_{\mathcal{N}} e' : t'$, thus we can derive $\Gamma \vDash_{\mathcal{N}} e' : t$ by using a [$\leqslant$-N] node.

[$\to$I-N] We have $e' \equiv \lambda x.\ (e_\lambda\{e'_\circ/e_\circ\})$ for some expression e_λ. For each $i \in I$, we can derive $\Gamma, x : \mathbf{u}_i \vDash_{\mathcal{N}} e_\lambda\{e'_\circ/e_\circ\} : t_i$ by applying the induction hypothesis on the premise $\Gamma, x : \mathbf{u}_i \vDash_{\mathcal{N}} e_\lambda : t_i$, and conclude by using a [$\to$I-N] node.

[$\times$I-N] We have $e' \equiv (e_1\{e'_\circ/e_\circ\}, e_2\{e'_\circ/e_\circ\})$ for some e_1 and e_2. We apply the induction hypothesis on the premises, as in the previous case.

[→E-N] We have $e \equiv e_1 e_2$ for some expressions e_1 and e_2. If $e_\circ$ is a subexpression of e_1 and/or e_2, we conclude by applying the induction hypothesis on the premises, as in the previous case.

Otherwise, $e_\circ \equiv e_1 e_2$ and thus the reduction $e_\circ \rightsquigarrow_\top e'_\circ$ uses the rule 4.1. Consequently, we know that $e_\circ \equiv e \equiv (\lambda x.\ e_\lambda) v$ for some expression e_λ and value v, and $e'_\circ \equiv e' \equiv e_\lambda\{v/x\}$.

We have the following premises:

1. $\Gamma \vDash_{\mathcal{N}} \lambda x.\ e_\lambda : t_1 \to t_2$ (with $t_2 \simeq t$)
2. $\Gamma \vDash_{\mathcal{N}} v : t_1$

As D is a weakly-canonical derivation, we know that the premise (1) is union-free. Thus, we can extract from (1) a collection of derivations $\{A_i\}_{i\in I}$ of the judgments $\Gamma, x : \mathbf{u}_i \vDash_{\mathcal{N}} e_\lambda : s_i$ for $i \in I$, such that $\forall i, j \in I.\ i \neq j \Rightarrow \mathbf{u}_i \wedge \mathbf{u}_j \simeq \mathbb{0}$, and such that $\bigwedge_{i\in I}(\mathbf{u}_i \to s_i) \leqslant t_1 \to t_2$. If the partition $\{\mathbf{u}_i\}_{i\in I}$ does not cover $\mathbb{1}$, we extend it with the part $\mathbb{1}\backslash(\bigvee_{i\in I} \mathbf{u}_i)$, yielding a new partition $\{\mathbf{u}_i\}_{i\in I'}$ with $I' = I \cup \{k\}$ (if $\{\mathbf{u}_i\}_{i\in I}$ already covers $\mathbb{1}$, then $I' = I$).

By applying Lemma 19 on the premise (2) and the partition $\{\mathbf{u}_i\}_{i\in I'}$, we are able to derive a minimal weakly-canonical derivation B of $\Gamma \vDash_{\mathcal{N}} v : t_1 \wedge \mathbf{u}_i$ for some $i \in I'$. We can deduce from Proposition 8 that $t_1 \wedge \mathbf{u}_i \not\simeq \mathbb{0}$ (otherwise we would have $v \in \mathbb{0}$), and thus we have $i \in I$. Using the fact that $\bigwedge_{i\in I}(\mathbf{u}_i \to s_i) \leqslant t_1 \to t_2$ and that all $\mathbf{u}_i$ are disjoint, we deduce $s_i \leqslant t_2$.

By inserting a [⩽-N] node at the root of B, we obtain a derivation B' of $\Gamma \vDash_{\mathcal{N}} v : \mathbf{u}_i$. Using the substitution lemma (Lemma 18) on A_i and B', we obtain a weakly-canonical derivation of $\Gamma \vDash_{\mathcal{N}} e_\lambda\{v/x\} : s_i$. By inserting a [⩽-N] node at its root, we obtain $\Gamma \vDash_{\mathcal{N}} e_\lambda\{v/x\} : t_2$.

[×E$_1$-N] We have $e \equiv \pi_1 e_1$ for some expression e_1. If $e_\circ$ is a subexpression of e_1, we conclude by applying the induction hypothesis on the premise.

Otherwise, $e_\circ \equiv \pi_1 e_1$ and thus the reduction $e_\circ \rightsquigarrow_\top e'_\circ$ uses the rule 4.2. Consequently, we know that $e_\circ \equiv e \equiv \pi_1(v_1, v_2)$ for some values v_1 and v_2, and $e'_\circ \equiv e' \equiv v_1$.

As D is a weakly-canonical derivation, we know that the premise $\Gamma \vDash_{\mathcal{N}} (v_1, v_2) : t_1 \times t_2$ (with $t_1 \simeq t$) is union-free, and thus we can extract from it a derivation A_1 of $\Gamma \vDash_{\mathcal{N}} v_1 : s$ and a derivation A_2 of $\Gamma \vDash_{\mathcal{N}} v_2 : s'$ such that $s \times s' \leqslant t_1 \times t_2$. We thus have $s \leqslant t_1$, as s' cannot be $\mathbb{0}$ (this would contradict Proposition 8).

Therefore, we can conclude this case by using a [⩽-N] node with the premise A_1 in order to derive $\Gamma \vDash_{\mathcal{N}} v_1 : t_1$.

[×E$_2$-N] Similar to the previous case.

[$\vee$-N] We have $e \equiv e_1\{e_2/\mathsf{x}\}$ for some expressions e_1 and e_2, and thus $e' \equiv (e_1\{e_2/\mathsf{x}\})\{e'_\circ/e_\circ\}$.

We have the following premises:

Definition premise $\Gamma \vDash_{\mathcal{N}} e_2 : s$

Body premises $\forall i \in I.\ \Gamma, \mathsf{x} : s \wedge \mathbf{u}_i \vDash_{\mathcal{N}} e_1 : t$

As D is minimal, we know that e_2 cannot be a value. Also, we know that $e_\circ$ does not contain x (otherwise there would be no occurrence of $e_\circ$ in e). Consequently, $e'_\circ$ does not contain x neither, because a reduction step cannot introduce a new free variable.

There are several cases:

- $e_\circ$ does not contain e_2 and e_2 does not contain $e_\circ$.
  In this case, we have:
  $e' \equiv (e_1\{e_2/\mathsf{x}\})\{e'_\circ/e_\circ\} \equiv (e_1\{e'_\circ/e_\circ\})\{e_2/\mathsf{x}\}$. Thus, we can conclude by keeping the definition premise of the [$\vee$-N] node and applying the induction hypothesis on the body premises.
- e_2 contains $e_\circ$.
  In this case, we pose $e'_2 = e_2\{e'_\circ/e_\circ\}$.
  We have $e' \equiv (e_1\{e_2/\mathsf{x}\})\{e'_\circ/e_\circ\} \equiv (e_1\{e'_\circ/e_\circ\})\{e'_2/\mathsf{x}\}$. We can derive $\Gamma \vDash_{\mathcal{N}} e'_2 : s$ by induction on the definition premise, and $\Gamma, \mathsf{x} : s \wedge \mathbf{u}_i \vDash_{\mathcal{N}} e_1\{e'_\circ/e_\circ\} : t$ for every $i \in I$ by induction on the body premises. Thus, we can derive $\Gamma \vDash_{\mathcal{N}} (e_1\{e'_\circ/e_\circ\})\{e'_2/\mathsf{x}\} : t$ using a [$\vee$-N] node.
- $e_\circ$ contains e_2 as a strict subexpression.
  Let $\{e_{\circ,i}\}_{i\in I}$ the set of subexpressions e'' of e_1 that satisfy $e''\{e_2/\mathsf{x}\} \equiv_\alpha e_\circ$. As e_2 is not a value, it can only appear in $e_\circ$ inside a λ-abstraction, and/or inside a branch of a type-case: otherwise, $e_\circ$ would not be reducible. Thus, we can deduce that, for every $i \in I$, $e_{\circ,i} \rightsquigarrow_\top e'_{\circ,i}$ for some $e'_{\circ,i}$. By numbering the elements of I from 1 to n, we have $e' \equiv (e_1\{e_2/\mathsf{x}\})\{e'_\circ/e_\circ\} \equiv (e_1\{e'_{\circ,1}/e_{\circ,1}\} \dots \{e'_{\circ,n}/e_{\circ,n}\})\{e_2/\mathsf{x}\}$.
  Consequently, we can conclude by keeping the definition premise of the [$\vee$-N] node and applying the induction hypothesis n times on the body premises (once for the reduction $e_{\circ,1} \rightsquigarrow_\top e'_{\circ,1}$, then once for the reduction $e_{\circ,2} \rightsquigarrow_\top e'_{\circ,2}$, etc.).

[$\mathbb{0}$-N] We have $e \equiv (e_1 \in \tau)\ ?\ e_2 : e_3$ for some e_1, e_2 and e_3. As values cannot have the type $\mathbb{0}$ (Proposition 8), we know that e_1 is not a value. Thus, $e' \equiv (e_1\{e'_\circ/e_\circ\} \in \tau)\ ?\ e_2\{e'_\circ/e_\circ\} : e_3\{e'_\circ/e_\circ\}$. We can derive $\Gamma \vDash_{\mathcal{N}} e_1\{e'_\circ/e_\circ\} : \mathbb{0}$ by applying the induction hypothesis on the premise, and then we can derive $\Gamma \vDash_{\mathcal{N}} e' : \mathbb{0}$ by using [$\mathbb{0}$-N].

[$\in_1$-N] We have $e \equiv (e_1 \in \tau)\ ?\ e_2 : e_3$ for some e_1, e_2 and e_3. There are three cases:

$e' \equiv (e_1\{e'_\circ/e_\circ\} \in \tau)\ ?\ e_2\{e'_\circ/e_\circ\} : e_3\{e'_\circ/e_\circ\}$ We can easily conclude by applying the induction hypothesis on the premises.

$e' \equiv e_2$ The second premise, unchanged, allows us to conclude.

$e' \equiv e_3$ This case is impossible. Indeed, it implies that e_1 is a value, and as $\Gamma \vDash_{\mathcal{N}} e_1 : \tau$ (first premise), we can deduce using Proposition 8 that $e_1 \in \tau$, which contradicts $e \rightsquigarrow_\top e_3$.

[$\in_2$-N] Similar to the previous case.

□

Theorem 2 (Subject reduction)**.** *Let Γ be an environment, e an expression, and t a type. Let D be a weakly-canonical derivation of $\Gamma \vDash_{\mathcal{N}} e : t$. Let e' be an expression such that $e \rightsquigarrow_{\mathcal{P}} e'$. Then, there exists a weakly-canonical derivation of $\Gamma \vDash_{\mathcal{N}} e' : t$.*

Proof. Using Lemma 20 on D, we can build a minimal weakly-canonical derivation of $\Gamma \vDash_{\mathcal{N}} e : t$.

Moreover, the root of the reduction step $e \rightsquigarrow_{\mathcal{P}} e'$ is a $[\kappa]$ node, with its premise being of the form $e_\circ \rightsquigarrow_\top e'_\circ$, and with $e' \equiv e\{e'_\circ/e_\circ\}$.

Thus, by using Lemma 21, we obtain $\Gamma \vDash_{\mathcal{N}} e' : t$. □

Corollary 1 (Subject reduction for programs)**.** *Let Γ be an environment, p and p' two programs such that $p \rightsquigarrow_{\mathcal{P},Pr} p'$, and t a type. Let D be a weakly-canonical derivation of $\Gamma \vDash_{\mathcal{N},Pr} p : t$. Then, there exists a weakly-canonical derivation of $\Gamma \vDash_{\mathcal{N},Pr} p' : t$.*

Proof. Straightforward induction on D, using Theorem 2 and Lemma 18. □

4.3.5 Progress

We now prove a progress theorem for the type system $\vDash_{\mathcal{N}}$ and the parallel semantics $\rightsquigarrow_{\mathcal{P}}$.

Usually, the progress property is proved for a closed expression, typeable under an empty environment $\varnothing$. However, as the union-elimination rule introduces a binding variable in the environment, we need to prove a more general property: the typing environment does not need to be empty as long as no variable appear under a reduction context in our expression.

Lemma 22. *Let Γ be an environment, e an expression, and t a type. Let D be a minimal derivation of $\Gamma \vDash_{\mathcal{N}} e : t$. If there is no evaluation context E and variable $\boldsymbol{x}$ such that $e \equiv E[\boldsymbol{x}]$, then either e is a value or $\exists e'.\ e \rightsquigarrow_{\mathcal{P}} e'$.*

Proof. We proceed by structural induction on D.
We consider the root of D:

[Const-N] Trivial (e is a value).

[Var$_\lambda$-N] Impossible case (contradict the hypotheses).

[Var$_\vee$-N] Impossible case (contradict the hypotheses).

[$\leqslant$-N] Trivial (by induction on the premise).

[$\to$I-N] Trivial (e is a value).

[$\times$I-N] We have $e \equiv (e_1, e_2)$ for some expressions e_1 and e_2.

- If e_1 is not a value, and as we have $\forall E, \boldsymbol{x}.\ e_1 \not\equiv E[\boldsymbol{x}]$, we know by applying the induction hypothesis that e_1 can be reduced. Thus, e can also be reduced under the evaluation context $([\,], e_2)$.
- If e_1 is a value, then we can apply the induction hypothesis on the second premise (as e_1 is a value, we know that $\forall E, \boldsymbol{x}.\ e_2 \not\equiv E[\boldsymbol{x}]$). It gives that either e_2 is a value or it can be reduced. We can easily conclude in both cases: if e_2 is a value, then e is also a value, otherwise, e can be reduced under the evaluation context $(e_1, [\,])$.

[$\to$E-N] We have $e \equiv e_1 e_2$ for some expressions e_1 and e_2, with $\Gamma \vDash_{\mathcal{N}} e_1 : s \to t$ and $\Gamma \vDash_{\mathcal{N}} e_2 : s$.

- If e_1 is not a value, and as we have $\forall E, \boldsymbol{x}.\ e_1 \not\equiv E[\boldsymbol{x}]$, we know by applying the induction hypothesis that e_1 can be reduced. Thus, e can also be reduced under the evaluation context $[\,]e_2$.
- If e_1 is a value, we can apply Proposition 8 on it: as $\Gamma \vDash_{\mathcal{N}} e_1 : \mathbb{0} \to \mathbb{1}$, it implies that $e_1 \in \mathbb{0} \to \mathbb{1}$ and thus $e_1 \equiv \lambda x.\ e_\lambda$ for some e_λ. Moreover, we can apply the induction hypothesis on the second premise (as e_1 is a value, we know that $\forall E, \boldsymbol{x}.\ e_2 \not\equiv E[\boldsymbol{x}]$). It gives that either e_2 is a value or it can be reduced. We can easily conclude in both cases: if e_2 is a value, then e can be reduced using the rule 4.1, otherwise, e can be reduced under the evaluation context $e_1[\,]$.

[$\times$E$_1$-N] We have $e \equiv \pi_1 e_1$ for some e_1, with $\Gamma \vDash_{\mathcal{N}} e_1 : t \times s$. By applying the induction hypothesis on the premise, we know that e_1 is either a value or it can be reduced. If e_1 can be reduced, then e can also be reduced under the evaluation context $\pi_1[\,]$. Otherwise, as $\Gamma \vDash_{\mathcal{N}} e_1 : \mathbb{1} \times \mathbb{1}$, we can apply Proposition 8 on it, yielding $e_1 \in \mathbb{1} \times \mathbb{1}$. Thus, $e_1 \equiv (v_1, v_2)$ for some values v_1 and v_2, and consequently e can be reduced using the rule 4.2.

[$\times$E$_2$-N] Similar to the previous case.

[$\vee$-N] We have $e \equiv e_1\{e_2/\mathsf{x}\}$ for some e_1 and e_2, with $\Gamma \vDash_{\mathcal{N}} e_2 : s$ and $\forall i \in I.\ \Gamma, \mathsf{x} : s \wedge \mathbf{u}_i \vDash_{\mathcal{N}} e_1 : t$. There are two cases:

- There exists an evaluation context E such that $e_1 \equiv E[\mathsf{x}]$. In this case, we know that $\forall E', \boldsymbol{y}.\ e_2 \not\equiv E'[\boldsymbol{y}]$, otherwise we would have $e \equiv E[E'[\boldsymbol{y}]]$. Thus, we can apply the induction hypothesis on the definition premise. It gives that either e_2 is a value or it can be reduced. As D is minimal, e_2 cannot be a value and thus e_2 can be reduced. Consequently, e can also be reduced under the evaluation context E.
- There is no evaluation context E such that $e_1 \equiv E[\mathsf{x}]$. In this case, we apply the induction hypothesis on one of the body premises. It gives that either e_1 is a value or it can be reduced. We can easily conclude in both cases: if e_1 is a value, then e is also a value, otherwise, e can be reduced.

[$\mathbb{0}$-N] We have $e \equiv (e_1 \in \tau)\,?\,e_2 : e_3$ for some e_1, e_2 and e_3, with $\Gamma \vDash_{\mathcal{N}} e_1 : \mathbb{0}$. As values cannot have the type $\mathbb{0}$ (Proposition 8), we know that e_1 is not a value. Thus, by applying the induction hypothesis on the premise, we know that e_1 can be reduced. Consequently, e can be reduced under the evaluation context $([\,] \in \tau)\,?\,e_2 : e_3$.

[$\in_1$-N] We have $e \equiv (e_1 \in \tau)\,?\,e_2 : e_3$ for some e_1, e_2 and e_3, with $\Gamma \vDash_{\mathcal{N}} e_1 : \tau$. We thus have $e_1 \in \tau$ (Proposition 8). By applying the induction hypothesis on the first premise, we know that e_1 is either a value or it can be reduced. If e_1 is a value, then e can be reduced using the rule 4.4. Otherwise, e_1 can be reduced, and thus e can also be reduced under the evaluation context $([\,] \in \tau)\,?\,e_2 : e_3$.

[$\in_2$-N] Similar to the previous case.

□

Theorem 3 (Progress)**.** *Let e be an expression. Let t be a type. Let D be a weakly-canonical derivation of $\varnothing \vDash_{\mathcal{N}} e : t$. Then, either e is a value or $\exists e'.\ e \rightsquigarrow_{\mathcal{P}} e'$.*

Proof. Using Lemma 20 on D, we can build a minimal derivation of $\varnothing \vDash_{\mathcal{N}} e : t$.

Moreover, we can deduce from $\varnothing \vDash_{\mathcal{N}} e : t$ that there is no evaluation context E and variable $\boldsymbol{x}$ such that $e \equiv E[\boldsymbol{x}]$. Thus, we can conclude using Lemma 22. □

Corollary 2 (Progress for programs)**.** *Let p be a program. Let t be a type. Let D be a weakly-canonical derivation of $\varnothing \vDash_{\mathcal{N},Pr} p : t$. Then, either p is a value or $\exists p'.\ p \rightsquigarrow_{\mathcal{P},Pr} p'$.*

Proof. Straightforward induction on D, using Theorem 3. □

Theorem 4 (Type safety for the parallel semantics)**.** *Let p be a program, and t a type. Let D be a weakly-canonical derivation of $\varnothing \vDash_{\mathcal{N},\mathsf{Pr}} p : t$. Then, either $p \rightsquigarrow^*_{\mathcal{P},\mathsf{Pr}} v$ with $\varnothing \vDash_{\mathcal{N},\mathsf{Pr}} v : t$ or $p \rightsquigarrow^\infty_{\mathcal{P},\mathsf{Pr}}$.*

Proof. Direct consequence of Corollary 1 and Corollary 2. □

4.3.6 Type safety for the source semantics

The final step is to deduce a type safety theorem for the source semantics (Figure 3.1) from the type safety theorem for the parallel semantics. The relation between these two semantics is not trivial (they are not equivalent). For instance, the following expression reduces to two distinct values depending on whether it is reduced using $\rightsquigarrow_{\mathcal{P}}$ or $\rightsquigarrow$:

$$((\lambda x.x)(\lambda x.x), \lambda y.(\lambda x.x)(\lambda x.x)) \rightsquigarrow^*_{\mathcal{P}} (\lambda x.x, \lambda y.(\lambda x.x))$$
$$((\lambda x.x)(\lambda x.x), \lambda y.(\lambda x.x)(\lambda x.x)) \rightsquigarrow^* (\lambda x.x, \lambda y.(\lambda x.x)(\lambda x.x))$$

In addition, the type safety theorem for the parallel semantics (Theorem 4) uses the type system $\vDash_{\mathcal{N},\mathsf{Pr}}$, but we want to state our final type safety theorem for the type system $\vDash_{\mathsf{Pr}}$. Although the type system $\vDash_{\mathsf{Pr}}$ is included in the type system $\vDash_{\mathcal{N},\mathsf{Pr}}$ (Lemma 16), the opposite is not true (in particular, it may be impossible to derive negative arrow types for λ-abstractions using $\vDash_{\mathsf{Pr}}$, while it is possible using $\vDash_{\mathcal{N},\mathsf{Pr}}$).

A consequence is that, for a program p such that $\varnothing \vDash_{\mathsf{Pr}} p : t$, we do not have the guarantee that after reduction using the source semantics, the value v we get (in the case where the expression does not diverge) satisfies $\varnothing \vDash_{\mathsf{Pr}} v : t$, whereas it is the case for reductions using the parallel semantics and the type system $\vDash_{\mathcal{N},\mathsf{Pr}}$ (cf. Theorem 4). For instance, consider this reduction:

$$((\lambda x.x)(\lambda x.x), \lambda y.(\lambda x.x)(\lambda x.x)) \rightsquigarrow^*_{\mathsf{Pr}} (\lambda x.x, \lambda y.(\lambda x.x)(\lambda x.x))$$

For the program before reduction, we can derive the following type using $\vDash_{\mathsf{Pr}}$:

$$((\mathbb{1} \to \mathbb{0}) \times (\mathbb{1} \to \mathbb{1} \to \mathbb{0})) \vee (\neg(\mathbb{1} \to \mathbb{0}) \times (\mathbb{1} \to \neg(\mathbb{1} \to \mathbb{0})))$$

This type can be derived by decomposing the type of both occurrences of $(\lambda x.x)(\lambda x.x)$ into $\mathbb{1} \to \mathbb{0}$ and $\neg(\mathbb{1} \to \mathbb{0})$ using the $[\vee]$ rule. However, after reduction, the $[\vee]$ rule cannot relate the type of the application in the right component of the pair with the type of the left component. Thus, it becomes impossible to derive the type above.

This example shows that the type safety as stated in Theorem 4 does not hold when using the source semantics and the type system $\vDash_{\mathsf{Pr}}$. However, we can prove a weaker version of it: instead of guaranteeing, after reduction of a non-diverging

expression e of type t, that we obtain a value v of the same type t, it only states that we obtain a value v satisfying $v \in \tau$ for every test type $\tau \geqslant t$. In particular, if t is a function type, it only guarantees that after reduction we get a value $v \in \mathbb{0} \to \mathbb{1}$ (or equivalently, that we get a λ-abstraction).

The idea of the proof of this result is based on the following observation: even though a single step of the parallel semantics may perform more reductions than a single step of the source semantics, these additional reductions will still happen in a future step of the source semantics, at the exception of the reductions happening under a λ-abstraction of the final value (because the source semantics does not perform reduction under λ-abstractions). Though, it is not an issue for the weaker version of the type safety theorem, as it does not give any guarantee about the body of the λ-abstractions contained in the final value.

In order to relate the parallel semantics with the source semantics, we introduce again another reduction semantics $\rightsquigarrow_{\mathcal{C}}$ on expressions. This semantics $\rightsquigarrow_{\mathcal{C}}$ can perform a reduction $\rightsquigarrow_{\top}$ under any context $\mathcal{C}$ (not just an evaluation context):

Definition 48 (Reduction under any context)**.** *A reduction step $\rightsquigarrow_{\mathcal{C}}$ is a top-level reduction step $\rightsquigarrow_{\top}$ (see Figure 4.3) that can happen under any context:*

$$\mathcal{C} ::= [\,] \mid \lambda x.\mathcal{C} \mid \mathcal{C}\ e \mid e\ \mathcal{C} \mid (\mathcal{C}, e) \mid (e, \mathcal{C}) \mid \pi_i \mathcal{C} \mid (\mathcal{C} \in \tau)\,?\,e : e \mid (e \in \tau)\,?\,\mathcal{C} : e \mid (e \in \tau)\,?\,e : \mathcal{C}$$

$$\frac{e \rightsquigarrow_{\top} e'}{\mathcal{C}[e] \rightsquigarrow_{\mathcal{C}} \mathcal{C}[e']}$$

Definition 49 (Subcontext)**.** *A context $\mathcal{C}_1$ is a* subcontext *of $\mathcal{C}_2$, noted $\mathcal{C}_1 \preceq \mathcal{C}_2$, if and only if there exists a context $\mathcal{C}'_1$ such that $\mathcal{C}_2 \equiv \mathcal{C}_1[\mathcal{C}'_1]$.*

Lemma 23. *For every expression e_1 and e_3, if we have a chain $e_1 \rightsquigarrow^*_{\mathcal{C}} e_3$ such that at least one of the reduction steps happens under an evaluation context, then there exists an expression e_2 such that $e_1 \rightsquigarrow e_2$ and $e_2 \rightsquigarrow^*_{\mathcal{C}} e_3$.*

Proof. Let e_1 and e_4 be two expressions such that $e_1 \rightsquigarrow^*_{\mathcal{C}} e_4$, where at least one of the reduction steps happen under an evaluation context.

Let us consider the first reduction step $e_2 \rightsquigarrow_{\mathcal{C}} e_3$ happening under an evaluation context. We have $e_1 \rightsquigarrow^*_{\mathcal{C}} e_2$ (with no reduction step happening under an evaluation context) and $e_3 \rightsquigarrow^*_{\mathcal{C}} e_4$. Moreover, we have $e_2 \equiv E[e'_2]$ and $e_3 \equiv E[e'_3]$ for some evaluation context E and expressions e'_2 and e'_3 such that $e'_2 \rightsquigarrow_{\top} e'_3$.

No reduction step in $e_1 \rightsquigarrow^*_{\mathcal{C}} e_2$ happen under a context $\mathcal{C}$ such that $\mathcal{C} \preceq E$: otherwise, it would also be an evaluation context, contradicting the fact that $e_2 \rightsquigarrow_{\mathcal{C}} e_3$ is the first reduction step happening under an evaluation context. Consequently, we can reverse in E and e'_2 the reduction steps happening in $e_1 \rightsquigarrow^*_{\mathcal{C}} e_2$ (one after the other, in reverse order):

- If a reduction step happens under a context $\mathcal{C}$ such that $E \preceq \mathcal{C}$, it only involves e'_2, we can thus apply the reverse rewriting to e'_2. After that, the expression we get is still reducible at top-level, as the reduction step that has been reversed cannot happen under an evaluation context (no reduction step in $e_1 \rightsquigarrow^*_{\mathcal{C}} e_2$ can happen under an evaluation context).
- Otherwise, if a reduction step happens under a context $\mathcal{C}$ such that $\mathcal{C} \not\preceq E$ and $E \not\preceq \mathcal{C}$, it only involves E, we can thus apply the reverse rewriting to E. After that, the new context we get is still an evaluation context, as the reduction step that has been reversed cannot happen under an evaluation context (no reduction step in $e_1 \rightsquigarrow^*_{\mathcal{C}} e_2$ can happen under an evaluation context).

After this reversing process, we get a new evaluation context E' and expression e'_1 such that $e_1 \equiv E'[e'_1]$ and $e'_1 \rightsquigarrow_\top e''_2$ for some e''_2 such that $E'[e''_2] \rightsquigarrow^*_{\mathcal{C}} E[e'_2]$.

Consequently, we have $e_1 \equiv E'[e'_1] \rightsquigarrow E'[e''_2]$, and $E'[e''_2] \rightsquigarrow^*_{\mathcal{C}} E[e'_2] \equiv e_3 \rightsquigarrow^*_{\mathcal{C}} e_4$, which concludes the proof. □

Lemma 24. *For every expression e_1, e_2 and e_3, if $e_1 \rightsquigarrow^*_{\mathcal{C}} e_2 \rightsquigarrow_{\mathcal{P}} e_3$, then there exists an expression e'_1 such that $e_1 \rightsquigarrow e'_1 \rightsquigarrow^*_{\mathcal{C}} e_3$.*

Proof. The step $e_2 \rightsquigarrow_{\mathcal{P}} e_3$ can be decomposed into several $\rightsquigarrow_{\mathcal{C}}$ steps with at least one happening under an evaluation context. Thus, this lemma is an immediate consequence of Lemma 23 applied on the chain $e_1 \rightsquigarrow^*_{\mathcal{C}} e_2 \rightsquigarrow^*_{\mathcal{C}} e_3$. □

Lemma 25. *For every expression e and value v, if $e \rightsquigarrow^*_{\mathcal{C}} v$, then either $e \rightsquigarrow^\infty$ or there exists a value v' such that $e \rightsquigarrow^* v' \rightsquigarrow^*_{\mathcal{C}} v$.*

Proof. If e is a value, this lemma is trivial. Thus, let us assume that e is not already a value. Then, there must be at least one step in $e \rightsquigarrow^*_{\mathcal{C}} v$ that happens under an evaluation context (it would not be possible for v to be a value otherwise). We can thus apply Lemma 23 successively, starting on $e \rightsquigarrow^*_{\mathcal{C}} v$ and continuing until the remaining $e' \rightsquigarrow^*_{\mathcal{C}} v$ chain is such that e' is a value. If this process terminates, it builds a chain $e \rightsquigarrow^* v'$ with $v' \rightsquigarrow^*_{\mathcal{C}} v$, otherwise it builds $e \rightsquigarrow^\infty$. □

Note that Lemma 25 could be strengthened into $e \rightsquigarrow^*_{\mathcal{C}} v \Rightarrow \exists v'.\ e \rightsquigarrow^* v' \rightsquigarrow^*_{\mathcal{C}} v$ by showing that $e \rightsquigarrow^\infty \Rightarrow e \not\rightsquigarrow^*_{\mathcal{C}} v$, but this is not required by our proof of type safety.

Lemma 26. *For every expression e_1, e_2, and value v, if $e_1 \rightsquigarrow^*_{\mathcal{C}} e_2 \rightsquigarrow^*_{\mathcal{P}} v$ then there exists v' such that $e_1 \rightsquigarrow^* v' \rightsquigarrow^*_{\mathcal{C}} v$ or $e_1 \rightsquigarrow^\infty$.*

Proof. By induction on the number of steps in $e_2 \rightsquigarrow_{\mathcal{P}}^* v$, we prove using Lemma 24 that $e_1 \rightsquigarrow^* e' \rightsquigarrow_{\mathcal{C}}^* v$ for some e'. Then, we conclude by applying Lemma 25 on $e' \rightsquigarrow_{\mathcal{C}}^* v$. □

Lemma 27. *For every expression e_1 and e_2, if $e_1 \rightsquigarrow_{\mathcal{C}}^* e_2 \rightsquigarrow_{\mathcal{P}}^\infty$ then $e_1 \rightsquigarrow^\infty$.*

Proof. We can arbitrarily extend a chain $e_1 \rightsquigarrow \dots$ using Lemma 24. □

The semantics $\rightsquigarrow_{\mathcal{C}}$ is extended into a semantics $\rightsquigarrow_{\mathcal{C},\mathsf{Pr}}$ for programs, with an extended context allowing to perform a reduction under any top-level definition of the program:

$$\mathcal{C}_{\mathsf{Pr}} ::= [] \mid \mathtt{let}\, x = [] \,;\, p \mid \mathtt{let}\, x = e \,;\, \mathcal{C}_{\mathsf{Pr}} \qquad \frac{e \rightsquigarrow_\top e'}{\mathcal{C}_{\mathsf{Pr}}[\mathcal{C}[e]] \rightsquigarrow_{\mathcal{C},\mathsf{Pr}} \mathcal{C}_{\mathsf{Pr}}[\mathcal{C}[e']]}$$

Lemma 28. *For every program p_1, p_2, and value v, if $p_1 \rightsquigarrow_{\mathcal{C},Pr}^* p_2 \rightsquigarrow_{\mathcal{P},Pr}^* v$, then there exists v' such that $p_1 \rightsquigarrow_{Pr}^* v' \rightsquigarrow_{\mathcal{C},Pr}^* v$ or $p_1 \rightsquigarrow_{Pr}^\infty$.*

Proof. Straightforward induction on the number of top-level definitions in p_2, using Lemma 26 (note that p_1 and p_2 must have the same number of top-level definitions as $p_1 \rightsquigarrow_{\mathcal{C},\mathsf{Pr}}^* p_2$). □

Lemma 29. *For every program p_1 and p_2, if $p_1 \rightsquigarrow_{\mathcal{C},Pr}^* p_2 \rightsquigarrow_{\mathcal{P},Pr}^\infty$, then $p_1 \rightsquigarrow_{Pr}^\infty$.*

Proof. Straightforward induction on the number of top-level definitions in p_2, using Lemma 27 (note that p_1 and p_2 must have the same number of top-level definitions as $p_1 \rightsquigarrow_{\mathcal{C},\mathsf{Pr}}^* p_2$). □

Lemma 30. *For every value v_1 and v_2 such that $v_1 \rightsquigarrow_{\mathcal{C}}^* v_2$, if $v_2 \in \tau$, then $v_1 \in \tau$.*

Proof. As v_1 is a value, every reduction step in $v_1 \rightsquigarrow_{\mathcal{C}}^* v_2$ can only happen under a λ-abstraction. Given that the $\in$ relation ignores the body of λ-abstractions (for every x and e, $(\lambda x.e) \in \mathbb{0} \to \mathbb{1}$), none of the reduction steps in $v_1 \rightsquigarrow_{\mathcal{C}}^* v_2$ can change the relation $\cdot \in \tau$. □

Theorem 5 (Type safety for the source semantics)**.** *For every program p and test type τ, if $\varnothing \vDash_{Pr} p : \tau$, then either $p \rightsquigarrow_{Pr}^* v$ for some $v \in \tau$ or $p \rightsquigarrow_{Pr}^\infty$.*

Proof. Straightforward combination of Lemma 16, which allows building a weakly-canonical derivation of $\varnothing \vDash_{\mathcal{N},\mathsf{Pr}} p : \tau$, of the type safety for the parallel semantics (Theorem 4), and of Lemmas 28 and 29.

In the case where we get $p \rightsquigarrow^*_{\mathsf{Pr}} v$ for some value v such that $\exists v'$. ($v \rightsquigarrow^*_{\mathcal{C}} v'$ and $\varnothing \vDash_{\mathcal{N},\mathsf{Pr}} v' : \tau$), we can deduce $\varnothing \vDash_{\mathcal{N}} v' : \tau$, then $v' \in \tau$ using Proposition 8, and finally $v \in \tau$ using Lemma 30. □

Chapter 5

Algorithmic Type System

Contents

The declarative type system given in Chapter 4 is expressive, combining the power of intersection types, union types, and parametric polymorphism. However, it is non-algorithmic. How can we define an algorithm that infers whether a given expression can be typed in this system?

The first step to answer this question is to identify the different sources of non-determinism in the declarative type system. Rules such as union-elimination and intersection-introduction are easy to understand, but they do not easily lend themselves to an implementation. There are two issues that make the type system non-algorithmic.

First, the type system is not syntax directed: several typing rules can be applied for the same syntactic expression. This is due to the non-structural rules that can be applied on any expression: $[\leqslant]$, [Inst], $[\vee]$, and $[\wedge]$. We can solve this issue by constraining the use of non-structural rules, following the structure of canonical typing derivations (Section 4.2). The typing rules $[\leqslant]$ and [Inst] are only necessary when a destructor is applied (application, projection, type-case), and thus they can be embedded in the corresponding structural rules. In order to constrain the use of $[\vee]$ rules, however, we need to give more structure to our expressions: this is done in Section 5.1, which introduces a *canonical form* for expressions.

The second source of non-determinism is due to the fact that some typing rules are non-analytic, meaning that some inputs of their premises (for instance, a type added to the typing environment of a premise) cannot be determined by the inputs of their conclusion. For instance, the type decomposition $\{\mathbf{u}_i\}_{i\in I}$ appearing in the premises of the $[\vee]$ rule does not appear in its conclusion, and thus we have to "guess" it. This second issue is solved in Section 5.2 by adding to our algorithmic

type system an additional input: an *annotation tree* that specifies which value to give to these non-analytic parameters.

In order to arrive to an effective implementation of our type system, we thus proceed in two steps: (*i*) in this chapter, we define an algorithmic type system following the ideas discussed above, and (*ii*) in Chapter 6, we define a reconstruction algorithm that aims to reconstruct the annotation tree used by the algorithmic type system.

5.1 Maximal Sharing Canonical forms

One thing that makes the declarative type system non-algorithmic is the fact that it is not *syntax-directed*. A type system is *syntax-directed* when the syntax of the expression we want to type indicates which rule to apply. This is the case when the type system only has structural rules, but in our case, we also have non-structural rules: $[\wedge]$, [Inst], and $[\leqslant]$.

The two rules $[\leqslant]$ and [Inst] can easily be eliminated, as suggested by the structure of canonical typing derivations (Section 4.2), by embedding them in destructor rules: $[\to\mathrm{E}]$, $[\times\mathrm{E}_1]$, $[\times\mathrm{E}_2]$, $[\mathbb{0}]$, $[\in_1]$, and $[\in_2]$.

The two other rules, $[\vee]$ and $[\wedge]$, cannot be eliminated in this way. However, for typing an expression e, the normalization lemmas of Section 4.2 suggest that $[\vee]$ rules only need to decompose the type of each subexpression of e once. Following this idea, this section aims to make the $[\vee]$ rule syntax-directed by introducing a new syntactic construction called *binding* for our expressions. The idea is to make the $[\vee]$ rule to apply only on these bindings, thus turning it into a structural rule. Expressions that use this new binding syntax are called *canonical forms*.

Two other constraints suggested by the normalization lemmas are that a $[\vee]$ rule decomposing the type of e' should appear in the derivation tree as close of the root as possible, and that it should decompose the type of all the occurrences of e' at once. Our canonical forms are thus further refined to match these constraints, yielding the notion of *Maximal Sharing Canonical (MSC)* form, presented in Section 5.1.2.

5.1.1 Canonical forms

As stated above, we need a syntactic way to determine when to apply the $[\vee]$ rule, and which subexpression to split. In order to achieve this, we represent our terms with a syntax called *Maximal Sharing Canonical (MSC)* form.

Definition 50 (Canonical forms and atoms)**.** Canonical forms *and* atomic expressions *(or atoms) are finite terms produced by the following grammar:*

$$\begin{array}{llll} \textbf{\textit{Atomic expressions}} & a & ::= & c \mid x \mid \lambda x.\kappa \mid (x, x) \mid xx \mid \pi_i x \mid (x{\in}\tau)\,?\,x : x \\ \textbf{\textit{Canonical forms}} & \kappa & ::= & x \mid \mathtt{bind}\, x = a\, \mathtt{in}\, \kappa \end{array}$$

For convenience, we use the metavariable η to range over both atoms and canonical forms.

Canonical forms, ranged over by κ, are *binding variables* (noted x, y, or z) possibly preceded by a list of definitions (from binding variables to atoms). Atoms are either a variable from a λ-abstraction (noted x, y, or z), or a constant, or a λ-abstraction whose body is a canonical form, or any other expression in which *all* proper subexpressions are binding variables. Canonical forms are similar to A-Normal Forms (Sabry and Felleisen, 1992), an intermediate representation of programs where arguments of functions can only be constants or variables, but are even more restrictive: every constructor and destructor (like an application) can only involve binding variables.

Notice the use of distinct sets of variables to represent λ-abstracted variables (x, y, or z) and bind-abstracted ones (x, y, or z). The reason is that we want each subexpression on which we might want to apply the union-elimination rule to be defined in a separate binding. As we could decide to apply the union-elimination rule on occurrences of a single lambda variable, we have to associate a binding to each lambda variable in our expression, but without allowing arbitrary aliasing as it would allow for more complex canonical forms with no benefit. Distinguishing binding variables from lambda variables solves this issue: a binding variable can then be associated to any subexpression, including a single lambda variable, but not to another binding variable, thus preventing arbitrary aliasing.

Definition 51 (Form context)**.** *A* form context $\mathcal{C}_F$ *is a finite term produced by the following grammar:*

$$\textbf{\textit{Form context}} \quad \mathcal{C}_F \quad ::= \quad [\,] \mid \mathsf{bind}\, x = \lambda x.\mathcal{C}_F\, \mathsf{in}\, \kappa \mid \mathsf{bind}\, x = a\, \mathsf{in}\, \mathcal{C}_F$$

Definition 52 (Atom context)**.** *An* atom context $\mathcal{C}_A$ *is a finite term produced by the following grammar:*

$$\textbf{\textit{Atom context}} \quad \mathcal{C}_A \quad ::= \quad \mathsf{bind}\, x = [\,]\, \mathsf{in}\, \kappa \mid \mathsf{bind}\, x = a\, \mathsf{in}\, \mathcal{C}_A \mid \mathsf{bind}\, x = \lambda x.\mathcal{C}_A\, \mathsf{in}\, \kappa$$

Definition 53 (Free variables)**.** *The set of* free variables *of a canonical form* κ

(resp. atom a), noted $\mathsf{fv}(\kappa)$ *(resp.* $\mathsf{fv}(a)$*), is inductively defined as follows:*

$$
\begin{aligned}
\mathsf{fv}(c) &= \varnothing \\
\mathsf{fv}(x) &= \{x\} \\
\mathsf{fv}(\lambda x.\kappa) &= \mathsf{fv}(\kappa)\backslash\{x\} \\
\mathsf{fv}(x_1 x_2) &= \{x_1, x_2\} \\
\mathsf{fv}((x_1, x_2)) &= \{x_1, x_2\} \\
\mathsf{fv}(\pi_i x) &= \{x\} && i = 1, 2 \\
\mathsf{fv}((x_1 \in \tau)\ ?\ x_2 : x_3) &= \{x_1, x_2, x_3\} \\
\mathsf{fv}(x) &= \{x\} \\
\mathsf{fv}(\mathtt{bind}\, x = a\, \mathtt{in}\, \kappa) &= \mathsf{fv}(a) \cup \mathsf{fv}(\kappa)\backslash\{x\}
\end{aligned}
$$

A canonical form can be transformed into an expression of the source language using the unwinding operator $\lceil . \rceil$ that basically inlines bindings:

Definition 54 (Unwinding)**.** *The* unwinding *of a canonical form* κ*, noted* $\lceil \kappa \rceil$*, is the expression inductively defined as follows:*

$$
\begin{aligned}
\lceil c \rceil &= c \\
\lceil x \rceil &= x \\
\lceil \lambda x.\kappa \rceil &= \lambda x.\lceil \kappa \rceil \\
\lceil x_1 x_2 \rceil &= x_1 x_2 \\
\lceil (x_1, x_2) \rceil &= (x_1, x_2) \\
\lceil \pi_i x \rceil &= \pi_i x && i = 1, 2 \\
\lceil (x \in \tau)\ ?\ x_1 : x_2 \rceil &= (x \in \tau)\ ?\ x_1 : x_2 \\
\lceil \mathtt{bind}\, x = a\, \mathtt{in}\, \kappa \rceil &= \lceil \kappa \rceil \{\lceil a \rceil / x\} \\
\lceil x \rceil &= x
\end{aligned}
$$

Note that if $\mathsf{fv}(\kappa) \cap \mathsf{Vars}_{\mathsf{B}} = \varnothing$, then $\lceil \kappa \rceil$ is a ground expression (cf. Definition 20).

The inverse direction, that is, producing from a source language expression a canonical form that unwinds to it, is also straightforward. For that, we introduce binding contexts, similar to the definition contexts from Chapter 4 (Definition 26), but this time with definitions that are atoms:

Definition 55 (Binding context)**.** *A* binding context *B is an ordered list of mappings from binding variables to atoms. Each mapping is written as a pair (x, a). We note these lists extensionally by separating elements by a semicolon, that is, $(x_1, a_1); \ldots; (x_n, a_n)$ and use ε to denote the empty list.*

Definition 56 (Application of binding context to an expression)**.** *The application of a binding context B to an expression e, noted eB, is the expression inductively defined as follows:*

$$e\varepsilon = e$$
$$e(B; (\mathsf{x}, a)) = (e\{\lceil a \rceil / \mathsf{x}\})B$$

Definition 57 (Application of a binding context to an expression order)**.** *Let $\sqsubseteq$ be an expression order. Let B be a binding context. The relation $\sqsubseteq_B$ is the expression order defined by $e_1 \sqsubseteq_B e_2 \Leftrightarrow e_1 B \sqsubseteq e_2 B$.*

We define an operation $\mathsf{term}(B, \kappa)$ which takes a binding context B and a canonical form κ and constructs the canonical form containing the bindings in B and ending with κ:

Definition 58. *Let B be a binding context. Let κ be a canonical form. The canonical form $\mathsf{term}(B, \kappa)$ is defined inductively as follows:*

$$\mathsf{term}(\varepsilon, \kappa) \stackrel{\text{def}}{=} \kappa$$
$$\mathsf{term}(((\mathsf{x}, a); B), \kappa) \stackrel{\text{def}}{=} \mathsf{bind}\, \mathsf{x} = a\, \mathsf{in}\, \mathsf{term}(B, \kappa)$$

We can now define the function $[\![e]\!]$ that transforms an expression e into a pair (B, x) formed by a binding context B and a binding variable x such that $\mathsf{term}(B, \mathsf{x})$ is a canonical form whose unwinding is e.

Definition 59. *Let e be an expression. We define $[\![e]\!]$ as the pair defined inductively as follows, where $\mathsf{x}_\circ$ is a fresh binding variable:*

$$[\![\mathsf{x}]\!] = (\varepsilon, \mathsf{x})$$
$$[\![c]\!] = ((\mathsf{x}_\circ, c), \mathsf{x}_\circ)$$
$$[\![x]\!] = ((\mathsf{x}_\circ, x), \mathsf{x}_\circ)$$
$$[\![\lambda x.e]\!] = ((\mathsf{x}_\circ, \lambda x.\mathsf{term}[\![e]\!]), \mathsf{x}_\circ)$$
$$[\![\pi_i e]\!] = ((B; (\mathsf{x}_\circ, \pi_i \mathsf{x})), \mathsf{x}_\circ) \quad \textit{where } (B, \mathsf{x}) = [\![e]\!]$$
$$[\![e_1 e_2]\!] = ((B_1; B_2; (\mathsf{x}_\circ, \mathsf{x}_1 \mathsf{x}_2)), \mathsf{x}_\circ) \quad \textit{where } (B_1, \mathsf{x}_1) = [\![e_1]\!],\ (B_2, \mathsf{x}_2) = [\![e_2]\!]$$
$$[\![(e_1, e_2)]\!] = ((B_1; B_2; (\mathsf{x}_\circ, (\mathsf{x}_1, \mathsf{x}_2))), \mathsf{x}_\circ) \quad \textit{where } (B_1, \mathsf{x}_1) = [\![e_1]\!],\ (B_2, \mathsf{x}_2) = [\![e_2]\!]$$
$$[\![(e \in \tau)\ ?\ e_1 : e_2]\!] = ((B; B_1; B_2; (\mathsf{x}_\circ, (\mathsf{x} \in \tau)\ ?\ \mathsf{x}_1 : \mathsf{x}_2)), \mathsf{x}_\circ) \quad \textit{where } (B, \mathsf{x}) = [\![e]\!],\ (B_1, \mathsf{x}_1) = [\![e_1]\!],\ (B_2, \mathsf{x}_2) = [\![e_2]\!]$$

Proposition 9. *For every expression of the source language e, $\lceil \mathsf{term}([\![e]\!]) \rceil \equiv e$.*

Proof. Straightforward structural induction on e. □

A canonical form κ ensures, by its syntax, that every subexpression of $\lceil \kappa \rceil$ is associated to (at least) one binding variable. However, different occurrences of the same subexpression could be associated to different binding variables. Consequently, for each expression of the source language, there are several canonical forms that unwind to it. For our algorithmic type system we need to associate each source language expression to a unique canonical form: the Maximal Sharing Canonical (MSC) form.

5.1.2 Maximal Sharing Canonical forms

Maximal Sharing Canonical (MSC) forms are introduced to drastically reduce the range of possible applications of the union-elimination rule, according to the normalization lemmas of Section 4.2. The characteristic of MSC forms is that they encode expressions and preserve typeability in the sense that every expression is typeable if and only if its unique MSC form is typeable.

We define a congruence on canonical forms and atoms:

Definition 60 (Canonical equivalence)**.** *We denote by $\equiv_\kappa$ the smallest congruence on canonical forms and atoms that is closed by α-conversion and such that*

$$\mathsf{bind}\, x_1 = a_1 \,\mathsf{in}\, \mathsf{bind}\, x_2 = a_2 \,\mathsf{in}\, \kappa \quad \equiv_\kappa \quad \mathsf{bind}\, x_2 = a_2 \,\mathsf{in}\, \mathsf{bind}\, x_1 = a_1 \,\mathsf{in}\, \kappa \qquad \text{if } x_1 \notin \mathsf{fv}(a_2), x_2 \notin \mathsf{fv}(a_1)$$

In other words, two canonical forms are $\equiv_\kappa$-equivalent if they are the same, up to α-equivalence and re-ordering of *independent* bindings.

Proposition 10. *If $\kappa \equiv_\kappa \kappa'$, then $\lceil \kappa \rceil \equiv_\alpha \lceil \kappa' \rceil$.*

Proof. If a reordering, as defined in Definition 60, applies at top-level on the expression $\mathsf{bind}\, \mathsf{x}_1 = a_1 \,\mathsf{in}\, \mathsf{bind}\, \mathsf{x}_2 = a_2 \,\mathsf{in}\, \kappa$, the unwinding remains unchanged: as $\mathsf{x}_1 \notin \mathsf{fv}(a_2)$ and $x_2 \notin \mathsf{fv}(a_1)$, we have $\kappa\{a_1/\mathsf{x}_1\}\{a_2/\mathsf{x}_2\} \equiv \kappa\{a_2/\mathsf{x}_2\}\{a_1/\mathsf{x}_1\}$.

The general case follows from the observation that $\forall \mathcal{C}_F, \kappa_1, \kappa_2.\ \lceil \kappa_1 \rceil \equiv_\alpha \lceil \kappa_2 \rceil \Rightarrow \lceil \mathcal{C}_F[\kappa_1] \rceil \equiv_\alpha \lceil \mathcal{C}_F[\kappa_2] \rceil$. □

Now, we can define Maximal Sharing Canonical (MSC) forms:

Definition 61 (MSC forms)**.** *A* Maximal Sharing Canonical (MSC) form *is (any canonical form α-equivalent to) a canonical form κ such that:*

1. *Maximal sharing: if* $\mathtt{bind}\, x_1 = a_1 \,\mathtt{in}\, \kappa_1$ *and* $\mathtt{bind}\, x_2 = a_2 \,\mathtt{in}\, \kappa_2$ *are distinct subexpressions of κ, then $a_1 \not\equiv_\kappa a_2$, and*
2. *Extrusion of bindings: if $\lambda x.\kappa_1$ is a subexpression of κ and* $\mathtt{bind}\, y = a \,\mathtt{in}\, \kappa_2$ *a subexpression of κ_1, then $\mathsf{fv}(a) \not\subseteq \mathsf{fv}(\lambda x.\kappa_1)$, and*
3. *No useless binding: if* $\mathtt{bind}\, x = a \,\mathtt{in}\, \kappa'$ *is a subexpression of κ, then $x \in \mathsf{fv}(\kappa')$.*

The first two properties of Definition 61 are important since they ensure that an expression of the source language is typeable *if and only if* it is the unwinding of a typeable MSC form.

For the first property, it states that distinct variables denote different definitions. It ensures the maximal sharing of common subexpressions in the source language, where *common subexpressions* designate sets composed by different occurrences of subexpressions that are equal (in our case, α-convertible). In other terms, if we start from a source language term and put it into a MSC form, then all common subexpressions occurring in it are bound by the same binding variable. For instance, if we start from the term $(\mathsf{f3}, \mathsf{f3})$, then its MSC form is ($\alpha$-equivalent to) $\mathtt{bind}\, \mathsf{x} = 3 \,\mathtt{in}\, \mathtt{bind}\, \mathsf{y} = \mathsf{fx} \,\mathtt{in}\, \mathtt{bind}\, \mathsf{z} = (\mathsf{y}, \mathsf{y}) \,\mathtt{in}\, \mathsf{z}$: both occurrences of $\mathsf{f3}$ are bound to y.

Intuitively, this first property is necessary because regrouping equivalent subexpressions together increases the typeability of a term as it better captures correlation between these two subexpression: if we can type a term in which two distinct variables bind the same subexpression, then the same term in which this subexpression is bound by a single variable can also be typed by assigning to the unique variable the intersection of the types of the distinct variables, but the converse does not hold. This condition is similar to the condition on canonical form derivations, in Definition 25, imposing $[\vee]$ rules to substitute all occurrences of the subexpression at issue (in other words, two occurrences of the same subexpression must be substituted by the same $[\vee]$ rule).

The second property of Definition 61 requires bindings to be extruded from λ-abstractions whenever possible. This is justified by the fact that outer bindings may produce better types. For instance, consider the expression $\mathtt{bind}\, \mathsf{x} = a \,\mathtt{in}\, \lambda y.\, \mathsf{x}$, where a is an expression that can be either an integer or a Boolean. This expression can be typed with $(\mathbb{1} \to \mathtt{Int}) \vee (\mathbb{1} \to \mathtt{Bool})$. However, for the expression $\lambda y.\, (\mathtt{bind}\, \mathsf{x} = a \,\mathtt{in}\, \mathsf{x})$ which has the same unwinding as the previous one, the most precise type one can deduce is $\mathbb{1} \to (\mathtt{Int} \vee \mathtt{Bool})$, which is strictly larger (i.e., less precise) than $(\mathbb{1} \to \mathtt{Int}) \vee (\mathbb{1} \to \mathtt{Bool})$. In the definition of canonical form derivations (Section 4.2), this condition corresponds to the notion of well-positioned $[\vee]$ node (Definition 31), which basically states that a $[\vee]$ rule must be applied as soon as possible.

The third condition of Definition 61 states that there is no useless binding (the bound variable must occur in the body of the bind). This is important because

it ensures that given a source language expression e there exists a unique (modulo α-conversion and reordering of independent bindings) MSC form whose unwinding is e. In other words, given an expression e of the source language, all its MSC forms (i.e., all MSC forms whose unwinding is e) are equivalent (modulo $\equiv_\kappa$).

Proposition 11 (Equivalence of MSC forms)**.** *Let κ_1, κ_2 be two MSC forms. If $\lceil\kappa_1\rceil \equiv_\alpha \lceil\kappa_2\rceil$, then $\kappa_1 \equiv_\kappa \kappa_2$.*

Proof. We show that κ_2 can be transformed into κ_1 just with α-renaming and reordering of independent bindings, as specified in the definition of $\equiv_\kappa$ (Definition 60).

In this proof, we represent *partially unwound canonical forms* by a pair (B, e), where B is a binding context and e an expression. With this representation, the unwinding of (B, e) is eB, but for clarity we can also use the notation $\lceil(B, e)\rceil$.

Let (B_1, x_1) be the representation of κ_1, with B_1 representing its top-level definitions and x_1 its final binding variable, and (B_2, x_2) be the representation of κ_2. Formally, we have $\mathsf{term}(B_1, \mathsf{x}_1) \equiv \kappa_1$ and $\mathsf{term}(B_2, \mathsf{x}_2) \equiv \kappa_2$.

As $\lceil\kappa_1\rceil \equiv_\alpha \lceil\kappa_2\rceil$, we have $\lceil(B_1, \mathsf{x}_1)\rceil \equiv_\alpha \lceil(B_2, \mathsf{x}_2)\rceil$. We can assume without loss of generality that $\mathsf{x}_1 = \mathsf{x}_2 = \mathsf{x}$ (otherwise we use α-renaming on κ_2), and thus $\lceil(B_1, \mathsf{x})\rceil \equiv \lceil(B_2, \mathsf{x})\rceil$.

Now, we prove the property below, from which Proposition 11 can be deduced. *Let B_1 and B_2 be two binding contexts. Let e be an expression such that:*

- *$\lceil(B_1, e)\rceil \equiv \lceil(B_2, e)\rceil$, and*
- *The body of λ-abstractions in B_1 and B_2 are in MSC form (Definition 61), and*
- *Both B_1 and B_2 satisfy the following properties (corresponding to the MSC form properties applied to the top-level definitions), written here for a binding context B:*
  1. *for every distinct $(\mathsf{x}_1, a_1), (\mathsf{x}_2, a_2) \in B$, we have $a_1 \not\equiv_\kappa a_2$*
  2. *for every $(\mathsf{x}, \lambda z.\kappa) \in B$, every binding $\mathsf{bind}\,\mathsf{y} = a\,\mathsf{in}\,\kappa'$ in κ is such that $\mathsf{fv}(a) \not\subseteq \mathsf{fv}(\lambda z.\kappa)$*
  3. *if $(\mathsf{x}, a) \in B$, then x is a free variable of one of the next definitions in B or of e*

Then, we can transform B_2 into B_1 just with α-renaming of binding variables not in $\mathsf{fv}(e)$, reordering of independent definitions, and replacement of an atom by a $\equiv_\kappa$-equivalent one.

We prove this property by induction on the total number of atoms appearing in B_1 (by counting top-level atoms as well as, for each top-level atom that is a λ-abstraction, the atoms it contains).

The base case ($B_1 = \varepsilon$) is trivial: as $\lceil(B_2, e)\rceil \equiv \lceil(B_1, e)\rceil \equiv \lceil(\varepsilon, e)\rceil \equiv e$, we deduce with Property 3 that $B_2 = \varepsilon$.

For the inductive case ($B_1 \neq \varepsilon$), we consider, among the binding variables defined in B_1 and B_2, a binding variable that unwinds to a maximal expression for the subexpression order. We note x such a binding variable.

We know that no other definition in B_1 or B_2 can use the binding variable x as it would contradict its maximality. We also know that e contains x (Property 3) and thus both B_1 and B_2 contain a definition for x. We move the definition of x at the end in both B_1 and B_2 (we can do so because no other definition in B_1 and B_2 can use x).

We pose $B_1 = B_1' ; (\mathsf{x}, a_1)$ and $B_2 = B_2' ; (\mathsf{x}, a_2)$. As $\lceil(B_1, e)\rceil \equiv \lceil(B_2, e)\rceil$, we can deduce that $\lceil(B_1', \lceil a_1\rceil)\rceil \equiv \lceil(B_2', \lceil a_2\rceil)\rceil$, and thus a_1 and a_2 are atoms of the same kind. We now show that we can obtain $a_1 \equiv a_2$ just by α-renaming binding variables not in $\mathsf{fv}(e)$ and by reordering independent bindings in a_1 and a_2.

- If a_1 and a_2 are atoms that are not λ-abstractions and that do not contain any binding variable (this is the case of constants and lambda variables), we directly have $a_1 \equiv a_2$.
- If a_1 and a_2 are atoms that are not λ-abstractions and that contain only one binding variable (this is the case of projections), we can α-rename binding variables in B_2 so that $a_1 \equiv a_2$.
- If a_1 and a_2 are atoms that are not λ-abstractions and that contain two binding variables x and y (this is the case of applications and pairs), we consider two cases:
  - If $\lceil(B_1', \mathsf{x})\rceil \equiv_\alpha \lceil(B_1', \mathsf{y})\rceil$, we necessarily have $\mathsf{x} = \mathsf{y}$. Indeed, if $\mathsf{x} \neq \mathsf{y}$, then we can find two distinct definitions in B_1' that share the same atom modulo $\equiv_\kappa$, contradicting Property 1. The same reasoning can be done for a_2, and thus we get that both a_1 and a_2 contain the same binding variable twice. Thus, we can α-rename binding variables in B_2 so that $a_1 \equiv a_2$.
  - Otherwise, x and y must be different, and the same applies to a_2. Thus, we can α-rename binding variables in B_2 so that $a_1 \equiv a_2$.
- If a_1 and a_2 are type-cases (containing 3 binding variables), we proceed similarly to the previous case to obtain $a_1 \equiv a_2$.
- In the case where a_1 and a_2 are λ-abstractions, let's say $\lambda x.\ \kappa_1$ and $\lambda x.\ \kappa_2$, we note $(B_{\mathsf{x}_1}, \mathsf{x}_1)$ and $(B_{\mathsf{x}_2}, \mathsf{x}_2)$ the representations of κ_1 and κ_2 respectively. As κ_1 is in MSC-form, we know that $(B_{\mathsf{x}_1}, \mathsf{x}_1)$ satisfies Properties 1, 2 and 3. Moreover, according to Property 2, we know that every atom a in B_{x_1} is such that $\mathsf{fv}(a) \nsubseteq \mathsf{fv}(\lambda x.\kappa_1)$, and thus the atoms in B_{x_1} and those in B_1' are distinct modulo $\equiv_\kappa$. Thus, $((B_1'; B_{\mathsf{x}_1}), \mathsf{x}_1)$ satisfies Properties 1, 2 and

3. Similarly, $((B'_2; B_{\mathsf{x}_2}), \mathsf{x}_2)$ satisfies Properties 1, 2 and 3, and we have $\lceil((B'_1; B_{\mathsf{x}_1}), \mathsf{x}_1)\rceil \equiv \lceil((B'_2; B_{\mathsf{x}_2}), \mathsf{x}_2)\rceil$.

Thus, we can apply the induction hypothesis on $(B'_1; B_{\mathsf{x}_1})$, $(B'_2; B_{\mathsf{x}_2})\{\mathsf{x}_1/\mathsf{x}_2\}$ and x_1. It gives us that $(B'_1; B_{\mathsf{x}_1})$ and $(B'_2; B_{\mathsf{x}_2})$ are equivalent modulo reordering of the definitions, α-renaming of binding variables and $\equiv_\kappa$-transformation of the atoms.

By using again the fact that every atom a in B_{x_1} is such that $\mathsf{fv}(a) \notin \mathsf{fv}(\lambda x.\kappa_1)$, we deduce that they all unwind to an expression containing x, unlike atoms in B'_1. Similarly, every atom a in B_{x_2} unwinds to an expression containing x, unlike atoms in B'_2. Thus, we can deduce that B_{x_1} and B_{x_2} are equivalent modulo reordering of the definitions, α-renaming and $\equiv_\kappa$-transformation of the atoms. Thus, we can α-rename binding variables in B_2 and $\equiv_\kappa$-transform some of its atoms so that $B_{\mathsf{x}_1} \equiv B_{\mathsf{x}_2}$, and thus so that $a_1 \equiv a_2$.

In any case, we get $a_1 \equiv a_2$, thus the last definition of B_1 is the same as the last definition of B_2. The same can be proven for the previous definitions by using the induction hypothesis on B'_1, B'_2 and $e\{a_1/\mathsf{x}\}$. □

Definition 62. *The MSC form of an expression e, noted $\mathsf{MSC}(e)$, is the unique MSC form (up to $\equiv_\kappa$-equivalence) whose unwinding is e.*

It is easy to transform a canonical form into a MSC form that has the same unwinding. This can be done by applying the rewriting rule $\dashrightarrow$ defined in Figure 5.1.

$$\mathsf{bind}\,\mathsf{x}_1 = a_1\,\mathsf{in}\ \mathsf{bind}\,\mathsf{x}_2 = a_2\,\mathsf{in}\,\kappa \dashrightarrow_\circ \mathsf{bind}\,\mathsf{x}_1 = a_1\,\mathsf{in}\,\kappa\{\mathsf{x}_1/\mathsf{x}_2\} \quad \text{if } a_1 \equiv_\kappa a_2 \tag{5.1}$$

$$\mathsf{bind}\,\mathsf{x} = a\,\mathsf{in}\,\kappa \dashrightarrow_\circ \kappa \quad \text{if } \mathsf{x} \notin \mathsf{fv}(\kappa) \tag{5.2}$$

$$\mathsf{bind}\,\mathsf{x} = \lambda y.(\ \mathsf{bind}\,\mathsf{z} = a\,\mathsf{in}\,\kappa_\circ\ )\ \mathsf{in}\,\kappa \dashrightarrow_\circ \mathsf{bind}\,\mathsf{z} = a\,\mathsf{in}\ \mathsf{bind}\,\mathsf{x} = \lambda y.\kappa_\circ\,\mathsf{in}\,\kappa \quad \text{if } y \notin \mathsf{fv}(a), \mathsf{z} \notin \mathsf{fv}(\kappa) \tag{5.3}$$

$$\kappa_1 \dashrightarrow \kappa_2 \quad \text{if } \exists \kappa'_1 \text{ s.t. } \kappa'_1 \equiv_\kappa \kappa_1 \text{ and } \kappa'_1 \dashrightarrow_\circ \kappa_2 \tag{5.4}$$

Figure 5.1: Canonical to MSC rewriting rules

Rule (5.1) implements the maximal sharing: if two variables bind atoms with the same unwinding (modulo α-conversion), then the variables are unified. Rule (5.2) removes useless bindings. Rule (5.3) extrudes bindings from abstractions of variables that do not occur in the argument of the binding. Rule (5.4) applies the previous rules modulo the canonical equivalence: in practice it applies the swap

of binding defined in Definition 60 as many times as it is needed to apply one of the other rules. These rules can be applied under any context. They are confluent (modulo $\equiv_\kappa$) and normalizing, as proved below.

Proposition 12. *If $\kappa \dashrightarrow \kappa'$, then $\lceil\kappa\rceil \equiv_\alpha \lceil\kappa'\rceil$.*

Proof. Similar to Proposition 10. □

Proposition 13 (Normalization)**.** *There is no infinite chain $\kappa_1 \dashrightarrow \kappa_2 \dashrightarrow \cdots$*

Proof. For an atom context $\mathcal{C}_A$, we define $\lambda\text{-}\mathsf{depth}(\mathcal{C}_A)$ inductively as follows:

$$\begin{aligned}\lambda\text{-}\mathsf{depth}(\mathsf{bind}\,\mathsf{x} = [\,]\,\mathsf{in}\,\kappa\,) &= 0\\ \lambda\text{-}\mathsf{depth}(\mathsf{bind}\,\mathsf{x} = a\,\mathsf{in}\,\mathcal{C}_A\,) &= \lambda\text{-}\mathsf{depth}(\mathcal{C}_A)\\ \lambda\text{-}\mathsf{depth}(\mathsf{bind}\,\mathsf{x} = \lambda x.\mathcal{C}_A\,\mathsf{in}\,\kappa\,) &= \lambda\text{-}\mathsf{depth}(\mathcal{C}_A) + 1\end{aligned}$$

Let n be the maximal λ-depth of atom contexts in κ_1:

$$n \stackrel{\text{def}}{=} \max_{\mathcal{C}_A \text{ s.t. } \exists a.\ \mathcal{C}_A[a] \equiv \kappa_1} \lambda\text{-}\mathsf{depth}(\mathcal{C}_A)$$

Let $N_\kappa(i)$ be the number of atom contexts of depth i in the canonical form κ:

$$N_\kappa(i) \stackrel{\text{def}}{=} |\{\mathcal{C}_A \mid \exists a.\mathcal{C}_A[a] \simeq \kappa \text{ and } \lambda\text{-}\mathsf{depth}(\mathcal{C}_A) = i\}|$$

Let $S(\kappa)$ be the following n-tuple:

$$S(\kappa) \stackrel{\text{def}}{=} (N_\kappa(n), N_\kappa(n-1), \ldots, N_\kappa(0))$$

For every chain $\kappa_1 \dashrightarrow \kappa_2 \dashrightarrow \cdots$, the sequence $S(\kappa_1)$, $S(\kappa_2)$, ... is strictly decreasing with respect to the lexicographic order. Thus, $\kappa_1 \dashrightarrow \kappa_2 \dashrightarrow \cdots$ cannot be infinite. □

Proposition 14. *If $\kappa \not\dashrightarrow$ (i.e., no $\dashrightarrow$ rule apply on κ), then κ is a MSC form.*

Proof. We assume $\kappa \not\dashrightarrow$ and show that all 3 MSC properties are satisfied.

The property 3 (no unused bindings) is trivially verified: any binding that does not satisfy this property can directly be eliminated with the rule 5.2. As the rule 5.2 does not apply, this property must be satisfied.

Now, we focus on the property 2 (extrusion of bindings). We assume that there exists a subexpression $\lambda x.\ \kappa_1$ of κ and a subexpression $\mathsf{bind}\,\mathsf{y} = a\,\mathsf{in}\,\kappa_2$ of κ_1 such that $\mathsf{fv}(a) \subseteq \mathsf{fv}(\lambda x.\ \kappa_1)$. We know that a does not depend on x, otherwise x would be in $\mathsf{fv}(a)$ and not in $\mathsf{fv}(\lambda x.\ \kappa_1)$. Thus, we can reorder the binding y

(rule 5.4) in the first position of its innermost containing λ-abstraction, and then apply the rule 5.3 on it, which contradicts $\kappa \not\mapsto$. Consequently, the property 2 is satisfied.

Finally, we show that the property 1 (sharing) is also satisfied. We assume that there are two distinct bindings $\mathtt{bind}\,\mathsf{x}_1 = a_1\,\mathtt{in}\,\ldots$ and $\mathtt{bind}\,\mathsf{x}_2 = a_2\,\mathtt{in}\,\ldots$ such that $a_1 \equiv_\kappa a_2$. As the property 2 is satisfied, and as $\mathsf{fv}(a_1) = \mathsf{fv}(a_2)$, we know that these two bindings are in the same λ-abstraction. Thus, we can reorder them (rule 5.4) to be the one next to the other so that the rule 5.1 is applicable, which contradicts $\kappa \not\mapsto$. Thus, the property 1 is satisfied. □

Proposition 15 (Confluence)**.** *Let κ_1, κ_2, and κ_2' be three canonical forms such that $\kappa_1 \dashrightarrow^* \kappa_2$ and $\kappa_1 \dashrightarrow^* \kappa_2'$. Then, there exists κ_3 and κ_3' such that $\kappa_2 \dashrightarrow^* \kappa_3$, $\kappa_2' \dashrightarrow^* \kappa_3'$, and $\kappa_3 \equiv_\kappa \kappa_3'$.*

Proof. Immediate consequence of Proposition 13 (normalization), Proposition 12 (preservation of $\lceil . \rceil$), Proposition 14 ($\not\mapsto$ implies MSC form), and Proposition 11 (equivalence of MSC forms). □

In summary, the transformation defined by the rules above transforms every canonical form into a MSC form that has the same unwinding. It thus allows computing $\mathsf{MSC}(e)$, the unique (modulo $\equiv_\kappa$) MSC form of e, for every expression e of the source language.

In practice, we do not need to apply these rewriting rules: the MSC form of an expression e can be computed efficiently by walking through e while maintaining a dictionary that maps every atom already defined to the associated binding variable. This has been implemented in the prototype presented in Chapter 9.

Note that, though the transformation of an expression into its MSC form preserves typing, it does not, intuitively, preserve the reduction semantics, since bindings regroup different occurrences of some subexpression that in the original expression might be evaluated at different stages of the reduction, or not evaluated at all. We said "intuitively" because no operational semantics is defined for canonical forms: we use them just for typing.

To summarize, MSC forms tell us when to apply a $[\vee]$ rule and on which subexpression: a term $\mathtt{bind}\,\mathsf{x} = a\,\mathtt{in}\,\kappa$ means (roughly) that the expression $\kappa\{a/\mathsf{x}\}$ must be typed by first applying the union-elimination rule to decompose the type of all occurrences of a. Putting an expression into its MSC form to type it thus corresponds to applying the $[\vee]$ rule on every occurrence of every subexpression of the original expression. This is a step toward a syntax-directed type system.

By associating each subexpression of the original expression e with a binding variable, MSC forms allow storing type information about any subexpression of e in the type environment. Prior to this work, we have explored in Castagna et al. (2022a) an alternative presentation, where type environments associate types to expressions (instead of variables). However, this implies the definition of several

auxiliary operations to manipulate such type environments, and result in a type system both more complex and less expressive.

5.2 Annotations and algorithmic type system

There are still two issues to solve before obtaining an algorithmic type system: (*i*) rules [$\wedge$], [Inst] and [$\leqslant$] are still not syntax-directed, and (*ii*) rules [$\vee$], [Inst], [$\to$I], and [$\leqslant$] are not analytic. Indeed, the [$\vee$] rule requires guessing a type decomposition (i.e., the monomorphic type $\mathbf{u}$ in the premises), the [Inst] rule requires guessing a substitution, the [$\to$I] rule requires guessing the domain $\mathbf{u}$ of the function, and the [$\leqslant$] rule requires guessing the type t' to subsume to.

The issue of [Inst] and [$\leqslant$] not being syntax directed can be solved by embedding them in some structural rules ([$\to$E], [$\times$E$_1$], [$\times$E$_2$], [$\mathbb{0}$], [$\in_1$], and [$\in_2$]), as suggested by the normalization lemmas of Section 4.2. Moreover, the rules in which we embed [$\leqslant$] can be made analytic by using the type operators $\mathsf{dom}()$, $\circ$, $\boldsymbol{\pi}_1$ and $\boldsymbol{\pi}_2$ defined in Section 2.5.

As for rule [$\wedge$], making it syntax-directed is trickier. Indeed, the usual approach of merging rules [$\to$I] and [$\wedge$] does not work here, since terms in MSC forms may hoist a bind definition outside the function where they are used (cf. extrusion property in Definition 61), causing rule [$\wedge$] to be needed on a term that is not, syntactically, a λ-abstraction.

Lastly, there is no easy way to guess the substitutions used by [Inst] rules, or the domain used in [$\to$I] rules, or the decomposition performed by [$\vee$] rules.

To tackle these issues, our algorithmic type system will not only take a canonical form as input, but also an annotation that (*i*) indicates when to apply an intersection, and (*ii*) indicates which type decomposition (for [$\vee$] rules), which type substitutions (for [Inst] rules), and which domain (for [$\to$I] rules) to use. Formally, our algorithmic system uses judgments of the form $\Gamma \vdash_{\mathcal{A}} [\kappa \mid \mathbb{k}] : t$ for a canonical form κ, and $\Gamma \vdash_{\mathcal{A}} [a \mid \mathbb{a}] : t$ for an atom a where $\mathbb{k}$ and $\mathbb{a}$ are respectively *form annotations* and *atom annotations* defined as follows:

Definition 63 (Annotation trees)**.** Atom annotation trees *and* form annotation trees *are finite terms produced by the following grammar:*

$$
\begin{array}{lrcl}
\textbf{Atom annotations} & \mathbb{a} & ::= & \varnothing \mid \lambda(\mathbf{u}, \mathbb{k}) \mid (\rho, \rho) \mid @(\Sigma, \Sigma) \mid \pi(\Sigma) \\
 & & & \mid \mathbb{0}(\Sigma) \mid \in_1(\Sigma) \mid \in_2(\Sigma) \mid \bigwedge(\{\mathbb{a}, \ldots, \mathbb{a}\}) \\
\textbf{Form annotations} & \mathbb{k} & ::= & \rho \mid \mathtt{keep}\ (\mathbb{a}, \{(\mathbf{u}, \mathbb{k}), \ldots, (\mathbf{u}, \mathbb{k})\}) \mid \mathtt{skip}\ \mathbb{k} \\
 & & & \mid \bigwedge(\{\mathbb{k}, \ldots, \mathbb{k}\})
\end{array}
$$

where, we recall, ρ ranges over renamings *of polymorphic type variables (i.e., injective substitutions from $\mathcal{V}_P$ to $\mathcal{V}_P$), and Σ ranges over instantiations (i.e., sets of substitutions from $\mathcal{V}_P$ to $\mathcal{T}$).*

For convenience, we use the metavariable $\mathbb{h}$ to range over both atom annotations and form annotations.

Each node of the annotation tree indicates which typing rule to apply, and in the case of a non-analytic rule, with which parameters. Note that each syntactic constructor in the grammar of annotation trees is dedicated to a specific syntactic expression: an annotation $\lambda(\mathbf{u}, \mathbb{k})$ should be paired with a λ-abstraction, an annotation $\mathbb{0}(\Sigma)$, $\in_1(\Sigma)$, or $\in_2(\Sigma)$ should be paired with a type-case, and an annotation $\mathsf{keep}\ (\mathbb{a}, \dots)$ or $\mathsf{skip}\ \mathbb{k}$ should be paired with a bind expression. The exception is the $\bigwedge(\dots)$ annotation, which indicates that a $[\wedge]$ rule must be applied, and that may be paired with any atom or canonical form.

Annotation trees encode canonical derivations of the declarative type system (cf. Section 4.2) for the MSC form they are paired with. They are a generalization of type annotations inserted in the code. Instead of annotating directly a MSC form with type-annotations, we use a separate annotation tree because of the union-elimination and intersection-introduction rules, which type several times the same expression under different type environments; this would, thus, require different annotations for the same subexpressions: this naturally yields tree-shaped annotations in which each branching corresponds either to the different deductions performed by a union-elimination rule or to the different deductions performed by an intersection-introduction rule.

The full algorithmic type system is given in Figure 5.2. Each rule is explained below. As usual, α-renaming can be applied implicitly on a canonical form in order to make a rule apply. Essentially, there is one typing rule for each annotation, the only exception being the $\varnothing$ annotation that is used both in the rule to type constants and in the two rules for variables.

$$[\textsc{Const-Alg}]\ \frac{}{\Gamma \vdash_{\mathcal{A}} [c \mid \varnothing] : \boldsymbol{b}_c} \qquad [\textsc{Var-Alg}]\ \frac{}{\Gamma \vdash_{\mathcal{A}} [x \mid \varnothing] : \Gamma(x)}$$

$$[{\to}\text{I-Alg}]\ \frac{\Gamma, x : \mathbf{u} \vdash_{\mathcal{A}} [\kappa \mid \mathbb{k}] : t}{\Gamma \vdash_{\mathcal{A}} [\lambda x.\kappa \mid \lambda(\mathbf{u}, \mathbb{k})] : \mathbf{u} \to t}$$

To type the atom $\lambda x.\kappa$, the annotation $\lambda(\mathbf{u}, \mathbb{k})$ provides the domain $\mathbf{u}$ of the function, and the annotation $\mathbb{k}$ for its body.

$$[{\to}\text{E-Alg}]\ \frac{}{\Gamma \vdash_{\mathcal{A}} [\mathsf{x}_1\mathsf{x}_2 \mid @(\Sigma_1, \Sigma_2)] : t_1 \circ t_2}\ \begin{array}{l} t_1 = \Gamma(\mathsf{x}_1)\Sigma_1,\ \ t_2 = \Gamma(\mathsf{x}_2)\Sigma_2 \\ t_1 \leqslant \mathbb{0} \to \mathbb{1},\ \ t_2 \leqslant \mathsf{dom}(t_1) \end{array}$$

To type an application one must apply an instantiation and a subsumption to both the type of the function and the type of the argument. We recall that instantiations (i.e., Σ_1 and Σ_2) are sets of type substitutions; their application to a type t is defined as $t\Sigma \stackrel{\text{def}}{=} \bigwedge_{\sigma\in\Sigma} t\sigma$. Since they cannot be directly guessed, they are given by the annotation. Subsumption instead is embedded in two type operators, defined in Section 2.5. The first operator, $\mathsf{dom}()$, computes the domain of the arrow and

$$[\textsc{Const-Alg}]\ \frac{}{\Gamma \vdash_{\mathcal{A}} [c \mid \varnothing] : \boldsymbol{b}_c} \qquad [\textsc{Var-Alg}]\ \frac{}{\Gamma \vdash_{\mathcal{A}} [x \mid \varnothing] : \Gamma(x)}$$

$$[\to\text{I-Alg}]\ \frac{\Gamma, x : \mathbf{u} \vdash_{\mathcal{A}} [\kappa \mid \Bbbk] : t}{\Gamma \vdash_{\mathcal{A}} [\lambda x.\kappa \mid \lambda(\mathbf{u}, \Bbbk)] : \mathbf{u} \to t}$$

$$[\to\text{E-Alg}]\ \frac{}{\Gamma \vdash_{\mathcal{A}} [\mathsf{x}_1\mathsf{x}_2 \mid @(\Sigma_1, \Sigma_2)] : t_1 \circ t_2}\ \begin{array}{l} t_1 = \Gamma(\mathsf{x}_1)\Sigma_1,\ \ t_2 = \Gamma(\mathsf{x}_2)\Sigma_2 \\ t_1 \leqslant \mathbb{0} \to \mathbb{1},\ \ t_2 \leqslant \mathsf{dom}(t_1) \end{array}$$

$$[\times\text{I-Alg}]\ \frac{}{\Gamma \vdash_{\mathcal{A}} [(\mathsf{x}_1, \mathsf{x}_2) \mid (\rho_1, \rho_2)] : t_1 \times t_2}\ t_1 = \Gamma(\mathsf{x}_1)\rho_1,\ \ t_2 = \Gamma(\mathsf{x}_2)\rho_2$$

$$[\times\text{E}_1\text{-Alg}]\ \frac{}{\Gamma \vdash_{\mathcal{A}} [\pi_1\mathsf{x} \mid \pi(\Sigma)] : \boldsymbol{\pi}_1(t)}\ \begin{array}{l} t = \Gamma(\mathsf{x})\Sigma \\ t \leqslant (\mathbb{1} \times \mathbb{1}) \end{array}$$

$$[\times\text{E}_2\text{-Alg}]\ \frac{}{\Gamma \vdash_{\mathcal{A}} [\pi_2\mathsf{x} \mid \pi(\Sigma)] : \boldsymbol{\pi}_2(t)}\ \begin{array}{l} t = \Gamma(\mathsf{x})\Sigma \\ t \leqslant (\mathbb{1} \times \mathbb{1}) \end{array}$$

$$[\mathbb{0}\text{-Alg}]\ \frac{}{\Gamma \vdash_{\mathcal{A}} [(\mathsf{x} \in \tau)\ ?\ \mathsf{x}_1 : \mathsf{x}_2 \mid \mathbb{0}(\Sigma)] : \mathbb{0}}\ \Gamma(\mathsf{x})\Sigma \simeq \mathbb{0}$$

$$[\in_1\text{-Alg}]\ \frac{}{\Gamma \vdash_{\mathcal{A}} [(\mathsf{x} \in \tau)\ ?\ \mathsf{x}_1 : \mathsf{x}_2 \mid \in_1(\Sigma)] : \Gamma(\mathsf{x}_1)}\ \Gamma(\mathsf{x})\Sigma \leqslant \tau$$

$$[\in_2\text{-Alg}]\ \frac{}{\Gamma \vdash_{\mathcal{A}} [(\mathsf{x} \in \tau)\ ?\ \mathsf{x}_1 : \mathsf{x}_2 \mid \in_2(\Sigma)] : \Gamma(\mathsf{x}_2)}\ \Gamma(\mathsf{x})\Sigma \leqslant \neg\tau$$

$$[\textsc{Bind}_1\text{-Alg}]\ \frac{\Gamma \vdash_{\mathcal{A}} [\kappa \mid \Bbbk] : t}{\Gamma \vdash_{\mathcal{A}} [\mathsf{bind}\,\mathsf{x} = a\,\mathsf{in}\,\kappa \mid \mathsf{skip}\ \Bbbk] : t}\ \mathsf{x} \notin \mathsf{dom}(\Gamma)$$

$$[\textsc{Bind}_2\text{-Alg}]\ \frac{\begin{array}{c} \Gamma \vdash_{\mathcal{A}} [a \mid \mathbb{a}] : s \\ (\forall i \in I) \quad \Gamma, \mathsf{x} : s \wedge \mathbf{u}_i \vdash_{\mathcal{A}} [\kappa \mid \Bbbk_i] : t_i \end{array}}{\Gamma \vdash_{\mathcal{A}} [\mathsf{bind}\,\mathsf{x} = a\,\mathsf{in}\,\kappa \mid \mathsf{keep}\ (\mathbb{a}, \{(\mathbf{u}_i, \Bbbk_i)\}_{i \in I})] : \bigvee_{i \in I} t_i}\ \{\mathbf{u}_i\}_{i \in I} \in \mathsf{Part}(\mathbb{1})$$

$$[\textsc{BindVar-Alg}]\ \frac{}{\Gamma \vdash_{\mathcal{A}} [\mathsf{x} \mid \rho] : \Gamma(\mathsf{x})\rho}$$

$$[\wedge\text{-Alg}]\ \frac{(\forall i \in I) \quad \Gamma \vdash_{\mathcal{A}} [\eta \mid \mathbb{h}_i] : t_i}{\Gamma \vdash_{\mathcal{A}} [\eta \mid \bigwedge(\{\mathbb{h}_i\}_{i \in I})] : \bigwedge_{i \in I} t_i}\ I \neq \varnothing$$

Figure 5.2: Algorithmic Type System

is used to check that the application is well-typed. The second type operator, $\circ$, computes the type of the result of the application.

$$[\times\text{E}_1\text{-Alg}]\ \frac{}{\Gamma \vdash_{\mathcal{A}} [\pi_1 \mathsf{x} \mid \pi(\Sigma)] : \boldsymbol{\pi}_1(t)}\ \begin{array}{l} t = \Gamma(\mathsf{x})\Sigma \\ t \leqslant (\mathbb{1} \times \mathbb{1}) \end{array}$$

$$[\times\text{E}_2\text{-Alg}]\ \frac{}{\Gamma \vdash_{\mathcal{A}} [\pi_2 \mathsf{x} \mid \pi(\Sigma)] : \boldsymbol{\pi}_2(t)}\ \begin{array}{l} t = \Gamma(\mathsf{x})\Sigma \\ t \leqslant (\mathbb{1} \times \mathbb{1}) \end{array}$$

The rules for projections $[\times\text{E}_1\text{-Alg}]$ and $[\times\text{E}_2\text{-Alg}]$ follow the same idea as the rule for application $[\to\text{E-Alg}]$, with the use of the two type operators $\boldsymbol{\pi}_1$ and $\boldsymbol{\pi}_2$ for computing the type of the result of the projection.

$$[\times\text{I-Alg}]\ \frac{}{\Gamma \vdash_{\mathcal{A}} [(\mathsf{x}_1, \mathsf{x}_2) \mid (\rho_1, \rho_2)] : t_1 \times t_2}\ t_1 = \Gamma(\mathsf{x}_1)\rho_1,\ \ t_2 = \Gamma(\mathsf{x}_2)\rho_2$$

To type a pair $(\mathsf{x}_1, \mathsf{x}_2)$ it is not necessary to instantiate $\Gamma(\mathsf{x}_1)$ or $\Gamma(\mathsf{x}_2)$. However, to avoid unwanted correlations, it is necessary to rename the polymorphic type variables of its components. For instance, when typing the pair (x, x) with $\mathsf{x} : \alpha \to \alpha$, we should type it with $(\alpha \to \alpha, \beta \to \beta)$ rather than $(\alpha \to \alpha, \alpha \to \alpha)$, since the former type has strictly more instances than the latter.

$$[\mathbb{0}\text{-Alg}]\ \frac{}{\Gamma \vdash_{\mathcal{A}} [(\mathsf{x} \in \tau)\,?\,\mathsf{x}_1 : \mathsf{x}_2 \mid \mathbb{0}(\Sigma)] : \mathbb{0}}\ \Gamma(\mathsf{x})\Sigma \simeq \mathbb{0}$$

$$[\in_1\text{-Alg}]\ \frac{}{\Gamma \vdash_{\mathcal{A}} [(\mathsf{x} \in \tau)\,?\,\mathsf{x}_1 : \mathsf{x}_2 \mid \in_1(\Sigma)] : \Gamma(\mathsf{x}_1)}\ \Gamma(\mathsf{x})\Sigma \leqslant \tau$$

$$[\in_2\text{-Alg}]\ \frac{}{\Gamma \vdash_{\mathcal{A}} [(\mathsf{x} \in \tau)\,?\,\mathsf{x}_1 : \mathsf{x}_2 \mid \in_2(\Sigma)] : \Gamma(\mathsf{x}_2)}\ \Gamma(\mathsf{x})\Sigma \leqslant \neg\tau$$

To type type-cases, the annotation indicates which of the three rules must be applied and how to instantiate the polymorphic type variables occurring in the type of the tested expression, so that it satisfies the side condition of the applied rule.

$$[\text{Bind}_1\text{-Alg}]\ \frac{\Gamma \vdash_{\mathcal{A}} [\kappa \mid \mathbb{k}] : t}{\Gamma \vdash_{\mathcal{A}} [\mathsf{bind}\,\mathsf{x} = a\,\mathsf{in}\,\kappa \mid \mathsf{skip}\ \mathbb{k}] : t}\ \mathsf{x} \notin \mathsf{dom}(\Gamma)$$

In rule $[\text{Bind}_1\text{-Alg}]$ the annotation indicates to skip the definition of the current binding. This rule is used when the binding variable is not required for typing the body κ under the current context Γ. For instance, this is the case when x only appears in a branch of a type-case that cannot be selected under the hypotheses Γ. The side condition $\mathsf{x} \notin \mathsf{dom}(\Gamma)$ prevents a potential unsound name conflict between binding variables: as occurrences of x in κ denote the x binding variable that is being skipped, having the type of a former binding variable x in our environment when typing κ would be unsound.

Note that this rule does not really have any equivalent in the declarative type system, but is required because of the use of canonical forms, where each subexpression is introduced by a binding before actually being used. Indeed, whereas in the declarative type system a subexpression is simply ignored when it appears in an unreachable branch of a type-case (cf. rules $[\mathbb{0}]$, $[\in_1]$, and $[\in_2]$), in the algorithmic type system this subexpression is introduced preemptively by a binding and this binding is skipped in the contexts where it is not used.

$$[\textsc{Bind}_2\text{-Alg}]\ \frac{\Gamma \vdash_{\mathcal{A}} [a \mid \mathbb{a}] : s \qquad (\forall i \in I) \quad \Gamma, \mathsf{x} : s \wedge \mathbf{u}_i \vdash_{\mathcal{A}} [\kappa \mid \mathbb{k}_i] : t_i}{\Gamma \vdash_{\mathcal{A}} [\mathtt{bind}\,\mathsf{x} = a\,\mathtt{in}\,\kappa \mid \mathtt{keep}\ (\mathbb{a}, \{(\mathbf{u}_i, \mathbb{k}_i)\}_{i\in I})] : \bigvee_{i\in I} t_i}\ \{\mathbf{u}_i\}_{i\in I} \in \mathsf{Part}(\mathbb{1})$$

Conversely, the rule [Bind$_2$-Alg] tries to type the bound atom and then decomposes its type according to the annotation. This decomposition corresponds to an application of the $[\vee]$ rule of the declarative type system.

$$[\textsc{BindVar}\text{-Alg}]\ \frac{}{\Gamma \vdash_{\mathcal{A}} [\mathsf{x} \mid \rho] : \Gamma(\mathsf{x})\rho}$$

The type of the final binding variable of a canonical form is simply read from the environment, and its polymorphic type variables are renamed according to ρ. Once again, this renaming is necessary to avoid undesirable correlations, in particular when regrouping the types $\{t_i\}_{i\in I}$ of different branches into a union $\bigvee_{i\in I} t_i$ in the [Bind$_2$-Alg] rule.

Finally, two annotations indicate when and how to apply rule $[\wedge]$ to atoms and canonical forms:

$$[\wedge\text{-Alg}]\ \frac{(\forall i \in I) \quad \Gamma \vdash_{\mathcal{A}} [a \mid \mathbb{a}_i] : t_i}{\Gamma \vdash_{\mathcal{A}} [a \mid \bigwedge(\{\mathbb{a}_i\}_{i\in I})] : \bigwedge_{i\in I} t_i}\ I \neq \varnothing$$

$$[\wedge\text{-Alg}]\ \frac{(\forall i \in I) \quad \Gamma \vdash_{\mathcal{A}} [\kappa \mid \mathbb{k}_i] : t_i}{\Gamma \vdash_{\mathcal{A}} [\kappa \mid \bigwedge(\{\mathbb{k}_i\}_{i\in I})] : \bigwedge_{i\in I} t_i}\ I \neq \varnothing$$

Let us illustrate the relation between annotation tree and typing derivation with an example. Consider the term $\lambda x.(fx{\in}\mathtt{Int})\,?\,g(fx)\,{:}\,x$, where $f : \alpha \to \alpha$ and $g : \mathtt{Int} \to \mathtt{Int}$. The MSC form corresponding to this expression is written in Figure 5.3. Figure 5.4 and Figure 5.5 give two examples of possible annotations for this MSC form. In Figure 5.4, the function is typed with a single λ annotation of domain $\boldsymbol{\beta}$. The interesting part is the annotation of the binding for u: the corresponding keep annotation represents an application of the union-elimination rule on the occurrences of the expression fx whose type $\boldsymbol{\beta}$ is split into $\boldsymbol{\beta}\wedge\mathtt{Int}$ and $\boldsymbol{\beta}\backslash\mathtt{Int}$. Each subcase is annotated accordingly. Notice in the second subcase that the annotation for v is skip, which indicates that this particular variable must not be used (as $g(fx)$ cannot be typed since in the "else" branch, fx has type $\neg\mathtt{Int}$). This annotation yields for our MSC form the non-overloaded type $\boldsymbol{\beta} \to \boldsymbol{\beta} \vee \mathtt{Int}$.

This example also shows why the condition of maximal sharing for our forms is necessary, not only for their uniqueness, but also for the completeness of the

```
bind f = f in
bind g = g in
bind z = λx.
    bind x = x in
    bind u = f x in
    bind v = g u in
    bind w = (u∈Int)?v:x in
    w
in z
```

Figure 5.3: MSC form of $\lambda x.(fx{\in}\mathtt{Int})\,?\,g(fx)\,{:}\,x$

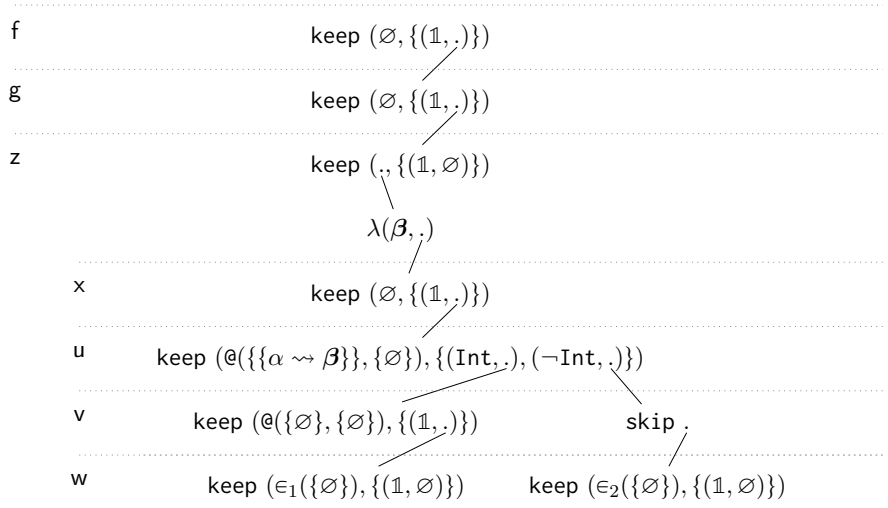


Figure 5.4: Annotation yielding the type $\boldsymbol{\beta} \to \boldsymbol{\beta} \vee \mathtt{Int}$

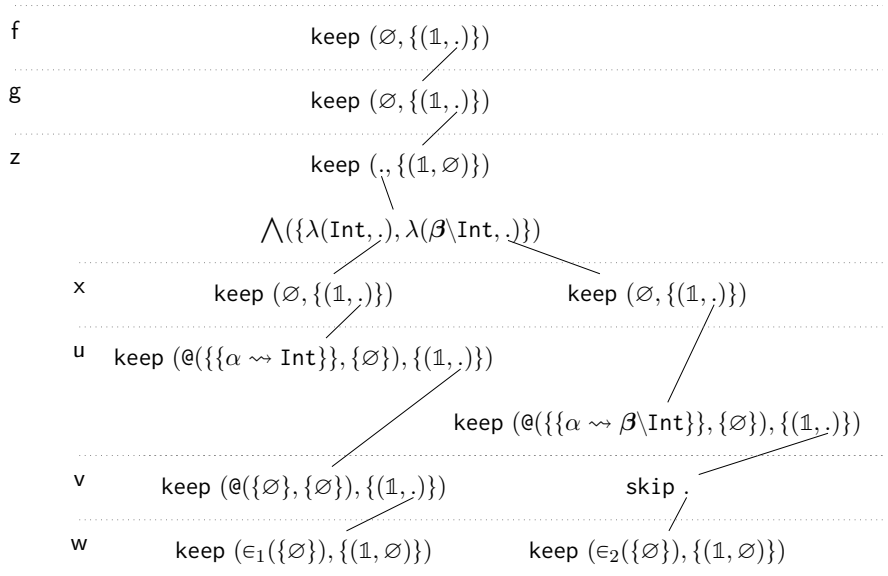


Figure 5.5: Annotation yielding the type $(\mathtt{Int} \to \mathtt{Int}) \wedge (\boldsymbol{\beta}\backslash\mathtt{Int} \to \boldsymbol{\beta}\backslash\mathtt{Int})$

algorithmic system: if the two occurrences of fx in "$\lambda x.(fx{\in}\mathtt{Int})\,?\,g(fx)\,{:}\,x$" were not bound by the same variable, viz., if the sharing were not maximal, then it would not be possible to deduce that $g(fx)$ is well typed using a type decomposition on fx.

A different annotation, yielding a better type, is the one presented in Figure 5.5. The intersection annotation used for the definition of `z` separates the domain of the λ-abstraction into two cases, each typed independently, yielding for the whole function the intersection type $(\mathtt{Int} \to \mathtt{Int}) \wedge (\mathscr{\beta}\backslash\mathtt{Int} \to \mathscr{\beta}\backslash\mathtt{Int})$.

5.3 Equivalence with the declarative type system

An expression e is typeable if and only if its unique (modulo $\equiv_\kappa$) MSC form is typeable, too:

Theorem 6 (Soundness and Completeness)**.** *For every ground expression e of the source language:*

$$\vdash e : t \quad \Leftarrow \quad \vdash_{\mathcal{A}} [\mathsf{MSC}(e) \mid \Bbbk] : t \qquad \textit{(soundness)}$$
$$\vdash e : t \quad \Rightarrow \quad \exists \Bbbk \vdash_{\mathcal{A}} [\mathsf{MSC}(e) \mid \Bbbk] : t' \leqslant t \qquad \textit{(completeness)}$$

These soundness and completeness properties are stated in terms of MSC forms and annotation trees. They essentially state that an expression e has type t in the declarative system if and only if there exists a tree annotation for the (unique) MSC form of e that is typeable in the algorithmic system with (a subtype of) t.

The rest of this chapter is focused on proving this theorem, establishing the equivalence between declarative and algorithmic type system. The soundness is proved in Section 5.3.1 with Theorem 7, and the completeness is proved in Section 5.3.2 with Theorem 8.

5.3.1 Soundness

Lemma 31. *Let e and e' be two expressions, Γ an environment, t a type, and x a binding variable such that $\mathsf{x} \notin \mathsf{dom}(\Gamma)$. Let D be a derivation of $\Gamma \vdash e : t$. Then, we have $\Gamma \vdash e\{e'/\mathsf{x}\} : t$.*

Proof. Straightforward structural induction on D to substitute occurrences of x by e'. The condition $\mathsf{x} \notin \mathsf{dom}(\Gamma)$ ensures that D does not contain any $[\textsc{Var}_\vee]$ node applied on x. □

Theorem 7 (Soundness)**.** *Let Γ be a type environment, η a canonical form or atom, $\Bbbh$ an annotation tree, and t a type. Let D be a derivation of $\Gamma \vdash_{\mathcal{A}} [\eta \mid \Bbbh] : t$. Then, $\Gamma \vdash [\eta] : t$ is derivable.*

Proof. We proceed by structural induction on D in order to build a derivation $\Gamma \vDash \lceil \eta \rceil : t$. We consider the root of D:

[Const-Alg] Trivial (we use a [Const] node).

[Var-Alg] Trivial (we use a [Var_λ] node).

[$\to$I-Alg] We have $\eta \equiv \lambda x.\ \kappa$, and thus $\lceil \eta \rceil \equiv \lambda x.\ \lceil \kappa \rceil$.

By induction on the premise, we get $\Gamma, x : \mathbf{u} \vDash \lceil \kappa \rceil : s$. By applying the rule [$\to$I], we get $\Gamma \vDash \lceil \eta \rceil : \mathbf{u} \to s$ (with $t \simeq \mathbf{u} \to s$).

[$\to$E-Alg] We have $\eta \equiv \mathsf{x}_1\mathsf{x}_2$. We pose $t_1 \stackrel{\text{def}}{=} \Gamma(\mathsf{x}_1)\Sigma_1$ and $t_2 \stackrel{\text{def}}{=} \Gamma(\mathsf{x}_2)\Sigma_2$.

With a [$\text{Var}_\vee$] node, we can derive $\Gamma \vDash \mathsf{x}_1 : \Gamma(\mathsf{x}_1)$ and $\Gamma \vDash \mathsf{x}_2 : \Gamma(\mathsf{x}_2)$. Using a [$\lhd$] pattern, we can derive from that $\Gamma \vDash \mathsf{x}_1 : t_1$ and $\Gamma \vDash \mathsf{x}_2 : t_2$. We have $t \simeq t_1 \circ t_2$. Thus, according to the definition of $\circ$, we know that $t_1 \leqslant t_2 \to t$. Thus, with an application of the [$\leqslant$] rule on $\Gamma \vDash \mathsf{x}_1 : t_1$, we can derive $\Gamma \vDash \mathsf{x}_1 : t_2 \to t$. We can then conclude with an application of the [$\to$E] rule.

[$\times$I-Alg] We have $\eta \equiv (\mathsf{x}_1, \mathsf{x}_2)$.

With a [$\text{Var}_\vee$] node, we can derive $\Gamma \vDash \mathsf{x}_1 : \Gamma(\mathsf{x}_1)\rho_1$ and $\Gamma \vDash \mathsf{x}_2 : \Gamma(\mathsf{x}_2)\rho_2$ (with ρ_1 and ρ_2 as in the [$\times$I-Alg] node). We can then conclude with an application of the [$\times$I] rule.

[$\times\text{E}_1$-Alg] We have $\eta \equiv \pi_1\mathsf{x}$. We pose $t_\circ = \Gamma(\mathsf{x})\Sigma$.

With a [$\text{Var}_\vee$] node, we can derive $\Gamma \vDash \mathsf{x} : \Gamma(\mathsf{x})$. Using a [$\lhd$] pattern, we can derive from that $\Gamma \vDash \mathsf{x} : t_\circ$. We have $t \simeq \boldsymbol{\pi}_1(t_\circ)$. Thus, according to the definition of $\boldsymbol{\pi}_1$, we know that $t_\circ \leqslant t \times \mathbb{1}$. Thus, with an application of the [$\leqslant$] rule, we can derive $\Gamma \vDash \mathsf{x} : t \times \mathbb{1}$. We can then conclude with an application of the [$\times\text{E}_1$] rule.

[$\times\text{E}_2$-Alg] Similar to the previous case.

[$\mathbb{0}$-Alg] Similar to the previous case.

[$\in_1$-Alg] Similar to the previous case.

[$\in_2$-Alg] Similar to the previous case.

[BindVar-Alg] Trivial (we use a [$\text{Var}_\vee$] node).

[Bind_1-Alg] We have $\eta \equiv \mathsf{bind}\,\mathsf{x} = a\,\mathsf{in}\,\kappa$ and thus $\lceil \eta \rceil \equiv \lceil \kappa \rceil\{\lceil a \rceil/\mathsf{x}\}$.

By induction on the premise, we get $\Gamma \vDash \lceil \kappa \rceil : t$. We can then derive $\Gamma \vDash \lceil \kappa \rceil\{\lceil a \rceil/\mathsf{x}\} : t$ by using Lemma 31.

[BIND$_2$-ALG] We have $\eta \equiv \mathsf{bind}\,\mathsf{x} = a\,\mathsf{in}\,\kappa$ and thus $\lceil\eta\rceil \equiv \lceil\kappa\rceil\{\lceil a\rceil/\mathsf{x}\}$.

By induction on the first premise, we get $\Gamma \vDash \lceil a\rceil : s$. For every $i \in I$, we apply the induction hypothesis on the corresponding premise. It gives $\Gamma, \mathsf{x} : s \wedge \mathbf{u}_i \vDash \lceil\kappa\rceil : t_i$. With a $[\leqslant]$ node, we can obtain $\Gamma, \mathsf{x} : s \wedge \mathbf{u}_i \vDash \lceil\kappa\rceil : t$ (with $t \simeq \bigvee_{i\in I} t_i$). We conclude with a $[\vee]$ node.

[$\wedge$-ALG] Straightforward application of the induction hypothesis on the premises.

□

5.3.2 Completeness

For an expression e, our definition of $\mathsf{MSC}(e)$ is unique modulo $\equiv_\kappa$. In particular, two MSC forms that differ by the order of their independent bindings are considered equivalent. We want to prove that the algorithmic type system is complete with respects to the declarative type system, and this regardless of the order of independent bindings in the MSC form used. For that, we define a new $\mathsf{MSC}_\sqsubseteq(e)$ operator that enforces an arbitrary expression order $\sqsubseteq$ on bindings and show that the completeness holds, regardless of $\sqsubseteq$.

Definition 64 (Binding context of a form context)**.** *Let $\mathcal{C}_F$ be a form context. The binding context of $\mathcal{C}_F$, noted $\mathit{bindings}(\mathcal{C}_F)$, is the binding context defined as follows:*

$$\begin{aligned} \mathit{bindings}([\,]) &= \varepsilon \\ \mathit{bindings}(\mathtt{bind}\,x = \lambda x.\mathcal{C}_F\,\mathtt{in}\,\kappa) &= \mathit{bindings}(\mathcal{C}_F) \\ \mathit{bindings}(\mathtt{bind}\,x = a\,\mathtt{in}\,\mathcal{C}_F) &= (\mathsf{x}, a); \mathit{bindings}(\mathcal{C}_F) \end{aligned}$$

Definition 65 (Ordered MSC form)**.** *Let $\sqsubseteq$ be a total expression order (cf. Definition 30). Let κ be a canonical form. We say that κ is a* $\sqsubseteq$-ordered MSC form *if and only if:*

- *κ is a MSC-form, and*
- *For every expression $\kappa' = \mathsf{bind}\,\mathsf{x}_1 = a_1\,\mathsf{in}\,\mathsf{bind}\,\mathsf{x}_2 = a_2\,\mathsf{in}\,\kappa_\circ$ and form context $\mathcal{C}_F$ such that $\mathcal{C}_F[\kappa'] \equiv \kappa$, we have $\lceil a_1\rceil B_1 \sqsubseteq \lceil a_2\rceil B_2$ with $B_1 = \mathit{bindings}(\mathcal{C}_F)$ and $B_2 = \mathit{bindings}(\mathcal{C}_F; (\mathsf{x}_1, a_1))$.*

Proposition 16 (Unicity of ordered MSC forms)**.** *Let $\sqsubseteq$ be a total expression order. Let κ_1 and κ_2 be two $\sqsubseteq$-ordered MSC forms. If $\lceil\kappa_1\rceil \equiv_\alpha \lceil\kappa_2\rceil$, then $\kappa_1 \equiv_\alpha \kappa_2$.*

Proof. Direct consequence of Proposition 11. □

Definition 66. *Let e be an expression of the source language. Let $\sqsubseteq$ be a total expression order. The ordered MSC form of e for the order $\sqsubseteq$, noted $\mathsf{MSC}_{\sqsubseteq}(e)$, is the unique $\sqsubseteq$-ordered MSC form κ modulo α-renaming such that $\lceil\kappa\rceil \equiv_\alpha e$.*

For proving the completeness of the algorithmic type system, we first need some intermediate definitions and lemmas to relate expressions from the source language and MSC forms. In particular, we introduce a notion of atomic source expression that characterizes which expressions of the source language can be represented by a single atom in a MSC form.

Definition 67 (Atomic source expression)**.** *An expression of the source language is an* atomic source expression *if it can be produced by the following grammar:*

$$\textbf{\textit{Atomic source expr.}} \quad \bar{a} \quad ::= \quad c \mid x \mid \lambda x.e \mid (x, x) \mid xx \mid \pi_i x \mid (x{\in}\tau)\,?\,x : x$$

and if, for the case $\lambda x.e$, all subexpressions of e are either a binding variable or they contain a lambda variable that is not in $\mathsf{fv}(\lambda x.e)$.

The metavariable $\bar{a}$ is used to range over atomic source expressions.

Intuitively, the condition on λ-abstractions ensures that, after transforming the body e of a λ-abstraction $\lambda x.e$ into a MSC form, every binding depends on x or depends on another lambda variable introduced in e. This is necessary to ensure that no binding can be extruded outside the definition of $\lambda x.e$.

Now, we can define for every atomic source expression $\bar{a}$ and total expression order $\sqsubseteq$ its unique representation (modulo α-renaming) as an atom, noted $\mathsf{MSCA}_{\sqsubseteq}(\bar{a})$.

Definition 68. *For every atomic source expression $\bar{a}$ and total expression order $\sqsubseteq$, the atom $\mathsf{MSCA}_{\sqsubseteq}(\bar{a})$ is defined as follows:*

$$\begin{aligned} \mathsf{MSCA}_{\sqsubseteq}(\lambda x.e) &= \lambda x.\mathsf{MSC}_{\sqsubseteq}(e) \\ \mathsf{MSCA}_{\sqsubseteq}(\bar{a}) &= \bar{a} \qquad\qquad \textit{if } \bar{a} \textit{ is not a } \lambda\textit{-abstraction} \end{aligned}$$

Proposition 17. *For every atomic source expression $\bar{a}$ and total expression order $\sqsubseteq$, we have $\lceil\mathsf{MSCA}_{\sqsubseteq}(\bar{a})\rceil \equiv_\alpha \bar{a}$.*

Proof. Directly follows from the definition of $\mathsf{MSCA}_{\sqsubseteq}(\bar{a})$. □

Proposition 18. *For every atomic source expression $\bar{a}$ and total expression order $\sqsubseteq$, the canonical form $\mathsf{bind}\, x = \mathsf{MSCA}_{\sqsubseteq}(\bar{a})\, \mathsf{in}\, x$ is a $\sqsubseteq$-ordered MSC form.*

Proof. The extrusion property is ensured by the condition on λ-abstractions in Definition 67. The two other properties are trivially satisfied. □

This notion of atomic source expression allows us to decompose the MSC form of any expression e, as stated by the following lemma.

Lemma 32 (Decomposition of an ordered MSC form)**.** *Let $\sqsubseteq$ be a total expression order. Let e be an expression. Then, $\mathsf{MSC}_{\sqsubseteq}(e)$ is either a binding variable (if e is a binding variable), or a binding $\mathsf{bind}\, x = a\, \mathsf{in}\, \kappa$ where:*

- *$\bar{a}$ is the smallest atomic source expression in e for the $\sqsubseteq$ order,*
- *$a = \mathsf{MSCA}_{\sqsubseteq}(\bar{a})$,*
- *$\kappa = \mathsf{MSC}_{\sqsubseteq_{(\mathsf{x},a)}}(e\{\mathsf{x}/\bar{a}\})$*

Proof. We show that the canonical form $\mathsf{bind}\, \mathsf{x} = a\, \mathsf{in}\, \kappa$ is a $\sqsubseteq$-ordered MSC form, which then allows concluding by uniqueness (Proposition 16).

By Property 18, we get that $\mathsf{bind}\, \mathsf{x} = a\, \mathsf{in}\, \mathsf{x}$ is a $\sqsubseteq$-ordered MSC form. By construction, we also get that κ is a $\sqsubseteq_{(\mathsf{x},a)}$-ordered MSC form. Thus, we only need to check that the definition of x in $\mathsf{bind}\, \mathsf{x} = a\, \mathsf{in}\, \kappa$ is not useless, that it satisfies maximal sharing with κ, and that it respects the $\sqsubseteq$ order:

No useless binding As $\bar{a}$ is a subterm of e, we have $\mathsf{x} \in \mathsf{fv}(e\{\mathsf{x}/\bar{a}\})$ and thus $\mathsf{x} \in \mathsf{fv}(\kappa)$.

Maximal Sharing $e\{\mathsf{x}/\bar{a}\}$ does not feature any subterm α-equivalent to $\bar{a}$. Thus, there is no atom a' in κ such that $a' \equiv_{\kappa} a$ (Proposition 10).

Order Consequence of the fact that $\bar{a}$ is chosen minimal for the order $\sqsubseteq$.

□

This decomposition of MSC forms allows us to prove an important lemma that establishes a relation between the use of a $[\vee]$ rule in a canonical form derivation for the expression e and the structure of $\mathsf{MSC}_{\sqsubseteq}(e)$.

Lemma 33 (Relation canonical form derivation $\leftrightarrow$ MSC form)**.** *Let $\sqsubseteq$ be a total expression order. Let D be a canonical form derivation of $\Gamma \vDash e : t$ for the order $\sqsubseteq$, whose root is a $[\vee]$ node that does not perform aliasing. Then, there exists a binding context B, a binding variable x, an atom a and canonical form κ such that $\mathsf{MSC}_{\sqsubseteq}(e) \equiv_{\alpha} \mathsf{term}(B, \mathsf{bind}\, x = a\, \mathsf{in}\, \kappa)$ and such that there exists a canonical form derivation of $\Gamma \vDash \lceil \mathsf{bind}\, x = a\, \mathsf{in}\, \kappa \rceil : t$ for the order $\sqsubseteq_B$ and whose root is a $[\vee]$ node doing the substitution $\lceil \kappa \rceil \{\lceil a \rceil / x\}$.*

Proof. We proceed by structural induction on $\mathsf{MSC}_{\sqsubseteq}(e)$.

The case of $\mathsf{MSC}_{\sqsubseteq}(e)$ being a binding variable is impossible: the $[\vee]$ root of D would perform aliasing. Thus, $\mathsf{MSC}_{\sqsubseteq}(e)$ is not a binding variable.

We apply Lemma 32 on $\mathsf{MSC}_{\sqsubseteq}(e)$, yielding that $\mathsf{MSC}_{\sqsubseteq}(e) \equiv \mathtt{bind\,x = }a\,\mathtt{in}\,\kappa$ for some a and κ such that:

- $\bar{a}$ is the smallest atomic source expression in e for the $\sqsubseteq$ order,
- $a = \mathsf{MSCA}_{\sqsubseteq}(\bar{a})$,
- $\kappa = \mathsf{MSC}_{\sqsubseteq_{(\mathsf{x},a)}}(e\{\mathsf{x}/\bar{a}\})$

There are two cases to consider:

- The $[\vee]$ root of D is performing a substitution $\{\lceil a \rceil/\mathsf{y}\}$ for some y, in which case we can conclude $(B = \varepsilon)$.
- Otherwise, by minimality of $\bar{a}$, D cannot contain any $[\vee]$ node performing a substitution $\{\lceil a \rceil/\mathsf{y}\}$ for some y. Thus, as $\bar{a}$ is an atomic source expression and that structural rules only occur in D as the definition premise of a $[\vee]$ node, we deduce that D has no subderivation for $\bar{a}$. Thus, we can substitute in D all the occurrences of the expression $\bar{a}$ by a binding variable y, yielding a derivation D' of $\Gamma \models e\{\mathsf{y}/\bar{a}\} : t$ that starts with a $[\vee]$ node that does not perform aliasing. We can thus conclude by applying the induction hypothesis on the derivation D' and order $\sqsubseteq_{(\mathsf{y},a)}$ (it is a valid structural induction as $\mathsf{MSC}_{\sqsubseteq_{(\mathsf{y},a)}}(e\{\mathsf{y}/\bar{a}\}) \equiv_\alpha \kappa\{\mathsf{y}/\mathsf{x}\}$).

□

Now, we have all the key elements to prove the completeness of the algorithmic type system. We start by stating a monotonicity property, which will be used in the proof of completeness for handling the case of the union-elimination.

Lemma 34 (Monotonicity)**.** *Let Γ be a type environment, η a canonical form or atom, $\mathbb{h}$ an annotation tree, and t a type. Let D be a derivation of $\Gamma \vdash_{\mathcal{A}} [\eta \mid \mathbb{h}] : t$. Then, for every type environment Γ' such that $\Gamma' \lhd \Gamma$, there exists an annotation tree $\mathbb{h}'$ and a type $t' \lhd t$ such that $\Gamma' \vdash_{\mathcal{A}} [\eta \mid \mathbb{h}'] : t'$ is derivable.*

Proof. Straightforward structural induction on D. □

Lemma 35 (Completeness)**.** *Let Γ be a type environment, e an expression, and t a type. Let $\sqsubseteq$ be a total expression order. Let D be a canonical form derivation of $\Gamma \models e : t$ for the order $\sqsubseteq$ such that D does not contain any $[\vee]$ node performing aliasing. Then, $\exists \mathbb{k}, t'.\ \Gamma \vdash_{\mathcal{A}} [\mathsf{MSC}_{\sqsubseteq}(e) \mid \mathbb{k}] : t'$ with $t' \lhd t$.*

Let Γ be a type environment, $\bar{a}$ an atomic source expression, and t a type. Let $\sqsubseteq$ be a total expression order. Let D be a canonical atomic derivation of $\Gamma \models \bar{a} : t$ for the order $\sqsubseteq$ such that D does not contain any $[\vee]$ node performing aliasing. Then, $\exists \mathbb{a}, t'.\ \Gamma \vdash_{\mathcal{A}} [\mathsf{MSCA}_{\sqsubseteq}(\bar{a}) \mid \mathbb{a}] : t'$ with $t' \lhd t$.

Proof. We proceed by induction on the depth of D.
We consider the root of D:

[$\leqslant$] Impossible case (D would not be canonical).

[Inst] Impossible case (D would not be canonical).

[$\wedge$] In this case, D is an atomic derivation (and the premises of this [$\wedge$] node are [$\to$I] rules). By induction on the premises, we get $\forall i \in I.\ \Gamma \vdash_{\mathcal{A}} [\mathsf{MSCA}_{\sqsubseteq}(\bar{a}) \mid \mathbb{a}_i] : t'_i$ with $t'_i \lhd t_i$. Thus, we can derive $\Gamma \vdash_{\mathcal{A}} [\mathsf{MSCA}_{\sqsubseteq}(\bar{a}) \mid \bigwedge(\{\mathbb{a}_i\}_{i\in I})] : \bigwedge_{i\in I} t'_i$ (with $\bigwedge_{i\in I} t'_i \lhd \bigwedge_{i\in I} t_i$, cf. Proposition 5).

[Const] Trivial.

[Var$_\lambda$] Trivial.

[$\to$I] We have $\bar{a} \equiv \lambda \mathsf{x}.\ e$ and thus $\mathsf{MSCA}_{\sqsubseteq}(\bar{a}) \equiv \lambda \mathsf{x}.\ \mathsf{MSC}_{\sqsubseteq}(e)$.

The premise of this [$\to$I] node is a canonical form derivation. Thus, by induction on this premise, we get $\Gamma, \mathsf{x} : \mathbf{u} \vdash_{\mathcal{A}} [\mathsf{MSC}_{\sqsubseteq}(e) \mid \mathbb{k}] : t'$ (with $t' \lhd t$). We can thus derive $\Gamma \vdash_{\mathcal{A}} [\lambda \mathsf{x}.\ \mathsf{MSC}_{\sqsubseteq}(e) \mid \lambda(\mathbf{u}, \mathbb{k})] : \mathbf{u} \to t'$, and we have $\mathbf{u} \to t' \lhd \mathbf{u} \to t$, which concludes this case.

[$\to$E] We have $\bar{a} \equiv \mathsf{x}_1\mathsf{x}_2$ and thus $\mathsf{MSCA}_{\sqsubseteq}(\bar{a}) \equiv \mathsf{x}_1\mathsf{x}_2$.

As D is a canonical atomic derivation, we know that the second premise, $\Gamma \vDash \mathsf{x}_2 : t_1$, is a [$\lhd$] pattern with no [$\leqslant$] node and whose premise is a [Var$_\vee$] node. Thus, we know that there exists Σ_2 such that $\Gamma(\mathsf{x}_2)\Sigma_2 \simeq t_1$. Similarly, the first premise, $\Gamma \vDash \mathsf{x}_1 : t_1 \to t_2$, is a [$\lhd$] pattern whose premise is a [Var$_\vee$] node. Thus, we know that there exists Σ_1 such that $\Gamma(\mathsf{x}_1)\Sigma_1 \leqslant t_1 \to t_2$.

Consequently, and by definition of $\circ$, we know that $(\Gamma(\mathsf{x}_1)\Sigma_1) \circ (\Gamma(\mathsf{x}_2)\Sigma_2) \leqslant t_2$. We can thus derive $\Gamma \vdash_{\mathcal{A}} [\mathsf{x}_1\mathsf{x}_2 \mid @(\Sigma_1, \Sigma_2)] : t'$ (with $t' \simeq (\Gamma(\mathsf{x}_1)\Sigma_1) \circ (\Gamma(\mathsf{x}_2)\Sigma_2)$) such that $t' \leqslant t_2$, which concludes this case.

[$\times$I] We have $\bar{a} \equiv (\mathsf{x}_1, \mathsf{x}_2)$ and thus $\mathsf{MSCA}_{\sqsubseteq}(\bar{a}) \equiv (\mathsf{x}_1, \mathsf{x}_2)$.

As D is a canonical atomic derivation, both premises can only be [Var$_\vee$] nodes. Thus, we can deduce that there exists two renamings of polymorphic type variables ρ_1 and ρ_2 such that $\Gamma(\mathsf{x}_1)\rho_1 \simeq t_1$ and $\Gamma(\mathsf{x}_2)\rho_2 \simeq t_2$. Thus, we can derive $\Gamma \vdash_{\mathcal{A}} [(\mathsf{x}_1, \mathsf{x}_2) \mid (\rho_1, \rho_2)] : t_1 \times t_2$.

[$\times$E$_1$] We have $\bar{a} \equiv \pi_1\mathsf{x}$ and thus $\mathsf{MSCA}_{\sqsubseteq}(\bar{a}) \equiv \pi_1\mathsf{x}$.

As D is a canonical atomic derivation, we know that the premise, $\Gamma \vDash \mathsf{x} : t_1 \times t_2$, is a [$\lhd$] pattern whose premise is a [Var$_\vee$] node. Thus, we know that there exists Σ such that $\Gamma(\mathsf{x})\Sigma \leqslant t_1 \times t_2$.

Consequently, and by definition of $\boldsymbol{\pi}_1$, we know that $\boldsymbol{\pi}_1(\Gamma(\mathsf{x})\Sigma) \leqslant t_1$. We can thus derive $\Gamma \vdash_{\mathcal{A}} [\pi_1\mathsf{x} \mid \pi(\Sigma)] : t'$ (with $t' \simeq \boldsymbol{\pi}_1(\Gamma(\mathsf{x})\Sigma)$) such that $t' \leqslant t_1$, which concludes this case.

$[\times \mathrm{E}_2]$ Similar to the previous case.

$[\mathbb{0}]$ We have $\bar{a} \equiv (\mathsf{x}{\in}\tau)\,?\,\mathsf{x}_1 : \mathsf{x}_2$ and thus $\mathsf{MSCA}_{\sqsubseteq}(\bar{a}) \equiv (\mathsf{x}{\in}\tau)\,?\,\mathsf{x}_1 : \mathsf{x}_2$.

As D is a canonical atomic derivation, we know that the premise, $\Gamma \models \mathsf{x} : \mathbb{0}$, is a $[\lhd]$ pattern with no $[\leqslant]$ node and whose premise is a $[\textsc{Var}_\vee]$ node. Thus, we know that there exists Σ such that $\Gamma(\mathsf{x})\Sigma \simeq \mathbb{0}$.

We can thus derive $\Gamma \vdash_{\mathcal{A}} [(\mathsf{x}{\in}\tau)\,?\,\mathsf{x}_1 : \mathsf{x}_2 \mid \mathbb{0}(\Sigma)] : \mathbb{0}$.

$[\in_1]$ We have $\bar{a} \equiv (\mathsf{x}{\in}\tau)\,?\,\mathsf{x}_1 : \mathsf{x}_2$ and thus $\mathsf{MSCA}_{\sqsubseteq}(\bar{a}) \equiv (\mathsf{x}{\in}\tau)\,?\,\mathsf{x}_1 : \mathsf{x}_2$.

As D is a canonical atomic derivation, we know that the first premise, $\Gamma \models \mathsf{x} : \tau$, is a $[\lhd]$ pattern whose premise is a $[\textsc{Var}_\vee]$ node. Thus, we know that there exists Σ such that $\Gamma(\mathsf{x})\Sigma \leqslant \tau$. Similarly, the second premise, $\Gamma \models \mathsf{x}_1 : t_1$, can only be a $[\textsc{Var}_\vee]$ rule. Thus, we know that there exists a renaming of polymorphic variables ρ such that $\Gamma(\mathsf{x}_1)\rho \simeq t_1$.

We can thus derive $\Gamma \vdash_{\mathcal{A}} [(\mathsf{x}{\in}\tau)\,?\,\mathsf{x}_1 : \mathsf{x}_2 \mid \in_1(\Sigma)] : \Gamma(\mathsf{x}_1)$ with $\Gamma(\mathsf{x}_1) \lhd t_1$.

$[\in_2]$ Similar to the previous case.

$[\textsc{Var}_\vee]$ Trivial.

$[\vee]$ By using Lemma 33, we know that there exists B such that $\mathsf{MSC}_{\sqsubseteq}(e) \equiv_\alpha \mathsf{term}(B, \mathsf{bind}\,\mathsf{x} = a\,\mathsf{in}\,\kappa)$, and such that there exists a canonical form derivation D' of $\Gamma \models \lceil \mathsf{bind}\,\mathsf{x} = a\,\mathsf{in}\,\kappa \rceil : t$ for the order $\sqsubseteq_B$ and whose root is a $[\vee]$ node doing the substitution $\lceil\kappa\rceil\{\lceil a\rceil/\mathsf{x}\}$.

By induction on the premises of D', we get $\Gamma \vdash_{\mathcal{A}} [a \mid \mathbb{a}] : s'$ (with $s' \lhd s$) and $\forall i \in I.\ \Gamma, \mathsf{x} : s \wedge \mathbf{u}_i \vdash_{\mathcal{A}} [\kappa \mid \mathbb{k}_i] : t_i$ (with $t_i \lhd t$). By monotonicity (Lemma 34), we can derive $\forall i \in I.\ \Gamma, \mathsf{x} : s' \wedge \mathbf{u}_i \vdash_{\mathcal{A}} [\kappa \mid \mathbb{k}'_i] : t'_i$ (with $t'_i \lhd t_i \lhd t$). We can thus derive $\Gamma \vdash_{\mathcal{A}} [\mathsf{bind}\,\mathsf{x} = a\,\mathsf{in}\,\kappa \mid \mathbb{k}] : \bigvee_{i\in I} t'_i$ with $\mathbb{k} = \mathsf{keep}\ (\mathbb{a}, \{(\mathbf{u}_i, \mathbb{k}'_i)\}_{i\in I})$.

From that, we can derive $\Gamma \vdash_{\mathcal{A}} [\mathsf{term}(B, \mathsf{bind}\,\mathsf{x} = a\,\mathsf{in}\,\kappa) \mid \mathbb{k}'] : \bigvee_{i\in I} t'_i$ with $\mathbb{k}'$ obtained by inserting at the root of $\mathbb{k}$ a `skip` annotation for each definition in B, which concludes the proof.

□

Theorem 8 (Completeness)**.** *Let $\sqsubseteq$ be a total expression order. Let Γ be a type environment, e a ground expression, and t a type. Let D be a derivation of $\Gamma \models e : t$. Then, $\exists \mathbb{k}, t'.\ \Gamma \vdash_{\mathcal{A}} [\mathsf{MSC}_{\sqsubseteq}(e) \mid \mathbb{k}] : t'$ with $t' \lhd t$.*

Proof. Direct application of Lemma 35 after using the normalization theorem (Theorem 1) on D for the order $\sqsubseteq$. Note that it is necessary for e to be a ground expression so that the normalized derivation does not contain any $[\vee]$ node performing aliasing. □

Chapter 6

Reconstruction Algorithm

Contents

The second of the two steps to achieve an effective implementation for the type system of Chapter 4 is to define a reconstruction algorithm for the algorithmic system described in Chapter 5. The statements of the soundness and completeness properties of the algorithmic system clearly suggest what this algorithm is expected to do: given an expression that defines a polymorphic function, the algorithm must transform it into its unique MSC form and then try to reconstruct an annotation tree for it, so that the pair of the MSC form and annotation tree is typeable with the algorithmic system.

At this point, however, it should be pretty obvious that such a reconstruction algorithm cannot be complete. Our system merges three well known systems: first-order parametric polymorphism, intersection types, union-elimination. Now, even if parametric polymorphism is decidable, in our system we can encode (and type) polymorphic fixed-point combinators[1], yielding a system with polymorphic recursion whose inference has been long known to be undecidable Henglein (1993); Kfoury et al. (1993). Still, despite being incomplete, our reconstruction algorithm is powerful enough to handle both complicated typing use-cases and common programming patterns of dynamic languages, as we will see in Chapter 9.

Note that performance considerations are not discussed in this chapter: practical aspects of the implementation are discussed in Chapter 8, and some experimental results are presented in Chapter 9.

The reconstruction is performed by a system of deduction rules that incrementally refines an annotation tree (initially composed of a single node "`infer`") while exploring the list of bindings of the MSC form of the expression to type. It mixes

[1] An example of implementation of Curry's fixpoint combinator is given in Chapter 9 (Section 9.1.1.3).

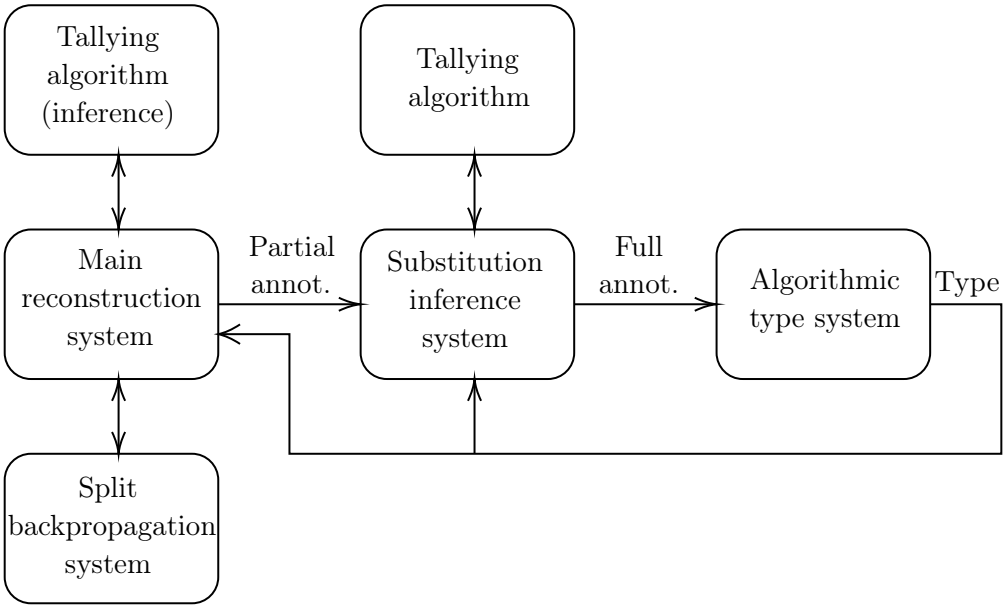


Figure 6.1: Structure of the reconstruction algorithm

two mechanisms: one that infers the domain(s) of λ-terms, and the other that performs occurrence typing (through type decomposition of bindings) when a type-case is encountered.

The first mechanism is inspired by algorithm $\mathcal{W}$ by Damas and Milner (1982): whenever the application of a destructor (e.g., a function application) is encountered, a procedure finds a substitution (if any) that makes this application well-typed. In the context of a Hindley-Milner type system, the algorithm at issue needs to solve a *unification problem* (i.e., whether for two given types s and t there exists a substitution ϕ such as $s\phi = t\phi$) which, if solvable, has a principal solution given by a single substitution (Robinson, 1965). In our system, which is based on subtyping, the algorithm at issue needs to solve a *tallying problem* (cf. Section 2.6) which, if solvable, has a principal solution given by *a finite set* of substitutions (Castagna et al., 2015). When multiple substitutions are found, they are all considered and explored in different branches by adding an intersection branching node in the current annotation tree.

The second mechanism refines the type decompositions applied to binding variables in order to perform occurrence typing. When the system encounters a type-case $(\mathsf{x} \in \tau)\,?\,\mathsf{y} : \mathsf{z}$, then the type of the binding variable x is split into $s \wedge \tau$ and $s \wedge \neg\tau$ (as per the $[\vee]$ rule of the declarative type system, cf. Chapter 4). This decomposition is in turn backward-propagated, splitting the type of the binding variables that appear in the definition of x, and so on.

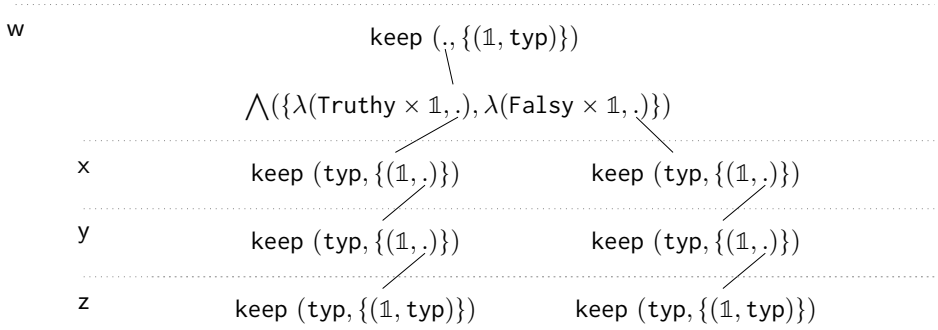
The reconstruction algorithm is structured in two systems of deduction rules: the *main reconstruction algorithm* (Section 6.2) which produces intermediate annotations containing information about the domains of λ-abstractions and the type de-

compositions to use in bindings, and the *substitution inference system* (Section 6.3) which converts these intermediate annotations into annotations for the algorithmic type system by computing instantiations Ψ for the destructors. Both the main reconstruction system and the substitution inference system rely on the tallying algorithm defined in Chapter 2 in order to infer the type of parameters and the polymorphic instantiations. The main reconstruction algorithm also relies on an independent auxiliary system, the *split backpropagation system* (Section 6.4), which is used to propagate type decompositions from a binding variable to the binding variables in its definition. This structure is summarized in Figure 6.1.

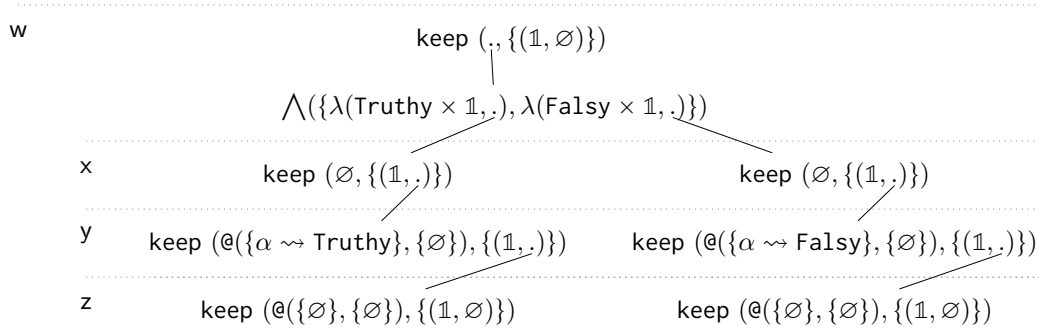
To illustrate the role of the main reconstruction system and the substitution inference system, consider the expression $\lambda x.\ \mathsf{f}\ (\mathsf{g}\ x)$, with $\mathsf{f} : (\mathtt{Truthy} \to \mathtt{True}) \wedge (\mathtt{Falsy} \to \mathtt{False})$ and $\mathsf{g} : \alpha \times \mathbb{1} \to \alpha$, whose MSC form is as follows:

```
bind w =
  λx.
    bind x = x in
    bind y = g x in
    bind z = f y in
    z
in w
```

For this MSC form, the main reconstruction system will produce annotations of the following form (some annotations have been simplified):



Notice how all atoms except λ-abstractions are just annotated with `typ`, meaning that this atom is typeable for some annotation, but this annotation is not provided. In particular, the substitutions required to type the application `g x` are not provided. The reason for not including them is that, during the process of reconstruction, the types of parameters and the type decompositions will be refined many times, each time invalidating the substitutions that have been computed for applications and projections. Thus, it would be inconvenient to store these substitutions in the annotation during the reconstruction, as it would force us to manually invalidate them each time a type is refined. Instead, these substitutions are re-computed whenever needed by the substitution inference system, which replaces `typ` annotations by annotations for the algorithmic type system:



Notice in particular the annotations for the definition of `y`, corresponding to the atom `g x`, that now specify the substitutions necessary to type this polymorphic application. This annotation tree can now be used to type our MSC form with the algorithmic type system, which in this case derives the type $(\texttt{Truthy} \times \mathbb{1} \to \texttt{True}) \wedge (\texttt{Falsy} \times \mathbb{1} \to \texttt{False})$.

6.1 The tallying algorithm

In Chapter 2, we defined the tallying problem, consisting in finding, given a set of constraints C (with each constraint being a pair of types) and a set of type variables Δ, all the substitutions ϕ such that $\phi \# \Delta$ and $\forall (t, s) \in C.\ t\phi \leqslant s\phi$. These conditions are also noted $\phi \Vdash_\Delta C$. This problem is decidable and the set of solutions can be characterized by a finite set Φ of substitutions (where every solution is an instance of a substitution in Φ).

Tallying is a key ingredient of the reconstruction algorithm presented in this chapter. It is mainly used in two contexts: (*i*) to find the result type of applications of polymorphic function types to polymorphic arguments, and (*ii*) to infer the type of the parameters of λ-abstractions.

The first case requires finding, for a function of type t and an argument of type s, all the substitutions σ (over polymorphic type variables) such that $t\sigma \leqslant s\sigma \to \gamma$ (with γ a type variable representing the type of the result of the application). For instance, in order to type `f x` where $\texttt{f} : \alpha \to \alpha$ and $\texttt{x} : \texttt{Int}$, we try to solve the tallying problem $\alpha \to \alpha \leqslant \texttt{Int} \to \gamma$, where γ is a fresh type variable capturing the type of the result of the application. The tallying algorithm will generate the set of constraints $\{\alpha \geqslant \texttt{Int}\ ;\ \alpha \leqslant \gamma\}$ which, when solved, yields the substitution $\{\alpha \rightsquigarrow \texttt{Int} \vee \alpha\ ;\ \gamma \rightsquigarrow \texttt{Int} \vee \alpha \vee \gamma\}$ (which, in our case, can be simplified into $\{\alpha \rightsquigarrow \texttt{Int}\ ;\ \gamma \rightsquigarrow \texttt{Int}\}$). This use is merely to perform the type-checking of the function application `f x`. For this purpose, we introduce a function $\mathsf{tally}(C)$ as follows:

Definition 69 (Tallying)**.** *The function* $\mathsf{tally}(C)$ *maps any set of constraints* C *to a set of substitutions* Σ *such that:*

$$\forall \sigma \in \Sigma.\ \sigma \Vdash_{\mathcal{V}_M} C \qquad \textit{(soundness)}$$
$$\forall \sigma''.\ (\sigma'' \Vdash_{\mathcal{V}_M} C) \Rightarrow (\exists \sigma \in \Sigma.\ \exists \sigma'.\ \sigma'' \simeq \sigma' \circ \sigma) \qquad \textit{(completeness)}$$

The second case deals with type reconstruction. Indeed, as with algorithm $\mathcal{W}$, when typing a λ-abstraction, a fresh monomorphic type variable $\boldsymbol{\alpha}$ is introduced as the type of its argument x. While typing the body, expressions involving x may constraint its type (e.g. $\pi_1 x$ forces the function to accept at most a pair). To deduce such constraints, tallying is used again, but this time we need to find some substitutions ψ over monomorphic type variables (as parameters of λ-abstractions have monomorphic types). For instance, let us consider the expression $\lambda x.f\ x$, with $f : \mathtt{Int} \to \mathtt{Int}$. Initially, x has type $\boldsymbol{\alpha}$. Then, while trying to type the body, the type of x is updated: for the application $f\ x$ to be typeable, we need to substitute $\boldsymbol{\alpha}$ by $\mathtt{Int} \wedge \boldsymbol{\alpha}$. To find this substitution, we introduce an additional tallying function $\mathsf{tally\_infer}(.)$, this time returning substitutions over monomorphic type variables:

Definition 70 (Tallying (inference))**.** *The function* $\mathsf{tally\_infer}(t_1 \dot{\leqslant} t_2)$ *maps any constraint* $t_1 \dot{\leqslant} t_2$ *to a set of substitutions* Ψ *such that:*

$$\forall \psi \in \Psi.\ \exists \sigma_1, \sigma_2.\ \psi \Vdash_{\mathcal{V}_P} \{t_1\sigma_1 \dot{\leqslant} t_2\sigma_2\} \qquad \textit{(soundness)}$$
$$\forall \psi''.\ (\exists \sigma_1, \sigma_2.\ \psi'' \Vdash_{\mathcal{V}_P} \{t_1\sigma_1 \dot{\leqslant} t_2\sigma_2\}) \Rightarrow (\exists \psi \in \Psi.\ \exists \psi'.\ \psi'' \simeq \psi' \circ \psi) \qquad \textit{(completeness)}$$

This function $\mathsf{tally\_infer}(.)$ can be computed from $\mathsf{tally}(.)$, using renamings and restrictions:

$$\mathsf{tally\_infer}(t_1 \dot{\leqslant} t_2) \stackrel{\text{def}}{=} \{(\sigma \circ \sigma' \circ \phi)\big|_{\mathcal{V}_M} \mid \sigma' \in \mathsf{tally}(\{\mathsf{fresh}(t_1)\phi \dot{\leqslant} \mathsf{fresh}(t_2)\phi\})\}$$

where $\mathsf{fresh}(t)$ denotes the type t where polymorphic type variables have been substituted by fresh ones; ϕ is a renaming from $(\mathsf{vars}(t_1) \cup \mathsf{vars}(t_2)) \cap \mathcal{V}_M$ to fresh polymorphic type variables; and σ is a substitution mapping each polymorphic type variable appearing in $\mathsf{vars}(\sigma' \circ \phi)$ to a fresh monomorphic type variable.

In a nutshell, polymorphic type variables in t_1 and t_2 are refreshed in order to decorrelate them, and monomorphic type variables are generalized using ϕ so that $\mathsf{tally}(.)$ is allowed to find solutions involving them. Each solution σ' is composed with ϕ in order to restore the connection with the initial monomorphic type variables, and the polymorphic type variables in the resulting substitution are transformed into monomorphic ones by composing σ with it. Finally, the substitution is restricted to $\mathcal{V}_M$ (its new domain is thus the same as the domain of ϕ).

For instance, when typing $f\ x$ from our earlier example (with $f : \mathtt{Int} \to \mathtt{Int}$ and $x : \boldsymbol{\alpha}$), the tallying instance $\mathsf{tally\_infer}(\mathtt{Int} \to \mathtt{Int} \dot{\leqslant} \boldsymbol{\alpha} \to \beta)$ is generated (with β a fresh polymorphic type variable that captures the resulting type). As expected, solving this instance yields one principal substitution $\{\boldsymbol{\alpha} \rightsquigarrow \mathtt{Int} \wedge \boldsymbol{\alpha}\}$. Note that this substitution does not involve β as β is a polymorphic type variable.

6.2 Main reconstruction algorithm

The *main reconstruction algorithm*, defined in this section, infers the domains of λ-abstractions and the decompositions of types into disjoint unions to use for bindings.

It works by successively refining *intermediate annotations* defined below. These intermediate annotations store information about the domains of λ-abstractions and the decompositions of bindings. However, the instantiations Σ used to type destructors (i.e., applications, projections, and type-cases) in the algorithmic type system are not stored in intermediate annotations, because they might get invalidated as the reconstruction progresses: when new information is found about the domain of a λ-abstraction or the decomposition of a binding, the algorithm re-types some intermediate definitions of the MSC form, thus invalidating the instantiations Σ of later definitions. Consequently, these instantiations Σ are recomputed whenever needed, using the substitution inference system (Section 6.3) that converts intermediate annotations into annotations for the algorithmic type system.

Definition 71 (Intermediate annotation trees)**.** Atom intermediate annotations *and* form intermediate annotations *are finite terms produced by the following grammar:*

Split annotations	$\mathcal{S}$	::=	$\{(\mathbf{u}, \mathcal{K}), \ldots, (\mathbf{u}, \mathcal{K})\}$
Atom intermediate annot.	$\mathcal{A}$	::=	$\mathtt{infer} \mid \mathtt{untyp} \mid \mathtt{typ}$
			$\mid \bigwedge(\{\mathcal{A}, \ldots, \mathcal{A}\}, \{\mathcal{A}, \ldots, \mathcal{A}\})$
			$\mid \in_1 \mid \in_2 \mid \lambda(\mathbf{u}, \mathcal{K})$
Form intermediate annot.	$\mathcal{K}$	::=	$\mathtt{infer} \mid \mathtt{untyp} \mid \mathtt{typ}$
			$\mid \bigwedge(\{\mathcal{K}, \ldots, \mathcal{K}\}, \{\mathcal{K}, \ldots, \mathcal{K}\})$
			$\mid \mathtt{try\text{-}skip}\ (\mathcal{K}) \mid \mathtt{try\text{-}keep}\ (\mathcal{A}, \mathcal{K}, \mathcal{K})$
			$\mid \mathtt{propagate}\ (\mathcal{A}, \mathbb{\Gamma}, \mathcal{S}, \mathcal{S})$
			$\mid \mathtt{skip}\ \mathcal{K} \mid \mathtt{keep}\ (\mathcal{A}, \mathcal{S}, \mathcal{S})$

where $\mathbb{\Gamma}$ ranges over sets of type environments.

For convenience, we use the metavariable $\mathcal{H}$ to range over both atom intermediate annotations and form intermediate annotations. The meaning of each constructor will be detailed later on.

The main reconstruction algorithm is presented as a deduction system, for judgments of the form $\Gamma \vdash^*_{\mathcal{R}} \langle \eta \mid \mathcal{H} \rangle \Rightarrow \mathbb{R}$, where, we recall, η ranges over atoms and canonical forms, and $\mathbb{R}$ is one of the following:

$$\textbf{Result}\quad \mathbb{R} \quad ::= \quad \mathtt{Ok}(\mathcal{H}) \mid \mathtt{Fail} \mid \mathtt{Split}(\Gamma, \mathcal{H}, \mathcal{H}) \mid \mathtt{Subst}(\Psi, \mathcal{H}, \mathcal{H}) \mid \mathtt{Var}\ (\mathtt{x}, \mathcal{H}, \mathcal{H})$$

Let us see what each result for $\Gamma \vdash^*_{\mathcal{R}} \langle \eta \mid \mathcal{H} \rangle$ means. The first two, $\mathtt{Ok}()$ and `Fail`, are terminal, meaning that they are definitive answers that cannot be further refined.

$\mathtt{Ok}(\mathcal{H}')$: the reconstruction successfully computed an intermediate annotation $\mathcal{H}'$. This annotation tree can be turned into an annotation tree $\mathbb{h}$ such that $\Gamma \vdash_{\mathcal{A}} [\eta \mid \mathbb{h}] : t$ for some type t (that is, such that η is typeable by the algorithmic system with the annotation $\mathbb{h}$).

`Fail`: the reconstruction has failed. The algorithm was not able to find an annotation that makes η typeable with the algorithmic type system.

The other three results are intermediary, and specify three arguments for their continuation: an object that generates new hypotheses to make η typeable when analyzed again, an annotation $\mathcal{H}_1$ to use in that case, and a default annotation $\mathcal{H}_2$ to be used when the new hypotheses do not hold.

`Subst`$(\Psi, \mathcal{H}_1, \mathcal{H}_2)$**:** the reconstruction found a set of substitutions Ψ that if applied to Γ may make η typeable. In practice, for each substitution $\psi \in \Psi$, the reconstruction will be called again on the environment $\Gamma\psi$ and annotation $\mathcal{H}_1\psi$. However, this does not necessarily mean that the reconstruction will fail on the current environment Γ: η might still be typeable but with a less precise type (e.g., it could yield a function type with a smaller domain). Thus, this "default" case which does not instantiate Γ is also explored, using the annotation $\mathcal{H}_2$ instead of $\mathcal{H}_1$.

`Split`$(\Gamma', \mathcal{H}_1, \mathcal{H}_2)$**:** the reconstruction found some splits for the variables in $\mathsf{dom}(\Gamma')$ that if applied to Γ may make η typeable. In practice, the system will generate several new environments: one is obtained by (pointwise) intersecting Γ with Γ' and will be used to re-type η with the annotation $\mathcal{H}_1$; the others are obtained by intersecting Γ with $\{(\mathsf{x} : \neg\Gamma'(\mathsf{x}))\}$ for any $\mathsf{x} \in \mathsf{dom}(\Gamma')$, and they will be used to re-type η with the annotation $\mathcal{H}_2$.

`Var` $(\mathsf{x}, \mathcal{H}_1, \mathcal{H}_2)$**:** the reconstruction found that in order to type η, the definition of the bind-abstracted variable x should be typed. Any branch that successfully types it continues with the annotation $\mathcal{H}_1$, otherwise it continues with the annotation $\mathcal{H}_2$.

Initially, any form or atom η is annotated with `infer`, and this annotation is then refined until it yields a terminal result (i.e., either `Ok`() or `Fail`).

There are two different forms of judgments: $\Gamma \vdash^*_{\mathcal{R}} \langle \eta \mid \mathcal{H} \rangle \Rightarrow \mathbb{R}$ and $\Gamma \vdash_{\mathcal{R}} \langle \eta \mid \mathcal{H} \rangle \Rightarrow \mathbb{R}$. We first define rules for the judgment $\vdash_{\mathcal{R}}$ for every canonical form and atom. The results of these judgments are not necessarily terminal and, therefore, it may be necessary to call the reconstruction again in order to refine them. This is the purpose of $\vdash^*_{\mathcal{R}}$ judgments which call repetitively $\vdash_{\mathcal{R}}$ judgments when relevant, so that in the end we get a terminal result. Let us first focus on $\vdash_{\mathcal{R}}$ judgments.

The rules below are presented by decreasing priority (i.e., the first rule that applies is used).

$$[\textsc{Ok}]\ \frac{}{\Gamma \vdash_{\mathcal{R}} \langle \eta \mid \mathtt{typ} \rangle \Rightarrow \mathtt{Ok(typ)}} \qquad [\textsc{Fail}]\ \frac{}{\Gamma \vdash_{\mathcal{R}} \langle \eta \mid \mathtt{untyp} \rangle \Rightarrow \mathtt{Fail}}$$

If a canonical form or atom η is annotated with `typ`, then reconstruction is finished for η, and it is typeable in the current context Γ. The annotation `typ` is never used on λ-abstractions and bindings because the system needs to store more information

for them. Likewise, if a form or atom η is annotated with `untyp`, then reconstruction is finished for η by failing in the current context.

$$[\textsc{Const}]\ \frac{}{\Gamma \vdash_{\mathcal{R}} \langle c \mid \texttt{infer} \rangle \Rightarrow \texttt{Ok(typ)}}$$

$$[\textsc{VarOk}]\ \frac{x \in \mathsf{dom}(\Gamma)}{\Gamma \vdash_{\mathcal{R}} \langle x \mid \texttt{infer} \rangle \Rightarrow \texttt{Ok(typ)}} \qquad [\textsc{VarFail}]\ \frac{}{\Gamma \vdash_{\mathcal{R}} \langle x \mid \texttt{infer} \rangle \Rightarrow \texttt{Fail}}$$

A constant c is typeable regardless of the environment, and thus the algorithm returns `Ok(typ)`. If a λ-abstracted variable x is in the environment, then it is typeable and the algorithm returns `Ok(typ)`. Otherwise, x is undefined and `Fail` is returned (we recall that the rules are presented in decreasing order of priority).

$$[\textsc{PairVar}_i]\ \frac{\mathsf{x}_i \notin \mathsf{dom}(\Gamma)}{\Gamma \vdash_{\mathcal{R}} \langle (\mathsf{x}_1, \mathsf{x}_2) \mid \texttt{infer} \rangle \Rightarrow \texttt{Var}\ (\mathsf{x}_i, \texttt{infer}, \texttt{untyp})}$$

$$[\textsc{PairOk}]\ \frac{\{\mathsf{x}_1, \mathsf{x}_2\} \subseteq \mathsf{dom}(\Gamma)}{\Gamma \vdash_{\mathcal{R}} \langle (\mathsf{x}_1, \mathsf{x}_2) \mid \texttt{infer} \rangle \Rightarrow \texttt{Ok(typ)}}$$

To type the pair $(\mathsf{x}_1, \mathsf{x}_2)$, we must ensure that $\{\mathsf{x}_1, \mathsf{x}_2\} \subseteq \mathsf{dom}(\Gamma)$. If it is not the case, then the two rules [PairVar$_i$] (for $i = 1, 2$) try to remedy it by returning `Var` $(\mathsf{x}_i, \texttt{infer}, \texttt{untyp})$, which is the result that asks the system to try to type the atom bound to x_i for $\mathsf{x}_i \notin \mathsf{dom}(\Gamma)$. If the attempt is successful, then the algorithm will continue with the annotation `infer` and $\mathsf{x}_i \in \mathsf{dom}(\Gamma)$, otherwise it will continue with the annotation `untyp` making the reconstruction fail on this pair. If both x_1 and x_2 are in the environment, then the pair is typeable and the rule [PairOk] returns `Ok(typ)`.

$$[\textsc{ProjVar}]\ \frac{\mathsf{x} \notin \mathsf{dom}(\Gamma)}{\Gamma \vdash_{\mathcal{R}} \langle \pi_i \mathsf{x} \mid \texttt{infer} \rangle \Rightarrow \texttt{Var}\ (\mathsf{x}, \texttt{infer}, \texttt{untyp})}$$

$$[\textsc{ProjInfer}]\ \frac{\Psi = \mathsf{tally\_infer}(\Gamma(\mathsf{x}) \dot{\leqslant} \alpha \times \beta)}{\Gamma \vdash_{\mathcal{R}} \langle \pi_i \mathsf{x} \mid \texttt{infer} \rangle \Rightarrow \texttt{Subst}(\Psi, \texttt{typ}, \texttt{untyp})}\ \alpha, \beta \in \mathcal{V}_P \text{ fresh}$$

When typing a projection $\pi_i \mathsf{x}$, the rule [ProjVar] handles the case where $\mathsf{x} \notin \mathsf{dom}(\Gamma)$, similarly to the rules [PairVar$_i$] for pairs. If $\mathsf{x} \in \mathsf{dom}(\Gamma)$, then the reconstruction continues with the rule [ProjInfer], which tries to find all instances of the current context in which the projection $\pi_i \mathsf{x}$ is typeable, by subsuming $\Gamma(\mathsf{x})$ to $\alpha \times \beta$. For that, it calls the tallying algorithm which returns a set of substitutions Ψ. Then, $\texttt{Subst}(\Psi, \texttt{typ}, \texttt{untyp})$ is returned, meaning that this projection should be typeable under every instance $\Gamma\psi$ of the current context Γ (with $\psi \in \Psi$). The default case (i.e., when the current context is unchanged, for example, when $\Psi = \varnothing$) cannot be typed, so it is annotated with `untyp` (see rule [Iterate$_2$] later on).

$$[\textsc{AppVar}_i]\ \frac{\mathsf{x}_i \notin \mathsf{dom}(\Gamma)}{\Gamma \vdash_{\mathcal{R}} \langle \mathsf{x}_1\mathsf{x}_2 \mid \mathtt{infer}\rangle \Rightarrow \mathtt{Var}\ (\mathsf{x}_i, \mathtt{infer}, \mathtt{untyp})}$$

$$[\textsc{AppInfer}]\ \frac{\Psi = \mathsf{tally\_infer}(\Gamma(\mathsf{x}_1) \dot{\leqslant} \Gamma(\mathsf{x}_2) \to \alpha)}{\Gamma \vdash_{\mathcal{R}} \langle \mathsf{x}_1\mathsf{x}_2 \mid \mathtt{infer}\rangle \Rightarrow \mathtt{Subst}(\Psi, \mathtt{typ}, \mathtt{untyp})}\ \alpha \in \mathcal{V}_P \text{ fresh}$$

Again, the two rules [AppVar$_i$] (for $i = 1, 2$) are similar to the two rules [PairVar$_i$]. Then, if $\{\mathsf{x}_1, \mathsf{x}_2\} \subseteq \mathsf{dom}(\Gamma)$, the rule [AppInfer] tries to find all instances of the current context in which the application $\mathsf{x}_1\mathsf{x}_2$ is typeable, by subsuming $\Gamma(\mathsf{x}_1)$ (the type of the function) to $\Gamma(\mathsf{x}_2) \to \alpha$ (a function type whose domain is the type of the argument). Again, this is done using the tallying algorithm, which computes a set of substitutions Ψ, and the annotation $\mathtt{Subst}(\Psi, \mathtt{typ}, \mathtt{untyp})$ is returned.

$$[\textsc{CaseVar}]\ \frac{\mathsf{x} \notin \mathsf{dom}(\Gamma)}{\Gamma \vdash_{\mathcal{R}} \langle (\mathsf{x} \in \tau)\ ?\, \mathsf{x}_1 : \mathsf{x}_2 \mid \mathtt{infer}\rangle \Rightarrow \mathtt{Var}\ (\mathsf{x}, \mathtt{infer}, \mathtt{untyp})}$$

$$[\textsc{CaseSplit}]\ \frac{\Gamma(\mathsf{x}) \not\leqslant \tau \qquad \Gamma(\mathsf{x}) \not\leqslant \neg\tau}{\Gamma \vdash_{\mathcal{R}} \langle (\mathsf{x} \in \tau)\ ?\, \mathsf{x}_1 : \mathsf{x}_2 \mid \mathtt{infer}\rangle \Rightarrow \mathtt{Split}(\{(\mathsf{x} : \tau)\}, \mathtt{infer}, \mathtt{infer})}$$

The key rule for type-cases is [CaseSplit], corresponding to the case where x is in Γ, but with a type that does not allow the selection of a specific branch. Thus, we need to partition the type of x in two, one part being a subtype of τ and the other a subtype of $\neg\tau$. This is achieved by returning $\mathtt{Split}(\{(\mathsf{x} : \tau)\}, \mathtt{infer}, \mathtt{infer})$: this result is backtracked up to the binding of x, where it will be used to split the associated type, accordingly.

Before showing the other type-case rules, we recall that in the algorithmic type system, there are three typing rules for type-cases:

1. When $\mathsf{x} \leqslant \mathbb{0}$, both branches are skipped (rule [$\mathbb{0}$-Alg]),
2. When $\mathsf{x} \leqslant \tau$, the second branch is skipped (rule [$\in_1$-Alg]),
3. When $\mathsf{x} \leqslant \neg\tau$, the first branch is skipped (rule [$\in_2$-Alg]).

The split performed by the rule [CaseSplit] guarantees that we are now either in the case 2 or in the case 3. Still, before typing the remaining branch of the type-case, we must check whether the rule [$\mathbb{0}$-Alg] could be used instead (case 1), as it would allow skipping this branch too.

$$[\textsc{CaseEmpty}]\ \frac{\Gamma(\mathsf{x}) \simeq \mathbb{0}}{\Gamma \vdash_{\mathcal{R}} \langle (\mathsf{x} \in \tau)\,?\,\mathsf{x}_1 : \mathsf{x}_2 \mid \mathtt{infer} \rangle \Rightarrow \mathtt{Ok}(\mathtt{typ})}$$

$$[\textsc{CaseThen}]\ \frac{\Gamma(\mathsf{x}) \leqslant \tau \qquad \Psi = \mathsf{tally\_infer}(\Gamma(\mathsf{x}) \dot{\leqslant} \mathbb{0})}{\Gamma \vdash_{\mathcal{R}} \langle (\mathsf{x} \in \tau)\,?\,\mathsf{x}_1 : \mathsf{x}_2 \mid \mathtt{infer} \rangle \Rightarrow \mathtt{Subst}(\Psi, \mathtt{typ}, \in_1)}$$

$$[\textsc{CaseElse}]\ \frac{\Gamma(\mathsf{x}) \leqslant \neg\tau \qquad \Psi = \mathsf{tally\_infer}(\Gamma(\mathsf{x}) \dot{\leqslant} \mathbb{0})}{\Gamma \vdash_{\mathcal{R}} \langle (\mathsf{x} \in \tau)\,?\,\mathsf{x}_1 : \mathsf{x}_2 \mid \mathtt{infer} \rangle \Rightarrow \mathtt{Subst}(\Psi, \mathtt{typ}, \in_2)}$$

When x has type $\mathbb{0}$, then [CaseEmpty] applies and returns $\mathtt{Ok}(\mathtt{typ})$ (the type-case is typeable using the algorithmic rule [$\mathbb{0}$-Alg]). Otherwise, when the type of x allows the selection of a branch, then either the rule [CaseThen] or the rule [CaseElse] applies. If we are in the case of [CaseThen], that is $\Gamma(\mathsf{x}) \leqslant \tau$, then we have to determine whether we will apply the algorithmic rule [$\mathbb{0}$-Alg] or the algorithmic rule [$\in_1$-Alg]. To determine it, the [CaseThen] rule calls $\mathsf{tally\_infer}(\Gamma(\mathsf{x}) \dot{\leqslant} \mathbb{0})$ which returns the set of contexts $\Gamma\psi$ (for $\psi \in \Psi$) under which the algorithmic rule [$\mathbb{0}$-Alg] is to be applied, that is, the contexts under which the tested expression x has an empty type. The default case, corresponding to the case in which the type of $\Gamma(\mathsf{x})$ is not guaranteed to be empty and, thus, in which the algorithmic rule [$\in_1$-Alg] must be applied, is annotated with $\in_1$. This annotation is handled by the rules [CaseVar$_1$] and [CaseOk$_1$] below, which force the system to type x_1, the binding variable associated to the first branch. The case of [CaseElse] and [CaseVar$_2$] is analogous.

$$[\textsc{CaseVar}_i]\ \frac{\mathsf{x}_i \notin \mathsf{dom}(\Gamma)}{\Gamma \vdash_{\mathcal{R}} \langle (\mathsf{x} \in \tau)\,?\,\mathsf{x}_1 : \mathsf{x}_2 \mid \in_i \rangle \Rightarrow \mathtt{Var}\ (\mathsf{x}_i, \mathtt{typ}, \mathtt{untyp})}$$

$$[\textsc{CaseOk}_i]\ \frac{}{\Gamma \vdash_{\mathcal{R}} \langle (\mathsf{x} \in \tau)\,?\,\mathsf{x}_1 : \mathsf{x}_2 \mid \in_i \rangle \Rightarrow \mathtt{Ok}(\mathtt{typ})}$$

The annotation $\in_1$ (resp. $\in_2$) is an intermediate annotation, it is only used to indicate that the rule [CaseThen] (resp. [CaseElse]) has already been applied, but that the first branch (resp. second branch) of the type-case may not have been analyzed yet. In the end, the type-case is either annotated with $\mathtt{typ}$ or $\mathtt{untyp}$.

One could wonder why the [CaseThen] and [CaseElse] rules are necessary. Indeed, when $\Gamma(\mathsf{x}) \leqslant \tau$, the [CaseVar$_1$] or [CaseOk$_1$] rule could directly be applied (and similarly when $\Gamma(\mathsf{x}) \leqslant \neg\tau$). However, the [CaseThen] and [CaseElse] rules allow finding contexts that may yield more precise types. For instance, consider the expression $\lambda x.\ (x \in \mathtt{Int})\,?\,\mathtt{true} : \mathtt{false}$. We would like to type it with the intersection type $(\mathtt{Int} \to \mathtt{True}) \wedge (\neg\mathtt{Int} \to \mathtt{False})$ rather than $\mathbb{1} \to \mathtt{Bool}$. Indeed, when x has type $\mathtt{Int}$, then the second branch of the type-case is not taken, and thus it does not need to be typed. Similarly, when x has type $\neg\mathtt{Int}$, then the first branch of the

type-case does not need to be considered. The goal of the rules [CaseThen] and [CaseElse] is to find these two particular contexts: the rule [CaseThen] searches the contexts under which the first branch of the type-case can be skipped, and the rule [CaseElse] searches the contexts under which the second branch of the type-case can be skipped.

$$[\textsc{LambdaInfer}]\ \frac{\Gamma \vdash_{\mathcal{R}} \langle \lambda x.\kappa \mid \lambda(\boldsymbol{\alpha}, \mathtt{infer})\rangle \Rightarrow \mathbb{R}}{\Gamma \vdash_{\mathcal{R}} \langle \lambda x.\kappa \mid \mathtt{infer}\rangle \Rightarrow \mathbb{R}}\ \boldsymbol{\alpha} \in \mathcal{V}_M \text{ fresh}$$

$$[\textsc{Lambda}]\ \frac{\Gamma, x : \mathbf{u} \vdash^*_{\mathcal{R}} \langle \kappa \mid \mathcal{K}\rangle \Rightarrow \mathbb{R}}{\Gamma \vdash_{\mathcal{R}} \langle \lambda x.\kappa \mid \lambda(\mathbf{u}, \mathcal{K})\rangle \Rightarrow \mathsf{map}(X \mapsto \lambda(\mathbf{u}, X),\ \mathbb{R})}$$

The rules for λ-abstractions mimic the algorithm $\mathcal{W}$. Rule [LambdaInfer] transforms the initial `infer` annotation into a $\lambda(\boldsymbol{\alpha}, \mathtt{infer})$ annotation. As in $\mathcal{W}$, λ-abstracted variables are initially typed with a fresh monomorphic type variable, which will then be substituted as needed while reconstructing the type of the body. Note that, although the type variable is monomorphic, it can still be substituted during the reconstruction of the body, as solutions found by tally_infer(.) involve monomorphic type variables (cf. Section 6.1). Rule [Lambda] adds the λ-abstracted variable to the environment with the type specified in the annotation, recursively calls reconstruction on the body, and reestablishes the variable type annotation on the result. The notation $\mathsf{map}(X{\mapsto}f(X),\ \mathbb{R})$ denotes the result $\mathbb{R}$ where f has been applied to every annotation X:

$$\begin{aligned}
\mathsf{map}(X \mapsto f(X),\ \mathtt{Ok}(\mathcal{H})) &\stackrel{\text{def}}{=} \mathtt{Ok}(f(\mathcal{H}))\\
\mathsf{map}(X \mapsto f(X),\ \mathtt{Fail}) &\stackrel{\text{def}}{=} \mathtt{Fail}\\
\mathsf{map}(X \mapsto f(X),\ \mathtt{Split}(\Gamma, \mathcal{H}_1, \mathcal{H}_2)) &\stackrel{\text{def}}{=} \mathtt{Split}(\Gamma, f(\mathcal{H}_1), f(\mathcal{H}_2))\\
\mathsf{map}(X \mapsto f(X),\ \mathtt{Subst}(\Psi, \mathcal{H}_1, \mathcal{H}_2)) &\stackrel{\text{def}}{=} \mathtt{Subst}(\Psi, f(\mathcal{H}_1), f(\mathcal{H}_2))\\
\mathsf{map}(X \mapsto f(X),\ \mathtt{Var}\ (\mathsf{x}, \mathcal{H}_1, \mathcal{H}_2)) &\stackrel{\text{def}}{=} \mathtt{Var}\ (\mathsf{x}, f(\mathcal{H}_1), f(\mathcal{H}_2))
\end{aligned}$$

Now that we have explained the rules for atoms, let us focus on those for canonical forms.

$$[\textsc{BindVar}]\ \frac{\mathsf{x} \notin \mathsf{dom}(\Gamma)}{\Gamma \vdash_{\mathcal{R}} \langle \mathsf{x} \mid \mathtt{infer}\rangle \Rightarrow \mathtt{Var}\ (\mathsf{x}, \mathtt{infer}, \mathtt{untyp})}$$

$$[\textsc{BindVarOk}]\ \frac{}{\Gamma \vdash_{\mathcal{R}} \langle \mathsf{x} \mid \mathtt{infer}\rangle \Rightarrow \mathtt{Ok}(\mathtt{typ})}$$

There is nothing new with the rules for binding variables. Note that they are different from the rules for lambda variables: while lambda variables are introduced by λ-abstractions, and thus they cannot be skipped (the variable is systematically added to the environment when entering the body of the λ-abstraction), binding

variables are introduced by bindings, and thus they may be skipped (cf. rule [BIND$_1$-ALG] of the algorithmic type system, Figure 5.2). That is why the rule [BINDVAR] returns `Var` $(\mathsf{x}, \mathtt{infer}, \mathtt{untyp})$ when the associated binding variable is not in the environment, while the rule [VARFAIL] given at page 124 returns `Fail` when the associated lambda variable is not in the environment (which can only happen if the source expression contains variables that are not bound by a λ-abstraction).

The rules for bindings are the most numerous. As bindings make the connection between the different atoms, there are many cases to consider, each case being handled by a rule.

$$[\textsc{BindInfer}]\ \frac{\Gamma \vdash_{\mathcal{R}} \langle \mathtt{bind}\,\mathsf{x} = a\,\mathtt{in}\,\kappa \ \mid\ \mathtt{try\text{-}skip}\ (\mathtt{infer}) \rangle \Rightarrow \mathbb{R}}{\Gamma \vdash_{\mathcal{R}} \langle \mathtt{bind}\,\mathsf{x} = a\,\mathtt{in}\,\kappa \ \mid\ \mathtt{infer} \rangle \Rightarrow \mathbb{R}}$$

The [BINDINFER] rule transforms an initial `infer` annotation into a `try-skip` (`infer`) annotation which skips the binding and annotates the body κ with `infer`. Indeed, we do not try to type the definition of a binding until it is actually used, because its variable might appear only in unreachable positions (e.g., in an unreachable branch of a type-case). In other words, we implement a lazy typing discipline for binding variables. If the variable is used at some point, then an attempt to type it will be initiated by the [BINDTRYSKIP$_1$] rule below:

$$[\textsc{BindTrySkip}_1]\ \frac{\begin{array}{c}\Gamma \vdash^*_{\mathcal{R}} \langle \kappa \ \mid\ \mathcal{K} \rangle \Rightarrow \mathtt{Var}\ (\mathsf{x}, \mathcal{K}_1, \mathcal{K}_2) \\ \Gamma \vdash_{\mathcal{R}} \langle \mathtt{bind}\,\mathsf{x} = a\,\mathtt{in}\,\kappa \ \mid\ \mathtt{try\text{-}keep}\ (\mathtt{infer}, \mathcal{K}_1, \mathcal{K}_2) \rangle \Rightarrow \mathbb{R}\end{array}}{\Gamma \vdash_{\mathcal{R}} \langle \mathtt{bind}\,\mathsf{x} = a\,\mathtt{in}\,\kappa \ \mid\ \mathtt{try\text{-}skip}\ (\mathcal{K}) \rangle \Rightarrow \mathbb{R}}$$

This rule tries to type the body of the binding, starting with the annotation $\mathcal{K}$ (initially `infer`). If the result is a `Var` $(\mathsf{x}, \mathcal{K}_1, \mathcal{K}_2)$, then it means that the current binding is used in the body κ and, thus, the system should try to type it. Consequently, the annotation for the current binding is changed into a `try-keep` $(\mathtt{infer}, \mathcal{K}_1, \mathcal{K}_2)$ so that, at the next iteration, its definition will be reconstructed.

If typing the body yields $\mathtt{Ok}(\mathcal{K}')$, it means that, in the current context, the body can be typed without using the binding variable `x`. Thus, the current annotation can be changed from `try-skip` $(\mathcal{K})$ to `skip` $\mathcal{K}'$:

$$[\textsc{BindTrySkip}_2]\ \frac{\Gamma \vdash^*_{\mathcal{R}} \langle \kappa \ \mid\ \mathcal{K} \rangle \Rightarrow \mathtt{Ok}(\mathcal{K}')}{\Gamma \vdash_{\mathcal{R}} \langle \mathtt{bind}\,\mathsf{x} = a\,\mathtt{in}\,\kappa \ \mid\ \mathtt{try\text{-}skip}\ (\mathcal{K}) \rangle \Rightarrow \mathtt{Ok}(\mathtt{skip}\ \mathcal{K}')}$$

Finally, if typing the body of the binding yields a result different from `Var` $(\mathsf{x}, \mathcal{K}_1, \mathcal{K}_2)$ and $\mathtt{Ok}(\mathcal{K}')$, then this result is just propagated as in [LAMBDA]:

$$[\textsc{BindTrySkip}_3]\ \frac{\Gamma \vdash^*_{\mathcal{R}} \langle \kappa \ \mid\ \mathcal{K} \rangle \Rightarrow \mathbb{R}}{\Gamma \vdash_{\mathcal{R}} \langle \mathtt{bind}\,\mathsf{x} = a\,\mathtt{in}\,\kappa \ \mid\ \mathtt{try\text{-}skip}\ (\mathcal{K}) \rangle \Rightarrow \mathbb{R}'}$$

where $\mathbb{R}' = \mathtt{map}(X \mapsto \mathtt{try\text{-}skip}\ (X),\ \mathbb{R})$.

Now, we need some rules to handle the annotation $\texttt{try-keep}\ (\mathcal{A}, \mathcal{K}_1, \mathcal{K}_2)$:

$$[\textsc{BindTryKeep}_1]\ \frac{\Gamma \vdash^*_{\mathcal{R}} \langle a \mid \mathcal{A}\rangle \Rightarrow \texttt{Ok}(\mathcal{A}') \qquad \Gamma \vdash_{\mathcal{R}} \langle \texttt{bind}\,\texttt{x} = a\,\texttt{in}\,\kappa \mid \texttt{keep}\ (\mathcal{A}', \{(\mathbb{1}, \mathcal{K}_1)\}, \varnothing)\rangle \Rightarrow \mathbb{R}}{\Gamma \vdash_{\mathcal{R}} \langle \texttt{bind}\,\texttt{x} = a\,\texttt{in}\,\kappa \mid \texttt{try-keep}\ (\mathcal{A}, \mathcal{K}_1, \mathcal{K}_2)\rangle \Rightarrow \mathbb{R}}$$

$$[\textsc{BindTryKeep}_2]\ \frac{\Gamma \vdash^*_{\mathcal{R}} \langle a \mid \mathcal{A}\rangle \Rightarrow \texttt{Fail} \qquad \Gamma \vdash_{\mathcal{R}} \langle \texttt{bind}\,\texttt{x} = a\,\texttt{in}\,\kappa \mid \texttt{skip}\ \mathcal{K}_2\rangle \Rightarrow \mathbb{R}}{\Gamma \vdash_{\mathcal{R}} \langle \texttt{bind}\,\texttt{x} = a\,\texttt{in}\,\kappa \mid \texttt{try-keep}\ (\mathcal{A}, \mathcal{K}_1, \mathcal{K}_2)\rangle \Rightarrow \mathbb{R}}$$

$$[\textsc{BindTryKeep}_3]\ \frac{\Gamma \vdash^*_{\mathcal{R}} \langle a \mid \mathcal{A}\rangle \Rightarrow \mathbb{R}}{\Gamma \vdash_{\mathcal{R}} \langle \texttt{bind}\,\texttt{x} = a\,\texttt{in}\,\kappa \mid \texttt{try-keep}\ (\mathcal{A}, \mathcal{K}_1, \mathcal{K}_2)\rangle \Rightarrow \mathbb{R}'}$$

where $\mathbb{R}' = \texttt{map}(X \mapsto \texttt{try-keep}\ (X, \mathcal{K}_1, \mathcal{K}_2),\ \mathbb{R})$.

As expected, if the current annotation for the binding is a $\texttt{try-keep}\ (\mathcal{A}, \mathcal{K}_1, \mathcal{K}_2)$, then the system tries to reconstruct the annotation for the definition. If it succeeds, then it becomes possible to type the definition and to continue the reconstruction of the body using $\mathcal{K}_1$. This is what [BindTryKeep$_1$] does by changing the current annotation to $\texttt{keep}\ (\mathcal{A}', \{(\mathbb{1}, \mathcal{K}_1)\}, \varnothing)$ (more details are given later). If the reconstruction of the definition fails (rule [BindTryKeep$_2$]), then we have no choice but to skip this definition and use the default annotation $\mathcal{K}_2$ to type the body. Finally, the rule [BindSkip$_3$] handles all the other possibles results for the reconstruction of the definition, and simply forwards them after reestablishing the annotation for the current binding.

The rules for handling $\texttt{skip}\ \mathcal{K}$ annotations are as follows:

$$[\textsc{BindSkip}_1]\ \frac{\Gamma \vdash^*_{\mathcal{R}} \langle \kappa \mid \mathcal{K}\rangle \Rightarrow \texttt{Var}\ (\texttt{x}, \mathcal{K}_1, \mathcal{K}_2) \qquad \Gamma \vdash_{\mathcal{R}} \langle \texttt{bind}\,\texttt{x} = a\,\texttt{in}\,\kappa \mid \texttt{skip}\ \mathcal{K}_2\rangle \Rightarrow \mathbb{R}}{\Gamma \vdash_{\mathcal{R}} \langle \texttt{bind}\,\texttt{x} = a\,\texttt{in}\,\kappa \mid \texttt{skip}\ \mathcal{K}\rangle \Rightarrow \mathbb{R}}$$

$$[\textsc{BindSkip}_2]\ \frac{\Gamma \vdash^*_{\mathcal{R}} \langle \kappa \mid \mathcal{K}\rangle \Rightarrow \mathbb{R}}{\Gamma \vdash_{\mathcal{R}} \langle \texttt{bind}\,\texttt{x} = a\,\texttt{in}\,\kappa \mid \texttt{skip}\ \mathcal{K}\rangle \Rightarrow \texttt{map}(X \mapsto \texttt{skip}\ X,\ \mathbb{R})}$$

The $\texttt{skip}\ \mathcal{K}$ annotation means that the definition of the binding must be skipped. Thus, if reconstructing annotations for body of the binding yields $\texttt{Var}\ (\texttt{x}, \mathcal{K}_1, \mathcal{K}_2)$ (rule [BindSkip$_1$]), we do not attempt reconstructing annotations for the definition as in the [BindTrySkip$_1$] rule, but instead the annotation for the body is changed to $\mathcal{K}_2$, and we try to reconstruct it again. The rule [BindSkip$_2$] forwards any other result, as usual.

Finally, we focus on the rules handling the $\texttt{keep}\ (\mathcal{A}, \mathcal{S}, \mathcal{S}')$ annotations. For a binding $\texttt{bind}\,\texttt{x} = a\,\texttt{in}\,\kappa$ and annotation $\texttt{keep}\ (\mathcal{A}, \mathcal{S}, \mathcal{S}')$:

$\mathcal{A}$ is the annotation for typing the definition of x,

$\mathcal{S}$ describes the type decomposition to use for x and, for each part of the decomposition, the annotation to use for the body. It only contains parts of the decomposition whose annotations have not been fully reconstructed yet.

$\mathcal{S}'$ also describe the type decomposition to use for x, but it contains only the parts whose annotations have already been fully reconstructed.

For instance, the annotation $\mathsf{keep}\ (\mathcal{A}', \{(\mathbb{1}, \mathcal{K}_1)\}, \varnothing)$ used in rule [BindTryKeep$_1$] means that the type of the definition does not need to be partitioned: there is only one part, covering $\mathbb{1}$, associated to an annotation $\mathcal{K}_1$ (for typing the body) which has not been fully reconstructed yet.

In the more general case, types and annotations in $\mathcal{S}$ are processed one a time. Those yielding a successful typing are recorded in $\mathcal{S}'$. This is handled by the set of rules below.

$$[\textsc{BindOk}]\ \frac{}{\Gamma \vdash_{\mathcal{R}} \langle \mathsf{bind}\,\mathsf{x} = a\,\mathsf{in}\,\kappa \ \mid\ \mathsf{keep}\ (\mathcal{A}, \varnothing, \mathcal{S}) \rangle \Rightarrow \mathsf{Ok}(\mathsf{keep}\ (\mathcal{A}, \varnothing, \mathcal{S}))}$$

If all the parts of the type decomposition have already been reconstructed, then the reconstruction is successful. Otherwise, the following rules are applied:

$$[\textsc{BindKeep}_1]\ \frac{\begin{array}{c}\Gamma \vdash_{\mathcal{S}} \langle a \mid \mathcal{A} \rangle \Rightarrow \mathbb{a} \\ \Gamma \vdash_{\mathcal{A}} [a \mid \mathbb{a}] : s \qquad \Gamma, \mathsf{x} : s \wedge \mathbf{u} \vdash^*_{\mathcal{R}} \langle \kappa \mid \mathcal{K} \rangle \Rightarrow \mathsf{Ok}(\mathcal{K}') \\ \Gamma \vdash_{\mathcal{R}} \langle \mathsf{bind}\,\mathsf{x} = a\,\mathsf{in}\,\kappa \ \mid\ \mathsf{keep}\ (\mathcal{A}, \mathcal{S}, \{(\mathbf{u}, \mathcal{K}')\} \cup \mathcal{S}') \rangle \Rightarrow \mathbb{R}\end{array}}{\Gamma \vdash_{\mathcal{R}} \langle \mathsf{bind}\,\mathsf{x} = a\,\mathsf{in}\,\kappa \ \mid\ \mathsf{keep}\ (\mathcal{A}, \{(\mathbf{u}, \mathcal{K})\} \cup \mathcal{S}, \mathcal{S}') \rangle \Rightarrow \mathbb{R}}$$

$$[\textsc{BindKeep}_2]\ \frac{\begin{array}{c}\Gamma \vdash_{\mathcal{S}} \langle a \mid \mathcal{A} \rangle \Rightarrow \mathbb{a} \qquad \Gamma \vdash_{\mathcal{A}} [a \mid \mathbb{a}] : s \\ \Gamma, \mathsf{x} : s \wedge \mathbf{u} \vdash^*_{\mathcal{R}} \langle \kappa \mid \mathcal{K} \rangle \Rightarrow \mathsf{Split}(\Gamma', \mathcal{K}_1, \mathcal{K}_2) \qquad \mathsf{x} \in \mathsf{dom}(\Gamma') \\ \Gamma \vdash_{\mathcal{B}} (a : \neg(\mathbf{u} \wedge \Gamma'(\mathsf{x}))) \Rightarrow \mathbb{\Gamma}_1 \qquad \Gamma \vdash_{\mathcal{B}} (a : \neg(\mathbf{u} \backslash \Gamma'(\mathsf{x}))) \Rightarrow \mathbb{\Gamma}_2\end{array}}{\Gamma \vdash_{\mathcal{R}} \langle \mathsf{bind}\,\mathsf{x} = a\,\mathsf{in}\,\kappa \ \mid\ \mathsf{keep}\ (\mathcal{A}, \{(\mathbf{u}, \mathcal{K})\} \cup \mathcal{S}, \mathcal{S}') \rangle \Rightarrow \mathbb{R}'}$$

where $\mathbb{R}' = \mathsf{Split}(\Gamma' \backslash \mathsf{x}, \mathcal{K}'_1, \mathcal{K}'_2)$ with $\mathcal{K}'_1 = \mathsf{propagate}\ (\mathcal{A}, \mathbb{\Gamma}_1 \cup \mathbb{\Gamma}_2, \{(\mathbf{u} \wedge \Gamma'(\mathsf{x}), \mathcal{K}_1), (\mathbf{u} \backslash \Gamma'(\mathsf{x}), \mathcal{K}_2)\} \cup \mathcal{S}, \mathcal{S}')$ and $\mathcal{K}'_2 = \mathsf{keep}\ (\mathcal{A}, \{(\mathbf{u}, \mathcal{K}_2)\} \cup \mathcal{S}, \mathcal{S}')$.

$$[\textsc{BindKeep}_3]\ \frac{\begin{array}{c}\Gamma \vdash_{\mathcal{S}} \langle a \mid \mathcal{A} \rangle \Rightarrow \mathbb{a} \\ \Gamma \vdash_{\mathcal{A}} [a \mid \mathbb{a}] : s \qquad \Gamma, \mathsf{x} : s \wedge \mathbf{u} \vdash^*_{\mathcal{R}} \langle \kappa \mid \mathcal{K} \rangle \Rightarrow \mathbb{R}\end{array}}{\Gamma \vdash_{\mathcal{R}} \langle \mathsf{bind}\,\mathsf{x} = a\,\mathsf{in}\,\kappa \ \mid\ \mathsf{keep}\ (\mathcal{A}, \{(\mathbf{u}, \mathcal{K})\} \cup \mathcal{S}, \mathcal{S}') \rangle \Rightarrow \mathbb{R}'}$$

where $\mathbb{R}' = \mathsf{map}(X \mapsto \mathsf{keep}\ (\mathcal{A}, \{(\mathbf{u}, X)\} \cup \mathcal{S}, \mathcal{S}'),\ \mathbb{R})$.

In each rule, the definition of the binding is typed using the annotation $\mathcal{A}$. For that, it is first converted into an annotation $\mathbb{a}$ of the algorithmic type system, using the deduction rules for the judgment $\Gamma \vdash_{\mathcal{S}} \langle a \mid \mathcal{A} \rangle \Rightarrow \mathbb{a}$, defined in Section 6.3. Then, the type s obtained for the definition is intersected with one of the parts $\mathbf{u}$ of the type decomposition, according to the second argument of the keep annotation (i.e., $\{(\mathbf{u}, \mathcal{K})\} \cup \mathcal{S}$ in each rule), and the corresponding annotation for the body is reconstructed recursively. Note that this type $s \wedge \mathbf{u}$ precisely corresponds to the one in the $[\vee]$ rule of the declarative type system (Figure 4.1) and [Bind$_2$] rule of the algorithmic type system (Figure 5.2). Also note that, since split annotations are sets, the order in which the parts are explored is arbitrary.

The rule [BindKeep$_1$], for an annotation keep $(\mathcal{A}, \mathcal{S}, \mathcal{S}')$, is responsible for moving a branch from $\mathcal{S}$ to $\mathcal{S}'$ when its reconstruction is over (i.e., when the result for the branch is Ok()). If instead the reconstruction of the body requires to further split the type of x, then the rule [BindKeep$_2$] splits the current branch into two branches. However, before exploring these two branches, some information about the split needs to be propagated in order to ensure that when a split is explored, it is under a context as precise as possible. This is where the backward propagation of types mentioned in our introduction occurs.

Let us explain this by an example. Assume we have, in the type environment, a binding variable f of type $\alpha \to \alpha$ and a lambda variable x of type Bool (i.e., $\Gamma = \{\mathsf{f} : \alpha \to \alpha \ ; \ x : \mathtt{Bool}\}$). We want to type the following canonical form, and deduce for it the type True (since x and y are always bound to the same value):

$$\texttt{bind x = } x \texttt{ in bind y = f x in bind z = (y}\in\texttt{True) ? x : true in z}$$

At some point, the type partition associated to y will change from $\{\mathbb{1}\}$ to $\{\mathtt{True}, \neg\mathtt{True}\}$ because of the type-case (rule [CaseSplit]). However, if the case corresponding to (y : True) is immediately explored, it will yield for the body the type Bool, because x still has the type Bool in the environment. In order to obtain the more precise type True, we must deduce, before exploring the case (y : True), that when f x (the definition of y) has type True, then x also has type True (since f is of type $\alpha \to \alpha$). Knowing that, the type of x should be split accordingly into $\{\mathtt{True}, \neg\mathtt{True}\}$. This way, the following environments will be considered by our reconstruction algorithm:

$$\Gamma_1 = \{\mathsf{x} : \mathtt{True};\ \mathsf{y} : \mathtt{True}\}\ ; \qquad \Gamma_2 = \{\mathsf{x} : \mathtt{True};\ \mathsf{y} : \mathbb{0}\}\ ;$$
$$\Gamma_3 = \{\mathsf{x} : \neg\mathtt{True};\ \mathsf{y} : \mathbb{0}\}\ ; \qquad \Gamma_4 = \{\mathsf{x} : \neg\mathtt{True};\ \mathsf{y} : \neg\mathtt{True}\}$$

Under Γ_1 and Γ_4, z will be typed True (using [$\in_1$-Alg] and [$\in_2$-Alg]), and under Γ_2 and Γ_3 it will be typed $\mathbb{0}$ (using [$\mathbb{0}$-Alg]). Thus, we obtain the type True for this expression. Note that it may seem redundant to explore the environments Γ_2 and Γ_4 (the environments Γ_2 and Γ_3 already capture every possible case). Still, these redundant cases are explored by the reconstruction algorithm: for instance, Γ_2 corresponds to the case where the True part is selected for x, and the False part is selected for y (to be intersected with the type True of the atom associated to y). We will see in Chapter 8 how to change the reconstruction algorithm to avoid exploring such redundant cases.

This mechanism of backward propagation of splits is initiated in the [BindKeep$_2$] rule with the two premises $\Gamma \vdash_{\mathcal{B}} (a : \neg(\mathbf{u} \wedge \Gamma'(\mathsf{x}))) \Rightarrow \mathbb{\Gamma}_1$ and $\Gamma \vdash_{\mathcal{B}} (a : \neg(\mathbf{u} \backslash \Gamma'(\mathsf{x}))) \Rightarrow \mathbb{\Gamma}_2$. This auxiliary judgment $\Gamma \vdash_{\mathcal{B}} (a : \mathbf{u}) \Rightarrow \mathbb{\Gamma}$, that will be formally defined in Section 6.4, can be read as follows: "intersecting the current environment Γ with one of the $\Gamma' \in \mathbb{\Gamma}$ makes the type $\mathbf{u}$ derivable for the atom a". The refinements $\mathbb{\Gamma}_1$ and $\mathbb{\Gamma}_2$ we obtain are stored in the annotation of the binding, using an annotation propagate $(\mathcal{A}, \mathbb{\Gamma}_1 \cup \mathbb{\Gamma}_2, \ldots, \ldots)$. This annotation is

then handled by the two following rules:

$$[\textsc{BindProp}_1]\ \frac{\Gamma' \in \mathbb{\Gamma} \qquad \mathsf{compatible}(\Gamma, \Gamma')}{\Gamma \vdash_{\mathcal{R}} \langle \mathtt{bind\,x = }a\,\mathtt{in}\,\kappa \ \mid\ \mathtt{propagate}\ (\mathcal{A}, \mathbb{\Gamma}, \mathcal{S}, \mathcal{S}') \rangle \Rightarrow \mathbb{R}'}$$

where $\mathbb{R}' = \mathtt{Split}(\Gamma'', \mathcal{K}_1, \mathcal{K}_2)$ with $\Gamma'' = \{(\mathsf{x} : \mathbf{u}) \in \Gamma' \mid \Gamma(\mathsf{x}) \not\leqslant \mathbf{u}\}$, $\mathcal{K}_1 = \mathtt{keep}\ (\mathcal{A}, \mathcal{S}, \mathcal{S}')$, and $\mathcal{K}_2 = \mathtt{propagate}\ (\mathcal{A}, \mathbb{\Gamma} \backslash \{\Gamma'\}, \mathcal{S}, \mathcal{S}')$.

$$[\textsc{BindProp}_2]\ \frac{\Gamma \vdash_{\mathcal{R}} \langle \mathtt{bind\,x = }a\,\mathtt{in}\,\kappa \ \mid\ \mathtt{keep}\ (\mathcal{A}, \mathcal{S}, \mathcal{S}') \rangle \Rightarrow \mathbb{R}}{\Gamma \vdash_{\mathcal{R}} \langle \mathtt{bind\,x = }a\,\mathtt{in}\,\kappa \ \mid\ \mathtt{propagate}\ (\mathcal{A}, \mathbb{\Gamma}, \mathcal{S}, \mathcal{S}') \rangle \Rightarrow \mathbb{R}}$$

The role of those two rules is to propagate the refinements in $\mathbb{\Gamma}$, so that when the reconstruction of this binding continues, it is either under an environment that refines one of the $\Gamma' \in \mathbb{\Gamma}$ or under an environment that is disjoint from all of them. The relation $\mathsf{compatible}(\Gamma, \Gamma')$ ensures that Γ' is a valid refinement for Γ and that it is not already disjoint from it. Formally, it is defined as follows:

$$\begin{aligned}\mathsf{compatible}(\Gamma, \Gamma') \Leftrightarrow\ & (\mathsf{dom}(\Gamma') \subseteq \mathsf{dom}(\Gamma)) \text{ and} \\ & (\forall \mathsf{x} \in \mathsf{dom}(\Gamma').\ (\Gamma(\mathsf{x}) \wedge \Gamma'(\mathsf{x}) \not\simeq \mathbb{0}) \text{ or } (\Gamma(\mathsf{x}) \simeq \mathbb{0}))\end{aligned}$$

If at least one of the $\Gamma' \in \mathbb{\Gamma}$ is compatible with the current environment Γ, then $[\textsc{BindProp}_1]$ initiates a split of the current environment Γ according to Γ' using a $\mathtt{Split}(\Gamma'', \mathcal{K}_1, \mathcal{K}_2)$ result, where Γ'' is just a filtered version of Γ' where only strict refinements are kept. The annotation $\mathcal{K}_1$ corresponds to the annotation to use in the case where the environment refines Γ': in this case, we can continue the reconstruction with a $\mathtt{keep}\ (\mathcal{A}, \mathcal{S}, \mathcal{S}')$ annotation. In the other cases, which use the annotation $\mathcal{K}_2$, we continue the backpropagation with the remaining refinements. When there is no compatible refinement left, Rule $[\textsc{BindProp}_2]$ continues the reconstruction with an annotation $\mathtt{keep}\ (\mathcal{A}, \mathcal{S}, \mathcal{S}')$.

Lastly, we need rules for intersections. Until now, we have not used intersection annotations, but they will be used by the $\vdash^*_{\mathcal{R}}$ judgments defined at the end of this section. Indeed, intersection annotations allow us to explore multiple typing derivations for a given atom or canonical form, which is useful when we have several cases to explore (in particular, when reconstructing annotations for an overloaded function). The rules for intersection annotations are the following (we recall that η denotes either an atom or a canonical form, and $\mathcal{H}$ denotes either an atom intermediate annotation or a form intermediate annotation):

$$[\textsc{InterEmpty}]\ \frac{}{\Gamma \vdash_{\mathcal{R}} \langle \eta \ \mid\ \bigwedge(\varnothing, \varnothing) \rangle \Rightarrow \mathtt{Fail}}$$

$$[\textsc{InterOk}]\ \frac{}{\Gamma \vdash_{\mathcal{R}} \langle \eta \ \mid\ \bigwedge(\varnothing, S) \rangle \Rightarrow \mathtt{Ok}(\bigwedge(\varnothing, S))}$$

$$[\textsc{Inter}_1]\ \frac{\Gamma \vdash^*_{\mathcal{R}} \langle \eta \ \mid\ \mathcal{H} \rangle \Rightarrow \mathtt{Ok}(\mathcal{H}') \qquad \Gamma \vdash_{\mathcal{R}} \langle \eta \ \mid\ \bigwedge(S, \{\mathcal{H}'\} \cup S') \rangle \Rightarrow \mathbb{R}}{\Gamma \vdash_{\mathcal{R}} \langle \eta \ \mid\ \bigwedge(\{\mathcal{H}\} \cup S, S') \rangle \Rightarrow \mathbb{R}}$$

$$[\textsc{Inter}_2]\ \frac{\Gamma \vdash^*_{\mathcal{R}} \langle \eta \mid \mathcal{H} \rangle \Rightarrow \mathtt{Fail} \qquad \Gamma \vdash_{\mathcal{R}} \langle \eta \mid \bigwedge(S, S') \rangle \Rightarrow \mathbb{R}}{\Gamma \vdash_{\mathcal{R}} \langle \eta \mid \bigwedge(\{\mathcal{H}\} \cup S, S') \rangle \Rightarrow \mathbb{R}}$$

$$[\textsc{Inter}_3]\ \frac{\Gamma \vdash^*_{\mathcal{R}} \langle \eta \mid \mathcal{H} \rangle \Rightarrow \mathbb{R}}{\Gamma \vdash_{\mathcal{R}} \langle \eta \mid \bigwedge(\{\mathcal{H}\} \cup S, S') \rangle \Rightarrow \mathtt{map}(X \mapsto (\bigwedge(\{X\} \cup S, S')),\ \mathbb{R})}$$

In an intersection annotation $\bigwedge(S, S')$, the annotations in S' are fully processed (i.e., the associated reconstruction returned $\mathtt{Ok}()$), while the annotations in S are not: they still have to be refined one after the other (rule [InterE$_3$]). If one of them becomes fully processed, it is moved in S' (rule [InterE$_1$]). Conversely, if one of them fails, it is removed (rule [InterE$_2$]). The process stops when S is empty: then, the reconstruction fails if S' is empty (rule [InterEmpty]), and succeed otherwise (rule [InterOk]).

Now, we formalize the rules for the judgments $\vdash^*_{\mathcal{R}}$. As said earlier, the purpose of $\vdash^*_{\mathcal{R}}$ is to repeatedly call $\vdash_{\mathcal{R}}$ judgments so that, in the end, we obtain a terminal result.

$$[\textsc{Iterate}_1]\ \frac{\Gamma \vdash_{\mathcal{R}} \langle \eta \mid \mathcal{H} \rangle \Rightarrow \mathtt{Split}(\Gamma', \mathcal{H}_1, \mathcal{H}_2) \qquad \Gamma \vdash^*_{\mathcal{R}} \langle \eta \mid \mathcal{H}_1 \rangle \Rightarrow \mathbb{R}}{\Gamma \vdash^*_{\mathcal{R}} \langle \eta \mid \mathcal{H} \rangle \Rightarrow \mathbb{R}}\ \Gamma' = \varnothing$$

$$[\textsc{Iterate}_2]\ \frac{\Gamma \vdash_{\mathcal{R}} \langle \eta \mid \mathcal{H} \rangle \Rightarrow \mathtt{Subst}(\{\psi_i\}_{i \in I}, \mathcal{H}_1, \mathcal{H}_2) \quad \Gamma \vdash^*_{\mathcal{R}} \langle \eta \mid \bigwedge(\{\mathcal{H}_1\psi_i\}_{i \in I} \cup \{\mathcal{H}_2\}, \varnothing) \rangle \Rightarrow \mathbb{R}}{\Gamma \vdash^*_{\mathcal{R}} \langle \eta \mid \mathcal{H} \rangle \Rightarrow \mathbb{R}}\ \forall i \in I.\ \psi_i \# \Gamma$$

where $\mathcal{H}\psi$ is the intermediate annotation $\mathcal{H}$ in which the substitution ψ has been applied recursively to every type in it.

The iteration continues as long as it yields non-terminal results that are immediately usable, that is, either they return a trivial split (i.e., $\Gamma' = \varnothing$) as in rule [Iterate$_1$], or they return substitutions that do not affect the current environment (i.e., $\psi_i \# \Gamma$) as in rule [Iterate$_2$]. For the latter rule, the iteration may need to introduce an intersection annotation (useless when I is empty) in order to explore all the cases $\mathcal{H}_1\psi_i$ and the default case $\mathcal{H}_2$ of a $\mathtt{Subst}(\{\psi_i\}_{i \in I}, \mathcal{H}_1, \mathcal{H}_2)$ result.

If the result is already terminal or if it is not immediately usable, then it is directly returned:

$$[\textsc{Stop}]\ \frac{\Gamma \vdash_{\mathcal{R}} \langle \eta \mid \mathcal{H} \rangle \Rightarrow \mathbb{R}}{\Gamma \vdash^*_{\mathcal{R}} \langle \eta \mid \mathcal{H} \rangle \Rightarrow \mathbb{R}}$$

In particular, if $\mathbb{R} = \mathtt{Split}(\Gamma', \mathcal{H}_1, \mathcal{H}_2)$ where $\Gamma' \neq \varnothing$ (i.e., [Iterate$_1$] does not apply), then [Stop] backtracks until Γ' becomes empty; likewise if $\mathbb{R} = \mathtt{Subst}(\{\psi_i\}_{i \in I}, \mathcal{H}_1, \mathcal{H}_2)$ and $\Gamma\psi_i \not\approx \Gamma$ for some i (i.e., [Iterate$_2$] does not apply), then [Stop] backtracks until it exits the scope of the binders of the variables that make the side condition of [Iterate$_2$] fail.

An important remark about the [Iterate$_2$] rule is that, in addition to all the instantiations $\mathcal{H}_1\psi_i$ (for $i \in I$), the default case $\mathcal{H}_2$ is always considered. This default case can be useful in order not to lose generality. The [CaseThen] and [CaseElse]

rules discussed earlier are a good illustration. These two rules look for some possible instantiations Ψ of the type environment that would make the first or second branch of a type-case unreachable. It is interesting to explore such environments separately as they allow to find a smaller type for the type-case. However, it is not necessary for a type-case to have an unreachable branch in order to be typeable: for instance, `(rand_bool () ∈True)?42:41` is typeable (with `rand_bool : Unit → Bool`) even though no instantiation of $\Gamma = \{\texttt{rand\_bool} : \texttt{Unit} \to \texttt{Bool}\}$ can make a branch unreachable.

Sometimes, however, the default case $\mathcal{H}_2$ considered by the [ITERATE$_2$] rule might be useless, for instance if the annotation $\mathcal{H}_2$ is `untyp`, or if one of the ψ_i is the identity substitution. In the first case, the branch $\mathcal{H}_2$ of the intersection annotation will be explored at some point, yielding `Fail`, which will make the [INTER$_2$] rule to apply and to remove this useless branch. In the second case, the branch $\mathcal{H}_2$ will also be explored, but its reconstruction might succeed, and thus the branch might be kept. While this is redundant, as the type obtained for this branch will be larger than the type obtained for the $\mathcal{H}_1$ branch, this is not an issue in the theory. In practice, however, redundant branches should be eliminated for performance reasons. This will be discussed in Chapter 8.

6.3 Substitution inference system

The substitution inference system defined in this section converts an intermediate annotation of the main reconstruction system into an annotation for the algorithmic type system. For that, it needs to retrieve the polymorphic instantiations Σ needed to type the atoms.

Formally, the algorithm takes as input an environment Γ, an atom or canonical form η, and an intermediate annotation $\mathcal{H}$, and produces an annotation $\mathbb{h}$ for the algorithmic type system. It is presented as a deduction system for judgments of the form $\Gamma \vdash_{\mathcal{S}} \langle \eta \mid \mathcal{H} \rangle \Rightarrow \mathbb{h}$. The intermediate annotation $\mathcal{H}$ given as input is assumed to be terminal: it should result from a judgment $\Gamma \vdash^*_{\mathcal{R}} \langle \eta \mid \mathcal{H}' \rangle \Rightarrow \mathsf{Ok}(\mathcal{H})$.

$$[\textsc{Const}]\ \frac{}{\Gamma \vdash_{\mathcal{S}} \langle c \mid \mathtt{typ} \rangle \Rightarrow \varnothing} \qquad [\textsc{Var}]\ \frac{}{\Gamma \vdash_{\mathcal{S}} \langle x \mid \mathtt{typ} \rangle \Rightarrow \varnothing}\ x \in \mathsf{dom}(\Gamma)$$

The rules for constants and variables just transform an intermediate annotation `typ` into an annotation $\varnothing$ for the algorithmic type system.

$$[\textsc{Pair}]\ \frac{\rho_1 = \mathsf{refresh}(\Gamma(\mathsf{x}_1)) \qquad \rho_2 = \mathsf{refresh}(\Gamma(\mathsf{x}_2))}{\Gamma \vdash_{\mathcal{S}} \langle (\mathsf{x}_1, \mathsf{x}_2) \mid \mathtt{typ} \rangle \Rightarrow (\rho_1, \rho_2)}$$

where $\mathsf{refresh}(t)$ denotes a renaming from $\mathsf{vars}(t) \cap \mathcal{V}_P$ to fresh polymorphic type variables.

The rule for pairs must produce an annotation (ρ_1, ρ_2), with ρ_1 a renaming for the polymorphic type variables of the first component x_1 of the pair, and ρ_2 a

renaming for polymorphic type variables of the second component x_2 of the pair. The polymorphic type variables of each component are renamed to fresh type variables, in order to avoid unnecessary correlations between the two components in the resulting product type.

The most important rules for this system are the one for projections and applications:

$$[\textsc{Proj}]\ \frac{\Sigma = \mathsf{tally}(\{\Gamma(\mathsf{x}) \mathrel{\dot{\leqslant}} \alpha \times \beta\})}{\Gamma \vdash_{\mathcal{S}} \langle \pi_i \mathsf{x} \mid \mathsf{typ} \rangle \Rightarrow \pi(\Sigma)} \begin{array}{l} \Sigma \neq \varnothing \\ \alpha, \beta \in \mathcal{V}_P \text{ fresh} \end{array}$$

$$[\textsc{App}]\ \frac{\begin{array}{c} t_1 = \Gamma(\mathsf{x}_1) \qquad t_2 = \Gamma(\mathsf{x}_2) \qquad \rho_1 = \mathsf{refresh}(t_1) \\ \rho_2 = \mathsf{refresh}(t_2) \qquad \Sigma = \mathsf{tally}(\{t_1\rho_1 \mathrel{\dot{\leqslant}} t_2\rho_2 \to \alpha\}) \end{array}}{\Gamma \vdash_{\mathcal{S}} \langle \mathsf{x}_1\mathsf{x}_2 \mid \mathsf{typ} \rangle \Rightarrow @(\{\sigma \circ \rho_1 \mid \sigma \in \Sigma\}, \{\sigma \circ \rho_2 \mid \sigma \in \Sigma\})} \begin{array}{l} \Sigma \neq \varnothing \\ \alpha \in \mathcal{V}_P \text{ fresh} \end{array}$$

For applications, an annotation of the form $@(\Sigma_1, \Sigma_2)$ must be produced. In order to find some instantiations Σ_1 and Σ_2 (for x_1 and x_2 respectively) that make the application typeable, the [App] rule solves the tallying instance $\mathsf{tally}(\{t_1\rho_1 \mathrel{\dot{\leqslant}} t_2\rho_2 \to \alpha\})$. The purpose of ρ_1 and ρ_2 is to decorrelate type variables in $\Gamma(\mathsf{x}_1)$ and in $\Gamma(\mathsf{x}_2)$. For instance, assume we want to reconstruct the instantiations for the atom $\mathsf{x}\,\mathsf{x}$ with $\Gamma(\mathsf{x}) = \beta \to \beta$. The tallying instance $\mathsf{tally}(\{\beta \to \beta \mathrel{\dot{\leqslant}} (\beta \to \beta) \to \alpha\})$ yields only a very specific, uninteresting solution (i.e., $\alpha = \beta = \mu X.\ X \to X$) because of the use of the same type variable β on both sides of $\mathrel{\dot{\leqslant}}$. But each occurrence of x has a polymorphic type that can be instantiated independently. Thus, we remove this useless and constraining dependency by refreshing the generic type variables yielding $\mathsf{tally}(\{\beta' \to \beta' \mathrel{\dot{\leqslant}} (\beta \to \beta) \to \alpha\})$ which has interesting solutions, in particular $\{\beta' \rightsquigarrow \beta \to \beta\ ;\ \alpha \rightsquigarrow \beta \to \beta\}$.

The side-condition $\Sigma \neq \varnothing$ ensures that the tallying instance has at least one solution (otherwise the annotation produced would be invalid).

$$[\textsc{Case}_{\mathbb{0}}]\ \frac{\sigma \in \mathsf{tally}(\{\Gamma(\mathsf{x}) \mathrel{\dot{\leqslant}} \mathbb{0}\})}{\Gamma \vdash_{\mathcal{S}} \langle (\mathsf{x} \in \tau)\ ?\,\mathsf{x}_1 : \mathsf{x}_2 \mid \mathsf{typ} \rangle \Rightarrow \mathbb{0}(\{\sigma\})}$$

$$[\textsc{Case}_1]\ \frac{\sigma \in \mathsf{tally}(\{\Gamma(\mathsf{x}) \mathrel{\dot{\leqslant}} \tau\})}{\Gamma \vdash_{\mathcal{S}} \langle (\mathsf{x} \in \tau)\ ?\,\mathsf{x}_1 : \mathsf{x}_2 \mid \mathsf{typ} \rangle \Rightarrow \in_1(\{\sigma\})}\ \mathsf{x}_1 \in \mathsf{dom}(\Gamma)$$

$$[\textsc{Case}_2]\ \frac{\sigma \in \mathsf{tally}(\{\Gamma(\mathsf{x}) \mathrel{\dot{\leqslant}} \neg\tau\})}{\Gamma \vdash_{\mathcal{S}} \langle (\mathsf{x} \in \tau)\ ?\,\mathsf{x}_1 : \mathsf{x}_2 \mid \mathsf{typ} \rangle \Rightarrow \in_2(\{\sigma\})}\ \mathsf{x}_2 \in \mathsf{dom}(\Gamma)$$

For type-cases, we need to determine which rule of the algorithmic type system should be applied between [$\mathbb{0}$-Alg], [$\in_1$-Alg], and [$\in_2$-Alg]. The rule [$\textsc{Case}_{\mathbb{0}}$] applies in priority, checking using tallying whether there exists a substitution σ such that $\Gamma(\mathsf{x})\sigma \leqslant \mathbb{0}$. If such an instantiation exists, then the type-case can be typed with the algorithmic rule [$\mathbb{0}$-Alg], and thus the annotation $\mathbb{0}(\{\sigma\})$ is returned. Note

that there might be several solutions to the tallying instance, but we only need to return one of them: this is different from the rules for projections and applications, where all the solutions to the tallying instance are returned in order for the resulting type to be as small as possible. Instead, we only want here to justify that $\mathbb{0}$ is an instance of $\Gamma(\mathsf{x})$, in order to satisfy the side-condition of [$\mathbb{0}$-Alg].

If the tallying instance has no solution, then we try to apply the [Case$_1$] rule which checks whether the type-case can be typed using the algorithmic rule [$\in_1$-Alg]. It proceeds similarly, using tallying to test whether there exists a substitution σ such that $\Gamma(\mathsf{x})\sigma \leqslant \tau$. Finally, if this tallying instance has no solution or if $\mathsf{x}_1 \notin \mathsf{dom}(\Gamma)$, then it means that the type-case should be typed using the algorithmic rule [$\in_2$-Alg]. In this case, the rule [Case$_2$] applies, and we know it will succeed (otherwise the intermediate annotation would not be $\mathtt{typ}$).

$$[\textsc{Lambda}]\ \frac{\Gamma, x : \mathbf{u} \vdash_{\mathcal{S}} \langle \kappa \mid \mathcal{K} \rangle \Rightarrow \Bbbk}{\Gamma \vdash_{\mathcal{S}} \langle \lambda x.\kappa \mid \lambda(\mathbf{u}, \mathcal{K}) \rangle \Rightarrow \lambda(\mathbf{u}, \Bbbk)}$$

The rule for λ-abstraction is straightforward: it just proceeds recursively on its children annotation.

$$[\textsc{BindVar}]\ \frac{\rho = \mathsf{refresh}(\Gamma(\mathsf{x}))}{\Gamma \vdash_{\mathcal{S}} \langle \mathsf{x} \mid \mathtt{typ} \rangle : \rho}$$

$$[\textsc{BindSkip}]\ \frac{\Gamma \vdash_{\mathcal{S}} \langle \kappa \mid \mathcal{K} \rangle \Rightarrow \Bbbk}{\Gamma \vdash_{\mathcal{S}} \langle \mathtt{bind}\,\mathsf{x} = a\,\mathtt{in}\,\kappa \mid \mathtt{skip}\ \mathcal{K} \rangle \Rightarrow \mathtt{skip}\ \Bbbk}\ \mathsf{x} \notin \mathsf{dom}(\Gamma)$$

$$[\textsc{BindKeep}]\ \frac{\begin{array}{c}\Gamma \vdash_{\mathcal{S}} \langle a \mid \mathcal{A} \rangle \Rightarrow \mathbb{a} \qquad \Gamma \vdash_{\mathcal{A}} [a \mid \mathbb{a}] : s \\ (\forall i \in I)\ \ \Gamma, \mathsf{x} : s \wedge \mathbf{u}_i \vdash_{\mathcal{S}} \langle \kappa \mid \mathcal{K}_i \rangle \Rightarrow \Bbbk_i\end{array}}{\Gamma \vdash_{\mathcal{S}} \langle \mathtt{bind}\,\mathsf{x} = a\,\mathtt{in}\,\kappa \mid \mathtt{keep}\ (\mathcal{A}, \varnothing, \{(\mathbf{u}_i, \mathcal{K}_i)\}_{i \in I}) \rangle \Rightarrow \Bbbk}\ \{\mathbf{u}_i\}_{i \in I} \in \mathsf{Part}(\mathbb{1})$$

where $\Bbbk = \mathtt{keep}\ (\mathbb{a}, \{(\mathbf{u}_i, \Bbbk_i)\}_{i \in I})$.

The rule [BindKeep] takes as input an intermediate annotation $\mathtt{keep}\ (\mathcal{A}, \mathcal{S}, \mathcal{S}')$, with $\mathcal{S} = \varnothing$ since the intermediate annotation given as input should be in a terminal state (all branches should have been fully reconstructed by the main reconstruction algorithm). The rule recursively transforms the intermediate annotation $\mathcal{A}$ for the definition a into an annotation $\mathbb{a}$ for the algorithmic type system, and uses it to type a. It can then update the environment and proceed recursively on the body κ, for each branch in $\mathcal{S}'$.

$$[\textsc{Inter}]\ \frac{(\forall i \in I)\ \ \Gamma \vdash_{\mathcal{S}} \langle \eta \mid \mathcal{H}_i \rangle \Rightarrow \mathbb{h}_i}{\Gamma \vdash_{\mathcal{S}} \langle \eta \mid \bigwedge(\varnothing, \{\mathcal{H}_i\}_{i \in I}) \rangle \Rightarrow \bigwedge(\{\mathbb{h}_i\}_{i \in I})}\ I \neq \varnothing$$

Lastly, the rule [Inter] takes as input an annotation $\bigwedge(S_1, S_2)$ and recursively builds annotations for each branch in S_2. Similarly to the [BindKeep] rule, it

requires $S_1 = \varnothing$ because the intermediate annotation given as input should be in a terminal state.

This substitution inference system is sound, meaning that if it produces an annotation for the algorithmic type system, then this annotation is valid (it allows deriving a type for the corresponding atom or form):

Theorem 9 (Soundness)**.** *If* $\Gamma \vdash_{\mathcal{S}} \langle \kappa \mid \mathcal{K} \rangle \Rightarrow \Bbbk$*, then* $\exists t.\ \Gamma \vdash_{\mathcal{A}} [\kappa \mid \Bbbk] : t$*. If* $\Gamma \vdash_{\mathcal{S}} \langle a \mid \mathcal{A} \rangle \Rightarrow \mathbb{a}$*, then* $\exists t.\ \Gamma \vdash_{\mathcal{A}} [a \mid \mathbb{a}] : t$*.*

Proof. We proceed by structural induction on the derivation of $\Gamma \vdash_{\mathcal{S}} \langle \kappa \mid \mathcal{K} \rangle \Rightarrow \Bbbk$ or $\Gamma \vdash_{\mathcal{S}} \langle a \mid \mathcal{A} \rangle \Rightarrow \mathbb{a}$.

If the derivation $\Gamma \vdash_{\mathcal{S}} \langle a \mid \mathcal{A} \rangle \Rightarrow \mathbb{a}$ has an [App] root, we construct a derivation $\Gamma \vdash_{\mathcal{A}} [a \mid \mathbb{a}] : t$ (for some t) with a [→E-Alg] root. In order to satisfy the side-conditions of the rule [→E-Alg], we need to prove that the annotation $@(\Sigma_1, \Sigma_2)$ generated by the [App] root satisfies $\Gamma(\mathsf{x}_1)\Sigma_1 \leqslant \mathbb{0} \to \mathbb{1}$ and $\Gamma(\mathsf{x}_2)\Sigma_2 \leqslant \mathsf{dom}(\Gamma(\mathsf{x}_1)\Sigma_1)$.

We have, in the premise of the [App] root, $\Sigma = \mathsf{tally}(\{\Gamma(\mathsf{x}_1)\rho_1 \mathrel{\dot{\leqslant}} \Gamma(\mathsf{x}_2)\rho_2 \to \alpha\})$ and $\Sigma \neq \varnothing$. Let $\sigma \in \Sigma$. By definition of the tallying problem, we have $(\Gamma(\mathsf{x}_1)\rho_1)\sigma \leqslant (\Gamma(\mathsf{x}_2)\rho_2 \to \alpha)\sigma$, which can be rewritten $\Gamma(\mathsf{x}_1)(\sigma \circ \rho_1) \leqslant (\Gamma(\mathsf{x}_2)(\sigma \circ \rho_2)) \to (\alpha\sigma)$. From that subtyping relation, we can deduce $\Gamma(\mathsf{x}_1)(\sigma \circ \rho_1) \leqslant \mathbb{0} \to \mathbb{1}$, and by definition of $\mathsf{dom}(.)$, $\Gamma(\mathsf{x}_2)(\sigma \circ \rho_2) \leqslant \mathsf{dom}(\Gamma(\mathsf{x}_1)(\sigma \circ \rho_1))$. As $(\sigma \circ \rho_2) \in \Sigma_2$ and $(\sigma \circ \rho_1) \in \Sigma_1$, we deduce $\Gamma(\mathsf{x}_1)\Sigma_1 \leqslant \mathbb{0} \to \mathbb{1}$ and $\Gamma(\mathsf{x}_2)\Sigma_2 \leqslant \mathsf{dom}(\Gamma(\mathsf{x}_1)\Sigma_1)$ (we use the fact that $\mathsf{dom}(.)$ is monotonically non-increasing, cf. Definition 10).

The other cases are similar or straightforward. □

6.4 Backpropagation of splits

The split backpropagation system defined in this section deals with the following problem: given an environment Γ, an atom a and a type t, how can Γ be constrained so that a has type t?

This system is used in the main reconstruction system, by the [BindKeep$_2$] rule, in order to propagate type decompositions made by bindings. It produces judgments of the form $\Gamma \vdash_{\mathcal{B}} (a : \mathbf{u}) \Rightarrow \mathbb{\Gamma}$, where $\mathbb{\Gamma}$ is a set of type environments containing only monomorphic types, and such that intersecting Γ with any type environment in $\mathbb{\Gamma}$ makes the type $\mathbf{u}$ derivable for a. Note that all the deduction rules below are axioms: they do not make any recursive call.

$$[\textsc{Const}_1]\ \frac{\boldsymbol{b}_c \leqslant \mathbf{u}}{\Gamma \vdash_{\mathcal{B}} (c : \mathbf{u}) \Rightarrow \{\varnothing\}} \qquad [\textsc{Const}_2]\ \frac{}{\Gamma \vdash_{\mathcal{B}} (c : \mathbf{u}) \Rightarrow \{\}}$$

The case of a constant c is straightforward, as the type of a constant does not depend on the type environment. If c has type $\mathbf{u}$, then $\{\varnothing\}$ is returned (with $\varnothing$

being the empty environment): this captures the fact that no additional assumption is required for c to have type $\mathbf{u}$. Otherwise, c cannot be typed with $\mathbf{u}$, and thus $\{\}$ is returned, meaning that there is no refinement of Γ that can make c have the type $\mathbf{u}$.

$$[\textsc{Var}_1]\ \frac{\Gamma(x) \leqslant \mathbf{u}}{\Gamma \vdash_{\mathcal{B}} (x : \mathbf{u}) \Rightarrow \{\varnothing\}} \qquad [\textsc{Var}_2]\ \frac{}{\Gamma \vdash_{\mathcal{B}} (x : \mathbf{u}) \Rightarrow \{\}}$$

The rules for lambda variables are quite similar to the rules for constants, which might seem surprising. Indeed, when $\Gamma(x) \not\leqslant \mathbf{u}$, we could nonetheless make x have the type $\mathbf{u}$ by adding the hypothesis $x : \mathbf{u}$ to our type environment. Thus, we could expect the rule $[\textsc{Var}_2]$ to return $\{\{(x : \mathbf{u})\}\}$ instead of $\{\}$. However, the type environments $\Gamma' \in \mathbb{\Gamma}$ generated by a $\Gamma \vdash_{\mathcal{B}} (a : \mathbf{u}) \Rightarrow \mathbb{\Gamma}$ judgment should only contain binding variables (i.e., $\mathsf{dom}(\Gamma') \subseteq \mathsf{Vars_B}$). Indeed, as explained in Section 6.2, $\vdash_{\mathcal{B}}$ judgments are used to propagate type decompositions between binding variables. Lambda variables are not involved in this mechanism: a type decomposition does not directly occur on a lambda variable x, but instead on the associated binding variable $\mathsf{bind\,x = }x\,\mathsf{in}\,\kappa$.

$$[\textsc{Pair}]\ \frac{\mathbf{u} \overset{\text{DNF}}{\simeq} \left(\bigvee_{i\in I}(\mathbf{u}_i \times \mathbf{v}_i)\right) \vee \ldots}{\Gamma \vdash_{\mathcal{B}} ((\mathsf{x}_1, \mathsf{x}_2) : \mathbf{u}) \Rightarrow \{\{\mathsf{x}_1 : \mathbf{u}_i\} \wedge \{\mathsf{x}_2 : \mathbf{v}_i\} \mid i \in I\}}$$

For pairs, we decompose the type $\mathbf{u}$ into a union of atomic products $\bigvee_{i\in I}(\mathbf{u}_i \times \mathbf{v}_i)$ and other atoms. For that, we proceed as follows:

1. Using the DNF decomposition defined in Section 2.5, we can decompose $\mathbf{u}$ into a union $\bigvee_{i\in I}\left(\bigwedge_{a\in P_i} a \wedge \bigwedge_{a\in N_i} \neg a\right)$ with a ranging over atoms $\mathcal{A}$.
2. Summands that contain atoms that are not products are ignored (in the rule above, they are captured by the $\ldots$ notation).
3. In every remaining summand, negative atomic products can be transformed into a union of positive atomic products ($\neg(t \times s) \simeq (\neg t \times \mathbb{1}) \vee (\mathbb{1} \times \neg s)$), and using distributivity we can push these unions at top-level, yielding a decomposition $\bigvee_{i\in I}\left(\bigwedge_{a\in P_i} a\right)$.
4. Every intersection $\bigwedge_{a\in P_i} a$ can be transformed into an atomic product type a_i ($(t_1 \times s_1) \wedge (t_2 \times s_2) \simeq t_1 \wedge t_2 \times s_1 \wedge s_2$), yielding a decomposition $\bigvee_{i\in I} a_i$.

Note that this decomposition is not unique: for instance, the type $(\mathsf{Bool} \times \mathsf{True}) \vee (\mathsf{True} \times \mathsf{Bool})$ is equivalent to $(\mathsf{False} \times \mathsf{True}) \vee (\mathsf{True} \times \mathsf{Bool})$ and to $(\mathsf{Bool} \times \mathsf{True}) \vee (\mathsf{True} \times \mathsf{False})$. Also note that we may have $I = \varnothing$ (in particular, if $\mathbf{u}$ is a subtype of $\neg(\mathbb{1} \times \mathbb{1})$).

$$[\textsc{Proj}_1]\ \frac{}{\Gamma \vdash_{\mathcal{B}} (\pi_1 \mathsf{x} : \mathbf{u}) \Rightarrow \{\{\mathsf{x} : \mathbf{u} \times \mathbb{1}\}\}} \qquad [\textsc{Proj}_2]\ \frac{}{\Gamma \vdash_{\mathcal{B}} (\pi_2 \mathsf{x} : \mathbf{u}) \Rightarrow \{\{\mathsf{x} : \mathbb{1} \times \mathbf{u}\}\}}$$

The rules for projections are straightforward.

For an application $\mathtt{x}_1\mathtt{x}_2$, we have the choice between trying to find a refinement for the type of $\mathtt{x}_1$, or trying to find a refinement for the type of $\mathtt{x}_2$ (or both at the same time). For instance, let us consider the example of Section 6.2, where we had a binding `bind y = f x in ...` with initially $\mathtt{f} : \alpha \to \alpha$ and $\mathtt{x} : \mathtt{Bool}$. After splitting the type of `y` into `True` and $\neg\mathtt{True}$, a $\vdash_{\mathcal{B}}$ judgment is used to propagate this split to `x`. In order for the application `f x` to have the type `True`, we could either require $\mathtt{x}_2 : \mathtt{True}$, or $\mathtt{x}_1 : (\alpha \to \alpha) \wedge (\mathtt{Bool} \to \mathtt{True})$.

In order to determine which of these two refinements is better, we should determine which one yields a better type decomposition. While decomposing the type of $\mathtt{x}_2$ (initially `Bool`) into `True` and `Bool\True` is useful, as seen in the example in Section 6.2, decomposing $\alpha \to \alpha$ into $(\alpha \to \alpha) \wedge (\mathtt{Bool} \to \mathtt{True})$ and $(\alpha \to \alpha)\backslash(\mathtt{Bool} \to \mathtt{True})$ is not really exploitable, as the second split $((\alpha \to \alpha)\backslash(\mathtt{Bool} \to \mathtt{True}))$ only differs with the initial type $\alpha \to \alpha$ by a negative arrow. Indeed, typing an expression with the hypothesis $\mathtt{f} : \alpha \to \alpha$ or with the hypothesis $\mathtt{f} : (\alpha \to \alpha)\backslash(\mathtt{Bool} \to \mathtt{True})$ barely makes any difference, as negative arrows have no impact on the typing (except on very specific examples). More generally, decomposing a functional type is most of the time useless, except when this functional type is a disjunction of arrows. For instance, we may want to split a functional type $(\mathbb{1} \to \mathtt{True}) \vee (\mathbb{1} \to \mathtt{False})$ into $\mathbb{1} \to \mathtt{True}$ and $\mathbb{1} \to \mathtt{False}$.

Our rule for applications follows from this reasoning, by decomposing the type of the function according to the summands of its DNF, and then trying to find a type for the argument that would make the application to have type $\mathbf{u}$:

$$[\textsc{App}]\ \frac{\Gamma(\mathtt{x}_1) \overset{\text{DNF}}{\cong} \bigvee_{i\in I} t_i \qquad \forall i \in I.\ \{\sigma_j\}_{j\in J_i} = \mathsf{tally}(\{t_i \dot{\leqslant} \alpha \to \mathbf{u}\})}{\Gamma \vdash_{\mathcal{B}} (\mathtt{x}_1\mathtt{x}_2 : \mathbf{u}) \Rightarrow \bigcup_{i\in I} \mathbb{\Gamma}_i}\ \alpha \in \mathcal{V}_P \text{ fresh}$$

where, for every $i \in I$, $\mathbb{\Gamma}_i = \{\{\mathtt{x}_1 : (t_i\sigma_j)\sigma'_j,\ \mathtt{x}_2 : (\alpha\sigma_j)\sigma'_j\} \mid j \in J_i\}$ with σ'_j a type substitution mapping each polymorphic type variable β appearing in $t_i\sigma_j$ or $\alpha\sigma_j$ to either:

- $\mathbb{1}$ if β only appears in covariant positions in $\alpha\sigma_j$,
- $\mathbb{0}$ if β only appears in contravariant positions in $\alpha\sigma_j$,
- a fresh monomorphic type variable otherwise.

In order to find some types for $\mathtt{x}_2$ that make the type $\mathbf{u}$ derivable for the application, we rely on tallying: it allows us to find all the substitutions σ such that $t_i\sigma \leqslant (\alpha \to \mathbf{u})\sigma$ (with t_i the type of the function, and α a fresh polymorphic type variable representing the type of the argument), which implies $(t_i\sigma) \circ (\alpha\sigma) \leqslant \mathbf{u}$, thus giving us the guarantee that the type $\mathbf{u}$ is derivable for $\mathtt{x}_1\mathtt{x}_2$ when $\mathtt{x}_2 : \alpha\sigma$ and $\mathtt{x}_1 : t_i\sigma$.

The type environments returned by $\vdash_{\mathcal{B}}$ judgments should not contain any polymorphic type variable, because those type environments are then used by the main reconstruction algorithm in order to refine the type decompositions made by bindings (and those decompositions cannot feature any polymorphic type variable). This

is the reason why the substitution σ'_j is applied on the resulting types, transforming any polymorphic type variable into a monomorphic one (or alternatively, into $\mathbb{1}$ or $\mathbb{0}$ if it weakens the constraint on $\mathtt{x}_2$).

$$[\textsc{Case}]\ \frac{}{\Gamma \vdash_{\mathcal{B}} ((\mathtt{x}\in\tau)\,?\,\mathtt{x}_1 : \mathtt{x}_2 : \mathbf{u}) \Rightarrow \{\{\mathtt{x} : \tau,\ \mathtt{x}_1 : \mathbf{u}\}, \{\mathtt{x} : \neg\tau,\ \mathtt{x}_2 : \mathbf{u}\}\}}$$

The rule for type-cases is straightforward: for a type-case $(\mathtt{x}\in\tau)\,?\,\mathtt{x}_1 : \mathtt{x}_2$ to have type $\mathbf{u}$, either the tested binding variable $\mathtt{x}$ must have type τ and the first branch must have type $\mathbf{u}$, or the tested binding variable $\mathtt{x}$ must have type $\neg\tau$ and the second branch must have type $\mathbf{u}$. Note that there is a third possibility for the type-case to have type $\mathbf{u}$: it is for the tested expression $\mathtt{x}$ to have type $\mathbb{0}$. However, this case is not interesting in our setting, as it does not yield any interesting type decomposition.

$$[\textsc{Lambda}]\ \frac{}{\Gamma \vdash_{\mathcal{B}} (\lambda x.\ \kappa : \mathbf{u}) \Rightarrow \{\}}$$

Finally, in the case of a λ-abstraction, we return $\{\}$, meaning that no assumption on the type environment can make the λ-abstraction to have type $\mathbf{u}$. Note that this is an approximation: there might exist some assumptions on the free variables of the λ-abstraction that would make it have the type $\mathbf{u}$. For instance, for an atom $a \stackrel{\text{def}}{=} \lambda x.\ \mathtt{y}$ and a type $\mathbf{u} \stackrel{\text{def}}{=} \mathbb{1} \to \mathtt{Int}$, a solution would be $\{(\mathtt{y} : \mathtt{Int})\}$. However, this scenario is very rare, as a type-case cannot directly generate a decomposition of a function type (test types τ cannot contain arrows other than $\mathbb{0} \to \mathbb{1}$), and as the type of the function is not decomposed when backpropagating a split over an application (cf. Rule [App] above). Additionally, backpropagating a split over a λ-abstraction would be quite complex as it would require entering the body of the λ-abstraction, and thus defining the $\vdash_{\mathcal{B}}$ judgment on canonical forms as well.

6.5 Discussion about the reconstruction algorithm

6.5.1 Termination

The deduction rules presented in this chapter define a terminating algorithm for reconstructing annotations: first, the main reconstruction system $\vdash_{\mathcal{R}}$ is used to infer an intermediate annotation, and then, if it succeeds, the substitution inference system $\vdash_{\mathcal{S}}$ converts this intermediate annotation into an annotation for the algorithmic type system.

In this section, we propose a sketch of proof justifying that the deduction rules for the reconstruction system define a terminating algorithm. The idea of this proof is similar to the proof of termination of the Kirby-Paris hydra game (Kirby and Paris, 1982): we can associate an ordinal number weight to each node, and this weight can only decrease as the game (or derivation) advances. Intuitively, this non-negative weight represents the advancement of the game (or derivation). Though subtrees

can sometimes be duplicated, their weight is always lowered before being duplicated, resulting in a lower weight overall.

For every type environment Γ, canonical form or atom η, and intermediate annotation $\mathcal{H}$, the weight $\mathsf{w}(\Gamma, \eta, \mathcal{H})$ is the ordinal number defined as follows:

$$\begin{aligned}
\mathsf{w}(\Gamma, \eta, \mathsf{typ}) &= 1 \\
\mathsf{w}(\Gamma, \eta, \mathsf{untyp}) &= 1 \\
\mathsf{w}(\Gamma, \eta, \textstyle\bigwedge(S_1, S_2)) &= \textstyle\sum\{\mathsf{w}(\Gamma, \eta, \mathcal{H}) \mid \mathcal{H} \in S_1\}
\end{aligned}$$

$$\begin{aligned}
\mathsf{w}(\Gamma, c, \mathsf{infer}) &= \omega & \mathsf{w}(\Gamma, \lambda x.\kappa, \mathsf{infer}) &= \omega^{\mathsf{w}(\Gamma,\kappa,\mathsf{infer})} \\
\mathsf{w}(\Gamma, x, \mathsf{infer}) &= \omega & \mathsf{w}(\Gamma, \lambda x.\kappa, \lambda(\mathbf{u}, \mathcal{K})) &= \omega^{\mathsf{w}(\Gamma,\kappa,\mathcal{K})}
\end{aligned}$$

$$\begin{aligned}
\mathsf{w}(\Gamma, \pi_i \mathsf{x}, \mathsf{infer}) &= \omega && \text{if } \mathsf{x} \in \mathsf{dom}(\Gamma) \\
\mathsf{w}(\Gamma, \pi_i \mathsf{x}, \mathsf{infer}) &= \omega^2 && \text{otherwise}
\end{aligned}$$

$$\begin{aligned}
\mathsf{w}(\Gamma, \mathsf{x}_1\mathsf{x}_2, \mathsf{infer}) &= \omega && \text{if } \{\mathsf{x}_1, \mathsf{x}_2\} \subseteq \mathsf{dom}(\Gamma) \\
\mathsf{w}(\Gamma, \mathsf{x}_1\mathsf{x}_2, \mathsf{infer}) &= \omega^2 && \text{otherwise, if } \mathsf{x}_1 \in \mathsf{dom}(\Gamma) \\
\mathsf{w}(\Gamma, \mathsf{x}_1\mathsf{x}_2, \mathsf{infer}) &= \omega^2 && \text{otherwise, if } \mathsf{x}_2 \in \mathsf{dom}(\Gamma) \\
\mathsf{w}(\Gamma, \mathsf{x}_1\mathsf{x}_2, \mathsf{infer}) &= \omega^3 && \text{otherwise}
\end{aligned}$$

$$\begin{aligned}
\mathsf{w}(\Gamma, (\mathsf{x}_1, \mathsf{x}_2), \mathsf{infer}) &= \omega && \text{if } \{\mathsf{x}_1, \mathsf{x}_2\} \subseteq \mathsf{dom}(\Gamma) \\
\mathsf{w}(\Gamma, (\mathsf{x}_1, \mathsf{x}_2), \mathsf{infer}) &= \omega^2 && \text{otherwise, if } \mathsf{x}_1 \in \mathsf{dom}(\Gamma) \\
\mathsf{w}(\Gamma, (\mathsf{x}_1, \mathsf{x}_2), \mathsf{infer}) &= \omega^2 && \text{otherwise, if } \mathsf{x}_2 \in \mathsf{dom}(\Gamma) \\
\mathsf{w}(\Gamma, (\mathsf{x}_1, \mathsf{x}_2), \mathsf{infer}) &= \omega^3 && \text{otherwise}
\end{aligned}$$

$$\begin{aligned}
\mathsf{w}(\Gamma, (\mathsf{x}_0 \in \tau)\,?\,\mathsf{x}_1 : \mathsf{x}_2, \in_i) &= \omega \\
\mathsf{w}(\Gamma, (\mathsf{x}_0 \in \tau)\,?\,\mathsf{x}_1 : \mathsf{x}_2, \mathsf{infer}) &= \omega^2 && \text{if } \Gamma(\mathsf{x}_0) \leqslant \tau \\
\mathsf{w}(\Gamma, (\mathsf{x}_0 \in \tau)\,?\,\mathsf{x}_1 : \mathsf{x}_2, \mathsf{infer}) &= \omega^2 && \text{otherwise, if } \Gamma(\mathsf{x}_0) \leqslant \neg\tau \\
\mathsf{w}(\Gamma, (\mathsf{x}_0 \in \tau)\,?\,\mathsf{x}_1 : \mathsf{x}_2, \mathsf{infer}) &= \omega^3 && \text{otherwise, if } \mathsf{x}_0 \in \mathsf{dom}(\Gamma) \\
\mathsf{w}(\Gamma, (\mathsf{x}_0 \in \tau)\,?\,\mathsf{x}_1 : \mathsf{x}_2, \mathsf{infer}) &= \omega^4 && \text{otherwise}
\end{aligned}$$

$$\begin{aligned}
\mathsf{w}(\Gamma, \mathsf{bind}\,\mathsf{x} = a\,\mathsf{in}\,\kappa\,, \mathsf{skip}\ \mathcal{K}) &= \omega^{\mathsf{w}(\Gamma,\kappa,\mathcal{K})} \\
\mathsf{w}(\Gamma, \mathsf{bind}\,\mathsf{x} = a\,\mathsf{in}\,\kappa\,, \mathsf{keep}\ (\mathcal{A}, \mathcal{S}_1, \mathcal{S}_2)) &= \omega^{\alpha} \\
&\quad \text{with } \alpha = \textstyle\sum\{\mathsf{w}((\Gamma, \mathsf{x} : \mathbf{u}), \kappa, \mathcal{K}) \mid (\mathbf{u}, \mathcal{K}) \in \mathcal{S}_1\} \\
\mathsf{w}(\Gamma, \mathsf{bind}\,\mathsf{x} = a\,\mathsf{in}\,\kappa\,, \mathsf{propagate}\ (\mathcal{A}, \mathbb{\Gamma}, \mathcal{S}_1, \mathcal{S}_2)) &= \omega^{\alpha + |\mathbb{\Gamma}|} \\
&\quad \text{with } \alpha = \textstyle\sum\{\mathsf{w}((\Gamma, \mathsf{x} : \mathbf{u}), \kappa, \mathcal{K}) \mid (\mathbf{u}, \mathcal{K}) \in \mathcal{S}_1\} \\
\mathsf{w}(\Gamma, \mathsf{bind}\,\mathsf{x} = a\,\mathsf{in}\,\kappa\,, \mathsf{try\text{-}keep}\ (\mathcal{A}, \mathcal{K}_1, \mathcal{K}_2)) &= \omega^{\alpha} \\
&\quad \text{with } \alpha = \textstyle\sum\{\mathsf{w}(\Gamma, a, \mathcal{A}),\ \mathsf{w}((\Gamma, \mathsf{x} : \mathbb{1}), \kappa, \mathcal{K}_1),\ \mathsf{w}(\Gamma, \kappa, \mathcal{K}_2)\} \\
\mathsf{w}(\Gamma, \mathsf{bind}\,\mathsf{x} = a\,\mathsf{in}\,\kappa\,, \mathsf{try\text{-}skip}\ (\mathcal{K})) &= \omega^{\alpha} \\
&\quad \text{with } \alpha = \textstyle\sum\{\mathsf{w}(\Gamma, a, \mathsf{infer}),\ \mathsf{w}(\Gamma, \kappa, \mathcal{K})\}
\end{aligned}$$

$$\begin{aligned} \mathtt{w}(\Gamma, \mathtt{x}, \mathtt{infer}) &= \omega && \text{if } \mathtt{x} \in \mathsf{dom}(\Gamma) \\ \mathtt{w}(\Gamma, \mathtt{x}, \mathtt{infer}) &= \omega^2 && \text{otherwise} \end{aligned}$$

where, for every multiset $\{\alpha_1, \alpha_2, \ldots, \alpha_n\}$, $\sum\{\alpha_1, \alpha_2, \ldots, \alpha_n\} \stackrel{\text{def}}{=} \alpha_1 + \alpha_2 + \cdots + \alpha_n$ with $\alpha_1 \geqslant \alpha_2 \geqslant \cdots \geqslant \alpha_n$.

Then, we define a weight $\mathtt{w}(\Gamma, \eta, \mathbb{R})$ for every type environment Γ, canonical form or atom η, and result $\mathbb{R}$:

$$\begin{aligned} \mathtt{w}(\Gamma, \eta, \mathtt{Ok}(\mathcal{H})) &= 1 \\ \mathtt{w}(\Gamma, \eta, \mathtt{Fail}) &= 1 \\ \mathtt{w}(\Gamma, \eta, \mathtt{Split}(\Gamma', \mathcal{H}_1, \mathcal{H}_2)) &= \\ &\sum \{\mathtt{w}(\Gamma \wedge \Gamma', \eta, \mathcal{H}_1)\} \cup \{\mathtt{w}((\Gamma \wedge \{\mathtt{x} : \neg\mathbf{u}\}), \eta, \mathcal{H}_2) \mid (\mathtt{x} : \mathbf{u}) \in \Gamma'\} \\ \mathtt{w}(\Gamma, \eta, \mathtt{Subst}(\{\psi_i\}_{i\in I}, \mathcal{H}_1, \mathcal{H}_2)) &= \sum \{\mathtt{w}(\Gamma, \eta, \mathcal{H}_2)\} \cup \{\mathtt{w}(\Gamma\psi_i, \eta, \mathcal{H}_1\psi_i) \mid i \in I\} \\ \mathtt{w}(\Gamma, \eta, \mathtt{Var}\ (\mathtt{x}, \mathcal{H}_1, \mathcal{H}_2)) &= \sum\{\mathtt{w}((\Gamma, \mathtt{x} : \mathbb{1}), \eta, \mathcal{H}_1),\ \mathtt{w}(\Gamma, \eta, \mathcal{H}_2)\} \end{aligned}$$

where

$$\begin{aligned} \Gamma_1 \wedge \Gamma_2 \stackrel{\text{def}}{=} &\left(\Gamma_1|_{\mathsf{dom}(\Gamma_1)\setminus\mathsf{dom}(\Gamma_2)}\right) \cup \left(\Gamma_2|_{\mathsf{dom}(\Gamma_2)\setminus\mathsf{dom}(\Gamma_1)}\right) \\ &\cup \{(\boldsymbol{x} : \Gamma_1(\boldsymbol{x}) \wedge \Gamma_2(\boldsymbol{x})) \mid \boldsymbol{x} \in \mathsf{dom}(\Gamma_1) \cap \mathsf{dom}(\Gamma_2)\} \end{aligned}$$

Lemma 36. *For every $\Gamma, \eta, \mathcal{H}$, and Γ' such that $\Gamma' \leqslant \Gamma$, we have $\mathtt{w}(\Gamma', \eta, \mathcal{H}) \leqslant \mathtt{w}(\Gamma, \eta, \mathcal{H})$.*

Proof. Structural induction on η and $\mathcal{H}$. □

Lemma 37. *For every $\Gamma, \eta, \mathcal{H}$, and ψ, we have $\mathtt{w}(\Gamma\psi, \eta, \mathcal{H}\psi) \leqslant \mathtt{w}(\Gamma, \eta, \mathcal{H})$.*

Proof. Structural induction on η and $\mathcal{H}$. For type-cases, we recall that test types τ do not contain type variables, and thus if $\Gamma(\mathtt{x}_0) \leqslant \tau$, then $\Gamma(\mathtt{x}_0)\psi \leqslant \tau\psi \simeq \tau$. □

Lemma 38. *If $\Gamma \vdash_{\mathcal{R}} \langle \eta \mid \mathcal{H} \rangle \Rightarrow \mathbb{R}$ or $\Gamma \vdash^*_{\mathcal{R}} \langle \eta \mid \mathcal{H} \rangle \Rightarrow \mathbb{R}$, then $\mathtt{w}(\Gamma, \eta, \mathcal{H}) \gneqq \mathtt{w}(\Gamma, \eta, \mathbb{R})$.*

Proof. Structural induction on the derivation of $\Gamma \vdash_{\mathcal{R}} \langle \eta \mid \mathcal{H} \rangle \Rightarrow \mathbb{R}$ or $\Gamma \vdash^*_{\mathcal{R}} \langle \eta \mid \mathcal{H} \rangle \Rightarrow \mathbb{R}$. □

Theorem 10 (Termination)**.** *The deduction rules $\vdash^*_{\mathcal{R}}$ and $\vdash_{\mathcal{R}}$ define a terminating algorithm: it can either fail (if no rule applies at some point) or return a result $\mathbb{R}$.*

Proof. There can only be finitely many [ITERATE$_1$] and [ITERATE$_2$] nodes applied on a given canonical form or atom, otherwise, according to the previous lemmas, we could extract from them an infinite decreasing chain of ordinal numbers. □

6.5.2 Incompleteness

Although terminating, the reconstruction algorithm is not complete: it may fail to find an annotation for a MSC form even if there exists such an annotation that would make the MSC form typeable with the algorithmic type system.

Non-principality The incompleteness of the reconstruction algorithm is inherent to our system and derives from the lack of type principality: for a type environment Γ and an expression e that is typeable under Γ, there does not necessarily exist a *principal* type t such that $\Gamma \vdash e : t$ and $\forall t'.\, \Gamma \vdash e : t' \Rightarrow t \leq t'$. For instance, consider the expression $(f\ 42, f\ 42)$ with $f : \mathbb{1} \to \mathbb{1}$ (our reconstruction algorithm infers the type $\mathbb{1} \times \mathbb{1}$ for this expression). For any type t, using the union-elimination rule, this expression can be typed $t \times t \vee \neg t \times \neg t$. Consequently, a principal type for this expression must be smaller than $t \times t \vee \neg t \times \neg t$ for every type t, thus capturing the fact that the left and right component of the pair are ultimately the same value. Unfortunately, this constraint cannot be expressed with our types, as we would need an infinite intersection.

Another more common example showing the lack of principal typing is the function `map`, which will be detailed in Chapter 9. Briefly, our type system can check that, for a fixed n, applying `map` on a list of length n yields a list of the same length n. However, no type can capture this property for every n: again, we would need an infinite intersection (or dependent types).

In other words, incompleteness stems from the fact that the declarative system can use all the infinitely many decompositions of unions in the union-elimination rule, and the infinitely many decompositions of the domain of a function when reconstructing its type as an intersection of arrows. The algorithmic counterpart of this, is that there are infinitely many annotations that the algorithmic system can use to type these expressions and that these infinite choices cannot be summarized by a notion of principal annotation: the reconstruction chooses one particular annotation, and therefore it will miss some solutions.

Type expansion There is a second source of incompleteness in the reconstruction algorithm: it does not perform the so-called "expansion" of intersection types.

Sometimes, typing an application may require to perform an *expansion* of the types. For instance, consider the application $g\ f$, where f is the polymorphic identity function of type $\alpha \to \alpha$, and g is a function of type $(\texttt{Int} \to \texttt{Int}) \wedge (\texttt{Bool} \to \texttt{Bool}) \to \mathbb{1}$ (we recall that $\wedge$ has precedence over $\to$). The tallying instance for the constraints $C = \{((\texttt{Int} \to \texttt{Int}) \wedge (\texttt{Bool} \to \texttt{Bool}) \to \mathbb{1},\ (\alpha \to \alpha) \to \beta)\}$ has no solution, because no type substitution can make $\alpha \to \alpha$ to become a subtype of $(\texttt{Int} \to \texttt{Int}) \wedge (\texttt{Bool} \to \texttt{Bool})$. In order to type this application, we need to expand the

type of f into $(\alpha_1 \to \alpha_1) \wedge (\alpha_2 \to \alpha_2)$. Unfortunately, we have no way, in general, to know how many times a type should be expanded.

In the rule [App] in Section 6.3, tallying is applied without expanding the types in the constraint, which is a source of incompleteness. In some related work, such as Castagna et al. (2015), expansion is automatically performed when tallying fails to type an application: if the tallying instance $\mathsf{tally}(\{t_1\rho_1 \mathrel{\dot{\leqslant}} t_2\rho_2 \to \alpha\})$ fails, then the type t_1 of the function is expanded, yielding the new tallying instance $\mathsf{tally}(\{t_1\rho_1 \wedge t_1\rho_3 \mathrel{\dot{\leqslant}} t_2\rho_2 \to \alpha\})$, and so on and so forth by alternating expansions on the function and on the argument types (see (Castagna et al., 2015, Section 3.2.3) for more details). However, this heuristics may not terminate (unless we fix a maximal number of expansions) and is still incomplete: it does not handle the case where the tallying instance has a solution but where performing an expansion may yield a strictly more precise type for the application.

Despite incompleteness, the declarative rules of Figure 4.1 form a reliable guide to which programs are accepted, provided we bear in mind that the reconstruction algorithm approximates data structures according to the tests performed on them. So, typically, the type reconstructed for a function on lists will probably differentiate the cases for empty and not-empty lists, but not for, say, lists of size 42, unless the function contains an explicit test for it. This (and to a lesser extent, expansion) is essentially the main difference with the declarative system, which has the liberty to deduce the type for the case of lists of size 42, even if this property is not tested in the body of the function.

We will see in Chapter 7 how explicit type annotations can be added to the language, allowing to guide the reconstruction in the cases where it fails to find a suitable type.

Part III

Towards a Practical Language

Chapter 7

Extensions

Contents

This chapter focuses on adding some new constructions to our language, and extending the type system accordingly.

The first extension adds to the language record types and record expressions, together with primitive operations to manipulate their values. Then we provide the user with a way to guide the type-inference algorithm by means of user type annotations for function parameters. We then show how to add local let-bindings to the language as well as extended type-case constructs. While interesting in their own rights these last two extensions are building blocks of a more powerful construct, namely pattern-matching.

7.1 Records

Many languages have extensible records as a built-in data structure. In particular, in languages such as JavaScript, the fundamental object type is implemented as an extensible record. More generally, object types from oriented-object languages can be encoded as record types mapping some labels to functional types (for methods) or data types (for attributes). When encoded in this way, the semantic subtyping relation over record types yields a structural subtyping relation over classes.

Record expressions are sometimes an afterthought in the definition of a calculus, as their semantics can typically be reduced to that of pairs. Extending their operations beyond projection to include field update and deletion, however, warrants additional attention.

Syntax and semantics

The empty record constant is added to the source language, as well as some operations: record update, field deletion, and field projection. Record values consist of

the empty records and the record update expressions whose constituent expressions are themselves values. This is formalized in Figure 7.1.

Syntax

$$
\begin{array}{llll}
\textbf{Expression} & e & ::= & c \mid x \mid \lambda x.e \mid e\ e \mid (e,e) \mid \pi_i e \mid \texttt{(}e \in \tau\texttt{)?}\, e : e \\
& & & \mid \texttt{\{\}} \mid \texttt{\{}e\ \texttt{with}\ \ell = e\texttt{\}} \mid e\backslash\ell \mid e.\ell \\
\textbf{Value} & v & ::= & c \mid \lambda x.e \mid (v,v) \mid \texttt{\{\}} \mid \texttt{\{}v\ \texttt{with}\ \ell = v\texttt{\}}
\end{array}
$$

Additional reduction rules

$$
\begin{array}{rl}
\texttt{\{}v'\ \texttt{with}\ \ell = v\texttt{\}}.\ell & \rightsquigarrow v \\
\texttt{\{\}}\backslash\ell & \rightsquigarrow \texttt{\{\}}
\end{array}
\qquad
\begin{array}{rll}
\texttt{\{}v'\ \texttt{with}\ \ell' = v\texttt{\}}.\ell & \rightsquigarrow v'.\ell & \text{if } \ell' \not\equiv \ell \\
\texttt{\{}v'\ \texttt{with}\ \ell = v\texttt{\}}\backslash\ell & \rightsquigarrow v'\backslash\ell & \\
\texttt{\{}v'\ \texttt{with}\ \ell' = v\texttt{\}}\backslash\ell & \rightsquigarrow \texttt{\{}v'\backslash\ell\ \texttt{with}\ \ell' = v\texttt{\}} & \text{if } \ell' \not\equiv \ell
\end{array}
$$

Evaluation Context

$$
\begin{array}{rcl}
E & ::= & [\,] \mid v\ E \mid E\ e \mid (v,E) \mid (E,e) \mid \pi_i E \mid \texttt{(}E \in \tau\texttt{)?}\, e : e \\
& & \mid \texttt{\{}E\ \texttt{with}\ \ell = e\texttt{\}} \mid \texttt{\{}v\ \texttt{with}\ \ell = E\texttt{\}} \mid E\backslash\ell \mid E.\ell
\end{array}
$$

Figure 7.1: Syntax and semantics of the language with records

Reduction of record expressions is straightforward. It is worth noting that in this representation, multiple identical labels may exist in a record expression, but this is equivalent to limiting to one label, as projection reduces to the last-applied field, and deletion removes all instances of a label from the record. Evaluation of the expressions is performed left-to-right, as in the rest of the language.

To reduce verbosity, we use syntactic sugar for nonempty records. Assuming all the labels are distinct, $\texttt{\{\{\{\{\}}\ \texttt{with}\ \ell_1 = e_1\texttt{\}}\ \texttt{with}\ \ell_2 = e_2\texttt{\}}\ldots\ \texttt{with}\ \ell_n = e_n\texttt{\}}$ is represented by $\texttt{\{}\ell_1 = e_1, \ell_2 = e_2, \ldots, \ell_n = e_n\texttt{\}}$.

Record types

In the syntax for record types, we distinguish between two kinds of record types. An open record type, denoted by $\{\ldots\ \texttt{..}\}$, is the type of records whose labels *include* those explicitly written. A closed record type, denoted by $\{\ldots\}$, is the type of records whose labels are *exactly* those explicitly written.

$$
\begin{array}{llll}
\textbf{Types} & t & ::= & \cdots \mid \{f, \ldots, f\} \mid \{f, \ldots, f\ \texttt{..}\} \\
\textbf{Fields} & f & ::= & \ell = t \mid \ell =? t
\end{array}
$$

The semantics of record types we use is the one by Frisch (2004), where record values are *quasi constant functions*, that is, functions that map labels into values of our interpretation domain $\mathcal{D}$ (cf. Definition 6) and are constant apart from on a finite number of labels. A value of a closed record type $\{\ell_1 = t_1, \ell_2 = t_2\}$ maps ℓ_1 into a value of type t_1, ℓ_2 into a value of type t_2 and *all other labels* into the constant `undef` meaning that the field is "absent". This constant `undef` is not a value of

our language. The associated type, Undef, can be seen as the type of the result of projection of a missing label. Undef is particular in that it is disjoint from $\mathbb{1}$, that is, $\mathsf{Undef} \wedge \mathbb{1} \simeq \mathbb{0}$. This representation allows us to encode open record types, whose interpretation contains quasi constant functions where all the labels not explicitly written are mapped to a value of $\mathcal{D}$ (not necessarily `undef`).

The syntax of types does not allow us to explicitly refer to the Undef type. The open/closed record syntax provides a way to set it for the infinitely many constant fields that are not explicitly written in the record type. For a single label, the access to the Undef type is provided via the syntax of fields. There are two kinds of fields: $\ell = t$, which indicates that the field is present in the record and that the associated value has type t, and $\ell =? t$, which is a syntactic sugar for $\ell = (t \vee \mathsf{Undef})$, indicating that a label ℓ may be present, and if so, the associated value has the type t. Note the special case $\ell =? \mathbb{0}$, which indicates that the field for ℓ is absent.

The subtyping relation defined in Chapter 2 can be extended to support records. This will not be formalized here, but the reader may refer to Frisch (2004) for more details. Using this subtyping relation, we can define the following three operators[1]:

$$t.\ell = \begin{cases} \min\{u \mid \{\ell = u\ \ldots\} \geqslant t\} & \text{if } t \leqslant \{\ell = \mathbb{1}\ \ldots\} \\ \min\{u \mid \{\ell =? u\ \ldots\} \geqslant t\} \vee \mathsf{Undef} & \text{otherwise} \end{cases} \tag{7.1}$$

$$t_1 + t_2 = \min\left\{ u \;\middle|\; \forall \ell \in \mathsf{Labels}.\ \begin{cases} u.\ell \geqslant t_2.\ell & \text{if } t_2.\ell \leqslant \mathbb{1} \\ u.\ell \geqslant t_1.\ell \vee (t_2.\ell \backslash \mathsf{Undef}) & \text{otherwise} \end{cases} \right\} \tag{7.2}$$

$$t\backslash\ell = \min\left\{ u \;\middle|\; \forall \ell' \in \mathsf{Labels}.\ \begin{cases} u.\ell' \geqslant \mathsf{Undef} & \text{if } \ell' \equiv \ell \\ u.\ell' \geqslant t.\ell' & \text{otherwise} \end{cases} \right\} \tag{7.3}$$

Record projection (7.1) represents the union of the possible types the label ℓ could have, and also contains `undef` if the record type does not surely have a label ℓ. Record concatenation (7.2) is the right-favored merging of two records. If a label is present in just one of the records, then the type of that label is used. If it is present in both records, the type of the right label is used. Record label deletion (7.3) marks the label as `undef`. These operators can be computed from the DNF of record types, similarly to other type operators defined in Section 2.5. Finally, the tallying algorithm is updated to support subtyping constraints involving records.

[1] The sets defined in definitions 7.2 and 7.3 may not have a minimal element. Any element of these sets can be used as an approximation of these operators.

Extension of the type system

Declarative type system The following typing rules are added to the declarative type system:

$$[\textsc{Record}]\ \frac{}{\Gamma \vdash \{\} : \mathbf{\{\}}} \qquad [\textsc{Update}]\ \frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash \{e_1\ \mathtt{with}\ \ell = e_2\} : t_1 + \mathbf{\{}\ell = t_2\mathbf{\}}}\ t_1 \leqslant \mathbf{\{\ ..\}}$$

$$[\textsc{Delete}]\ \frac{\Gamma \vdash e : t}{\Gamma \vdash e\backslash\ell : t\backslash\ell}\ t \leqslant \mathbf{\{\ ..\}} \qquad [\textsc{Select}]\ \frac{\Gamma \vdash e : t}{\Gamma \vdash e.\ell : t.\ell}\ t \leqslant \mathbf{\{}\ell = \mathbb{1}\ \mathbf{..\}}$$

The empty record value has the closed record type. Record update uses the type operator for extension and is defined provided that the type of e_1 is a record type (i.e., $t_1 \leqslant \mathbf{\{\ ..\}}$). Field deletion uses the corresponding type operator and so does field projection provided that the selected field ℓ is present in the expression e (i.e., $t \leqslant \mathbf{\{}\ell = \mathbb{1}\ \mathbf{..\}}$). This ensures that $t.\ell$ does not contain `undef`.

Algorithmic type system The grammar of atoms is extended with the empty record constant, record updates, record deletions and record projections:

$$\begin{array}{rrcl} \textbf{Atomic expr} & a & ::= & c \mid x \mid \lambda x.\kappa \mid (\mathsf{x},\mathsf{x}) \mid \mathsf{x}\mathsf{x} \mid \pi_i\mathsf{x} \mid (\mathsf{x}{\in}\tau)\,?\,\mathsf{x}:\mathsf{x} \\ & & & \mid \{\} \mid \{\mathsf{x}\ \mathtt{with}\ \ell = \mathsf{x}\} \mid \mathsf{x}\backslash\ell \mid \mathsf{x}.\ell \end{array}$$

The $[\![.]\!]$ transformation that maps expressions of the source language to canonical forms is extended with the following cases, where $\mathsf{x}_\circ$ is a fresh binding variable:

$$\begin{aligned} [\![\{\}]\!] &= ((\mathsf{x}_\circ, \{\}), \mathsf{x}_\circ) \\ [\![\{e_1\ \mathtt{with}\ \ell = e_2\}]\!] &= ((B_1; B_2; (\mathsf{x}_\circ, \{\mathsf{x}_1\ \mathtt{with}\ \ell = \mathsf{x}_2\})), \mathsf{x}_\circ) \\ &\qquad \text{where } (B_1, \mathsf{x}_1) = [\![e_1]\!],\ (B_2, \mathsf{x}_2) = [\![e_2]\!] \\ [\![e\backslash\ell]\!] &= ((B; (\mathsf{x}_\circ, \mathsf{x}\backslash\ell)), \mathsf{x}_\circ) \qquad \text{where } (B, \mathsf{x}) = [\![e]\!] \\ [\![e.\ell]\!] &= ((B; (\mathsf{x}_\circ, \mathsf{x}.\ell)), \mathsf{x}_\circ) \qquad \text{where } (B, \mathsf{x}) = [\![e]\!] \end{aligned}$$

The algorithmic type system can then be extended with the following rules:

$$[\textsc{Record-Alg}]\ \frac{}{\Gamma \vdash_{\mathcal{A}} [\{\} \mid \varnothing] : \mathbf{\{\}}}$$

$$[\textsc{Update-Alg}]\ \frac{}{\Gamma \vdash_{\mathcal{A}} [\{\mathsf{x}\ \mathtt{with}\ \ell = \mathsf{y}\} \mid =(\Sigma, \rho)] : t_1 + \mathbf{\{}\ell = \Gamma(\mathsf{y})\rho\mathbf{\}}}\ \begin{array}{l} t_1 = \Gamma(\mathsf{x})\Sigma \\ t_1 \leqslant \mathbf{\{\ ..\}} \end{array}$$

$$[\textsc{Delete-Alg}]\ \frac{}{\Gamma \vdash_{\mathcal{A}} [\mathsf{x}\backslash\ell \mid \backslash(\Sigma)] : t\backslash\ell}\ \begin{array}{l} t = \Gamma(\mathsf{x})\Sigma \\ t \leqslant \mathbf{\{\ ..\}} \end{array}$$

$$[\textsc{Select-Alg}]\ \frac{}{\Gamma \vdash_{\mathcal{A}} [\mathsf{x}.\ell \mid .(\Sigma)] : t.\ell}\ \begin{array}{l} t = \Gamma(\mathsf{x})\Sigma \\ t \leqslant \mathbf{\{}\ell = \mathbb{1}\ \mathbf{..\}} \end{array}$$

In these rules:

- $=(\Sigma, \rho)$ annotates record updates $\{\mathsf{x}\ \mathtt{with}\ \ell = \mathsf{y}\}$, where Σ is used to instantiate x so that it satisfies the side condition, and ρ is just a renaming of polymorphic type variables applied to the type of y to avoid correlations with the type of x,
- $\backslash(\Sigma)$ annotates field deletions $\mathsf{x}\backslash\ell$, where Σ is used to instantiate x so that it satisfies the side condition,
- $.(\Sigma)$ annotates field projections $\mathsf{x}.\ell$, where Σ is used to instantiate x so that it satisfies the side condition.

The sets of substitutions Σ in these annotations are needed for completeness. For instance, when typing $\mathsf{x}.\ell$ with $\mathsf{x} : \alpha$, we can use the [Select-Alg] rule with the set of substitutions $\{\ \{\alpha \rightsquigarrow \mathbf{\{}\ell = t\mathbf{\}}\}\ \}$ (for any t) in order to satisfy the side-condition and derive the type t.

Main reconstruction system The main reconstruction algorithm is extended with the following rules:

$$[\textsc{UpdateVar}_i]\ \frac{\mathsf{x}_i \notin \mathsf{dom}(\Gamma)}{\Gamma \vdash_{\mathcal{R}} \langle \{\mathsf{x}_1\ \mathtt{with}\ \ell = \mathsf{x}_2\}\ \mid\ \mathtt{infer}\rangle \Rightarrow \mathtt{Var}\ (\mathsf{x}_i, \mathtt{infer}, \mathtt{untyp})}$$

$$[\textsc{UpdateInfer}]\ \frac{\Psi = \mathsf{tally\_infer}(\Gamma(\mathsf{x}_1) \dot{\leqslant} \mathbf{\{}\ \mathbf{..\}}) \qquad \mathsf{x}_2 \in \mathsf{dom}(\Gamma)}{\Gamma \vdash_{\mathcal{R}} \langle \{\mathsf{x}_1\ \mathtt{with}\ \ell = \mathsf{x}_2\}\ \mid\ \mathtt{infer}\rangle \Rightarrow \mathtt{Subst}(\Psi, \mathtt{typ}, \mathtt{untyp})}$$

$$[\textsc{DeleteVar}]\ \frac{\mathsf{x} \notin \mathsf{dom}(\Gamma)}{\Gamma \vdash_{\mathcal{R}} \langle \mathsf{x}\backslash\ell\ \mid\ \mathtt{infer}\rangle \Rightarrow \mathtt{Var}\ (\mathsf{x}, \mathtt{infer}, \mathtt{untyp})}$$

$$[\textsc{DeleteInfer}]\ \frac{\Psi = \mathsf{tally\_infer}(\Gamma(\mathsf{x}) \dot{\leqslant} \mathbf{\{}\ \mathbf{..\}})}{\Gamma \vdash_{\mathcal{R}} \langle \mathsf{x}\backslash\ell\ \mid\ \mathtt{infer}\rangle \Rightarrow \mathtt{Subst}(\Psi, \mathtt{typ}, \mathtt{untyp})}$$

$$[\textsc{SelectVar}]\ \frac{\mathsf{x} \notin \mathsf{dom}(\Gamma)}{\Gamma \vdash_{\mathcal{R}} \langle \mathsf{x}.\ell\ \mid\ \mathtt{infer}\rangle \Rightarrow \mathtt{Var}\ (\mathsf{x}, \mathtt{infer}, \mathtt{untyp})}$$

$$[\textsc{SelectInfer}]\ \frac{\Psi = \mathsf{tally\_infer}(\Gamma(\mathsf{x}) \dot{\leqslant} \mathbf{\{}\ell = \alpha\ \mathbf{..\}})}{\Gamma \vdash_{\mathcal{R}} \langle \mathsf{x}.\ell\ \mid\ \mathtt{infer}\rangle \Rightarrow \mathtt{Subst}(\Psi, \mathtt{typ}, \mathtt{untyp})}\ \alpha \in \mathcal{V}_P \text{ fresh}$$

These rules are similar to the rules for pair projection. For instance, [DeleteVar] ensures that the binding variable x is in the environment, while [DeleteInfer] uses tallying to find some instances of the current context that make x a subtype of $\mathbf{\{}\ \mathbf{..\}}$, thus making the side conditions of the rule [Delete-Alg] satisfiable.

Substitution inference system The substitution inference system is extended with the following rules:

$$[\textsc{Update}]\ \frac{\Sigma = \mathsf{tally}(\{\Gamma(\mathsf{x}_1) \mathrel{\dot{\leqslant}} \mathbf{\{\ ..\}}\}) \qquad \rho = \mathsf{refresh}(\Gamma(\mathsf{x}_2))}{\Gamma \vdash_{\mathcal{S}} \langle \{\mathsf{x}_1\ \mathtt{with}\ \ell = \mathsf{x}_2\} \mid \mathsf{typ}\rangle \Rightarrow =(\Sigma, \rho)}$$

$$[\textsc{Delete}]\ \frac{\Sigma = \mathsf{tally}(\{\Gamma(\mathsf{x}) \mathrel{\dot{\leqslant}} \mathbf{\{\ ..\}}\})}{\Gamma \vdash_{\mathcal{S}} \langle \mathsf{x}\backslash\ell \mid \mathsf{typ}\rangle \Rightarrow \backslash(\Sigma)}$$

$$[\textsc{Select}]\ \frac{\Sigma = \mathsf{tally}(\{\Gamma(\mathsf{x}) \mathrel{\dot{\leqslant}} \mathbf{\{}\ell = \alpha\ \mathbf{..\}}\})}{\Gamma \vdash_{\mathcal{S}} \langle \mathsf{x}.\ell \mid \mathsf{typ}\rangle \Rightarrow .(\Sigma)}\ \alpha \in \mathcal{V}_P \text{ fresh}$$

Again, these rules follow those for pair projection. They use tallying to find an instantiation Σ that satisfy the side conditions of the corresponding rules in the algorithmic type system.

Split backpropagation system The split backpropagation system is extended with the following rules:

$$[\textsc{Update}]\ \frac{\mathbf{u} \wedge \mathbf{\{}\ell = \mathbb{1}\ \mathbf{..\}} \overset{\text{DNF}}{\cong} (\bigvee_{i\in I} a_i) \vee \ldots}{\Gamma \vdash_{\mathcal{B}} (\{\mathsf{x}_1\ \mathtt{with}\ \ell = \mathsf{x}_2\} : \mathbf{u}) \Rightarrow \{\{\mathsf{x}_1 : (a_i\backslash\ell) + \mathbf{\{}\ell =? \mathbb{1}\mathbf{\}}, \mathsf{x}_2 : a_i.\ell\} \mid i \in I\}}$$

$$[\textsc{Delete}]\ \frac{\mathbf{u} \wedge \mathbf{\{}\ell =? \mathbb{0}\ \mathbf{..\}} \overset{\text{DNF}}{\cong} (\bigvee_{i\in I} a_i) \vee \ldots}{\Gamma \vdash_{\mathcal{B}} (\mathsf{x}\backslash\ell : \mathbf{u}) \Rightarrow \{\{\mathsf{x} : a_i + \mathbf{\{}\ell =? \mathbb{1}\mathbf{\}}\} \mid i \in I\}}$$

$$[\textsc{Select}]\ \frac{}{\Gamma \vdash_{\mathcal{B}} (\mathsf{x}.\ell : \mathbf{u}) \Rightarrow \{\{\mathsf{x} : \mathbf{\{}\ell = \mathbf{u}\ \mathbf{..\}}\}\}}$$

with a_i ranging over atomic record types (open or closed).

An expression $\{\mathsf{x}_1\ \mathtt{with}\ \ell = \mathsf{x}_2\}$ can only produce records that surely have a field ℓ. Thus, we can intersect $\mathbf{u}$ with $\mathbf{\{}\ell = \mathbb{1}\ \mathbf{..\}}$ in Rule [Update] without loss of generality. Then, in the DNF of the result, we consider among the summands of the outer union those composed only of an atomic record type (the other summands are captured by ...), similarly to the [Pair] rule in Section 6.4. For each such summand a_i, we can ensure that $\{\mathsf{x}_1\ \mathtt{with}\ \ell = \mathsf{x}_2\}$ has type a_i (and thus $\mathbf{u}$) by giving to x_2 the type $a_i.\ell$ and to x_1 the type $(a_i\backslash\ell) + \mathbf{\{}\ell =? \mathbb{1}\mathbf{\}}$, which intuitively corresponds to the type a_i in which we removed any information about the field ℓ.

The idea behind the [Delete] rule is similar. The type $\mathbf{\{}\ell =? \mathbb{0}\ \mathbf{..\}}$ corresponds to records whose field ℓ is absent, while the type $a_i + \mathbf{\{}\ell =? \mathbb{1}\mathbf{\}}$ corresponds to the type a_i in which we removed any information about the field ℓ (the deletion of the field ℓ is not necessary this time as the field ℓ is already absent in a_i).

Finally, the [Select] rule is similar to the rules [Proj$_i$] of Section 6.4: in order for $\mathsf{x}.\ell$ to have type $\mathbf{u}$, we can just suppose that x has type $\mathbf{\{}\ell = \mathbf{u}\ \mathbf{..\}}$.

7.2 User type annotations

Our type system features type inference, so that the user does not have to write type annotations. Nevertheless, we present in this section an extension of the core language that allows one to annotate λ-abstractions. Type annotations may serve several purposes:

Incompleteness As discussed in Chapter 6, our reconstruction algorithm is incomplete: it may fail to reconstruct an annotation tree for an expression that is typeable with the declarative type system. User type annotations provide a way to guide reconstruction, and thus to type more programs.

Precision This point is related to the previous one. As we do not have principality of typings, the user may need to write type annotations in order to derive a more precise type for a function, or at the opposite, a simpler but less precise type (which may improve the readability of types).

Performance The inference of the type of parameters made by the reconstruction algorithm has a cost: it requires backtracking and inserting intersection nodes that may lead to an explosion of the number of branches in the annotation tree (this will be detailed in Chapter 8). Writing user type annotations is a way to prevent this explosion and to improve performance.

Signature restriction Type annotations allow the user to restrict the domain of a function. For instance, the user might want to define a function $\lambda x.x + x$ that returns the double of an integer. However, if the `+` operator is overloaded, allowing to also concatenate strings, then the type inferred for the function $\lambda x.x + x$ will be $(\texttt{Int} \to \texttt{Int}) \wedge (\texttt{String} \to \texttt{String})$. User type annotations provide a way to restrict the domain of this function to `Int`.

Better error messages User type annotations can be used to generate more relevant error messages. Indeed, when the body of a λ-abstraction is not typeable under a given context, the type system does not know whether it is because of an error in the program or if it is because the domain of the λ-abstraction needs to be restrained, in which case it may remove the current branch without raising any error message. Adding type annotations to the parameter of a λ-abstraction solves this issue: if the body cannot be typed for a parameter whose type has been explicitly written by the user, then we know that an error should be raised.

Syntax and semantics

We add to the source language a new construct for λ-abstractions that allows specifying its domain:

$$\begin{array}{rrcl} \textbf{Expression} & e & ::= & c \mid x \mid \lambda x.e \mid e\ e \mid (e,e) \mid \pi_i e \mid (e \in \tau)\ ?\ e : e \\ & & & \mid \lambda(x : \mathbf{u}\ ;\ \ldots\ ;\ \mathbf{u}).e \end{array}$$

There are two things to note about this new construct $\lambda(x : \mathbf{u}_1 \ ; \ \ldots \ ; \ \mathbf{u}_n).e$.

First, the types used for annotations are monomorphic types. Indeed, as this corresponds to the type of the parameter x of the λ-abstraction, it should not be polymorphic. Actually, we restrict it even further: we consider a special subset $\mathcal{V}_U$ of $\mathcal{V}_M$, whose elements are called user type variables, and we require that the types used for annotations only contain user type variables. Formally, we require $\forall i \in 1 \,..\, n.\ \mathsf{vars}(\mathbf{u}_i) \subseteq \mathcal{V}_U$. Doing so allows keeping track of which type variables have been introduced by the user.

Secondly, the parameter x is annotated with a list of types instead of a unique type. This allows specifying multiple types for the parameter, each of those types yielding a separate branch and thus allowing to capture overloaded behaviors.

Note that these type annotations are different from the explicitly-typed λ-abstractions that are usually used in set-theoretic type systems, such as in Castagna et al. (2014), and that are annotated with the full function type $\bigwedge_{i\in I} s_i \to t_i$ instead of only the domains $\{s_i\}_{i\in I}$.

User type annotations are erased by the semantics:

$$\lambda(x : \mathbf{u} \ ; \ \ldots \ ; \ \mathbf{u}).e \quad \rightsquigarrow \quad \lambda x.e$$

Extension of the type system

Declarative type system From the point of view of the declarative type system, these annotations are seen as a constraint rather than an indication:

$$[\to\text{I-Ann}]\ \frac{\Gamma \vdash \lambda x.e : t}{\Gamma \vdash \lambda(x : \mathbf{u}_1 \ ; \ \ldots \ ; \ \mathbf{u}_n).e : t}\ \mathsf{dom}(t) \simeq \bigvee_{i\in 1..n} \mathbf{u}_i$$

Algorithmic type system The grammar of atoms is extended with user-annotated λ-abstractions:

$$\begin{array}{rrcl} \textbf{Atomic expr} & a & ::= & c \mid x \mid \lambda x.\kappa \mid (\mathsf{x},\mathsf{x}) \mid \mathsf{x}\mathsf{x} \mid \pi_i\mathsf{x} \mid (\mathsf{x}{\in}\tau)\,?\,\mathsf{x}:\mathsf{x} \\ & & & \mid \lambda(x : \mathbf{u} \ ; \ \ldots \ ; \ \mathbf{u}).\kappa \end{array}$$

We extend the $[\![.]\!]$ transformation that converts a term of the source language into a canonical form as follows:

$$[\![\lambda(x : \mathbf{u}_1 \ ; \ \ldots \ ; \ \mathbf{u}_n).e]\!] = ((\mathsf{x}_\circ, \lambda(x : \mathbf{u}_1 \ ; \ \ldots \ ; \ \mathbf{u}_n).\mathsf{term}[\![e]\!]), \mathsf{x}_\circ)$$

The following rule is added to the algorithmic type system, following the behavior of the [→I-Ann] rule of the declarative type system:

$$[\to\text{I-Ann-Alg}]\ \frac{\Gamma \vdash_{\mathcal{A}} [\lambda x.\kappa \mid \mathbb{a}] : t}{\Gamma \vdash_{\mathcal{A}} [\lambda(x : \mathbf{u}_1 \ ; \ \ldots \ ; \ \mathbf{u}_n).\kappa \mid \mathbb{a}] : t}\ \mathsf{dom}(t) \simeq \bigvee_{i\in 1..n} \mathbf{u}_i$$

Reconstruction system In the reconstruction algorithm, user type annotations are seen as an indication to initialize the annotation tree:

$$[\textsc{LambdaInfer-Ann}]\ \frac{\Gamma \vdash_{\mathcal{R}} \langle \lambda x.\kappa \mid \bigwedge(\{\lambda(\mathbf{u}_i, \texttt{infer}) \mid i \in 1\,..\,n\}, \varnothing)\rangle \Rightarrow \mathbb{R}}{\Gamma \vdash_{\mathcal{R}} \langle \lambda(x : \mathbf{u}_1\ ;\ \ldots\ ;\ \mathbf{u}_n).\kappa \mid \texttt{infer}\rangle \Rightarrow \mathbb{R}}$$

$$[\textsc{Lambda-Ann}]\ \frac{\Gamma \vdash_{\mathcal{R}} \langle \lambda x.\kappa \mid \lambda(\mathbf{u}, \mathcal{K})\rangle \Rightarrow \mathbb{R}}{\Gamma \vdash_{\mathcal{R}} \langle \lambda(x : \mathbf{u}_1\ ;\ \ldots\ ;\ \mathbf{u}_n).\kappa \mid \lambda(\mathbf{u}, \mathcal{K})\rangle \Rightarrow \mathbb{R}}$$

The reconstruction of user-annotated λ-abstractions is the same as the reconstruction of unannotated λ-abstractions, except for the initialization. While the parameter of an unannotated λ-abstraction is initially typed with a fresh $\boldsymbol{\alpha} \in \mathcal{V}_M \backslash \mathcal{V}_U$, the parameter of a user-annotated λ-abstraction is initially typed according to the user annotation (Rule [LambdaInfer-Ann]). An intersection annotation is used to handle the case where multiple types are specified in the user annotation.

Two additional changes must be applied to the main reconstruction system. First, the type variables in $\mathcal{V}_U$ should not be substituted during the reconstruction. Indeed, if the user specified a domain for a λ-abstraction, it is this exact domain that should be used and not a specific instance of it. Thus, the tally_infer(.) function should be modified so that the solutions produced do not substitute type variables in $\mathcal{V}_U$. This change is straightforward: instead of generalizing every monomorphic type variable, tally_infer(.) should only generalize type variables in $\mathcal{V}_M \backslash \mathcal{V}_U$ before calling tally(.).

Secondly, the following rule should be added to the rules handling the intersection annotations, with a higher priority over them:

$$[\textsc{InterFail-Ann}]\ \frac{\Gamma \vdash^{*}_{\mathcal{R}} \langle \lambda(x : \mathbf{u}_1\ ;\ \ldots\ ;\ \mathbf{u}_n).\kappa \mid \mathcal{A}\rangle \Rightarrow \texttt{Fail}}{\Gamma \vdash_{\mathcal{R}} \langle \lambda(x : \mathbf{u}_1\ ;\ \ldots\ ;\ \mathbf{u}_n).\kappa \mid \bigwedge(\{\mathcal{A}\} \cup S, S')\rangle \Rightarrow \texttt{Fail}}$$

This rules modifies the behavior of the reconstruction algorithm when a branch of an intersection fails, if this intersection comes from a user-annotated λ-abstraction. Instead of removing the branch and continuing the reconstruction with the other branches, it makes the reconstruction fail for this λ-abstraction, because it is not typeable for one of the domains specified by the user.

7.3 Let-bindings

Syntax and semantics

An essential construct of every functional language is the let-binding. With the notion of programs, our language already features top-level let constructs. However, we are interested here in adding a *local* let-binding to the syntax of expressions (and not programs), as defined in Figure 7.2. We use the usual call-by-value semantics, where the definition is reduced before the body.

Syntax

$$\begin{array}{llll} \textbf{Expression} & e & ::= & c \mid x \mid \lambda x.e \mid e\ e \mid (e,e) \mid \pi_i e \mid (e \in \tau)\,?\,e : e \mid \texttt{let}\, x = e\, \texttt{in}\, e \\ \textbf{Value} & v & ::= & c \mid \lambda x.e \mid (v,v) \end{array}$$

Additional reduction rules

$$\texttt{let}\, x = v\, \texttt{in}\, e \quad \rightsquigarrow \quad e\{v/x\}$$

Evaluation Context

$$E ::= [\,] \mid v\ E \mid E\ e \mid (v,E) \mid (E,e) \mid \pi_i E \mid (E \in \tau)\,?\,e : e \mid \texttt{let}\, x = E\, \texttt{in}\, e$$

Figure 7.2: Syntax and semantics of the language with let-bindings

Declarative type system

Usually, let-bindings can be typed by adding a single rule to the type system:

$$[\textsc{Let}]\ \frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma, x : t_1 \vdash e_2 : t_2}{\Gamma \vdash \texttt{let}\, x = e_1\, \texttt{in}\, e_2 \ : t_2}$$

However, this solution is not satisfactory in our case, because of the presence of the union-elimination rule. Indeed, let-bindings can introduce aliasing, which might prevent the application of a $[\vee]$ rule, thus limiting occurrence typing. For instance, consider the following expression, with $f : \mathbb{1} \to \mathbb{1}$:

$$\lambda x.\ \texttt{let}\, y = x\, \texttt{in}\, (f\ y \in \texttt{Int})\,?\,(f\ x) + 1 : 42$$

We should be able to derive the type $\mathbb{1} \to \texttt{Int}$ for this expression: as x and y are just aliases, $f\ y$ and $f\ x$ both reduce to the same value (or both diverge) and thus we can deduce in the first branch of the type-case that $f\ x$ has the type $\texttt{Int}$.

The difficulty here is that the two expressions $f\ x$ and $f\ y$ are not syntactically equivalent, and thus the union-elimination rule cannot apply on those two occurrences. Note that the same problem is present if we choose to encode let-bindings as applications: by encoding $\texttt{let}\, x = e_1\, \texttt{in}\, e_2$ as the application $(\lambda x.e_2)\ e_1$, the correlation between x and e_1 is lost. In order to overcome this issue, we choose to remove aliasing before applying the declarative type system. For that, we introduce an intermediate language featuring an alternative version of let-bindings:

Definition 72 (Intermediate language)**.** Intermediate expressions *are finite terms produced by the following grammar:*

$$\begin{array}{llll} \textbf{\textit{Intermediate expression}} & \xi & ::= & c \mid x \mid \lambda x.\xi \mid \xi\xi \mid (\xi,\xi) \mid \pi_i \xi \mid (\xi \in \tau)\,?\,\xi : \xi \\ & & & \quad \mid \texttt{let}\, \xi\, \texttt{in}\, \xi \end{array}$$

In the intermediate expression $\texttt{let}\,\xi_1\,\texttt{in}\,\xi_2$, ξ_1 corresponds to the definition of the let-binding, and ξ_2 corresponds to the body. However, in order to prevent aliasing, the definition ξ_1 is inlined in ξ_2, so that there is no need to associate the definition ξ_1 to a variable. Still, it is necessary to keep the definition of ξ_1 (even if this definition is not bound to a variable) in order to preserve the call-by-value behavior of let-bindings. Indeed, in a let-binding $\texttt{let}\,x = e_1\,\texttt{in}\,e_2$, if the definition e_1 is not typeable then the let-binding expression should not be typeable either, even if x is not used in the body e_2.

Let-bindings of the source language can be transformed into this intermediate form using a transformation $(\!|.|\!)$:

Definition 73. *Let e be a ground expression of the source language. The intermediate expression $(\!|e|\!)$ is defined inductively on e as follows:*

$$
\begin{aligned}
(\!|c|\!) &= c \\
(\!|x|\!) &= x \\
(\!|\lambda x.e|\!) &= \lambda x.(\!|e|\!) \\
(\!|e_1 e_2|\!) &= (\!|e_1|\!)(\!|e_2|\!) \\
(\!|(e_1, e_2)|\!) &= ((\!|e_1|\!), (\!|e_2|\!)) \\
(\!|\pi_i e|\!) &= \pi_i(\!|e|\!) && i = 1, 2 \\
(\!|(e \in \tau)\,?\,e_1 : e_2|\!) &= ((\!|e|\!) \in \tau)\,?\,(\!|e_1|\!) : (\!|e_2|\!) \\
(\!|\texttt{let}\,x = e_1\,\texttt{in}\,e_2|\!) &= \texttt{let}\,(\!|e_1|\!)\,\texttt{in}\,(\!|e_2|\!)\{(\!|e_1|\!)/x\}
\end{aligned}
$$

For instance, our earlier example $\lambda x.\ \texttt{let}\,y = x\,\texttt{in}\,(f\ y \in \texttt{Int})\,?\,(f\ x) + 1 : 42$ is transformed into the intermediate expression $\lambda x.\ \texttt{let}\,x\,\texttt{in}\,(f\ x \in \texttt{Int})\,?\,(f\ x) + 1 : 42$, which can then be typed by the declarative type system by applying the union-elimination rule on both occurrences of $f\ x$.

As another example, consider the expression $\lambda x.\ \texttt{let}\,y = 42\ 42\,\texttt{in}\,(x \in \texttt{Int})\,?\,x : y$. It is transformed into $\lambda x.\ \texttt{let}\,42\ 42\,\texttt{in}\,(x \in \texttt{Int})\,?\,x : 42\ 42$, which is not typeable: the definition of the let-binding, 42 42, must be typed in all contexts, even if it is only used in the second branch of the type-case.

Note that, just like the $[\vee]$ rule, this transformation relies on the fact that our language is pure. In the presence of side effects, this transformation may not preserve the semantics, as inlining the definition e_1 in the body e_2 could duplicate the side effects of e_1.

Now, the declarative type system takes as input an intermediate expression instead of a source expression, and is extended with this rule:

$$
[\textsc{Let}]\ \frac{\Gamma \vdash \xi_1 : t_1 \qquad \Gamma \vdash \xi_2 : t_2}{\Gamma \vdash \texttt{let}\,\xi_1\,\texttt{in}\,\xi_2\ : t_2}
$$

It checks that the definition ξ_1 is typeable, and then proceed with typing the body ξ_2.

Notice that, by inlining let-definitions, the $(\!|.|\!)$ transformation may duplicate some subterms and thus increase the size of the term. Fortunately, this expansion does not happen in the MSC form, defined below, as the duplicated subterms are factorized into a unique binding.

Algorithmic type system

Let-bindings are added to MSC forms as new atoms:

$$\textbf{Atomic expr} \quad a \quad ::= \quad c \mid x \mid \lambda x.\kappa \mid (\mathsf{x},\mathsf{x}) \mid \mathsf{x}\mathsf{x} \mid \pi_i \mathsf{x} \mid (\mathsf{x} \in \tau)\,?\,\mathsf{x} : \mathsf{x} \mid \mathtt{let}\,\mathsf{x}\,\mathtt{in}\,\mathsf{x}$$

The intuition is the same as for the declarative type system: we want to get rid of the aliasing caused by let-bindings. To produce an atom for the expression $\mathtt{let}\,x = e_1\,\mathtt{in}\,e_2$, each subexpression must be replaced by a binding variable, yielding $\mathtt{let}\,x = \mathsf{x}_1\,\mathtt{in}\,\mathsf{x}_2$. Now, it means that the variable x is only an alias for x_1, which is undesirable. Consequently, we make let-binding atoms not introduce any new variable, which explains the $\mathtt{let}\,\mathsf{x}_1\,\mathtt{in}\,\mathsf{x}_2$ syntax.

We extend the definition of $[\![.]\!]$ of Section 5.1.1 with the let-binding case. The $[\![.]\!]$ operator is used to transform an expression of the source language into a canonical form expression, but from now on it will take as input an intermediate expression instead of an expression of the source language (the transformation from source language to intermediate language is handled by $(\!|.|\!)$).

$$[\![\mathtt{let}\,\xi_1\,\mathtt{in}\,\xi_2]\!] = ((B_1; B_2; (\mathsf{x}_\circ, \mathtt{let}\,\mathsf{x}_1\,\mathtt{in}\,\mathsf{x}_2)), \mathsf{x}_\circ)$$
$$\text{where } (B_1, \mathsf{x}_1) = [\![\xi_1]\!], (B_2, \mathsf{x}_2) = [\![\xi_2]\!]$$
$$\text{and } \mathsf{x}_\circ \text{ is a fresh binding variable}$$

For instance, the expression $\mathtt{let}\,x = \lambda y.y\,\mathtt{in}\,(x, x)$ of the source language is first transformed into the expression $\mathtt{let}\,\lambda y.y\,\mathtt{in}\,(\lambda y.y, \lambda y.y)$ of the intermediate language using the operator $(\!|.|\!)$, and then it is transformed into the following canonical form using $[\![.]\!]$:

$$\begin{aligned}
&\mathtt{bind}\,\mathsf{x}_1 = (\lambda y.\mathtt{bind}\,\mathsf{y} = y\,\mathtt{in}\,\mathsf{y})\,\mathtt{in}\\
&\mathtt{bind}\,\mathsf{x}_2 = (\lambda y.\mathtt{bind}\,\mathsf{y} = y\,\mathtt{in}\,\mathsf{y})\,\mathtt{in}\\
&\mathtt{bind}\,\mathsf{x}_3 = (\lambda y.\mathtt{bind}\,\mathsf{y} = y\,\mathtt{in}\,\mathsf{y})\,\mathtt{in}\\
&\mathtt{bind}\,\mathsf{x}_4 = (\mathsf{x}_2, \mathsf{x}_3)\,\mathtt{in}\\
&\mathtt{bind}\,\mathsf{x}_5 = (\mathtt{let}\,\mathsf{x}_1\,\mathtt{in}\,\mathsf{x}_4)\,\mathtt{in}\\
&\mathsf{x}_5
\end{aligned}$$

Finally, the different occurrences of $\lambda y.\mathtt{bind}\,\mathsf{y} = y\,\mathtt{in}\,\mathsf{y}$ are factorized by applying the $\dashrightarrow$ rewriting rules (Figure 5.1), yielding the following MSC form:

$$\begin{aligned}
&\mathtt{bind}\,\mathsf{x}_1 = (\lambda y.\mathtt{bind}\,\mathsf{y} = y\,\mathtt{in}\,\mathsf{y})\,\mathtt{in}\\
&\mathtt{bind}\,\mathsf{x}_2 = (\mathsf{x}_1, \mathsf{x}_1)\,\mathtt{in}\\
&\mathtt{bind}\,\mathsf{x}_3 = (\mathtt{let}\,\mathsf{x}_1\,\mathtt{in}\,\mathsf{x}_2)\,\mathtt{in}\\
&\mathsf{x}_3
\end{aligned}$$

The algorithmic type system can then be extended with this simple rule (with $\varnothing$ the annotation associated to let-bindings):

$$[\textsc{Let-Alg}]\ \frac{}{\Gamma \vdash_{\mathcal{A}} [\mathtt{let}\,\mathsf{x}_1\,\mathtt{in}\,\mathsf{x}_2 \mid \varnothing] : \Gamma(\mathsf{x}_2)}\ \mathsf{x}_1 \in \mathsf{dom}(\Gamma)$$

Reconstruction algorithm

The reconstruction rules are straightforward, as a let-binding is similar to a pair: for a let-binding to be typeable, the two binding variables composing it must be typeable.

Main reconstruction system

$$[\textsc{LetVar}_i]\ \frac{\mathsf{x}_i \notin \mathsf{dom}(\Gamma)}{\Gamma \vdash_{\mathcal{R}} \langle \mathtt{let}\,\mathsf{x}_1\,\mathtt{in}\,\mathsf{x}_2 \mid \mathtt{infer}\rangle \Rightarrow \mathtt{Var}\ (\mathsf{x}_i, \mathtt{infer}, \mathtt{untyp})}$$

$$[\textsc{LetOk}]\ \frac{}{\Gamma \vdash_{\mathcal{R}} \langle \mathtt{let}\,\mathsf{x}_1\,\mathtt{in}\,\mathsf{x}_2 \mid \mathtt{infer}\rangle \Rightarrow \mathtt{Ok}(\mathtt{typ})}$$

If one of the binding variables composing the let-binding has not been typed yet, the rule [LetVar$_i$] applies and ask to type it. Otherwise, [LetOk] applies and states that the let-binding is typeable.

Substitution inference system

$$[\textsc{Let}]\ \frac{}{\Gamma \vdash_{\mathcal{S}} \langle \mathtt{let}\,\mathsf{x}_1\,\mathtt{in}\,\mathsf{x}_2 \mid \mathtt{typ}\rangle \Rightarrow \varnothing}$$

This rule is trivial since let-bindings are annotated with $\varnothing$ in the algorithmic type system.

Split backpropagation system

$$[\textsc{Let}]\ \frac{}{\Gamma \vdash_{\mathcal{B}} (\mathtt{let}\,\mathsf{x}_1\,\mathtt{in}\,\mathsf{x}_2 : \mathbf{u}) \Rightarrow \{\{\mathsf{x}_2 : \mathbf{u}\}\}}$$

In order for a let-binding $\mathtt{let}\,\mathsf{x}_1\,\mathtt{in}\,\mathsf{x}_2$ to have type $\mathbf{u}$, we just need x_2 to have type $\mathbf{u}$.

7.4 Extended type-cases

In this section, we extend our language with a more general type-case construct, allowing to dispatch between an arbitrary number of branches at once instead of just two.

Syntax and semantics

Extended type-cases $\texttt{tcase}\, e\, \texttt{of}\, \tau_1 \to e_1 \mid \ldots \mid \tau_n \to e_n$ are quite similar to ternary type-cases $(e \in \tau)\,\texttt{?}\, e\,\texttt{:}\, e$. Both of them act like a dynamic dispatch, executing a branch selected at run-time depending on the type of an expression. However, instead of dispatching among two branches depending on the result of a unique type test, extended type-cases can have an arbitrary number of branches, each guarded by a type.

The syntax and semantics of extended type-cases is formalized in Figure 7.3.

Syntax

$$\begin{array}{lrcl} \textbf{Expression} & e & ::= & c \mid x \mid \lambda x.e \mid e\ e \mid (e,e) \mid \pi_i e \mid (e \in \tau)\,\texttt{?}\, e\,\texttt{:}\, e \mid \texttt{let}\, x\,\texttt{=}\, e\,\texttt{in}\, e \\ & & & \mid (\texttt{tcase}\, e\, \texttt{of}\, \tau \to e \mid \ldots \mid \tau \to e) \\ \textbf{Value} & v & ::= & c \mid \lambda x.e \mid (v,v) \end{array}$$

Additional reduction rules

For every $n \in \mathbb{N}^*$ and $k \in 1\,..\,n$:

$$\texttt{tcase}\, v\, \texttt{of}\, \tau_1 \to e_1 \mid \ldots \mid \tau_n \to e_n \quad \rightsquigarrow \quad e_k \qquad \text{if } v \in \tau_k \backslash (\textstyle\bigvee_{i \in 1..k-1} \tau_i)$$

Evaluation Context

$$\begin{array}{rcl} E & ::= & [\,] \mid v\ E \mid E\ e \mid (v,E) \mid (E,e) \mid \pi_i E \mid (E \in \tau)\,\texttt{?}\, e\,\texttt{:}\, e \mid \texttt{let}\, x\,\texttt{=}\, E\,\texttt{in}\, e \\ & & \mid (\texttt{tcase}\, E\, \texttt{of}\, \tau \to e \mid \ldots \mid \tau \to e) \end{array}$$

Figure 7.3: Syntax and semantics of the language with extended type-cases

Type constraints

From the perspective of the type system, we choose to encode extended type-cases using ternary type-cases in order to avoid having redundant typing rules. However, one ingredient is missing: while we can encode a dispatch between n branches by combining $n-1$ dispatches between two branches, we cannot encode the fact that extended type-cases might not cover $\mathbb{1}$. For instance, consider the following expression:

$$\lambda x.\ \texttt{tcase}\, x\, \texttt{of}\, \texttt{Int} \to x + 1 \mid \texttt{Bool} \to x$$

This type-case only covers $\texttt{Int} \vee \texttt{Bool}$. If x has any other type, for instance `String`, then the reduction is stuck as no branch can be selected. This is different from the behavior of the expression $\lambda x.\ (x \in \texttt{Int})\,\texttt{?}\, x + 1\,\texttt{:}\, x$, where the type-case reduces to a value for every x. For this reason, we first add a new construct $(\xi \in \tau)$ to our intermediate language that allows expressing a type constraint, for instance "*x must have type* $\texttt{Int} \vee \texttt{Bool}$". This new construct will allow us to encode extended type-cases, by combining it with ternary type-cases and let-bindings.

$$\textbf{Intermediate expr.}\quad \xi \;::=\; c \mid x \mid \lambda x.\xi \mid \xi\xi \mid (\xi,\xi) \mid \pi_i\xi \mid (\xi{\in}\tau)\,?\,\xi : \xi \mid \mathtt{let}\,\xi\,\mathtt{in}\,\xi \mid (\xi \,\mathtt{⦂}\, \tau)$$

Intuitively, this $(\xi \,\mathtt{⦂}\, \tau)$ construct corresponds to a dynamic type cast: if the expression ξ does not reduce to a value of type τ (and does not diverge), then the reduction is stuck (though it is not necessary to give it a semantics as it only exists in the intermediate language).

Type constraints should not be confused with user type annotations defined in Section 7.2. Indeed, user type annotations are only inserted on the parameters of λ-abstractions, and they fully determine the domain of the λ-abstraction. Instead, type constraints can be inserted around any expression e, and they only check the type of e: the expression $(\mathtt{true} \,\mathtt{⦂}\, \mathtt{Bool})$ can still be typed `True` or $\mathbb{1}$, as long as the type `Bool` is derivable for `true`.

Declarative type system The declarative type system is extended with this rule:

$$[\textsc{Constr}]\ \frac{\Gamma \vdash \xi : \tau \qquad \Gamma \vdash \xi : t}{\Gamma \vdash (\xi \,\mathtt{⦂}\, \tau) : t}$$

Note that the type derived for an expression $(\xi \,\mathtt{⦂}\, \tau)$ is not necessarily τ: the rule [Constr] only checks whether ξ has type τ, but then independently derives another type for ξ.

Algorithmic type system A type constraint construct is added to atoms:

$$\textbf{Atomic expr}\quad a \;::=\; c \mid x \mid \lambda x.\kappa \mid (\mathsf{x},\mathsf{x}) \mid \mathsf{x}\mathsf{x} \mid \pi_i\mathsf{x} \mid (\mathsf{x}{\in}\tau)\,?\,\mathsf{x}:\mathsf{x} \mid \mathtt{let}\,\mathsf{x}\,\mathtt{in}\,\mathsf{x} \mid \mathsf{x} \,\mathtt{⦂}\, \tau$$

The $[\![.]\!]$ transformation, for transforming expressions of the intermediate language to canonical forms, is extended with this case:

$$[\![\xi \,\mathtt{⦂}\, \tau]\!] = ((B; (\mathsf{x}_\circ, \mathsf{x} \,\mathtt{⦂}\, \tau)), \mathsf{x}_\circ) \quad \text{where } (B,\mathsf{x}) = [\![\xi]\!] \text{ and } \mathsf{x}_\circ \text{ is a fresh binding variable}$$

The algorithmic type system can then be extended with the following rule, with $\mathtt{⦂}(\Sigma)$ a new annotation dedicated to type constraints:

$$[\textsc{Constr-Alg}]\ \frac{}{\Gamma \vdash_{\mathcal{A}} [\mathsf{x} \,\mathtt{⦂}\, \tau \mid \mathtt{⦂}(\Sigma)] : \Gamma(\mathsf{x})}\ \Gamma(\mathsf{x})\Sigma \leqslant \tau$$

Main reconstruction system The main reconstruction system is extended with this rule:

$$[\textsc{ConstrVar}]\ \frac{\mathsf{x} \notin \mathsf{dom}(\Gamma)}{\Gamma \vdash_{\mathcal{R}} \langle \mathsf{x} \,\mathtt{⦂}\, \tau \mid \mathtt{infer} \rangle \Rightarrow \mathtt{Var}\ (\mathsf{x}, \mathtt{infer}, \mathtt{untyp})}$$

$$[\textsc{ConstrInfer}]\ \frac{\Psi = \mathsf{tally\_infer}(\Gamma(\mathsf{x}) \mathrel{\dot{\leqslant}} \tau)}{\Gamma \vdash_{\mathcal{R}} \langle \mathsf{x} \,\mathtt{⦂}\, \tau \mid \mathtt{infer} \rangle \Rightarrow \mathtt{Subst}(\Psi, \mathtt{typ}, \mathtt{untyp})}$$

These two rules are similar to the rules for projections: [ConstrVar] ensures that the binding variable x is in the environment, while [ConstrInfer] uses tallying to find some instances of the current context that make x have type τ.

Substitution inference system The substitution inference system is extended with this rule:

$$[\textsc{Constr}]\ \frac{\sigma \in \mathsf{tally}(\{\Gamma(\mathsf{x}) \mathrel{\dot{\leqslant}} \tau\})}{\Gamma \vdash_{\mathcal{S}} \langle \mathsf{x} \ltimes \tau \mid \mathsf{typ} \rangle \Rightarrow \ltimes(\{\sigma\})}$$

This rule uses tallying to find an instantiation σ such that $\Gamma(\mathsf{x})\sigma \leqslant \tau$, and annotates the atom with $\ltimes(\{\sigma\})$ in order to satisfy the side-condition of the algorithmic [Constr-Alg] rule. Note that, similarly to the rules for type-cases presented in Section 6.3, we only need to keep one such σ: there is no need to return the whole set of solutions Σ as our objective is only to satisfy the side-condition $\Gamma(\mathsf{x})\Sigma \leqslant \tau$, and not to find the smallest possible type for a destructor.

Split backpropagation system The split backpropagation system is extended with this rule:

$$[\textsc{Constr}]\ \frac{}{\Gamma \vdash_{\mathcal{B}} (\mathsf{x} \ltimes \tau : \mathbf{u}) \Rightarrow \{\{\mathsf{x} : \mathbf{u}\}\}}$$

For an atom $\mathsf{x} \ltimes \tau$ to have type $\mathbf{u}$, we just need x to have type $\mathbf{u}$. Actually, x should also have type τ (otherwise the atom would be untypeable), but there is no need to check that in this rule as the $\vdash_{\mathcal{B}}$ system is only called on atoms that have already been typed.

Encoding in the type system

Now that our intermediate language has let-bindings and type constraints, we can encode extended type-cases of the source language into the intermediate language by extending the $(\!|.|\!)$ transformation (Definition 73) with the following case:

$$\begin{aligned}
&(\!|\mathsf{tcase}\, e\, \mathsf{of}\, \tau_1 \to e_1 \mid \ldots \mid \tau_n \to e_n|\!) \\
&\qquad = \mathsf{let}\, ((\!|e|\!) \ltimes \textstyle\bigvee_{i\in 1..n} \tau_i)\, \mathsf{in}\, \mathsf{c}_{(\!|e|\!)}(\tau_1 \to (\!|e_1|\!)\,;\ldots;\,\tau_n \to (\!|e_n|\!)) \\
&\text{with}\quad \mathsf{c}_\xi(\tau \to \xi') = \xi' \\
&\quad \mathsf{c}_\xi(\tau \to \xi';\, C) = (\xi \in \tau)\,?\,\xi' : \mathsf{c}_\xi(C)
\end{aligned}$$

Basically, an extended type-case $\mathsf{tcase}\, e\, \mathsf{of}\, \tau_1 \to e_1 \mid \ldots \mid \tau_n \to e_n$ is encoded in two steps. First, we ensure that the expression e reduces to a value v that is matched by at least one pattern. This is done by the $((\!|e|\!) \ltimes \bigvee_{i\in 1..n} \tau_i)$ expression. Second, we select the first branch that applies using several consecutive regular type-cases: this is done by the $\mathsf{c}_{(\!|e|\!)}(\tau_1 \to (\!|e_1|\!)\,;\ldots;\,\tau_n \to (\!|e_n|\!))$ expression which transforms a type-case with multiple branches into a sequence of ternary type-cases nested on the right. If no pattern can capture v, then the last branch would be selected, but this case cannot happen thanks to the first step.

7.5 Pattern matching

Pattern matching is a fundamental feature of functional languages, and even some dynamic languages such as Python have finally implemented it (Python documentation, 2021). In this section, we add this feature to our source language by encoding it using let-bindings and extended type-cases.

Syntax and semantics

We first define patterns and some operations on them:

Definition 74 (Patterns)**.** *The set of* patterns *is the set of finite terms produced by the following grammar:*

$$\textbf{Patterns} \quad p \quad ::= \quad \tau \mid x \mid p\&p \mid p \mid p \mid (p, p) \mid x := c$$

Definition 75. *Let p be a pattern, and v a value. The substitution generated by the pattern p for the value v, noted v/p, is defined inductively as follows:*

$$\begin{aligned}
v/\tau &= \varnothing && \text{if } v \in \tau \\
v/x &= \{v/x\} && \\
v/(p_1\&p_2) &= (v/p_1) \cup (v/p_2) && \text{if } v/p_1 \neq \texttt{fail} \text{ and } v/p_2 \neq \texttt{fail} \\
v/(p_1 \mid p_2) &= v/p_1 && \text{if } v/p_1 \neq \texttt{fail} \\
v/(p_1 \mid p_2) &= v/p_2 && \text{if } v/p_1 = \texttt{fail} \\
(v_1, v_2)/(p_1, p_2) &= (v_1/p_1) \cup (v_2/p_2) && \text{if } v_1/p_1 \neq \texttt{fail} \text{ and } v_2/p_2 \neq \texttt{fail} \\
v/(x := c) &= \{c/x\} && \\
v/p &= \texttt{fail} && \text{otherwise}
\end{aligned}$$

where $\cup$ denotes the union of two substitutions with disjoint domains (if the two substitutions have overlapping domains, then it yields `fail`).

The pattern τ only matches values of type τ and does not capture them. Conversely, the pattern x matches any value and captures it in the variable x.

The pattern $p_1\&p_2$ matches any value v that is matched by p_1 and p_2, and captures both values captured by p_1 and by p_2 in v (it requires the capture variables in p_1 and p_2 to be disjoint).

The pattern $p_1 \mid p_2$ matches any value v that is matched by p_1 or p_2, and captures values captured by p_1 in v if v is matched by p_1, and values captured by p_2 in v otherwise. The set of capture variables in p_1 is equal to the set of capture variables in p_2: expressions featuring a `|` pattern for which it is not the case will not be typeable.

The pattern (p_1, p_2) matches any pair (v_1, v_2), and captures values captured by p_1 in v_1 and values captured by p_2 in v_2. Similarly to $p_1\&p_2$ patterns, it requires the capture variables in p_1 and p_2 to be disjoint.

Finally, the pattern $x := c$ just assigns the constant c to the capture variable x. It can be useful in the presence of a union of patterns: for instance, in the pattern $(x, \mathbb{1})\,|\,(x := \texttt{nil})$, the variable x captures the first component of the matched value if this value is a pair, and otherwise it captures the constant `nil`.

The next operator we define, $\lbag p \rbag$, computes the type of values matched by a pattern p. In other words, it returns a test type τ such that $\forall v \notin \tau.\ v/p = \texttt{fail}$ (and such that $\forall v \in \tau.\ v/p \neq \texttt{fail}$, provided that the pattern p does not have conflicting capture variables).

Definition 76. *Let p be a pattern. The type captured by the pattern p, noted $\lbag p \rbag$, is defined inductively on p as follows:*

$$\begin{aligned}
\lbag \tau \rbag &= \tau \\
\lbag x \rbag &= \mathbb{1} \\
\lbag p_1 \& p_2 \rbag &= \lbag p_1 \rbag \wedge \lbag p_2 \rbag \\
\lbag p_1 \,|\, p_2 \rbag &= \lbag p_1 \rbag \vee \lbag p_2 \rbag \\
\lbag (p_1, p_2) \rbag &= \lbag p_1 \rbag \times \lbag p_2 \rbag \\
\lbag x := c \rbag &= \mathbb{1}
\end{aligned}$$

With these operators defined, it is now straightforward to add pattern-matching to our language and define its semantics. This is formalized in Figure 7.4.

This definition of pattern-matching is borrowed from Frisch (2004), with some additional restrictions: (i) the same restriction applies on test types τ as for type-cases (the only arrow type allowed is $\mathbb{0} \to \mathbb{1}$), and (ii) we do not support recursive patterns[2].

Encoding in the type system

In terms of typing, we do not need to extend the type system: we can encode pattern-matching using let-bindings and extended type-cases. The idea is to transform each branch of the pattern-matching into a branch of an extended type-case, where every capture variable in the associated pattern is extracted and introduced using let-bindings.

First, we need a way to extract the value captured by a variable in a pattern:

[2] In Frisch (2004), patterns are extended with a recursive construct. For instance, the pattern $p = (x\&\texttt{Int}, p)\,|\,(\mathbb{1}, p)\,|\,(x := \texttt{nil})$ captures in any list, encoded as nested pairs terminating with a `nil` value, the sublist of its integer elements.

Syntax

$$
\begin{array}{lrcl}
\textbf{Expression} & e & ::= & c \mid x \mid \lambda x.e \mid e\ e \mid (e,e) \mid \pi_i e \mid (e{\in}\tau)\,?\,e : e \mid \mathtt{let}\, x = e\, \mathtt{in}\, e \\
& & & \mid (\mathtt{tcase}\, e\, \mathtt{of}\, \tau \to e \mid \ \ldots \ \mid \tau \to e) \\
& & & \mid (\mathtt{match}\, e\, \mathtt{with}\, p \to e \mid \ \ldots \ \mid p \to e) \\
\textbf{Value} & v & ::= & c \mid \lambda x.e \mid (v, v)
\end{array}
$$

Additional reduction rules

For every $n \in \mathbb{N}^*$ and $k \in 1\,..\,n$:

$$
\mathtt{match}\, v\, \mathtt{with}\, p_1 \to e_1 \mid \ \ldots \ \mid p_n \to e_n \quad \rightsquigarrow \quad e_k(v/p_k) \qquad \begin{array}{l} \text{if } v \in \wr p_k \int \setminus (\bigvee_{i \in 1..k-1} \wr p_i \int) \\ \text{and } v/p_k \neq \mathtt{fail} \end{array}
$$

Evaluation Context

$$
\begin{array}{rcl}
E & ::= & [\,] \mid v\ E \mid E\ e \mid (v, E) \mid (E, e) \mid \pi_i E \mid (E{\in}\tau)\,?\,e : e \mid \mathtt{let}\, x = E\, \mathtt{in}\, e \\
& & \mid (\mathtt{tcase}\, E\, \mathtt{of}\, \tau \to e \mid \ \ldots \ \mid \tau \to e) \\
& & \mid (\mathtt{match}\, E\, \mathtt{with}\, p \to e \mid \ \ldots \ \mid p \to e)
\end{array}
$$

Figure 7.4: Syntax and semantics of the language with pattern-matching

Definition 77. *For a pattern p, an expression e and a variable x, we define the expression $\mathsf{d}_x(p, e)$ inductively on p as follows:*

$$
\begin{array}{rcll}
\mathsf{d}_x(x, e) & = & e & \\
\mathsf{d}_x(x := c, e) & = & c & \\
\mathsf{d}_x((p_1, p_2), e) & = & \mathsf{d}_x(p_1, \pi_1 e) & \text{if } x \in \mathsf{vars}(p_1), x \notin \mathsf{vars}(p_2) \\
\mathsf{d}_x((p_1, p_2), e) & = & \mathsf{d}_x(p_2, \pi_2 e) & \text{if } x \in \mathsf{vars}(p_2), x \notin \mathsf{vars}(p_1) \\
\mathsf{d}_x(p_1 \& p_2, e) & = & \mathsf{d}_x(p_1, e) & \text{if } x \in \mathsf{vars}(p_1), x \notin \mathsf{vars}(p_2) \\
\mathsf{d}_x(p_1 \& p_2, e) & = & \mathsf{d}_x(p_2, e) & \text{if } x \in \mathsf{vars}(p_2), x \notin \mathsf{vars}(p_1) \\
\mathsf{d}_x(p_1 \mid p_2, e) & = & (e{\in}\wr\, p_1 \,\int)\,?\,\mathsf{d}_x(p_1, e) : \mathsf{d}_x(p_2, e) & \\
\mathsf{d}_x(p, e) & = & \text{undefined} & \text{otherwise}
\end{array}
$$

Intuitively, the operator $\mathsf{d}_x(p, e)$ returns an expression that extracts from the expression e the value captured by x in the pattern p. For instance, $\mathsf{d}_{x_1}((x_1, x_2), e)$ returns $\pi_1 e$.

Now, we extend the transformation $(\!|.|\!)$ to encode pattern-matching of the source language into the intermediate language. This yields the following case:

$$
\begin{array}{l}
(\!|\mathtt{match}\, e\, \mathtt{with}\, p_1 \to e_1 \mid \ \ldots \ \mid p_n \to e_n|\!) \\
\qquad = (\!|\mathtt{let}\, x = e\, \mathtt{in}\, \mathtt{tcase}\, x\, \mathtt{of}\ \wr\, p_1 \int \to e'_1 \mid \ \ldots \ \mid \ \wr\, p_n \int \to e'_n\,|\!)
\end{array}
$$

where x is a fresh variable, and where for every $i \in 1 \,..\, m$,

$$e'_i \stackrel{\text{def}}{=} \texttt{let}\, x_1 = \mathsf{d}_{x_1}(p_i, x)\, \texttt{in} \,\ldots\, \texttt{let}\, x_m = \mathsf{d}_{x_m}(p_i, x)\, \texttt{in}\, e_i$$

with $x_1, ..., x_m$ the variables appearing in p_i.

Note that, in this definition, $(\!|.|\!)$ is called recursively on an expression of the source language that is not a strict subexpression of the initial expression. Still, it is a well-founded inductive definition: each recursive call in the definition of $(\!|.|\!)$ is either on a strict subexpression, or it makes the number of pattern-matching expressions to decrease.

When this transformation $(\!|.|\!)$ is undefined, then we consider that the source expression is untypeable. This can happen, for instance, if the source expression contains an invalid pattern such as $x\&x$ (or more generally, a pattern p such that $\forall v.\ v/p = \texttt{fail}$).

Chapter 8

Practical Aspects

Contents

In Chapters 5 and 6 respectively, we have formalized an algorithmic type system and an algorithm to reconstruct annotation trees. Combined, they allow us to infer the type of expressions of the source language. However, though the reconstruction algorithm is terminating, a naive implementation would be unpractical due to performance issues. We present now several high-level optimizations that we implemented to mitigate the high branching factor of our backtracking algorithm.

A first source of inefficiency comes from the generation of intersection nodes when a destructor is reconstructed. This can lead to an explosion of the number of branches to explore, even though many of them may be redundant. This can be mitigated by trimming branches when we estimate that they will not contribute to make the final type strictly smaller. This optimization is explored in Section 8.1.

Another source of inefficiency comes from the type decomposition performed after each binding. Although these type decompositions are usually small (e.g., the type of a binding is seldom split in more than two parts), it becomes an issue when typing large expressions with multiple type-cases, since all the decompositions of successive bindings compose one with another, yielding an exponential explosion of the number of cases to explore. This is addressed in Section 8.2.

A third difficulty comes from the complexity of the operations on set-theoretic types (subtyping, tallying, etc.). The types manipulated during the reconstruction may get more and more complex, making these type operations slower. Performing simplifications of types and tallying solutions can thus have a significant positive impact on performance. This is discussed in Section 8.3.

Finally, in Section 8.4, we discuss the use of memoization techniques in order to prevent the reconstruction algorithm from retyping the same subexpression multiple times under equivalent contexts.

Note that the implementation of set-theoretic types and the associated type operators (subtyping, DNF, tallying, etc.) are not discussed here. Efficient algorithms for these operators are described in Castagna et al. (2015) and Castagna (2020). An efficient implementation of these operators in OCaml can also be found in the polymorphic version of CDuce.

8.1 Intersection nodes pruning

8.1.1 An explosion of the number of branches

While the reconstruction algorithm presented in Chapter 6 is guaranteed to terminate, the number of explored branches can explode if we implement it naively.

Consider the following example, written in an OCaml-like syntax where typecases are noted `if` e `is` t `then` e_1 `else` e_2 (this is the syntax used by our prototype implementation, which will be presented in Chapter 9):

```
let typeof arg =
    if arg is Int then "number"
    else if arg is Bool then "boolean"
    else if arg is Unit then "unit"
    else if arg is String then "string"
    else "object"
```

This function `typeof` is inspired by JavaScript's `typeof` operator. The expected type for this function is:

$$(\texttt{Int} \to \text{"number"}) \wedge (\texttt{Bool} \to \text{"boolean"}) \wedge (\texttt{Unit} \to \text{"unit"}) \wedge$$
$$(\texttt{String} \to \text{"string"}) \wedge (\mathbb{1} \backslash (\texttt{Int} \vee \texttt{Bool} \vee \texttt{Unit} \vee \texttt{String}) \to \text{"object"})$$

The corresponding MSC form is the following (instead of defining a binding for each string constant, they have been inlined for concision):

$\lambda x.$
```
    bind x = x in
    bind x1 = (x∈String)?"string":"object" in
    bind x2 = (x∈Unit)?"unit":x1 in
    bind x3 = (x∈Bool)?"boolean":x2 in
    bind x4 = (x∈Int)?"number":x3 in
    x4
```

During the reconstruction, x is initially given the type $\boldsymbol{\alpha}$. The first binding definition to be explored is the definition of $\texttt{x}_4$, triggering the reconstruction of `x` (Rule [CaseVar]), which is typed $\boldsymbol{\alpha}$, and then decomposing this type into two parts, namely `Int` and $\neg$`Int` (Rule [CaseSplit]). While exploring the `Int` part, the [CaseThen] rule is applied on the definition of $\texttt{x}_4$, yielding a `Subst` result that generates two branches (regrouped by an intersection annotation at the root of the annotation tree):

1. One issued from the substitution $\{\boldsymbol{\alpha} \rightsquigarrow \boldsymbol{\alpha}\backslash\texttt{Int}\}$ and for which, when exploring the `Int` part for `x`, the type-case is typed using the rule [$\mathbb{0}$-Alg].
2. The other issued from the identity substitution $\varnothing$ (the default case) and for which, when exploring the `Int` part for `x`, the type-case is typed using the rule [$\in_1$-Alg].

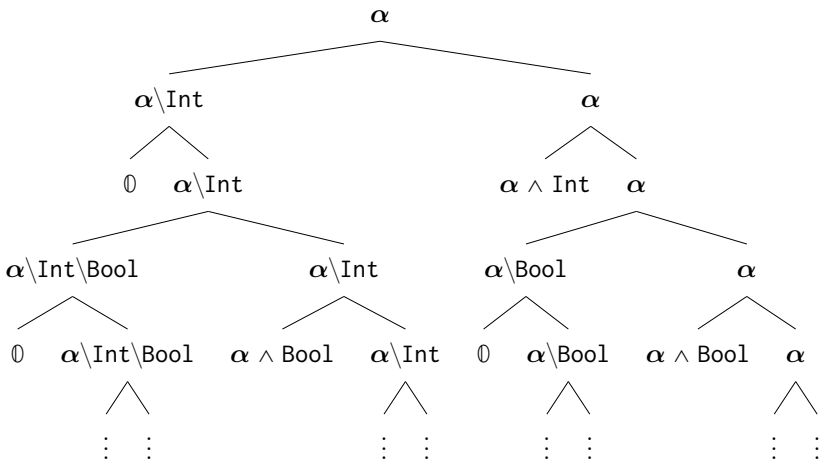
In both branches, the exploration of the part $\neg$`Int` for `x` yields two branches again, this time when applying the rule [CaseElse] on the definition of $\texttt{x}_4$:

1. One issued from the substitution $\{\boldsymbol{\alpha} \rightsquigarrow \boldsymbol{\alpha} \wedge \texttt{Int}\}$ and for which, when exploring the $\neg$`Int` part for `x`, the type-case is typed using the rule [$\mathbb{0}$-Alg].
2. The other issued from the identity substitution $\varnothing$ (the default case) and for which, when exploring the $\neg$`Int` part for `x`, the type-case is typed using the rule [$\in_2$-Alg].

We end up with 4 branches:

1. One where `x` has type $\mathbb{0}$ and where the type-case is typed with [$\mathbb{0}$-Alg] for both parts `Int` and $\neg$`Int`.
2. One where `x` has type $\boldsymbol{\alpha}\backslash\texttt{Int}$ and where the type-case is typed with [$\mathbb{0}$-Alg] for the part `Int` and with [$\in_2$-Alg] for the part $\neg$`Int`.
3. One where `x` has type $\boldsymbol{\alpha} \wedge \texttt{Int}$ and where the type-case is typed with [$\in_1$-Alg] for the part `Int` and with [$\mathbb{0}$-Alg] for the part $\neg$`Int`.
4. One where `x` has type $\boldsymbol{\alpha}$ and where the type-case is typed with [$\in_1$-Alg] for the part `Int` and with [$\in_2$-Alg] for the part $\neg$`Int`.

While the exploration of the branches 1 and 3 does not yield any new branch, branches 2 and 4 require typing $\texttt{x}_3$ and thus exploring them will again generate four branches each. This process continues with $\texttt{x}_2$ and $\texttt{x}_1$. We can see with this example that the number of cases to explore grows exponentially. The tree below illustrates this explosion: each node corresponds to an intersection node in the derivation tree, and is labeled with the domain of `typeof` that it captures at the moment it is generated.



This explosion of the number of branches is problematic. Though, many of these branches are redundant: our objective is thus to eliminate them. In particular, among the nodes at depth 2 in the tree above ($\mathbb{0}$, $\boldsymbol{\alpha}\backslash\mathtt{Int}$, $\boldsymbol{\alpha} \wedge \mathtt{Int}$, $\boldsymbol{\alpha}$), it is only necessary to explore $\boldsymbol{\alpha}\backslash\mathtt{Int}$ and $\boldsymbol{\alpha} \wedge \mathtt{Int}$: the branch with the domain $\mathbb{0}$ can only yield for this function the type $\mathbb{0} \to \mathbb{1}$ (the supertype of all functions), and the domain $\boldsymbol{\alpha}$ is already covered by the union of the two more specific branches $\boldsymbol{\alpha}\backslash\mathtt{Int}$ and $\boldsymbol{\alpha} \wedge \mathtt{Int}$ (i.e., we have $\boldsymbol{\alpha} \leqslant (\boldsymbol{\alpha} \wedge \mathtt{Int}) \vee (\boldsymbol{\alpha}\backslash\mathtt{Int})$).

The generation of redundant branches can also originate from applications. For instance, consider the canonical form $\lambda x.\ \mathtt{bind\,x=}x\,\mathtt{in\ f\ x}$ with $\mathtt{f} : (\mathtt{Int} \to \mathtt{Int}) \wedge (\mathtt{Bool} \to \mathtt{Bool})$. When calling the reconstruction on the application $\mathtt{f\ x}$, with $\mathtt{x}$ initially typed $\boldsymbol{\alpha}$, the [AppInfer] rule generates three substitutions, each corresponding to a solution to the associated tallying instance:

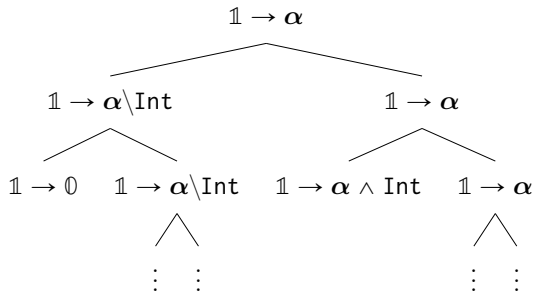
1. $\{\boldsymbol{\alpha} \rightsquigarrow \boldsymbol{\alpha} \wedge (\mathtt{Bool} \vee \mathtt{Int})\}$: in this case, the result is of type $\mathtt{Int} \vee \mathtt{Bool}$,
2. $\{\boldsymbol{\alpha} \rightsquigarrow \boldsymbol{\alpha} \wedge \mathtt{Int}\}$: in this case, the result is of type $\mathtt{Int}$,
3. $\{\boldsymbol{\alpha} \rightsquigarrow \boldsymbol{\alpha} \wedge \mathtt{Bool}\}$: in this case, the result is of type $\mathtt{Bool}$.

Each substitution is then applied to the environment and explored in a separate branch. The first branch yields the type $\mathtt{Bool} \vee \mathtt{Int} \to \mathtt{Bool} \vee \mathtt{Int}$ for our λ-abstraction, while the two others yield the types $\mathtt{Int} \to \mathtt{Int}$ and $\mathtt{Bool} \to \mathtt{Bool}$ respectively. The first branch is thus redundant as $(\mathtt{Int} \to \mathtt{Int}) \wedge (\mathtt{Bool} \to \mathtt{Bool}) \leqslant \mathtt{Bool} \vee \mathtt{Int} \to \mathtt{Bool} \vee \mathtt{Int}$. We will see in the next section how we can detect and trim those redundant branches.

Unfortunately, sometimes the combinatorics explosion cannot be avoided. For instance, consider this slightly modified version of the `typeof` example, where x is a variable of type $\mathbb{1}$ already in the context:

```
let typeof_image f =
    if f x is Int then "number"
    else if f x is Bool then "boolean"
    else if f x is Unit then "unit"
    else if f x is String then "string"
    else "object"
```

This time, the same explosion happens (the reasoning is similar) but the domains explored by the branches are different:



Here, none of the nodes at depth 2 is redundant. In particular, the domain $\mathbb{1} \to \boldsymbol{\alpha}$ is not covered by the union of the two domains $\mathbb{1} \to \boldsymbol{\alpha}\backslash\texttt{Int}$ and $\mathbb{1} \to \boldsymbol{\alpha} \wedge \texttt{Int}$: the type $(\mathbb{1} \to \boldsymbol{\alpha} \wedge \texttt{Int}) \vee (\mathbb{1} \to \boldsymbol{\alpha}\backslash\texttt{Int})$ is strictly smaller than $\mathbb{1} \to \boldsymbol{\alpha}$ (it becomes clear when substituting $\boldsymbol{\alpha}$ by $\mathbb{1}$: the type $(\mathbb{1} \to \texttt{Int}) \vee (\mathbb{1} \to \neg\texttt{Int})$ does not contain functions that return integers for some inputs and Boolean values for some others inputs, whereas the type $\mathbb{1} \to \mathbb{1}$ does).

8.1.2 A heuristic for trimming redundant branches

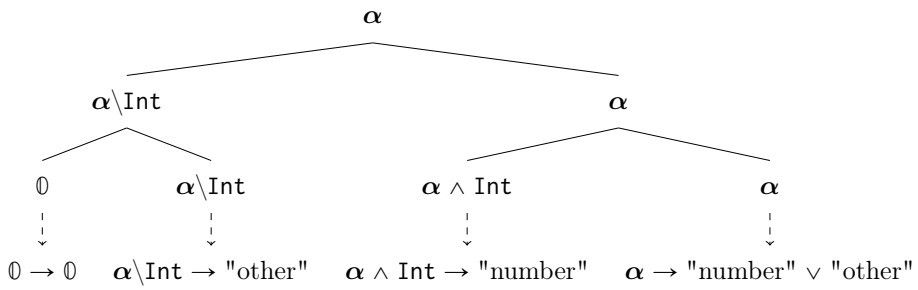
We see in this section a heuristic to trim branches that do not contribute to obtain a smaller type. It consists in three steps: (*i*) we explore "more specific" branches first, (*ii*) we remember, for each branch, the domain it covers for each λ-abstraction, and (*iii*) before exploring a branch, we check if this branch explores a domain that has not been covered yet (if not, it will be trimmed).

8.1.2.1 Order of exploration

Let us consider a simplified version of the `typeof` example above:

```
let typeof_simplified arg =
    if arg is Int then "number"
    else "other"
```

When reconstructing the annotation tree of `typeof_simplified`, we get the following branches (again, each node represents an intersection node and is labeled by the domain it captures at the moment of its generation):



The type under the dashed arrow is the type we obtain for each branch (note that we only know this type after having finished the reconstruction for the branch). The situation is similar to the one of the `typeof` example in the previous section: the rightmost branch does not contribute to the final type, as the intersection of the other branches yields a smaller type: we have $(\boldsymbol{\alpha} \wedge \texttt{Int} \to \texttt{"number"}) \wedge (\boldsymbol{\alpha}\backslash\texttt{Int} \to \texttt{"other"}) \leqslant \boldsymbol{\alpha} \to \texttt{"number"} \vee \texttt{"other"}$. However, when this branch is generated, we do not know yet the type it will yield.

Still, we know that it will yield a less precise type that its neighbor branch $\boldsymbol{\alpha} \wedge \texttt{Int}$ for the atom from which their parent intersection node originates, that is, the atom corresponding to the type-case `if arg is Int then "number" else`

`"other"`. Indeed, the $\boldsymbol{\alpha}$ branch requires to type both branches of the type-case, while the $\boldsymbol{\alpha} \wedge \mathtt{Int}$ branch allows skipping the second one. Consequently, we choose to first explore the more specific $\boldsymbol{\alpha} \wedge \mathtt{Int}$ branch, and only then, if we estimate that the branches already explored do not cover all the possible domains, we will explore the $\boldsymbol{\alpha}$ branch.

We apply the same idea to the intersection node at the root. In the end, the leaves are explored in the following order: $\mathbb{0}$, then $\boldsymbol{\alpha}\backslash\mathtt{Int}$, then $\boldsymbol{\alpha} \wedge \mathtt{Int}$, and finally $\boldsymbol{\alpha}$. Here, every intersection node originates from a type-case, but the same idea can be applied for intersection nodes coming from an application: among the solutions to the tallying instance, if we know that one of them yields a smaller type than another one for the application at issue, then the corresponding branch is explored first.

Formally, we change the intermediate annotations for intersections, so that sets are replaced by ordered lists:

$$\begin{array}{llll} \textbf{Atom intermediate annot.} & \mathcal{A} & ::= & \ldots \mid \bigwedge((\mathcal{A}\ ;\ \ldots\ ;\ \mathcal{A}), (\mathcal{A}\ ;\ \ldots\ ;\ \mathcal{A})) \\ \textbf{Form intermediate annot.} & \mathcal{K} & ::= & \ldots \mid \bigwedge((\mathcal{K}\ ;\ \ldots\ ;\ \mathcal{K}), (\mathcal{K}\ ;\ \ldots\ ;\ \mathcal{K})) \end{array}$$

We update the reconstruction rules for intersections accordingly, so that branches are explored following the order of the list. The [ITERATE$_2$] rule, that generates intersection nodes, is updated accordingly:

$$\textbf{Result} \quad \mathbb{R} \quad ::= \quad \ldots \mid \mathsf{Subst}((\psi\ ;\ \ldots\ ;\ \psi), \mathcal{H}, \mathcal{H})$$

$$[\textsc{Iterate}_2]\ \frac{\begin{array}{c}\Gamma \vdash_{\mathcal{R}} \langle \eta \mid \mathcal{H} \rangle \Rightarrow \mathsf{Subst}((\psi_1\ ;\ \ldots\ ;\ \psi_n), \mathcal{H}_1, \mathcal{H}_2) \\ \Gamma \vdash^*_{\mathcal{R}} \langle \eta \mid \bigwedge((\mathcal{H}_1\psi_1\ ;\ \ldots\ ;\ \mathcal{H}_1\psi_n\ ;\ \mathcal{H}_2), \varepsilon) \rangle \Rightarrow \mathbb{R}\end{array}}{\Gamma \vdash^*_{\mathcal{R}} \langle \eta \mid \mathcal{H} \rangle \Rightarrow \mathbb{R}}\ \forall i \in 1\,..\,n.\ \psi_i \# \Gamma$$

Note that the default case $\mathcal{H}_2$ is inserted at the last position in the intersection.

Finally, we update the rules that generate a `Subst` result so that they give a suitable order to the substitutions: when a rule [PROJINFER] or [APPINFER] generates a $\mathsf{Subst}((\psi_1\ ;\ \ldots\ ;\ \psi_n), \mathcal{H}_1, \mathcal{H}_2)$ result, the substitutions ψ_i should be ordered by increasing estimated result type for the $\lhd$ order (if the current atom is expected to have a smaller type under the environment $\Gamma\psi_i$ compared to $\Gamma\psi_j$, then ψ_i should precede ψ_j). We formalize it for the rule [APPINFER] below (other rules such as [PROJINFER] are similar):

$$[\textsc{AppInfer}]\ \frac{\{\psi_i\}_{i\in 1..n} = \mathsf{tally\_infer}(\Gamma(\mathsf{x}_1) \mathrel{\dot{\leqslant}} \Gamma(\mathsf{x}_2) \to \boldsymbol{\alpha})}{\Gamma \vdash_{\mathcal{R}} \langle \mathsf{x}_1\mathsf{x}_2 \mid \mathtt{infer} \rangle \Rightarrow \mathsf{Subst}((\psi'_1\ ;\ \ldots\ ;\ \psi'_n), \mathsf{typ}, \mathsf{untyp})}\ \boldsymbol{\alpha} \in \mathcal{V}_M \text{ fresh}$$

with $\forall i \in 1\,..\,n,\ \psi'_i \stackrel{\text{def}}{=} \psi_i|_{\mathcal{V}_M \setminus \{\boldsymbol{\alpha}\}}$, and such that:

$$\forall i \in 1\,..\,n.\ \forall j \in i+1\,..\,n.\ \mathsf{gen}(\boldsymbol{\alpha}\psi_i) \lhd_{\mathsf{T}} \boldsymbol{\alpha}\psi_j \text{ or } \mathsf{gen}(\boldsymbol{\alpha}\psi_j) \not\lhd_{\mathsf{T}} \boldsymbol{\alpha}\psi_i$$

where $\mathsf{gen}(t)$ renames all monomorphic type variables in t with fresh polymorphic ones, and where $\lhd_{\mathsf{T}}$ is defined as follows:

Definition 78. *We define the binary relation $\lhd_{\mathsf{T}}$ as follows:*

$$t_1 \lhd_{\mathsf{T}} t_2 \quad \stackrel{\text{def}}{\iff} \quad \exists \sigma.\ t_1\sigma \leqslant t_2$$

This relation $\lhd_{\mathsf{T}}$ (where the subscript T stands for "Tallying") intuitively corresponds to an approximation of the relation $\lhd$ that can be decided using the following tallying instance:

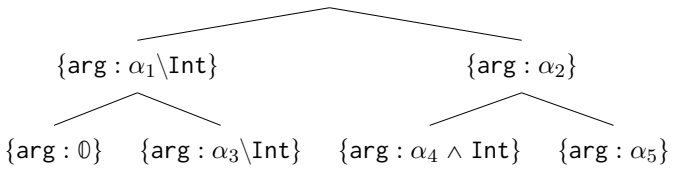
$$t_1 \lhd_{\mathsf{T}} t_2 \quad \iff \quad \mathsf{tally}(\{t_1 \stackrel{\cdot}{\leqslant} \mathsf{mono}(t_2)\}) \neq \varnothing$$

where $\mathsf{mono}(t)$ renames all polymorphic type variables in t with fresh monomorphic ones.

8.1.2.2 Trimming redundant branches

Now that we have determined an order of exploration for branches of intersection nodes, we have to decide when a pending branch should be trimmed. We do that by estimating whether it allows exploring new domains or not. To be able to do such an estimation, whenever a $\mathtt{Subst}((\psi_1\ ;\ \ldots\ ;\ \psi_n), \mathcal{H}_1, \mathcal{H}_2)$ result is generated, it is decorated with the domains of the λ-abstractions it crosses while backtracking, and this information is then stored into the generated intersection node.

Intuitively, we want to label branches of intersection nodes similarly as the nodes of the trees we have drawn in the previous section, but in a more general setting where there might be several λ-abstractions. For instance, here is the intersection tree that we want to build for `typeof_simplified`:



To achieve this, an extra parameter Γ is added to `Subst` results:

$$\textbf{Result} \quad \mathbb{R} \quad ::= \quad \ldots \mid \mathtt{Subst}((\psi\ ;\ \ldots\ ;\ \psi), \Gamma, \mathcal{H}, \mathcal{H})$$

When a `Subst` result is generated, this parameter is initially set to $\varnothing$:

$$[\textsc{AppInfer}]\ \frac{\{\psi_i\}_{i\in 1..n} = \mathsf{tally\_infer}(\Gamma(\mathsf{x}_1) \stackrel{\cdot}{\leqslant} \Gamma(\mathsf{x}_2) \to \boldsymbol{\alpha})}{\Gamma \vdash_{\mathcal{R}} \langle \mathsf{x}_1\mathsf{x}_2 \mid \mathtt{infer}\rangle \Rightarrow \mathtt{Subst}((\psi'_1\ ;\ \ldots\ ;\ \psi'_n), \varnothing, \mathsf{typ}, \mathsf{untyp})}\ \boldsymbol{\alpha} \in \mathcal{V}_M$$

Then, this parameter is populated when leaving a λ-abstraction while backtracking:

$$[\textsc{LambdaSubst}]\ \frac{\Gamma, x : \mathbf{u} \vdash^*_{\mathcal{R}} \langle \kappa \mid \mathcal{K}\rangle \Rightarrow \mathtt{Subst}(L, \Gamma', \mathcal{K}_1, \mathcal{K}_2)}{\Gamma \vdash_{\mathcal{R}} \langle \lambda x.\kappa \mid \lambda(\mathbf{u}, \mathcal{K})\rangle \Rightarrow \mathtt{Subst}(L, (\Gamma', x : \mathbf{u}), \lambda(\mathbf{u}, \mathcal{K}_1), \lambda(\mathbf{u}, \mathcal{K}_2))}$$

Finally, when an intersection is created from this `Subst` result by a [ITERATE$_2$] rule, each branch $\mathcal{H}_1\psi_i$ is decorated with the associated environment $\Gamma'\psi_i$, and the default branch $\mathcal{H}_2$ is decorated with Γ':

$$\begin{array}{lrcl}
\textbf{Decorated atom annot.} & \mathcal{A}^\circ & ::= & (\mathcal{A}, \Gamma) \\
\textbf{Atom annotations} & \mathcal{A} & ::= & \ldots \mid \bigwedge((\mathcal{A}^\circ\ ;\ \ldots\ ;\ \mathcal{A}^\circ), (\mathcal{A}^\circ\ ;\ \ldots\ ;\ \mathcal{A}^\circ)) \\
\textbf{Decorated form annot.} & \mathcal{K}^\circ & ::= & (\mathcal{K}, \Gamma) \\
\textbf{Form annotations} & \mathcal{K} & ::= & \ldots \mid \bigwedge((\mathcal{K}^\circ\ ;\ \ldots\ ;\ \mathcal{K}^\circ), (\mathcal{K}^\circ\ ;\ \ldots\ ;\ \mathcal{K}^\circ))
\end{array}$$

$$[\text{ITERATE}_2]\ \frac{\begin{array}{c}\Gamma \vdash_{\mathcal{R}} \langle \eta \mid \mathcal{H} \rangle \Rightarrow \mathtt{Subst}((\psi_1\ ;\ \ldots\ ;\ \psi_n), \Gamma', \mathcal{H}_1, \mathcal{H}_2) \\ \forall i \in 1\,..\,n.\ \mathcal{H}_i^\circ = (\mathcal{H}_1\psi_i, \mathsf{gen}_\Gamma(\Gamma'\psi_i)) \\ \Gamma \vdash^*_{\mathcal{R}} \langle \eta \mid \bigwedge((\mathcal{H}_1^\circ\ ;\ \ldots\ ;\ \mathcal{H}_n^\circ\ ;\ (\mathcal{H}_2, \mathsf{gen}_\Gamma(\Gamma'))), \varepsilon) \rangle \Rightarrow \mathbb{R}\end{array}}{\Gamma \vdash^*_{\mathcal{R}} \langle \eta \mid \mathcal{H} \rangle \Rightarrow \mathbb{R}}\ \forall i \in 1\,..\,n.\ \psi_i \# \Gamma$$

where $\mathsf{gen}_\Gamma(\Gamma')$ is obtained by substituting in Γ' all monomorphic type variables in $\mathsf{vars}(\Gamma')\backslash\mathsf{vars}(\Gamma)$ by fresh polymorphic ones.

Now that branches of intersection nodes are labelled with an environment representing the domains they cover, we remember the labels corresponding to the branches that we have already explored while walking through the annotation tree. More precisely, we parametrize our reconstruction algorithm by a set of environments $\mathbb{\Gamma}$. When reconstructing an intersection node, we reconstruct its first branch, add its label to $\mathbb{\Gamma}$, then reconstruct the next branch, and so on. For instance, for the intersection tree of `typeof_simplified`, before exploring the rightmost leaf, we should have collected the following set of environments $\mathbb{\Gamma}$:

$$\mathbb{\Gamma} \quad = \quad \{\ \{\mathtt{arg} : \alpha_1 \backslash \mathtt{Int}\}\ ,\ \{\mathtt{arg} : \alpha_4 \wedge \mathtt{Int}\}\ \}$$

The first environment comes from the label of the left branch of the intersection node at the root (this left branch should have been fully explored before the right one according to the previous section), and the second environment comes from the label of the left branch of the $\{\mathtt{arg} : \alpha_2\}$ intersection node.

Before exploring a branch of an intersection node, the set of environments $\mathbb{\Gamma}$ is compared, using several tallying instances, with the label Γ of this branch. We proceed in two steps: (*i*) $\mathbb{\Gamma}$ is filtered in order to only keep environments that are "more specific" than Γ, then (*ii*) we determine whether the "union" of the remaining environments in $\mathbb{\Gamma}$ covers Γ.

The purpose of the first step is to consider, among the branches that have already been explored, only those that have been generated from a type-case or application in the same λ-abstraction as the pending branch, and for which the body of this λ-abstraction is typed with a smaller type than it will (probably) be in the pending branch. Indeed, if the label Γ' of a previously-explored branch is smaller than the label Γ of the pending branch, it means that the body of the λ-abstraction that triggered these branches has been typed under a smaller context and, thus, that it should yield a smaller type. In other words, we want to select, among the explored branches, those that were generated in the same λ-abstraction as the pending branch

and that type the body of this λ-abstraction more precisely. This filtering is achieved as follows:

1. $\mathbb{\Gamma}$ is first filtered by removing any $\Gamma' \in \mathbb{\Gamma}$ such that $\mathsf{dom}(\Gamma') \neq \mathsf{dom}(\Gamma)$. This has as effect to filter out every previously-explored branch that does not originate from the same λ-abstraction as the pending branch.

2. Then, for each remaining $\Gamma' \in \mathbb{\Gamma}$, we generate the following tallying instance:

$$\mathsf{tally}(\{\Gamma'(x) \mathrel{\dot{\leqslant}} \Gamma(x) \mid x \in \mathsf{dom}(\Gamma)\})$$

   If there is at least one solution, then Γ' is kept, otherwise it is filtered away.

We note $\mathsf{filter}(\Gamma, \mathbb{\Gamma})$ the result of this filtering. In our `typeof_simplified` example, with $\mathbb{\Gamma} = \{\{\mathtt{arg} : \alpha_1 \backslash \mathtt{Int}\}, \{\mathtt{arg} : \alpha_4 \wedge \mathtt{Int}\}\}$ and $\Gamma = \{\mathtt{arg} : \alpha_5\}$, no branch is filtered out (we have $\mathsf{filter}(\Gamma, \mathbb{\Gamma}) = \mathbb{\Gamma}$).

For the second step, which consists in determining whether the environments in this filtered $\mathbb{\Gamma}$ cover the environment Γ of the pending branch, we first define a representation of an environment as a type. Let $\mathsf{encoding}(\Gamma)$ be the type representation of Γ, defined as the following record type:

$$\mathsf{encoding}(\Gamma) \stackrel{\text{def}}{=} \mathbf{\{}\mathsf{label}(x) = \Gamma(x) \mid x \in \mathsf{dom}(\Gamma)\mathbf{\}}$$

with $\mathsf{label}(x)$ an arbitrary fresh label associated to the variable x. Note that, as all our environments have the same fixed domain, we could also define $\mathsf{encoding}(\Gamma)$ using a product type by fixing an arbitrary order over the variables in $\mathsf{dom}(\Gamma)$.

Now, the idea that "the environment Γ can be covered with the set of environments $\mathbb{\Gamma}$" can be encoded with the following formula:

$$\left(\bigvee_{\Gamma' \in \mathsf{filter}(\Gamma, \mathbb{\Gamma})} \mathsf{encoding}(\Gamma')\right) \rhd_{\mathsf{T}} \mathsf{encoding}(\Gamma)$$

where $\rhd_{\mathsf{T}}$ is a binary relation over types defined as follows:

Definition 79. *We define the binary relation $\rhd_{\mathsf{T}}$ as follows:*

$$t_1 \rhd_{\mathsf{T}} t_2 \quad \stackrel{\text{def}}{\iff} \quad \exists \sigma.\ t_1\sigma \geqslant t_2$$

Intuitively, the relation $t_1 \rhd_{\mathsf{T}} t_2$ means that t_1 has a better coverage than t_2. Note that $t_1 \rhd_{\mathsf{T}} t_2$ is not equivalent to $t_2 \lhd_{\mathsf{T}} t_1$ (cf. Definition 78): the first tries to instantiate t_1, while the second tries to instantiate t_2. The relation $t_1 \rhd_{\mathsf{T}} t_2$ can be decided using the following tallying instance:

$$t_1 \rhd_{\mathsf{T}} t_2 \quad \iff \quad \mathsf{tally}(\{\mathsf{mono}(t_2) \mathrel{\dot{\leqslant}} t_1\}) \neq \varnothing$$

where $\mathsf{mono}(t)$ renames all polymorphic type variables in t with fresh monomorphic ones.

If $\bigvee_{\Gamma' \in \mathsf{filter}(\Gamma, \mathbb{\Gamma})} \mathsf{encoding}(\Gamma') \rhd_{\mathsf{T}} \mathsf{encoding}(\Gamma)$ holds, meaning that the domains of the next branch to explore are already covered by the domains of the branches already explored, then we can trim this branch. In our `typeof_simplified` example, the branch labeled with $\Gamma = \{\mathsf{arg} : \alpha_5\}$ is trimmed as we have $\mathbf{\{}\mathsf{arg} = \alpha_1 \backslash \mathtt{Int}\mathbf{\}} \vee \mathbf{\{}\mathsf{arg} = \alpha_4 \wedge \mathtt{Int}\mathbf{\}} \rhd_{\mathsf{T}} \mathbf{\{}\mathsf{arg} = \alpha_5\mathbf{\}}$.

This yields the following rule, to be added to the reconstruction system with higher priority over other rules for intersections:

$$[\textsc{InterTrim}] \quad \frac{\bigvee_{\Gamma' \in \mathsf{filter}(\Gamma_1, \mathbb{\Gamma})} \mathsf{encoding}(\Gamma') \rhd_{\mathsf{T}} \mathsf{encoding}(\Gamma_1) \quad \Gamma \vdash_{\mathcal{R}} \langle \eta \mid \bigwedge((\mathcal{H}_2^\circ \,;\, \ldots \,;\, \mathcal{H}_n^\circ), L) \rangle \Rightarrow \mathbb{R}}{\Gamma \vdash_{\mathcal{R}} \langle \eta \mid \bigwedge(((\mathcal{H}_1, \Gamma_1) \,;\, \ldots \,;\, \mathcal{H}_n^\circ), L) \rangle \Rightarrow \mathbb{R}}$$

where $\mathbb{\Gamma}$ is the set of labels of the branches already explored.

Note that intersection nodes that are created after a user type annotation should not be pruned: in this way, the user can force the exploration of a branch, giving them more control.

8.2 Type decompositions pruning

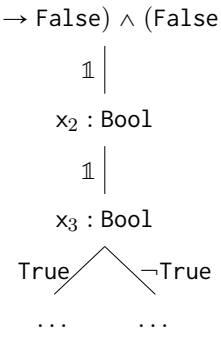
In addition to intersection nodes, branching can also come from type decompositions made after each binding. Let us consider the following expression, with $\mathsf{not} : (\mathtt{True} \to \mathtt{False}) \wedge (\mathtt{False} \to \mathtt{True})$ and $\mathsf{arg} : \mathtt{Bool}$:

```
if not arg is True then arg else false
```

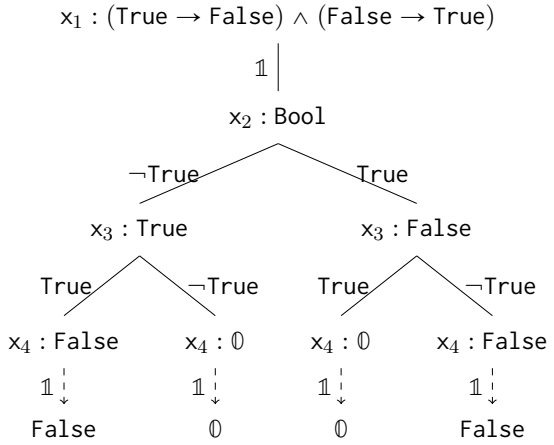
The associated MSC form is the following (the binding for the constant `false` has been inlined for concision):

```
bind x1 = not in
bind x2 = arg in
bind x3 = x1 x2 in
bind x4 = (x3∈True)?x2:false in
x4
```

When reconstructing the definition of x_4, a type decomposition is generated for x_3. This is illustrated by the following tree, where each node represents a binding and the type of the associated definition, and each edge represents a part of the type decomposition performed for this binding:



Then, the type decomposition performed on x_3 is propagated to x_2:



As we can see, some of these branches are useless: for instance, in the case where $\mathsf{x}_3 : \mathsf{True}$, the $\neg\mathsf{True}$ part of the type decomposition of x_3 is useless, as the other part True is enough to cover the whole type of x_3. If we want to avoid exploring such useless parts, we need to allow the type decompositions to cover only the type of the associated definition, instead of always covering $\mathbb{1}$. In this section, we propose some changes in this direction for the algorithmic type system and for the reconstruction algorithm.

The binding annotation $\mathsf{keep}\ (\mathbb{a}, \{(\mathbf{u}, \mathbb{k}), \dots, (\mathbf{u}, \mathbb{k})\})$ is modified to take a set of substitutions Σ as extra parameter:

$$\textbf{Form annotations} \quad \mathbb{k} \quad ::= \quad \dots \mid \mathsf{keep}\ (\mathbb{a}, \{(\mathbf{u}, \mathbb{k}), \dots, (\mathbf{u}, \mathbb{k})\}, \Sigma) \mid \dots$$

Then, the [Bind$_2$-Alg] rule of the algorithmic type system can be modified as follows:

$$[\textsc{Bind}_2\text{-Alg}]\ \frac{\Gamma \vdash_{\mathcal{A}} [a \mid \mathbb{a}] : s \qquad (\forall i \in I)\ \ \Gamma, \mathsf{x} : s \wedge \mathbf{u}_i \vdash_{\mathcal{A}} [\kappa \mid \mathbb{k}_i] : t_i}{\Gamma \vdash_{\mathcal{A}} [\mathsf{bind}\,\mathsf{x} = a\,\mathsf{in}\,\kappa \mid \mathsf{keep}\ (\mathbb{a}, \{(\mathbf{u}_i, \mathbb{k}_i)\}_{i\in I}, \Sigma)] : \bigvee_{i\in I} t_i}\ \begin{array}{l} I \neq \varnothing \\ s\Sigma \leqslant \bigvee_{i\in I} \mathbf{u}_i \end{array}$$

The guard condition $\{\mathbf{u}_i\}_{i\in I} \in \mathsf{Part}(\mathbb{1})$ of the original [Bind$_2$-Alg] rule has been replaced by $s\Sigma \leqslant \bigvee_{i\in I} \mathbf{u}_i$: instead of requiring the type decomposition to cover $\mathbb{1}$, we only need it to cover (any instantiation of) the type of the definition. Note that this instantiation Σ is only used to justify that the type decomposition covers the type of the definition: it is not applied to the type s of the definition when recursively typing the body, in order not to pollute the type of x with instantiations that might be useless. The admissibility of this new [Bind$_2$-Alg] rule follows from the monotonicity of the algorithmic type system (Lemma 34).

Now we can change the reconstruction so that parts of type decompositions that are disjoint from (an instance of) the type of the definition are not explored. We change the intermediate annotations for bindings as follows:

$$\begin{array}{rrcl} \textbf{Form intermediate annot.} & \mathcal{K} & ::= & \dots \mid \mathsf{keep}\ (\mathcal{A}, \mathcal{S}, \mathcal{S}, \mathbb{u}) \\ & & & \mid \mathsf{propagate}\ (\mathcal{A}, \mathbb{\Gamma}, \mathcal{S}, \mathcal{S}, \mathbb{u}) \end{array}$$

with $\mathbb{u}$ a set of monomorphic types representing the parts of the decomposition that will not be explored because they are not necessary to cover the type of the definition.

The `keep` and `propagate` annotations are given this additional parameter $\mathbb{u}$ to keep track of the parts of the decomposition that do not need to be explored.

The rule [BindKeep] of the substitution inference system, for transforming intermediate annotations into annotations for the algorithmic type system, is modified accordingly:

$$[\textsc{BindKeep}]\ \frac{\begin{array}{c}\Gamma \vdash_{\mathcal{S}} \langle a \mid \mathcal{A}\rangle \Rightarrow \mathbb{a} \qquad \Gamma \vdash_{\mathcal{A}} [a \mid \mathbb{a}] : s \\ (\forall i \in I)\ \ \Gamma, \mathsf{x} : s \wedge \mathbf{u}_i \vdash_{\mathcal{S}} \langle \kappa \mid \mathcal{K}_i\rangle \Rightarrow \mathbb{k}_i \\ (\forall j \in J)\ \ \sigma_j \in \mathsf{tally}(\{s \mathrel{\dot{\leqslant}} \neg \mathbf{u}'_j\})\end{array}}{\Gamma \vdash_{\mathcal{S}} \langle \mathsf{bind}\,\mathsf{x} = a\,\mathsf{in}\,\kappa \mid \mathsf{keep}\ (\mathcal{A}, \varnothing, \{(\mathbf{u}_i, \mathcal{K}_i)\}_{i\in I}, \{\mathbf{u}'_j\}_{j\in J})\rangle \Rightarrow \mathbb{k}}\ (\star)$$

where $\mathbb{k} = \mathsf{keep}\ (\mathbb{a}, \{(\mathbf{u}_i, \mathbb{k}_i)\}_{i\in I}, \{\sigma_j\}_{j\in J})$, and $(\star)$ is $(\bigvee_{i\in I} \mathbf{u}_i) \vee (\bigvee_{j\in J} \mathbf{u}'_j) \simeq \mathbb{1}$. In short, the set $\mathbb{u}$ of unexplored parts is used by the rule [BindKeep] to produce by tallying the set of substitutions $\Sigma = \{\sigma_j\}_{j\in J}$ required by the algorithmic type system to check that the type decomposition covers the type of the definition.

Now, we can amend the main reconstruction system in order not to explore useless type decomposition parts. First, we add a rule that trims a decomposition part if it is disjoint from the type of the definition:

$$[\textsc{BindEmpty}]\ \frac{\Gamma \vdash_{\mathcal{R}} \langle \mathsf{bind}\,\mathsf{x} = a\,\mathsf{in}\,\kappa \mid \mathsf{keep}\ (\mathcal{A}, \{(\mathbb{0}, \mathsf{infer})\}, \varnothing, \mathbb{u})\rangle \Rightarrow \mathbb{R}}{\Gamma \vdash_{\mathcal{R}} \langle \mathsf{bind}\,\mathsf{x} = a\,\mathsf{in}\,\kappa \mid \mathsf{keep}\ (\mathcal{A}, \varnothing, \varnothing, \mathbb{u})\rangle \Rightarrow \mathbb{R}}$$

$$[\textsc{BindTrim}]\ \frac{\begin{array}{c}\Gamma \vdash_{\mathcal{S}} \langle a \mid \mathcal{A}\rangle \Rightarrow \mathbb{a} \qquad \Gamma \vdash_{\mathcal{A}} [a \mid \mathbb{a}] : s \\ \Gamma \vdash_{\mathcal{R}} \langle \mathsf{bind}\,\mathsf{x} = a\,\mathsf{in}\,\kappa \mid \mathsf{keep}\ (\mathcal{A}, \mathcal{S}, \mathcal{S}', \{\mathbf{u}\} \cup \mathbb{u})\rangle \Rightarrow \mathbb{R}\end{array}}{\Gamma \vdash_{\mathcal{R}} \langle \mathsf{bind}\,\mathsf{x} = a\,\mathsf{in}\,\kappa \mid \mathsf{keep}\ (\mathcal{A}, \{(\mathbf{u}, \mathcal{K})\} \cup \mathcal{S}, \mathcal{S}', \mathbb{u})\rangle \Rightarrow \mathbb{R}}\ \begin{array}{l}\mathbf{u} \not\simeq \mathbb{0} \\ s \wedge \mathbf{u} \simeq \mathbb{0}\end{array}$$

The rule [BindEmpty] ensures that, even when the type of the definition is $\mathbb{0}$, we explore the body of the canonical form at least once. Note that the side condition $s \wedge \mathbf{u} \simeq \mathbb{0}$ may be replaced by a tallying instance $s \wedge \mathbf{u} \lhd_{\mathsf{T}} \mathbb{0}$ (cf. Definition 78) to trim more branches.

The [BindKeep] and [BindProp] rules are mostly unchanged (we just add the additional parameter $\mathbb{u}$ to the `keep` and `propagate` annotations).

8.3 Simplification of types

Through the process of reconstruction, the types inferred for the parameters of λ-abstractions might get more and more complex as they are successively substituted. In particular, the number of distinct type variables appearing in those types can grow exponentially if we are not careful about it. It is important to keep types as simple as possible and to minimize the number of type variables: in addition to significantly improving performance of subtyping and tallying, doing so will result in the inference of types more readable for the user.

8.3.1 Simplification of function types

We focus here on the simplification of the arrow part of types. Function types are subtle to manipulate as they mix covariance and contravariance. Two equivalent function types may have very different representations of different complexity, and the simplification of the DNF of a function type is a hard problem. Still, even though we have no systematic method of simplifying a function type, applying some simple simplification rules can greatly reduce the size of the representation of many function types generated during the reconstruction.

8.3.1.1 Simplification during the reconstruction

We recall that the DNF of a function type is as follows (see Section 2.5):

$$t \overset{\text{DNF}}{\simeq} \bigvee_{i\in I}\left(\bigwedge_{p'\in P'_i}\alpha_{p'} \wedge \bigwedge_{n'\in N'_i}\neg\alpha'_{n'} \wedge \bigwedge_{p\in P_i}(s_p \to t_p) \wedge \bigwedge_{n\in N_i}\neg(s'_n \to t'_n)\right)$$

where α can denote either a polymorphic or monomorphic type variable, and where for each $i \in I$, the whole summand is not equivalent to $\mathbb{0}$.

A first simplification consists in removing useless summands: if, for some $j \in I$, $\bigvee_{i\in I\setminus\{j\}}\left(\bigwedge_{p'\in P'_i}\alpha_{p'} \wedge \bigwedge_{n'\in N'_i}\neg\alpha'_{n'} \wedge \bigwedge_{p\in P_i}(s_p \to t_p) \wedge \bigwedge_{n\in N_i}\neg(s'_n \to t'_n)\right) \geqslant t$, then the summand associated to j can be removed from the DNF of t.

Similarly, we can remove useless literals: if, for some $j \in I$ and $q \in P_j$, $\bigvee_{i\in I}\left(\bigwedge_{p'\in P'_i}\alpha_{p'} \wedge \bigwedge_{n'\in N'_i}\neg\alpha'_{n'} \wedge \bigwedge_{p\in P''_i}(s_p \to t_p) \wedge \bigwedge_{n\in N_i}\neg(s'_n \to t'_n)\right) \leqslant t$ with $P''_j = P_j\setminus\{q\}$ and $\forall i \in I\setminus\{j\}.\ P''_i = P_i$, then the literal $s_q \to t_q$ can be removed from the DNF of t. The same applies to negative literals.

Another way to reduce the number of literals is to merge two literals together: if, for some $i \in I$, there are two $p, q \in P_i$ such that $s_p \simeq s_q$, then the conjunction of the two literals $s_p \to t_p$ and $s_q \to t_q$ can be transformed into a single literal $s_p \to t_p \wedge t_q$. Similarly, if $t_p \simeq t_q$, then $(s_p \to t_p) \wedge (s_q \to t_q)$ can be merged into $s_p \vee s_q \to t_p$. Note that this does not apply to disjunctions: $(s \to t_1) \vee (s \to t_2)$ is not equivalent to $s \to t_1 \vee t_2$ in general, nor is $(s_1 \to t) \vee (s_2 \to t)$ equivalent to $s_1 \wedge s_2 \to t$. This does not apply to negative literals either.

In addition to performing these simplifications on the DNF of t, we can also perform it recursively on all the types s_p, t_p, s'_n, and t'_n. However, as types are defined coinductively (they can be infinite trees), their memory representation may have cycles. Thus, it is necessary to remember nodes that have already been explored.

These simplifications preserve the semantic equivalence $\simeq$, and can thus be performed at any time during the reconstruction.

8.3.1.2 Simplification at top-level

Some additional simplifications can be performed at top-level, after the type of a top-level definition has been generalized.

A first straightforward simplification consists in substituting polymorphic type variables that only appear in covariant positions by $\mathbb{0}$, and those that only appear in contravariant positions by $\mathbb{1}$. For instance, for the type $\texttt{Int} \wedge \alpha \to \texttt{Bool} \vee \beta$, we apply the substitution $\{\alpha \rightsquigarrow \mathbb{1}\ ;\ \beta \rightsquigarrow \mathbb{0}\}$, yielding the simpler type $\texttt{Int} \to \texttt{Bool}$. Note that this transformation does not preserve semantic equivalence, but the resulting type is an instance of the initial type, and is smaller than the initial type.

It is also worth noting that the transformation above is more effective when the type variables of the initial type are not unnecessarily correlated. For instance, the type $(\texttt{Int} \wedge \alpha \to \texttt{Int} \wedge \alpha) \wedge (\neg\texttt{Int} \wedge \alpha \to \texttt{False})$ has its two branches using the same type variable α. An alternative version of this type without this unnecessary correlation would be $(\texttt{Int} \wedge \alpha \to \texttt{Int} \wedge \alpha) \wedge (\neg\texttt{Int} \wedge \beta \to \texttt{False})$ (this alternative type can be obtained by expansion of the initial type and by subsumption). This allows applying the simplification we have seen on β: as β only appears in a contravariant position, it can be substituted by $\mathbb{1}$, yielding the type $(\texttt{Int} \wedge \alpha \to \texttt{Int} \wedge \alpha) \wedge (\neg\texttt{Int} \to \texttt{False})$. Thus, it is generally worth decorrelating the different branches of top-level types, even though this might increase the number of different type variables.

Lastly, we can try to remove redundant literals of top-level types, just as we did in the previous section, but this time by reasoning modulo instantiation. For instance, the polymorphic type $(\alpha \to \alpha) \wedge (\texttt{Int} \to \texttt{Int})$ should intuitively be simplified into $(\alpha \to \alpha)$ (as the branch $\texttt{Int} \to \texttt{Int}$ is just an instance of the branch $(\alpha \to \alpha)$). More generally, let us assume that our top-level type has the following DNF:

$$t \overset{\text{DNF}}{\simeq} \bigvee_{i \in I} \left(\bigwedge_{p \in P_i} (s_p \to t_p) \right)$$

Note that this DNF does not contain top-level type variables nor negative arrow literals: this is always the case for the type inferred for a λ-abstraction. We assume that the sets of polymorphic type variables of each summand are disjoint (if it is not the case, we can temporarily substitute conflicting ones by monomorphic type variables). Then, for every $i \in I$ and $q \in P_i$, the literal $s_q \to t_q$ can be removed if $\left(\bigwedge_{p \in P_i \setminus \{q\}} (s_p \to t_p)\right) \lhd_{\mathsf{T}} s_q \to t_q$ (that is, if there exists an instance of the other literals that is smaller than $s_q \to t_q$).

The simplification above does not preserve the semantic equivalence $\simeq$: in fact, even for the same initial type t, it may yield different non-equivalent types depending on which order the literals of t are tested. Thus, it should not be performed during the reconstruction of a definition, but only on top-level types because the reconstruction algorithm expects the algorithmic type system to be stable by semantic type equivalence.

8.3.2 Simplification of tallying solutions

Many tallying instances are generated through the reconstruction process. In the main reconstruction algorithm, tally_infer(.) is used to find substitutions involving the monomorphic type variables in order to infer the types of the parameters of

λ-abstractions. In the substitution inference system, $\mathsf{tally}(.)$ is used to instantiate polymorphic types for applications and projections.

The solutions to a tallying instance are not uniquely characterized: they can be captured by different sets of substitutions. We consider the $\mathsf{tally}(.)$ and $\mathsf{tally\_infer}(.)$ functions to be black boxes that can return any such set of substitutions. Still, for better performance, it is important to simplify these solutions in order to keep types as simple as possible and to minimize backtracking. We give in this section some guidelines following this direction.

We start with some possible simplifications for the solutions to tallying instances $\mathsf{tally}(.)$ used in the substitution inference system:

Removing useless substitutions When calling $\mathsf{tally}(.)$ to find instantiations for an application involving polymorphic types, some substitutions found may be useless. For instance, consider the application $\mathsf{f}\ \mathsf{x}$ with f of type $(\alpha \to \mathtt{Bool}) \to \alpha \to \mathtt{Bool}$ and x the identity function of type $\beta \to \beta$. The tallying instance generated for this application is the following:

$$\mathsf{tally}(\{(\alpha \to \mathtt{Bool}) \to \alpha \to \mathtt{Bool} \mathrel{\dot{\leqslant}} (\beta \to \beta) \to \gamma\})$$

The solutions can be captured by these two substitutions $\{\sigma_1, \sigma_2\}$:

$$\sigma_1 = \{\alpha \rightsquigarrow \mathtt{Bool} \wedge \beta \wedge \alpha\ ;\ \beta \rightsquigarrow \mathtt{Bool} \wedge \beta\ ;\ \gamma \rightsquigarrow (\mathtt{Bool} \wedge \beta \wedge \alpha \to \mathtt{Bool}) \vee \gamma\}$$

$$\sigma_2 = \{\alpha \rightsquigarrow \mathbb{0}\ ;\ \gamma \rightsquigarrow \mathbb{0} \to \mathbb{1}\}$$

However, the substitution σ_2 is not really interesting in our case, as our objective is to get the smallest type for the result of the application (captured by the type variable γ). Indeed, applying the substitution σ_2 to f and x yields for the application the type $\mathbb{0} \to \mathbb{1}$, which is larger than the type we would obtain with σ_1. Thus, we should ignore the substitution σ_2 and only instantiate f and x with σ_1 in order to keep types as simple as possible.

Minimizing the number of type variables In the previous example, the substitution σ_1 can be greatly simplified. Indeed, as our objective is to get the smallest possible type for the result of the application (captured by γ in our example), we can substitute the type variables that only appear in the result (captured by the type variable γ) in a covariant position by $\mathbb{0}$, and substitute those that only appear in a contravariant position by $\mathbb{1}$. Doing so on σ_1 yields a new substitution $\{\alpha \rightsquigarrow \mathtt{Bool}\ ;\ \beta \rightsquigarrow \mathtt{Bool}\ ;\ \gamma \rightsquigarrow \mathtt{Bool} \to \mathtt{Bool}\}$.

We continue with some guidelines for simplifying solutions to tallying instances $\mathsf{tally\_infer}(.)$ used in the main reconstruction algorithm:

Substituting monomorphic type variables only when necessary Let us consider the projection $\pi_1\mathsf{x}$, with x the binding variable associated to a lambda variable x of type $(\boldsymbol{\beta}_1 \times \boldsymbol{\beta}_2) \wedge \boldsymbol{\alpha}$ (with $\boldsymbol{\alpha}$, $\boldsymbol{\beta}_1$ and $\boldsymbol{\beta}_2$ monomorphic). When

encountering the atom $\pi_1\mathsf{x}$ for the first time, the main reconstruction algorithm generates the following tallying instance:

$$\mathsf{tally\_infer}((\boldsymbol{\beta}_1 \times \boldsymbol{\beta}_2) \wedge \boldsymbol{\alpha} \mathrel{\dot{\leqslant}} \gamma_1 \times \gamma_2)$$

The solutions to this tallying instance can be captured in different ways. For simplicity, we ignore solutions that make the left-hand side of the tallying instance empty. The set of solutions to this tallying instance can then be captured by this substitution ϕ (for clarity, we keep polymorphic type variables in the domain of ϕ, even though $\mathsf{tally\_infer}(.)$ is supposed to restrict the domain of the solutions to $\mathcal{V}_M$):

$$\phi = \{\boldsymbol{\alpha} \rightsquigarrow ((\boldsymbol{\gamma}_1 \times \boldsymbol{\gamma}_2) \vee \neg(\boldsymbol{\beta}_1 \times \boldsymbol{\beta}_2)) \wedge \boldsymbol{\alpha} \ ; \ \gamma_1 \rightsquigarrow \boldsymbol{\gamma}_1 \ ; \ \gamma_2 \rightsquigarrow \boldsymbol{\gamma}_2\}$$

If we keep this solution as is, we would have to backtrack to the definition of x and apply ϕ, yielding for the parameter x the new type $(\boldsymbol{\beta}_1 \times \boldsymbol{\beta}_2) \wedge (\boldsymbol{\gamma}_1 \times \boldsymbol{\gamma}_2) \wedge \boldsymbol{\alpha}$. However, this seems unnecessary: we could instead have kept the type of x as it is, as the current type $(\boldsymbol{\beta}_1 \times \boldsymbol{\beta}_2) \wedge \boldsymbol{\alpha}$ of x already allows typing $\pi_1\mathsf{x}$ with $\boldsymbol{\beta}_1$, which is a type "precise enough".

To avoid backtracking unnecessarily and adding complexity to the types of parameters, we can simplify ϕ by making the following observation: the monomorphic type variable $\boldsymbol{\alpha}$ can be removed from the domain of ϕ by composing ϕ with the substitution $\{\boldsymbol{\gamma}_1 \rightsquigarrow \boldsymbol{\beta}_1 \ ; \ \boldsymbol{\gamma}_2 \rightsquigarrow \boldsymbol{\beta}_2\}$, yielding a new substitution $\phi' = \{\gamma_1 \rightsquigarrow \boldsymbol{\beta}_1 \ ; \ \gamma_2 \rightsquigarrow \boldsymbol{\beta}_2\}$. This new substitution does not substitute any monomorphic type variable anymore, and thus backtracking is not necessary anymore.

The substitution ϕ' is less general than ϕ, in the sense that ϕ' can be obtained by composing ϕ with another substitution, but not the other way around. However, once restricted to $\mathcal{V}_M$, ϕ' becomes as general as ϕ, and it still allows getting a precise type for the result of the projection. This kind of simplification can be achieved systematically using additional tallying instances.

Avoiding the introduction of new type variables New monomorphic type variables should only be introduced when necessary. For instance, a substitution $\{\boldsymbol{\alpha} \rightsquigarrow \mathtt{Int} \wedge \boldsymbol{\alpha} \wedge \boldsymbol{\beta}\}$ that would be solution to a tallying instance (with $\boldsymbol{\beta}$ a new fresh type variable) should be simplified before being applied: the new type variable $\boldsymbol{\beta}$ is unnecessary as it plays the same role as $\boldsymbol{\alpha}$. The simpler substitution $\{\boldsymbol{\alpha} \rightsquigarrow \mathtt{Int} \wedge \boldsymbol{\alpha}\}$ should be preferred.

Introducing unnecessary type variables may have a dramatically negative impact on performance: it may imply more solutions to the future tallying instances, which may in turn generate more redundant intersection branches in the annotation tree.

8.4 Memoization

Due to the branching nature of the reconstruction algorithm, some atoms are reconstructed and re-typed many times in the different branches of the annotation tree. Even within the same branch, backtracking may force an atom to be re-typed. While it is sometimes inevitable to re-type an atom, for instance if the context has changed, in many cases it could be avoided by performing memoization to prevent an atom to be re-typed twice under an equivalent context (with respect to its free variables).

Memoization may be implemented at three levels: main reconstruction system, substitution inference system, and algorithmic type system.

Algorithmic type system Implementing caching for the algorithmic type system is very simple: the algorithmic type system should return the same type when it is called on the same physical annotation object. Although the type returned by the algorithmic type system also depends on the expression and the context (it does not only depend on the annotation), annotations for the algorithmic type system are specific to an expression and environment: if the environment or expression change, then the annotation will be regenerated. Thus, caching for the algorithmic type system can easily be implemented by adding a mutable field `cached_type` to the structure of annotations: the first time the algorithmic type system is called on this annotation, it computes the type of the expression and stores it in the `cached_type` field, and the next times it can just return the type stored in `cached_type`.

Substitution inference system The substitution inference system takes as input an environment, an atom or canonical form, and a partial annotation, and it returns an annotation for the algorithmic type system. This time, we cannot store a cache as a mutable field in the structure of partial annotations, as partial annotations are destroyed, reconstructed and duplicated many times by the main reconstruction algorithm: it would be a nightmare to make copies of these mutable fields and invalidate them when necessary.

Instead, we can perform some basic memoization. For that, we use a mapping from triplets $(a, \mathcal{A}, \Gamma)$ to annotations for the algorithmic type system. Each time the substitution inference system is called on a given atom a, partial annotation $\mathcal{A}$, and environment Γ, we look in this mapping for a value associated to the key $(a, \mathcal{A}, \Gamma|_{\mathsf{fv}(a)})$. If it has one, we return this value, otherwise we compute the annotation and store it in the mapping. The equality relation over the keys (to use for the mapping) should be based on the semantic subtyping equivalence for the environments Γ, and for the partial annotations $\mathcal{A}$ we can implement a naive structural equality relation. It may be difficult to use an efficient mapping structure, such as a hashtable or a binary tree, because defining a hashcode or a total order that is compatible with semantic equivalence is still an open problem. However, using a naive data structure such as a list of pairs (key, value) is already enough to drastically improve performance of the reconstruction, even though searching in the mapping may be slow.

Note that it is not necessary to perform memoization for the reconstruction of canonical forms: it is enough to perform it for the reconstruction of atoms. Indeed, canonical forms are just a succession of atoms: the expensive part in the reconstruction of a canonical form is the reconstruction of the atoms composing it. Also, a canonical form is less likely to be reconstructed multiple times in an equivalent context, as its free variables are the union of all of those in the atoms composing it.

Main reconstruction system The implementation of memoization for the main reconstruction algorithm is similar to the implementation of memoization for the substitution inference system. We can implement it by using a mapping from triplets $(a, \mathcal{A}, \Gamma)$ to results $\mathbb{R}$. The same data structure can be used to implement this mapping as the type of the keys is the same (only the type of values is different: a result $\mathbb{R}$ instead of an annotation $\mathbb{a}$).

Memoization can be made more effective by reusing the same type variables when possible: we should avoid having two different type variables in two different branches that "play the same role". For instance, if in a branch of our annotation tree the type variable $\boldsymbol{\alpha}$ is used to type the parameter of a λ-abstraction, and if in another branch we use the type variable $\boldsymbol{\beta}$ for the same purpose, then it might prevent memoization to occur between those two branches for atoms that depend on this parameter. The rule [LambdaInfer] of the main reconstruction system may be modified accordingly, so that when applied to the same λ-abstraction in two different subderivations, it types its parameter with the same type variable $\boldsymbol{\alpha}$.

The impact on performance of implementing caching and other optimizations seen in this chapter will be evaluated and discussed in Chapter 9 (Section 9.2.2).

Chapter 9

Prototype Implementation

Contents

This chapter presents a prototype implementation (Castagna et al., 2024b) for the algorithmic type system and reconstruction algorithm. It implements the extensions and practical aspects discussed in the previous chapters. The features of the prototype are described in Section 9.1. Then, the prototype is tested on several examples in Section 9.2, where we evaluate its performance.

9.1 Presentation of the prototype

The prototype implementation fully implements the algorithmic type system, the reconstruction algorithm, and the extensions presented in Chapter 7. The syntax for the source language is inspired by the syntax of OCaml, with some modifications and the addition of type-cases. This section presents this language and its different features.

9.1.1 Language and features

9.1.1.1 Definition of types

Figure 9.1 presents some constants and built-in types of our language. For every constant of the language, there exists a corresponding singleton type. In order to distinguish them, type identifiers start with an uppercase, while constants start with a lowercase. It is possible to declare a new constant, together with the associated singleton type:

```
(* Defines a new value null and a singleton type Null *)
atom null
```

Constant	Singleton type
`true`	`True`
`false`	`False`
`nil`	`Nil`
i (with $i \in$ `Integer`)	i
`'`c`'` (with $c \in$ `Char`)	`'`c`'`

where `Integer` represents integers written in decimal, and `Char` represents characters.

Type	Inhabitants
`Any`	Every value
`Empty`	$\varnothing$
`Bool`	{`true`, `false`}
`Int`	`Integer`
i_1`--`i_2	$\{i \in$ `Integer` $\mid i_1 \leqslant i \leqslant i_2\}$
`(--`i_2`)`	$\{i \in$ `Integer` $\mid i \leqslant i_2\}$
`(`i_1`--)`	$\{i \in$ `Integer` $\mid i_1 \leqslant i\}$
`Char`	`Char`

Figure 9.1: Built-in constants and types

Function types can be constructed with the arrow type constructor `S -> T`, pair types with the product type constructor `(T1,T2)`, and record types with the constructor `{ fields ..}` (for open records) and `{ fields }` (for closed records). We can also use the union `|`, the intersection `&`, the difference `\`, and the negation `~`:

```
type TotalPredicate = Any -> Bool
type Coord = (Int, Int)
(* 'subtitle' field is optional, other fields are required *)
type Book = { id=Int, title=String, author=String, subtitle=?String ..}
type Falsy = False | "" | 0 | Null
type Truthy = ~Falsy
```

In the code above, `TotalPredicate`, `Coord`, `Falsy` and `Truthy` are type aliases.

Identifiers for type variables start with a `'` (for instance, `'a`, `'b`, `'typ`, etc.). There is no syntactic distinction between monomorphic and polymorphic type variables: whether a type variable can be instantiated or not depends on the context. In particular, type variables appearing in the type of a top-level definition are polymorphic.

Recursive and parametric types are supported, allowing to encode lists:

```
type MyList 'a = Nil | ('a, MyList 'a)
type IntList = MyList Int (* Instantiation of a parametric type *)
```

Lists are encoded as chains of pairs ending with the constant `nil`, as defined by the type `MyList` above. A built-in syntax for the types of lists is available. This syntax allows expressing lists whose elements follow a regular expression, as defined in Figure 9.2.

For instance, the type `[Int ; Bool* ; Int]` is inhabited by lists starting with an integer, followed by any number of Boolean values, and finishing with an integer. This type is equivalent to the type `Pattern` defined below:

```
type Chain 'a 'b = 'b | ('a, Chain 'a 'b)
type Pattern = (Int, Chain Bool (Int, Nil))
```

Type	Inhabitants
`List`	Lists
`[`r`]`	Lists satisfying r
`[]`	Same as `Nil`
`String`	Same as `[Char*]`
`"abc"`	Same as `['a';'b';'c']`

Regular expr r	Meaning
T	Element of type T
r`||`r	Union
r`;`r	Concatenation
r`*`	Zero or more times
r`+`	One or more times
r`?`	Zero or one times

Figure 9.2: Built-in types for lists, regular expressions, and strings

Any regular expression can be encoded similarly.

In the rest of this chapter, we will use the syntax of our prototype for writing types, instead of the formal syntax. In particular, the type $\mathbb{1}$ will be noted `Any`, the product $t_1 \times t_2$ will be noted `(T1, T2)`, etc.

9.1.1.2 Programs and expressions

In addition to atom declarations and type definitions, a program can contain top-level `let` definitions. The syntax for expressions is close to the OCaml syntax, with the addition of type-cases (`if . is . then . else .`).

For instance, the example from the introduction can be written as follows (the types inferred for these functions, and more, are detailed in Section 9.2):

```
atom null
type Falsy = False | "" | 0 | Null
type Truthy = ~Falsy

let toBoolean =
    fun x -> if x is Truthy then true else false

let lOr (x,y) =
    if toBoolean x (* is True (implicit) *)
    then x else y

let id x = lOr (x,x)
```

Here is another example with operations on records:

```
let add_f_field r =
    if r is {f=Int ..} then r else {r with f=0}

let remove_f_field r = r\f

let test =
    ((add_f_field { }).f, (add_f_field { f=42 }).f)
```

An expression can also contain let-bindings. These let-bindings fully benefit from occurrence typing, allowing the example below to be typed `('a & Truthy -> 'a & Truthy) & (Falsy -> False)`:

```
let test x =
    let y = toBoolean x in
    if y is True then x else false
```

However, let-bindings in expressions are not generalized as generalization only happens at top-level (cf. Section 4.1.2). Consequently, without type annotations, the definition `test` below is typed `(Bool, Bool)`[1]:

```
let test =
    let id x = x in
    (id true, id false)
```

while this one is typed `(True, False)`:

```
let id x = x
let test = (id true, id false)
```

Our language also features pattern-matching. Internally, pattern-matching is transformed into let-bindings and type-cases, as detailed in Section 7.5. The syntax of patterns is detailed in Figure 9.3.

Pattern p	Meaning
$:T$	Match values of type T
x	Capture matched value into variable x
x=c	Assign constant c to variable x (match anything)
(p, p)	Destruct a pair and pattern-match its components
{ l=p }	Destruct a record of exactly one field `l` and pattern-match it
{ l=p ..}	Destruct a record containing a field `l` and pattern-match it
p & p	Conjunction of patterns
p | p	Disjunction of patterns

Figure 9.3: Syntax for patterns

The example below uses pattern-matching to implement the `hd` function that takes a list and returns its first element or `nil` if the list is empty. The first pattern, `:Nil`, only matches empty lists (i.e., the constant `nil`). The second pattern, `(h, _) & :List`, matches lists that are non-empty (`:List` matches lists, and among

[1] Even though it would be possible to infer the type `(True, False)` for `test` by first inferring the overloaded type `(True -> True) & (False -> False)` for the local definition `id`, it would require user annotations: the reconstruction algorithm by itself only infers an overloaded type for a function when it is suggested by its body (and not by its future applications). This will be discussed in Section 9.1.1.5.

them `(h, _)` can only match non-empty ones) and captures its first element into the variable `h`.

```
let hd l =
    match l with
    | :Nil -> nil
    | (h, _) & :List -> h
    end
```

9.1.1.3 Recursive functions

The source language presented in Chapter 3 does not include recursive functions, since *from a theoretical viewpoint* they are useless: Milner (1978, page 356) justifies the addition of a "$\mathsf{fix}\, x.e$" expression by the fact that their system cannot type Curry's fixpoint combinator, but our system can. For instance, here is an implementation of Curry's fixpoint combinator (for a call-by-value language):

```
let fixpoint = fun f ->
    let delta = fun x ->
        f ( fun  v -> ( x x v ))
    in delta delta
```

For this function `fixpoint`, our prototype infers the following type:

```
(('a -> 'b) -> ('a -> 'b) & 'c) -> ('a -> 'b) & 'c
```

Though the fixpoint combinator is traditionally given the type `(('a -> 'b) -> ('a -> 'b)) -> 'a -> 'b`, our prototype infers a slightly more precise type by intersecting the co-domain of the argument with `'c`.

Using this function `fixpoint`, we can then implement recursive functions, such as the factorial `fact`:

```
let fact_stub fact n =
    if n is 0 then 1 else (fact (n-1)) * n

let fact = fixpoint fact_stub
```

The prototype is able to infer the type `((--1) | (1--) -> Int) & (0 -> 1)` for `fact` (assuming that the multiplication `*` has type `Int -> Int -> Int`). Note that this type is overloaded: the branch `0 -> 1` captures the behavior of the base case (`n = 0`), while the branch `(--1) | (1--) -> Int` captures the recursive case. This overloading is suggested by the type-case in `fact_stub`, yielding the following type for `fact_stub`:

```
(Any -> 0 -> 1) & ((Int -> Int) -> (0 -> 1) & ((--1) | (1--) -> Int))
```

Then, the tallying instance generated by the application of the fixpoint combinator to `fact_stub` generates several solutions that account for the different branches: in particular, one solution yields the resulting type `(0 -> 1)` and another yields the resulting type `((--1) | (1--) -> Int)` [2]. We will see more examples of recursive functions in Section 9.2. Each time the same behavior occurs: recursive functions are typed with an overloaded type that distinguishes the different cases of the body.

Although we do not strictly need recursive functions in the source language, *from a practical viewpoint* the use of `let rec` definitions instead of a fixpoint combinator is not only more convenient, but it may also improve the speed of reconstruction. Thus, we implemented the classic `let rec` definitions:

```
let rec fact n =
    if n is 0 then 1 else (fact (n-1)) * n
```

This code is transformed by the prototype into the version that uses the fixpoint combinator, with some additional type annotations to take the arity of the function into account for improved performance, as we will see in Section 9.1.1.4. The typing of recursive functions is thus composed of two phases: first, a type is inferred for the `stub` function generated, and secondly, the type of the application `fixpoint stub` is computed. As we will see in Section 9.2, the second step is usually significantly slower than the first, as it requires solving a tallying instance that may involve many type variables.

9.1.1.4 User type annotations

Type annotations, defined in Section 7.2, can be added to help the reconstruction algorithm or to restrict the domain of a function. For instance, the `hd` function can be written as follows, yielding the type `['a*] -> Nil | 'a` (instead of `(('a, Any) -> 'a) & ([ ] -> [ ])` without type annotation):

```
let hd (l:['a*]) =
    match l with
    | :Nil -> nil
    | (h, _) -> h
    end
```

For a given parameter, multiple types can be specified using a "`;`" as separator. Each type is then considered separately, yielding the overloaded type `(['a+] -> 'a) & ([ ] -> Nil)` for the function `hd` below:

```
let hd (l:['a+];[]) =
    match l with
```

[2] The tallying instance at issue is `(('a -> 'b) -> ('a -> 'b) & 'c) -> ('a -> 'b) & 'c` $\dot{\leqslant}$ `(Any -> 0 -> 1) & ((Int -> Int) -> (0 -> 1) & ((--1) | (1--) -> Int)) -> 'd`. The substitution {`'a` $\rightsquigarrow$ `0` ; `'b` $\rightsquigarrow$ `1` ; `'c` $\rightsquigarrow$ `Any` ; `'d` $\rightsquigarrow$ `0 -> 1`} is a solution that yields the resulting type `0 -> 1`, and the substitution {`'a` $\rightsquigarrow$ `Int` ; `'b` $\rightsquigarrow$ `Int` ; `'c` $\rightsquigarrow$ `Any` ; `'d` $\rightsquigarrow$ `Int -> Int`} is another solution that yields the resulting type `Int -> Int`.

```
    | :Nil -> nil
    | (h, _) -> h
    end
```

For recursive functions, it is also possible to annotate the type of the result:

```
let rec fact (n:Int) : Int =
    if n is 0 then 1 else (fact (n-1)) * n
```

This type information is forwarded to the `stub` function generated when encoding the recursive function as a fixpoint. In our case, the first parameter of `fact_stub` will be annotated with `Int -> Int`. Note that annotating the type of the result is only possible for recursive functions (since it only impacts the type of the first parameter of the `stub` function), and that the final type inferred (after applying the fixpoint) may have a different codomain. For instance, in the special case where the function is indicated as being recursive but never calls itself, then writing an annotation for the result type has no impact. This is illustrated by the following definition, for which the type `'a -> 'a` is inferred:

```
let rec id x : Int = x
```

Type variables used in user type annotations are not substituted by the reconstruction algorithm. If we want to specify an initial type for a parameter but still allow its type variables to be substituted during the reconstruction process, we can use type variable identifiers starting with `'_`. For instance, a parameter annotated with `'_a -> '_b` is initially given the type `'_a -> '_b` (instead of a single fresh type variable), but the reconstruction algorithm is still allowed to substitute `'_a` and `'_b` during the reconstruction of the body. This kind of constraint can be used to reduce the search space when we already know the shape of a parameter, thus improving performance. For instance, the recursive `fact` function can be encoded more efficiently with the following code:

```
let fact_stub (fact:'_a -> '_b) n =
    if n is 0 then 1 else (fact (n-1)) * n

let fact = fixpoint fact_stub
```

As we can see, the parameter `fact` of `fact_stub` is annotated with `'_a -> '_b` as we know that this must be a function of arity at least one. If `fact_stub` had an additional parameter, then the parameter `fact` would be annotated with `'_a -> '_b -> '_c`. While this annotation may seem trivial, it actually constrains the type of the parameter `fact` to be a single arrow (and, in particular, not an intersection of arrows), which can sometimes greatly reduce the search space and improve performance. While this does not make any difference on this simple function `fact_stub` (as `fact` is only called once on an argument of type `Int`, the reconstruction never tries to give its parameter `fact` an overloaded function type), we will see in Sec-

tion 9.2 another example of recursive function, `flatten`, whose inference is about 4 times faster with this simple annotation.

Lastly, type annotations can be provided for top-level definitions. These type annotations are ignored when typing the body of the definition, but are compared afterwards with the type inferred for the definition: if the inferred type is smaller than the annotated type (by the $\lhd_{\mathsf{T}}$ order, cf. Definition 78), then the type-checking is successful, and the type used for the top-level definition is the annotated one. Otherwise, the type-checking fails. As an example, consider the two following top-level definitions:

```
let a : Int -> Int = fun x -> x
let b : Bool = fun x -> x
```

The top-level definition `a` type-checks, yielding the type `Int -> Int`, while the top-level definition `b` does not type-check ("the type inferred is not a subtype of the type specified"). This can be useful if the user wants to check that a top-level definition has a given type, or wants to propose a simpler type for this top-level definition.

9.1.1.5 Generalizing let-bindings

Our type system only generalizes top-level definitions (cf. Chapter 4). While this is not an issue from the perspective of the declarative and algorithmic type system, as intersection types can be used locally in place of parametric polymorphism, it does have an impact on the reconstruction algorithm.

For instance, consider the following definition:

```
let test_no_gen x y =
    let pack u v = (u,v) in
    (pack x y, pack y x)
```

The type inferred for `pack` is `'a -> 'b -> ('a,'b)`. However, as it is not a top-level definition, the type variables `'a` and `'b` are not generalized: they stay monomorphic during the typing of the body of `test_no_gen`. Consequently, in order for both applications `pack x y` and `pack y x` to be typeable, the reconstruction algorithm infers that `x` must be of type `'a` and of type `'b`, and similarly that `y` must be of type `'b` and of type `'a`. After some simplifications, our prototype yields the following type for `test_no_gen`: `'a -> 'a -> (('a,'a),('a,'a))`.

While this type is correct, it could be more general: the parameters `x` and `y` do not need to have the same type, and `pack` could be typed `'a -> 'b -> (('a,'b),('b,'a))`. Indeed, this is the type we obtain if we move `pack` at top-level:

```
let pack u v = (u,v)
let test_toplevel x y =
    (pack x y, pack y x)
```

By doing so, the type of pack is generalized before reconstructing the annotations for the two applications `pack x y` and `pack y x`, and thus the reconstruction algorithm is able to type these two applications independently, using different instantiations.

In order to infer the more general type while still allowing the user to define pack inside `test_no_gen`, a new `let gen` keyword has been introduced:

```
let test_gen x y =
    let gen pack u v = (u,v) in
    (pack x y, pack y x)
```

Let-definitions that use the `gen` keyword are moved at top-level so that they can be generalized. They receive variables in their closure as extra parameters. For instance, the code above is transformed into the following top-level definitions:

```
let pack x y u v = (u,v)
let test_gen x y =
    (pack x y x y, pack x y y x)
```

Although this transformation is advantageous in some cases, as it allows generalizing intermediate let definitions, it comes with a counterpart: the correlation between occurrences of a subexpression inside the `let gen` definition and those outside is lost, making occurrence typing inoperative on those occurrences. Indeed, by moving a let-definition at top-level, the subexpressions that it had in common with the rest of the initial expression no longer share the same bindings, but instead are bound to other bindings in a different top-level definition. As an example, consider the following definition `test_no_occ_typ` (where the syntax `<T>` is used to "magically" construct an expression of type `T`):

```
let f = <Any -> Any> (* "magic" expression of type Any -> Any *)
let test_no_occ_typ x =
    let gen g y = if f x is Int then x else y in
    if f x is Int then g false else g x
```

In the definition `test_no_occ_typ`, the occurrence of `f x` inside the definition of `g` and the one after will be bound to two independent bindings in two distinct top-level definitions. As a result, it cannot be deduced that, during the call `g false` in the first branch of the type-case, only the first branch of the type-case in `g` can be taken. In the end, the type inferred for `test_no_occ_typ` is `'a -> False | 'a`, while it would be `'a -> 'a` without the `gen` keyword.

This choice between generalization and occurrence typing is not a limitation of the algorithmic type system, as intersections can be used in place of parametric polymorphism for local definitions. However, the reconstruction algorithm is not complete and cannot systematically infer the necessary intersections. In particular, in the absence of user type annotations, it does not try to type a λ-abstraction with an intersection type if it is not suggested by its own body. Consequently, later applications of this λ-abstraction in the same top-level definition may result in a

unification of the types of the arguments on which this λ-abstraction is applied. This can be solved by adding user type annotations to force the inference of an overloaded type for a λ-abstraction even when it is not suggested by its body:

```
let test_inter x y =
    let pack (u:'a;'b) (v:'a;'b) = (u,v) in
    (pack x y, pack y x)
```

The `let gen` construct is an alternative solution, relying on parametric polymorphism rather than intersections. This solution may lead to simpler types (because no intersection is used) and thus better performance, but is incompatible with occurrence typing.

9.1.2 Architecture of the implementation

The prototype is implemented in OCaml (about 4600 lines of code). It uses the implementation of set-theoretic types of the CDuce library. Note that it does not feature an interpreter (the code can be typed but cannot be executed). An online version can be tested at the following address: `https://www.cduce.org/dynlang/`, and the source code is available on Zenodo: Castagna et al. (2024b). The online version of the prototype is compiled using `Js_of_ocaml` and is about 8 times slower than the native version.

The global structure of the source code is as follows:

webeditor/ This directory contains code for the interface of the online version. It consists in an integration of the type-checker, compiled from the OCaml sources to JavaScript using `Js_of_ocaml`, into the Monaco Editor (Microsoft).

src/ Contains the OCaml code for the parser and type-checker.

src/main/ Contains the code for the command line interface (for the native version), as well as the code for the JavaScript wrappers (for the online version). These wrappers allow calling the type-checker from JavaScript code.

src/parsing/ Contains the lexer, parser and the *Abstract Syntax Tree (AST)* of the source language.

src/system/ Contains the AST for MSC forms and the implementation of the algorithmic type system and reconstruction algorithm. The main ideas and algorithms presented in this manuscript are implemented in this directory.

src/types/ Wrapper around the CDuce library that provides the operations on types (subtyping, DNF, tallying, etc.) and implements some auxiliary functions over types (simplification, etc.). One difference with the interface of the CDuce library is that, in the interface of our wrapper, there are two kinds of type variables (monomorphic and polymorphic), while in CDuce this distinction is not at the level of type variables but at the level of type operators

(type operators that manipulate type variables take an additional parameter specifying which type variables should be considered as monomorphic).

src/utils/ Some straightforward utility modules and functions.

In order to clarify the contributions of this prototype, here is a summary of what comes from the CDuce library, and what is implemented in the prototype:

Provided by CDuce Representation of set-theoretic types using Binary Decision Diagrams (BDD) with support for records and type variables, subtyping, extraction of the DNF, type operators (π_1, π_2, $\circ$, etc.), substitutions, tallying, pretty-printing of types.

Implemented in this prototype Parser for the source language, conversion to MSC form, algorithmic type system and reconstruction algorithm (with the extensions of Chapter 7 and the optimizations of Chapter 8). It also implements some specific auxiliary operators on types, such as a simplification function for function types.

The implementation of the prototype aims to be as close to the formalization as possible. For instance, here is an excerpt implementing the rules for type-cases of the reconstruction system:

```
98  | Ite (v, tau, _, _), InferA ->
99      if memvar v then (* not [CaseVar] *)
100         let t = vartype v in
101         if subtype t empty then Ok TypA (* [CaseEmpty] *)
102         else if subtype t tau (* [CaseThen] *)
103         then
104             let psi = tallying_infer [(t, empty)] in
105             (* Simplifications of Chapter 8 *)
106             let psi = simplify_tallying_infer env empty psi in
107             if List.exists Subst.is_identity psi
108             then Ok TypA
109             else (* Conclusion of [CaseThen] *)
110                 needsubst psi TypA ThenVarA
111         ... (* Other rules *)
```

9.2 Results and performance

9.2.1 Type inference

In this section, we discuss the type inferred by our prototype on several examples. Types inferred are written in comments, above each top-level definition. The time taken for inferring the type of each example will be detailed in Section 9.2.2.

```
type Falsy = False | "" | 0
type Truthy = ~Falsy

(* (Truthy -> True) & (Falsy -> False) *)
let toBoolean x =
    if x is Truthy then true else false

(* (('a \ Falsy, Any) -> 'a \ Falsy) & ((Falsy,'b) -> 'b) *)
let lOr (x,y) =
    if toBoolean x then x else y

(* 'a -> 'a *)
let id x = lOr (x,x)
```

The code above features the examples used in the introduction. Note that the type inferred for `id` is the simple polymorphic type `'a -> 'a`: though the reconstruction algorithm generates two branches `'a & Falsy -> 'a & Falsy` and `'a & Truthy -> 'a & Truthy`, the prototype simplifies the resulting overloaded type into `'a -> 'a`.

```
(* (Nil -> "Nil") &
   ([Char+] -> "String") &
   (Char -> "Char") &
   (Int -> "Number") &
   (Bool -> "Boolean") &
   (Any \ String \ Char \ Int \ Bool -> "Object") *)
let typeof x =
    match x with
    | :Nil -> "Nil"
    | :String -> "String"
    | :Char -> "Char"
    | :Int -> "Number"
    | :Bool -> "Boolean"
    | :Any -> "Object"
    end
```

This code implements a `typeof` function, inspired by JavaScript, that returns a string describing the type of the argument. One interesting thing to note here is the type of the second branch: `[Char+] -> "String"`. While we could expect it to be `String -> "String"`, this would actually be incorrect because of our encoding of strings: the empty string `""` is encoded by the constant `nil`, and thus it takes the first branch of the pattern-matching, yielding the result `"Nil"`.

```
type Expr = ("const", Int) | ("add", (Expr, Expr)) | ("uminus", Expr)
```

```
(* (("const", 'a) -> 'a) & (Expr -> Int) *)
let rec eval e =
    match e with
    | (:"add", (e1, e2)) -> (eval e1) + (eval e2)
    | (:"uminus", e) -> 0 - (eval e)
    | (:"const", x) -> x
    end
```

The `eval` function implements the evaluation of a very basic AST for arithmetic expressions. The type of the AST, `Expr`, is defined beforehand, but only to improve the readability of the pretty printing: without this definition, the type inferred for `eval` would be equivalent, though written in a less readable way.

The branch `("const", 'a) -> 'a` captures the fact that if our arithmetic expression is only composed of a constant (with no arithmetic operation performed on it), then this constant does not even need to be an integer as it is directly returned. While this branch is correct for this implementation of `eval`, we might want to forbid this case, which can be done either by making the last case of the pattern-matching more restrictive, or by adding a type annotation on the parameter `e` as follows:

```
(* Expr -> Int *)
let rec eval_ann (e:Expr) =
    match e with
    | (:"add", (e1, e2)) -> (eval_ann e1) + (eval_ann e2)
    | (:"uminus", e) -> 0 - (eval_ann e)
    | (:"const", x) -> x
    end
```

The traditional `map` function can be implemented as follows:

```
(* (('a -> 'b) -> (['a] -> ['b]) & (['a+] -> ['b+]) & ([ ] -> [ ])) &
   (Any -> [ ] -> [ ]) *)
let rec map f l =
    match l with
    | :Nil -> nil
    | (e, l) & :List -> ((f e), map f l)
    end
```

Again, the type inferred is overloaded. It subsumes the usual type inferred by Hindley-Milner type systems, stating that an application of `map` yields a function that maps lists of α's into lists of β's. In particular, our type captures the specific case where `map` receives an empty list as second argument (in which case the first argument can be of any type, and it returns an empty list), as well as the case where it receives a singleton list as second argument (in which case it returns a singleton list).

This example can be used to illustrate the lack of principal types. Indeed, the two specific cases specified above can be generalized: for any natural number n, if

`map` receives as second argument a list of length n, then it returns a list of length n. Each of these behaviors can be captured by a type (for instance, the type `('a -> 'b) -> ['a;'a] -> ['b;'b]` for $n = 2$), and each of these types can be checked by the algorithmic type system (provided that the function `map` is annotated accordingly). Unfortunately, it is not possible to capture all these behaviors in a single type, as it would require an infinite intersection (which is not allowed, as it would make subtyping undecidable).

A type annotation can be added in order to infer a simpler type for `map`:

```
(* ('a -> 'b) -> [ 'a* ] -> [ 'b* ] *)
let rec map_ann f (l:['a*]) =
    match l with
    | :Nil -> nil
    | (e,l) -> ((f e), map_ann f l)
    end
```

Note that we do not need to annotate the type of every parameter: here, we just annotated `l`. Still, this annotation has consequences on the whole reconstruction: in particular, the case where `f` has type `Any` is not typeable anymore.

The following code is an example of a common programming pattern that becomes typeable thanks to the expressivity of set-theoretic types combined with occurrence typing:

```
(* ('b -> Any \ True) & ('a -> Any) -> [('a | 'b)*] -> [('a \ 'b)*] *)
let rec filter_ann (f: ('a->Any) & ('b -> ~True)) (l:[('a|'b)*]) =
    match l with
    | :Nil -> nil
    | (e,l) ->
        if f e is True
        then (e, filter_ann f l)
        else filter_ann f l
    end

(* (Nil -> False) & (Any \ Nil -> True) *)
let not_nil x = if x is Nil then false else true

(* [(1 | 3 | 42)*] *)
let filtered_list = filter_ann not_nil [1;3;nil;42]

(* ([ Int* ] -> Int) & ([ ] -> 0) *)
let rec sum l =
    match l with
    | :Nil -> 0
    | (n,tl) -> n + (sum tl)
    end
```

```
(* Int *)
let test = sum filtered_list
```

First, a `filter_ann` function is defined, taking as first parameter a characteristic function for the set `'a | 'b` whose type precises that the elements in `'b` do not satisfy the predicate, and as second parameter a list of `'a | 'b` elements. Our prototype is able to infer that the result is a list of elements in `'a \ 'b`. This is only possible thanks to occurrence typing: for deriving this type, the type-checker has to deduce that when `f e` is `True`, then `e` has type `a \ 'b`.

This `filter_ann` function is then used to remove `nil` values in a list whose elements are either integers or `nil` values. The resulting type captures the fact that the list does not contain `nil` values anymore. This allows applying the function `sum` on this new filtered list.

Note that this precise typing of the `filter_ann` function requires type annotations. By removing type annotations from the parameters of the definition `filter_ann`, then the reconstruction does not terminate after a minute. An intermediate case is when only the second parameter `l` is annotated:

```
(* (('b -> Any) -> ['b*] -> ['b*]) &
   (('a -> Any \ True) -> ['a*] -> [ ]) *)
let rec filter f (l:['a*]) =
    match l with
    | :Nil -> nil
    | (e,l) ->
        if f e is True
        then (e, filter f l)
        else filter f l
    end
```

This type annotation may seem trivial, but actually it significantly reduces the search space as the prototype does not explore additional specific cases for `l` (when `l` is the empty list `[ ]`, when it is a singleton list `['a]`, etc.).

The type inferred for `filter` is less precise than the one we get for `filter_ann`, but it is still more precise than the traditional type `('a -> Bool) -> ['a*] -> ['a*]`. In particular, the branch `('a -> Any \ True) -> ['a*] -> [ ]` captures the fact that if the characteristic function is never true for elements in `'a`, then filtering a list of `'a` elements yields an empty list.

The last example is an implementation of a deep flatten:

```
(* ([ ] -> 'e -> 'e) & (['c] -> 'd -> ('c,'d)) & (['a+] -> 'b -> ('a,T))
   where T = ('a,T) | 'b *)
let rec concat x y =
  match x with
  | :Nil -> y
  | (h, t) -> (h, concat t y)
```

```
  end
(* ['a*] -> ['b*] -> ['a* ; 'b*] *)
let concat : ['a*] -> ['b*] -> ['a* ; 'b*] = concat

type Tree 'a = ('a \ List) | [(Tree 'a)*]

(* (Tree 'a -> [('a \ List)*]) & ('b \ List -> ['b \ List]) *)
let rec flatten x =
    match x with
    | :Nil -> nil
    | (h, t) & :List -> concat (flatten h) (flatten t)
    | _ -> [x]
    end
(* Tree 'a -> ['a*] *)
let flatten : Tree 'a -> ['a*] = flatten

(* [(1--7)*] *)
let flatten_test = flatten [[1;[2];3];4;[5;6;[];[7]]]
```

This code first defines a `concat` function. The type inferred for it is overloaded, separating the cases where the first argument is an empty list, a singleton list, and a non-empty list. Another interesting thing to note is that it does not require the second argument to be a list: indeed, the second argument could be anything as it is never inspected. While this type is correct, we might prefer for `concat` the simpler and more readable type `['a*] -> ['b*] -> ['a* ; 'b*]`. This is the purpose of the second top-level definition that explicitly specify this type using a top-level type annotation.

Then, the `flatten` function transforms arbitrary nested lists into the list of their elements. Greenberg (2019) considers this function to be the ultimate test for any type system: as he explains, this simple polymorphic function defies all type systems since of all existing languages, none can reconstruct a type for it and only a couple of languages can check its explicitly typed version: CDuce and Haskell (the latter by resorting to complex metaprogramming constructions). Our system reconstructs a precise type for `flatten` as shown by the first arrow in its intersection type, which states that `flatten` is a function that takes a tree (i.e., either a list of elements that are trees, or a value different from a list) and returns the list of elements of the tree that are not lists; the other arrow of the intersection states that when flatten is applied to an element different from a list, then it returns the list containing only that element.

Note that the type of `flatten` is reconstructed without needing any type annotation. However, for readability reasons, we have rewritten the inferred type: the type actually printed by the prototype does not feature any reference to `Tree 'a`, but instead *redefines* it: the problem of recognizing patterns of previously-defined type aliases for the pretty-printing is hard, and our prototype delegates this task to

	Naive	Pruning	Caching	P + C
`typeof`	817	38	227	**30**
`eval`	> 100 000	836	466	**325**
`eval_ann`	130	135	**42**	49
`map`	> 100 000	214	196	**112**
`map_ann`	53	62	32	**30**
`filter`	494	424	**198**	205
`filter_ann`	64	77	**22**	**22**
`filtered_list`	11	9	**5**	**5**
`concat`	522	185	112	**101**
`concat_ann`	39	50	20	**19**
`flatten`	2 006	499	338	**259**
`flatten_ann`	112	113	**40**	47
`flatten_test`	16	16	**8**	9
`bal`	21 560	19 678	2 263	**2 174**

Figure 9.4: Time taken for the reconstruction (in milliseconds)

the CDuce library.

The type of `flatten` is then changed for a simpler one (the second branch is cut) using top-level type annotations, and it is tested on a tree whose leaves are the integers from 1 to 7. As expected, the prototype gives to the result the type `[(1--7)*]`.

This example also illustrates the necessity to keep types simple. This is done here using top-level type annotations, though it could also be achieved using type annotations for the parameters of `concat` and `flatten`. If we keep the more complex overloaded type inferred by the prototype for `concat`, then it becomes unable to infer the type of `flatten` in a reasonable time (unless it is annotated): this is because it tries to explore even more cases when reconstructing the type of `concat (flatten h) (flatten t)`. This limitation will be discussed in Chapter 10.

9.2.2 Performance

Our prototype focuses on proximity with the formalization, rather than on performance: we used it mainly to explore and test our system, which is why it is implemented in a purely functional style with persistent data structures. Nonetheless, the optimizations presented in Chapter 8 have been implemented in order to mitigate the cost of backtracking and branching. This section aims to evaluate performance of the prototype and the impact of these optimizations.

Figure 9.4 regroups the times taken to reconstruct the types of the definitions of the previous section, and for an additional longer definition `bal` available at the end of this section. Times are written in milliseconds and correspond to the native version of the prototype.

The first column lists some definitions of the previous section (suffixed with `_ann` when parameters have been explicitly annotated with their types). The second column contains the time taken for type checking each definition, in a version of the prototype where caching and intersection nodes pruning have been disabled (cf. Chapter 8). Note that the type simplification heuristics and type decompositions pruning are still effective (these are lightweight optimizations that we do not measure in this section). The third column contains times for an implementation with pruning enabled and caching disabled, the fourth column contains times for an implementation with caching enabled and pruning disabled, and the last column contains times for the actual prototype with both pruning and caching enabled.

The first definition, `typeof`, illustrates the efficiency of pruning. This example only contains constants and type-cases (resulting from the encoding of pattern-matching). Without pruning, for each possible combination of branch selection of the type-cases, the corresponding type for the parameter `x` is explored: `Any`, `Any \ Nil`, `Any \ String`, ..., `Any \ Nil \ String`, ... This high number of branches implies a lot of redundancy (some atoms are typed many times under equivalent contexts), which can be partially mitigated by implementing caching. However, the smallest reconstruction time is obtained with pruning: instead of exploring all the types listed above for `x`, the most specific cases are explored first (`Nil`, `String \ Nil`, `Char`, `Int`, `Bool`, `Any \ String \ Char \ Int \ Bool`) and the other cases are trimmed as they do not cover new domains. In addition to a faster typing, this results in a simpler type. Without pruning, the type inferred is a huge intersection of arrow types, featuring one arrow type per combination of type-case branch selection. This huge intersection can be reduced using type simplification heuristics, but this takes time and may still yield a less readable type. Indeed, the type inferred for `typeof` when the pruning is disabled is the following:

```
(Bool | Char | Nil -> "Boolean" | "Char" | "Nil") &
(Any \ (Bool | Char | [Char+]) -> "Nil" | "Number" | "Object") &
(Any \ (Bool | Char | [Char+]) -> "Nil" | "Number" | "Object") &
(Any \ (Bool | Int | [ ]) -> "Char" | "Object" | "String") &
(Bool | Int | [Char+] -> "Boolean" | "Number" | "String")
```

While we can check that this type is equivalent to the more intuitive type `(Nil -> "Nil") & ([Char+] -> "String") & ...`, it is far less readable as the domain of the branches are not disjoint.

The second definition, `eval`, cumulates type-cases with more complex atoms (projections, applications). As `eval` is a recursive function, it is transformed into a stub function that takes a function as first parameter (intuitively, itself), before applying the fixpoint combinator. Without pruning nor caching, the reconstruction of the stub function generates too many intersection branches for it to be typed in a reasonable time (the execution has been stopped after 100 seconds). Surprisingly, adding caching only (without pruning) is enough to drastically reduce the reconstruction time below a second. In addition to avoiding redundancy by not

reconstructing and/or retyping atoms under equivalent contexts, using caching has another emerging consequence: it reduces the number of intersection branches generated by the reconstruction algorithm. Indeed, by reusing previous reconstruction results and types as much as possible, it reduces the number of different type variables and different substitutions generated by the reconstruction algorithm. In other words, it makes similar branches to generate similar results, which reduces the complexity of types and the number of different type variables overall. Pruning can also be used to avoid the combinatorial explosion, though it implies more overhead than caching. While caching *passively* reduces the number of intersection branches generated by increasing the sharing of type variables, pruning *actively* tries to trim redundant branches. In this case, these two optimizations are complementary: the best inference time is obtained by combining them.

The same observations can be made for the other non-annotated recursive functions: `map`, `filter`, `concat`, and `flatten`. Though `concat` can be typed in less than a second even without pruning nor caching, it can be worth annotating it in order to obtain a simpler, not overloaded type. Indeed, if we keep in the context the overloaded type inferred for `concat`, our prototype fails to infer a type for the non-annotated version of `flatten` in a reasonable time (the fixpoint application timeouts because the type of the stub has too many branches and type variables). This illustrates that user type annotations are not only useful for speeding up the type inference, but also for ensuring that types remain as simple as possible.

The annotated versions of the recursive functions are a lot faster to type-check: even the implementation without pruning nor caching types them in less than 150ms. For those annotated versions, adding pruning does not improve performance: as the parameters are annotated with their types, the main reconstruction algorithm does not generate any new intersection node. Similarly, definitions that do not feature lambda-abstractions, such as `filtered_list` and `flatten_test`, are fast to type-check as there is no parameter type to infer.

Lastly, the `bal` definition, whose code is written below, is a longer example combining several pattern-matching expressions and type-cases (6 different pattern matches and 4 type-cases). This function is adapted from the `bal`(ance) function used in the module `Map` of the OCaml standard library (OCaml, 2023). For each type-case and pattern to match, the reconstruction algorithm decomposes the type of the associated binding variable. These type decompositions compose, yielding a wide annotation tree: the implementation without caching takes about 20 seconds to reconstruct it. In contrast to the previous examples, this one illustrates a combinatorial explosion due to type decompositions (because of the type-cases and pattern matches), while for the previous examples it was mostly due to intersection nodes (because of the inference of the types of the parameters). This is the reason why our prototype shows poor performance for this example even in the presence of type annotations. The caching system has a huge impact on performance, dividing the type-checking time by almost 10. This is quite expected, as caching partially mitigates the negative impact of type decompositions: though a type decomposition duplicates a whole subtree of the annotation tree, thus causing redundancy as

the same atom must now be typed in several branches, a significant part of this redundancy can be removed with memoization.

```
(* <T> is a "magic" expression of type T *)
let (>=) = <Int -> Int -> Bool>
let (>) = <Int -> Int -> Bool>
let invalid_arg = <String -> Empty>

atom key

type T 'a =
  Nil | (T 'a, Key, 'a, T 'a, Int)

let height x =
  match x with
  | :Nil -> 0
  | (_,_,_,_,h) -> h
  end

let create l x d r =
  let hl = height l in
  let hr = height r in
  (l, x, d, r, (if hl >= hr then hl + 1 else hr + 1))

let bal (l:T 'a) (x: Key) (d:'a) (r:T 'a) =
  let hl = match l with :Nil -> 0 | (_,_,_,_,h) -> h end in
  let hr = match r with :Nil -> 0 | (_,_,_,_,h) -> h end in
  if hl > (hr + 2) then
    match l with
    | :Nil -> invalid_arg "Map.bal"
    | (ll, lv, ld, lr, _) ->
      if (height ll) >= (height lr) then
        create ll lv ld (create lr x d r)
      else
        match lr with
        | :Nil -> invalid_arg "Map.bal"
        | (lrl, lrv, lrd, lrr, _)->
          create (create ll lv ld lrl) lrv lrd (create lrr x d r)
        end
    end
  else if hr > (hl + 2) then
    match r with
    | :Nil -> invalid_arg "Map.bal"
    | (rl, rv, rd, rr, _) ->
```

```
      if (height rr) >= (height rl) then
        create (create l x d rl) rv rd rr
      else
        match rl with
        | :Nil -> invalid_arg "Map.bal"
        | (rll, rlv, rld, rlr, _) ->
          create (create l x d rll) rlv rld (create rlr rv rd rr)
        end
    end
  else (l, x, d, r, (if hl >= hr then hl + 1 else hr + 1))

let bal : T 'a -> Key -> 'a -> T 'a -> T 'a = bal
```

Conclusion While the simple optimizations seen in Chapter 8 significantly improve performance, they are still far from what would be considered acceptable for real applications. To be used in mainstream languages, the type system has to be adapted and restricted to ensure better and uniform performance. To this purpose, once the type of a function has been inferred, its parameters should be annotated accordingly so that the whole type inference does not need to run again on this function. This way, although it may take some time to type-check a function for the first time as the types of its parameters need to be inferred, subsequent calls to the type-checker should be faster. This could be further coupled with slicing to limit the combinatorial explosion that may be caused by the type decompositions in large functions such as `bal`: our reconstruction could be applied to very delimited regions that would limit the possibility of backtracking. Pieces of code that are estimated to be independent of the rest (in terms of occurrence typing) could be enclosed in blocks that contain their own copies of bindings, and so type decompositions would not escape those blocks and thus would not compose with type decompositions of other blocks. Additionally, we believe that some more language-oriented optimization techniques could be of help. An example is what the development team of Luau did on the occasion of its recent switch to semantic subtyping (Jeffrey, 2022). The developers did this switch by implementing a two-phase approach: first, a sound syntactic system, fast but imprecise, is used to try to prove subtyping, and only if it fails, the computationally expensive semantic subtyping inference is used (Luau targets the game developers' community, which is quite picky on system responsiveness). We think not only that such a staged approach could be applied in our case, but also that the partial results of the first phase could be used to improve performance of the later phases. These techniques are language-dependent, and quite different from the algorithmic aspects developed here, though they will completely rely on it.

Chapter 10

Discussion and Conclusion

Contents

This manuscript describes an approach for typing dynamic languages using expressive types, featuring ad hoc polymorphism, parametric polymorphism, and occurrence typing. This implies, however, that type inference cannot be complete (in particular due to the lack of principality). This contrasts with other approaches, such as MLstruct (Parreaux and Chau, 2022), that favors a decidable inference with principal typing, at the cost of expressivity.

This choice of expressivity over principality is motivated by idioms of dynamic languages, such as overloading, and by frequent patterns that require intersection of arrow types and occurrence typing in order to be typed, such as the `filter` example described in Chapter 9. The price to pay is the absence of principal types and the complexity of inference. Among the multiple types derivable for an expression, the user may sometimes have to find a sweet spot between precision and simplicity and write a type annotation accordingly. These limitations are discussed in Section 10.1. Section 10.2 explains in more details the different sources of incompleteness of the type reconstruction, and how user type annotations could be used to retrieve the expressivity of the algorithmic type system. Section 10.3 provides a comparison between our approach and related work. Finally, Section 10.4 concludes this manuscript and describes future work that may be necessary in the perspective of applying this type system to a real programming language.

10.1 Limitations

The system presented in this manuscript has several limitations, which are detailed in this section.

Incompleteness. The first limitation that comes in mind is probably the incompleteness of the reconstruction algorithm (Chapter 6), which derives from the lack of type principality. This has already been discussed in Section 6.5.2, and we saw in Chapter 9 an example of function for which no principal type exists: the recursive function `map`. Incompleteness can be mitigated by adding explicit type annotations in the language: this will be discussed in Section 10.2.

Performance. The reconstruction algorithm uses backtracking, and at each of its passes it may try to re-type the same piece of code several times. This is inherent to the use of unions and intersections: the union-elimination rule repeatedly type-checks the same expression, using different type hypotheses for a given subexpression; the intersection-introduction rule verifies that an expression has all the types of an intersection, by checking each of them separately. Both features are very penalizing in terms of performance, and any naive implementation of the reconstruction algorithm would yield type-inference times that grow exponentially with the size of the program. Clearly, this is an issue that must be addressed if we want to apply our system to real-world dynamic languages, and further work is needed to frame and/or constrain the current system so that its performance becomes acceptable. Fortunately, the room for improvement is significant: the prototype presented in Chapter 9 is a proof of concept whose implementation was defined to faithfully simulate the reconstruction inference rules, rather than to obtain an efficient execution; but the simple addition of textbook memoization techniques improved its performance by an order of magnitude (cf. Section 9.2).

Side effects. Another limitation of our system is that it is not sound in the presence of side effects. Indeed, the union-elimination rule regroups different occurrences of the same subexpression into the same variable, and decomposes the type of this variable. This is sound only if all evaluations of these different occurrences return results whose types are all in the same part of the decomposition. While this is true for pure expressions, where syntactically equivalent expressions evaluate to the same value, this can be invalidated by the presence of side effects. From the perspective of the algorithmic type system, this transpires in the transformation of an initial expression into its MSC form. As we explain in Section 5.1, these forms are called "maximal sharing" since all equivalent subexpressions (in the sense stated by Definition 60) of the initial expression must be bound by the same variable, so that any refinement of the type of one subexpression (e.g., as a consequence of a type-case) is passed-through to all equivalent subexpressions. Again, this is sound only if all evaluations of equivalent subexpressions return results that have the same types. In the future work section (Section 10.4), we suggest some research directions on how to modify the equivalence relation of Definition 60 to make our system work in the presence of side effects. Nevertheless, the work presented here is closer to be adapted/adaptable to pure functional languages such as Erlang and Elixir, than to languages such as JavaScript or Python. It may also be worth pointing out that our

approach works only for strict languages, since it uses a semantic subtyping relation that is unsound for call-by-name evaluation strategies (Petrucciani et al., 2018).

10.2 Towards completeness

While the reconstruction is not complete, the algorithmic type system has been proved complete with respect to the declarative one: any program typeable with the declarative type system can be typed with the algorithmic one, provided that the right annotation tree is used. Consequently, by giving the user a way to control the different aspects of the annotation tree, for instance through type annotations in the language, we could expect any program typeable with the declarative type system to be typeable with the actual type-checker implementation. We detail here different aspects of the annotation trees that may need feedback from the user, and discuss, for each, how it can be integrated in the language.

Types of λ-abstraction parameters The user should be able to indicate the domain of a λ-abstraction (or, in the case of an overloaded function, the list of domains to explore independently). This has already been formalized in Chapter 7 (Section 7.2) and implemented in the prototype.

While the reconstruction algorithm is able, in many cases, to infer a precise type for a λ-abstraction without indications from the user, this is at the cost of performance: type-checking is significantly slower for λ-abstractions with no type annotation (see Section 9.2). Moreover, the reconstruction algorithm must be combined with some heuristics in order to improve its performance, in particular the pruning of intersection nodes and the simplification of the tallying solutions. These heuristics may impact the inference of the types of the parameters, and they might be subject to changes as the language and type-checker evolve.

Consequently, in order to ensure good performance and stability of the typing (that is, if a program type-checks with a given version of the type-checker, it should still type-checks with later versions), we believe that function parameters should be explicitly annotated by their type. Instead of a fully autonomous system, the reconstruction algorithm could then be used as an interactive tool, integrated into the development environment, which would suggest domains (or arrow types) to the programmer for each λ-abstraction. The programmer could then simply pick, among the suggested types, the ones to be turned into type annotations.

Having explicit type annotations also participates into having simpler and more readable types, as only the domains selected by the programmer are considered. For instance, the `Any -> [ ] -> [ ]` branch of the `map` function (cf. Section 9.2) probably does not cover any use-case the programmer had in mind when writing this function. In addition, explicit type annotations allow for better error messages, as they indicate to the reconstruction algorithm which domain a function is supposed to cover, and the reconstruction algorithm can then trigger a precise error message if the body of the function is not typeable for a part of this domain.

Type decompositions for the different subexpressions In addition to the type of λ-abstraction parameters, the annotation trees for the algorithmic type system also specify the type decompositions to apply on bind definitions. In the reconstruction algorithm, these type decompositions are inferred from type-cases and are then propagated to the different bindings involved.

However, type decompositions are sometimes useful even in the absence of type-case. For instance, consider the following program:

```
let test =
    let b = bool () in
    lOr (neg b) b
```

where `bool` has type `Unit -> Bool`, `lOr` has type `(True -> Bool -> True) & (False -> (True -> True) & (False -> False))`, and `neg` has type `(True -> False) & (False -> True)`.

Intuitively, `test` should be typed `True`, as `lOr (neg b) b` yields `true` for any Boolean value of `b`. However, in order to derive the type `True` for `test`, the type of `b` must be decomposed in two parts, `True` and `False`. This decomposition is not inferred by the reconstruction algorithm, as the program does not contain any type-case, and thus the type `Bool` is inferred instead of `True`.

In order to increase the expressivity of our type system, we can give the programmer the ability to impose a type decomposition on any subexpression. This feature has been added in the prototype, inferring the type `True` for the following definition:

```
let test =
    let b = (bool () : True ; False) in
    lOr (neg b) b
```

The syntax `(bool () : True ; False)` means that the type of the subexpression `bool ()` should be decomposed into `True` and `False`.

Here, the decomposition `True ; False` already covers the type of the expression `bool ()`, so the reconstruction algorithm only has to split the type of the binding variable `x` associated to `bool ()`: when reconstructing the atom corresponding to `(bool () : True ; False)`, it simply yields a $\texttt{Split}(\{\texttt{x} : \texttt{True}\}, \texttt{typ}, \texttt{typ})$ result.

However, if the user writes a split annotation `(e : T1 ; ... ; Tn)` where the decomposition `T1 ; ... ; Tn` does not cover the type of `e`, the reconstruction algorithm first tries to turn the type of `e` into a subtype of `T1 | ... | Tn` using tally_infer(.). This notation `(e : T1 ; ... ; Tn)` is thus a generalization of type constraints defined in Section 7.4, that allows the user to express a type constraint and to force a type decomposition at the same time.

Note that a type decomposition could also be forced just by introducing a (semantically insignificant) type-case, for instance:

```
let test =
    let b = bool () in
    if b is True then lOr (neg b) b else lOr (neg b) b
```

Instantiations for polymorphic applications When typing an application $x_1\ x_2$, the algorithmic type system instantiates x_1 and x_2 according to two sets of substitutions Σ_1 and Σ_2 in the annotation tree.

These two sets of substitutions are computed by the reconstruction system using tallying. Even though the tallying algorithm is complete, the instance may admit no solution even for valid applications, because it may be necessary to expand the type of x_1 and/or x_2 (see Section 6.5.2).

For instance, the application `f id` with `f:(Int -> Int) & (Bool -> Bool) -> Any` and `id:'a -> 'a` can only be reconstructed by our system if the type of `id` is expanded to `('a -> 'a) & ('b -> 'b)` (in which case the reconstruction system finds the substitution $\{$`'a` $\rightsquigarrow$ `Int` ; `'b` $\rightsquigarrow$ `Bool`$\}$). Sometimes, the application can be typed without performing expansion, but with a resulting type that is less general than if an expansion was performed. This is the case, for instance, when typing `f x` with `f:('a -> 'a) -> ('a -> 'a)` and `x:(Int -> Int) | (Bool -> Bool)`. If no expansion is performed, the type reconstructed for this application is `Empty -> Empty`, while it could be `(Int -> Int) | (Bool -> Bool)` by performing one expansion to the type of `f`.

Some systems perform expansion automatically based on heuristics, such as (Castagna et al., 2015, Appendix C.2.3). However, this offers no formal guarantees, and adds even more complexity to the (already complex) reconstruction algorithm. Performing an expansion is expensive, as it can make the tallying instance much longer to solve (the number of substitutions needed to characterize the solutions may greatly increase as the number of polymorphic type variables increases).

Our solution is to perform an expansion only when it is suggested by an explicit annotation from the programmer. The programmer can annotate an expression with an explicit conjunction of types, for instance for the previous example, `f` could be annotated with the types `(Int -> Int) -> (Int -> Int)` and `(Bool -> Bool) -> (Bool -> Bool)`. The type system then checks that these annotated types are each (supertype of) an instance of the type of `f`. Then, it types the application by considering that `f` has type `((Int -> Int) -> (Int -> Int)) & ((Bool -> Bool) -> (Bool -> Bool))`. As this type is a conjunction of two different instances of `f`, it couldn't have been inferred by a tallying instance without performing expansion.

This feature has been added to the prototype. Our example can be written as follows, where `test_expansion` does not perform any expansion while `test_expansion_annot` performs one expansion on the type of `f` according to the user annotation:

```
let f = < ('a -> 'a) -> ('a -> 'a) >
```

```
let x = < (Int -> Int) | (Bool -> Bool) >
(* Arrow *)
let test_expansion = f x
(* (Int -> Int) | (Bool -> Bool) *)
let test_expansion_annot =
    (f :> ((Int -> Int) -> (Int -> Int)) ;
          ((Bool -> Bool) -> (Bool -> Bool))) x
```

In this code, the syntax `(expr :> T1 ; T2)` denotes a (double) type coercion: the type system checks that the type inferred for `expr` has an instance that is a subtype of `T1`, and that it has another instance that is a subtype of `T2`. If so, it types the expression `(expr :> T1 ; T2)` with the type `T1 & T2`. There can be more than two types: for an expression `(expr :> T1 ; ... ; Tn)`, the type system checks each type `T1`, `T2`, ..., `Tn` separately, and then uses the conjunction of all the `Ti` as a type for `(expr :> T1 ; ... ; Tn)`.

Note that the type coercion operator `(expr :> T1 ; ... ; Tn)` should not be confused with the type constraint/split operator `(expr : T1 ; ... ; Tn)` introduced in the previous point. Indeed, the former *checks* that `expr` has type `T1`, ..., `Tn` and then uses the *conjunction* of these types as a type for `(expr :> T1 ; ... ; Tn)`. The latter *ensures* (if possible) that the type of `expr` is a subtype of the *disjunction* of all the `Ti`, and then ensures that the type of `expr` is decomposed according to the `Ti`.

Here is another example, where this time an expansion is needed for the argument of the application:

```
(* 'a -> 'a *)
let id x = x
(* (True | 123, True | 123) *)
let test_expansion2 =
    let f = (fun x -> (x 123, x true)) in
    f id
(* (123, True) *)
let test_expansion2_annot =
    let f = (fun x -> (x 123, x true)) in
    f (id :> (123 -> 123) ; (True -> True))
```

In the definitions `test_expansion2` and `test_expansion2_annot`, the function `f` is typed `(123 -> 'a) & (True -> 'b) -> ('a,'b)`. Giving to this function the identity function (`'a -> 'a`) as parameter without performing any expansion yields the resulting type `(True | 123, True | 123)` (both `'a` and `'b` must be substituted by the same type `True | 123`), while with an expansion of the argument we obtain the type `(123, True)`.

Conclusion Although our type system does not have principality of typings, and our reconstruction algorithm is incomplete, we can still reach the expressivity of the declarative type system by requiring explicit type annotations in some situations: the programmer should provide explicit type annotations when an expansion is needed, and when the type system should perform a type decomposition that does not originate from a type-case. It should also be recommended to annotate the type of function parameters to make type-checking faster and to avoid relying on the heuristics used by the type inference, as these heuristics may change in the future. This could be done through an interactive process where the reconstruction algorithm suggests different domains for a function, and the programmer selects the ones they need.

10.3 Related work

10.3.1 Formalizations using set-theoretic types

Our work is based on the formalization for set-theoretic types introduced by Frisch (2004). In particular, it relies on the definition and implementation of subtyping, and on the notion of DNF for computing type operators. Set-theoretic types have later been extended with type variables (Castagna and Xu, 2011), and used to design a set-theoretic type system featuring parametric polymorphism (with explicitly-typed λ-abstractions) by Castagna et al. (2014, 2015). The latter work also introduces the tallying problem and an algorithm for deciding it. The approach developed in this manuscript reuses the ideas of these articles, but with two new objectives: performing type narrowing, and featuring a type inference capable of reconstructing types for polymorphic and overloaded functions.

Type narrowing In our approach, the essence of occurrence typing is captured by the combination of the union-elimination rule with three simple rules for type-cases ($[\mathbb{0}]$, $[\in_1]$ and $[\in_2]$). In the algorithmic type system, the union-elimination rule is implemented by using MSC forms. We first explored these ideas in a system without parametric polymorphism and with a limited type inference in Castagna et al. (2022b).

Other approaches based on set-theoretic types implement type narrowing differently. In particular, Schimpf et al. (2023) and Castagna et al. (2023) have developed set-theoretic type systems for the languages Erlang and Elixir respectively. These languages feature guarded pattern matching, allowing to express dynamic conditions over the type of an expression. When typing the branch associated to a guarded pattern, some assumptions can be made about the type of the variables appearing in the guard. The type systems developed by Schimpf et al. (2023) and Castagna et al. (2023) perform type narrowing directly in the typing rule for patterns. As the exact set of values captured by a guard cannot always be expressed as a type (in particular, guards can test the equality between two variables), they define an operator to compute a type that over-approximates this set, and another operator to compute

a type that under-approximates it. Those operators are then used to perform type narrowing and to check the exhaustiveness of a pattern-matching. This is different from our approach, where type narrowing is not performed at the level of type-cases (or pattern-matching expressions) but upstream, using the union-elimination rule to decompose the type of variables. While our approach is more general and fully captures the essence of occurrence typing, allowing to narrow the type of any subexpression (applications, projections, etc.), it comes at the cost of performance, as applying the union-elimination rule requires typing the same subexpression multiple times.

Inference of function types Inferring the type of λ-abstractions in a set-theoretic type system is not very common: most set-theoretic type systems use explicitly-typed λ-abstractions. Still, this problematic has been explored by Castagna et al. (2016b) and further developed by Petrucciani (2019). In their work, type inference is performed by encoding expressions into a language of constraints, mostly consisting in subtyping constraints. Then, these constraints are solved all at once by an algorithm that uses tallying. This approach is faster than our, as it does not require backtracking, however it only infers single-arrow types for λ-abstractions (overloaded behaviors are not captured). In comparison, our work does infer intersection types for functions, capturing overloaded behaviors, but this has a cost in terms of performance since backtracking is necessary during the reconstruction. Note that backtracking might still be necessary in the approach of Petrucciani (2019), but in a more subtle way. Indeed, for a given top-level definition, the set of constraints generated by their inference system may have several solutions, in which case one solution is chosen over the others. Sometimes, this choice might be challenged later, requiring either to backtrack in order to make another choice, or to fail and ask for user type annotations.

10.3.2 Other formalizations

Using set-theoretic types is not the only option to define subtyping. In the section, we compare our type system with other approaches that do not rely on semantic subtyping. In particular, we focus on (*i*) work on the union-elimination rule and on occurrence typing, (*ii*) work on intersection types and the expression of overloaded behaviors, and (*iii*) work on type inference for Hindley-Milner systems and intersection type systems.

Union-elimination and occurrence typing The use of a union-elimination rule in a type system has been studied by Barbanera et al. (1995). In addition to a union-elimination rule, their type system features a rule for the introduction of unions and for the elimination of intersections: this is different from our approach, where the introduction of unions and the elimination of intersections derive from the subtyping relation on set-theoretic types (rule $[\leqslant]$). The novelty of our work

is to combine the union-elimination rule with three basic rules for type-cases, thus capturing the essence of occurrence typing.

This approach to occurrence typing is very different from the approach of Tobin-Hochstadt and Felleisen (2008, 2010) implemented in Typed Racket. In this and subsequent work, types are annotated by two logical propositions that record the type of the input depending on the (Boolean) value of the output. For instance, the type of the `number?` function states that when the output is `true`, then the argument has type `Number`, and when the output is `false`, the argument does not. These propositions are propagated and used in conditionals to refine the type of variables in the "then" and "else" branches. However, this analysis only focuses on a particular set of pure operations. Contrary to these works, we try not to depend on an external logic but, rather, to express these conditions with set-theoretic types. Our approach is more global since, not only our analysis strives to infer type information by analyzing all types of results (and not just `true` or `false`), but also tries to perform this analysis for all possible expressions (and not just for a restricted set of operations). This allows our system to type all the examples given in Tobin-Hochstadt and Felleisen (2010) and, contrary to the cited work, to do so without needing any type annotation.

Intersection types and overloaded behaviors The use of trees to annotate calculi with full-fledged intersection types is common. In the presence of explicitly-typed overloaded functions, one must be able to precisely describe how the types of nested λ-abstractions relate to the various "branches" of the outermost function. The work most similar to ours is Liquori and Ronchi Della Rocca (2007), since the deductions are performed on pairs of marked terms and proof terms. A marked term is an untyped term where variables are marked with integers and a proof term is a tree that encodes the structure of the typing derivation and relates marks to types. Other approaches, such as Ronchi Della Rocca (2002); Wells et al. (2002); Bono et al. (2008), duplicate the term typed with an intersection, such that each copy corresponds exactly to one member of the intersection. Lastly, the work of Wells and Haack (2002) does not duplicate terms but rather decorate λ-abstractions with a richer concept of *branching shape* which essentially allows one to give names to the various branches of an overloaded function and to use these names in the annotations of nested λ-abstraction. Note that none of these works features type reconstruction, which was our main motivation to eschew annotations within terms, since the backtracking nature of our reconstruction would imply rewriting terms over and over.

Work by Oliveira et al. (2016) and Rioux et al. (2023) study disjoint intersection and union types. They allow expressing overloaded behaviors by a general deterministic merge operator (which may also work on non-functional values, such as records). In our work, we do not have a general merge operator: overloaded behaviors only emerge through the use of type-case expressions (or the application of an overloaded function). Our work can be extended with pattern-matching, in which

case the first matching branch is selected. This is an approach different from the one used with disjoint intersection types, where branches are disjoint and have no priority, and where ambiguous programs are rejected using a notion of mergeability and distinguishability. These notions make it possible to define a general merge operator and to support nested composition, which may be useful in some contexts such as compositional programming (Zhang et al., 2021).

Type inference Our type reconstruction algorithm combines several ideas from prior works on type inference. When thinking about type inference, the first type systems that come in mind are Hindley-Milner type systems. Hindley-Milner type systems have first-order prenex polymorphism, principal types, and type inference is decidable: Damas and Milner (1982) describes a simple method, referred as algorithm $\mathcal{W}$, for inferring types using unification. Our current approach for reconstruction takes inspiration from algorithm $\mathcal{W}$, where λ-abstracted variables are first given a fresh type $\boldsymbol{\alpha}$, which is then substituted as required while going through the body. Algorithm $\mathcal{W}$ uses unification to compute, for each application, the substitution to apply to the current context in order for this application to be typeable. In our case, we rely on tallying instead of unification as we have semantic subtyping. The main difference is that, while unification either returns one principal substitution or fail, tallying returns a set of substitutions. The consequence is that, while algorithm $\mathcal{W}$ can just apply the substitution to its context and continue typing the rest of the expression, we need to explore the different solutions using an intersection annotation[1].

Inference for ML systems with subtyping, unions, and intersections has been studied in MLsub (Dolan and Mycroft, 2017) and extended with richer types and a limited form of negation in MLstruct (Parreaux and Chau, 2022). Both work trade expressivity in favor of principality. They define a lattice of types and an algebraic subtyping relation that ensures principality, but forbids intersections of arrow types. This precludes them from capturing overloaded behavior of functions, but allows them to define a polymorphic type inference with principal types. We justify our choice of set-theoretic types, with no principality and a complex inference, by our aim to type dynamic languages, such as Erlang or JavaScript, where overloading plays an important role. We favor the expressivity necessary to type many idioms of these languages, and rely on user-defined annotations when necessary to compensate for the incompleteness of type inference.

Jim (2000) presents a polar type system which features intersections and parametric polymorphism. In Jim's type system, quantifiers may appear only in positive positions in types, while intersections may only appear in negative positions. This yields a system that is more expressive than rank-2 intersection types, and therefore more expressive than ML. Furthermore, the system features principal types, and a decidable type inference. Some aspects of this work are similar to ours, in particular the use of MGS, an algorithm to compute the most general solution to a (syntactic)

[1] In addition, it is necessary to backtrack in order to retype some intermediate definitions.

sub-typing problem, that plays the same role as our tallying algorithm. Despite these similarities, the approaches differ in the kind of programs they handle: in Jim (2000), intersections are only deduced by applying higher-order function parameters to arguments of distinct types within the body of a function, while in our approach, they can also be caused by a type-case.

In a recent work, Parreaux et al. (2024) define $F_{\{\leqslant\}}$, a λ-calculus with first-class polymorphism (in particular, quantification is not prenex and can happen in the left-hand side of an arrow) and MLstruct-like subtyping. It features intersections and unions, but the use of those is restricted: intersections can only appear in negative positions, and unions can only appear in positive positions. They show that this is equivalent to having first-class polymorphism with multiple bounds on type variables. They define a type inference for this system, SuperF, inspired in part by the polar type system of Jim (2000). In practice, SuperF yields better results than all first-class-polymorphic type inference systems proposed so far, although it is incomplete: for instance, it can infer the type $(123, \texttt{True})$ for the expression $(\lambda x.(x\ 123, x\ \texttt{True}))\ \texttt{id}$. This approach takes a different direction from ours: while we choose to restrict polymorphism to be prenex, they use unrestricted first-order polymorphism but restrict the use of intersections and unions in negative and positive positions respectively (which makes it impossible to capture the behavior of an overloaded function at top-level).

Ângelo and Florido (2022) provide a principal type inference for a type system with rank-2 intersection types[2]. In their work, overloaded behaviors are expressible using intersection types, but they are limited by the rank-2 restriction. Union types are not supported, nor are equirecursive types (actually, their work does not feature a general notion of subtyping between two arbitrary types). Their inference does not require backtracking: it generates a set of constraints that are then solved using a *set unification algorithm*. This approach for inference has some similarities with the one by Castagna et al. (2016b) improved and further developed by Petrucciani (2019) in a context with set-theoretic types, where the set unification algorithm is replaced by tallying in the presence of subtyping.

10.3.3 Dynamic languages

The language Julia is dynamic, offering the possibility to test the type of an expression at run-time, and yet functions can be annotated with types. However, these type annotations are not intended for type safety, but rather semantics, since they are used to determine, at run-time, which definition of an overloaded function should be called depending on the type of the parameters (dynamic dispatch). The dynamic dispatch of Julia is quite powerful as it takes into account the type of all the arguments passed to the function, following a policy referred as *triangular dispatch*. However, this can lead to two kinds of errors at run-time: (*i*) no definition of the function is compatible with the types of the arguments given, or (*ii*) more

[2] A type t satisfies the rank-2 restriction if no path from the root of the syntactic tree of t to an intersection passes to the left of 2 or more arrows.

than one definition of the function is compatible with the types of the arguments given, and none of these definitions is strictly more specific than the others (thus, which function to call is ambiguous). We believe our type system could be applied to a language such as Julia in order to detect some of these errors statically: our types are expressive enough to capture those of Julia (in particular, we have singleton types and union types) and to capture the behavior of complex overloaded functions. However, modeling the dispatch system of Julia using set-theoretic types may be complex: in Julia, adding a definition for a function may reduce its domain, as it may introduce new ambiguous cases that are rejected at run-time. It is different from the behavior of the extended type-cases introduced in Chapter 7, where the first branch that applies is selected (rather than the more specific one). Some features of Julia are also missing from our type system and would require future work, such as the support for abstract data types with nominal subtyping (cf. Section 10.4).

Other work is aimed at adding a static type system to an existing dynamic language. This is the case of TypeScript (Microsoft) for JavaScript, or Mypy (Jukka Lehtosalo) for Python (Python already has a syntax for type annotations, but no static type checker). Unfortunately, these approaches lack a formal foundation. They are not designed to provide strong static guarantees (the type checker of TypeScript is unsafe on many aspects, prioritizing flexibility for the programmer and time performance over type safety), but rather to detect some type inconsistencies (but not all of them), and to provide type information for the documentation and the toolchain. Mypy provides a type inference for the parameters of functions, but it is limited and hard to predict. TypeScript does not provide such type inference. We hope that our work could form the basis of a formal static type system for dynamic languages, providing more expressive types (Python does not have intersection types, and neither Python nor TypeScript have negation types), occurrence typing, type inference, and strong type safety guarantees. However, our type system cannot be applied as it stands to these languages: in addition to the missing features (row polymorphism, support for side effects, etc.), compromises will have to be found in order to obtain homogeneous time performance and not to constrain the programmer (for instance with the integration of gradual typing, cf. Section 10.4).

Another work that adds a static type system to an existing dynamic language is Luau. Luau uses a type system featuring semantic subtyping to statically type Lua code. In addition to the safety guarantees it brings, the static type information is used by their interpreter to perform optimizations. Their implementation of semantic subtyping, which they call *pragmatic semantic subtyping*, is inspired by set-theoretic types, though it differs from the set-theoretic interpretation for functions (Jeffrey, 2022). As with set-theoretic types, they support singleton types, unions, and intersections. However, negation is only supported on test types (those that can appear in their type-cases), and not on structural types (in particular, arrow types). Their type system is gradual: it features a type any that can be used as an arbitrary type (it should not be mistaken with our top type $\mathbb{1}$, which they call unknown). Overall, they have opted for a pragmatic approach that restricts some

features of set-theoretic types (in particular, the absence of arbitrary negation types is a limitation for implementing our general approach for occurrence typing) and with a limited type inference, but with an efficient type checking algorithm that shows homogeneous performance.

10.4 Conclusion and future work

This work aims at providing a formal and expressive type system for dynamic languages, where type-cases can be used to give functions an overloaded behavior. It features a type inference that mixes both parametric polymorphism (for modularity) and intersection polymorphism (to capture overloaded behaviors). In that sense, our work is more than a simple study on typeability: as a matter of fact, monomorphic intersection and union types are sufficient to type a closed program where all function applications are known, but this would be bad from a language design point of view, and it is the reason why people program using ML-style programming languages rather than intersection based ones. Separate compilation and modular definitions are requirements of any reasonable programming language. The essence of this work is thus to challenge the limits of how much precision one can obtain (through intersection types)—ideally precise enough to type idioms of dynamic languages—while preserving modularity (thanks to parametric polymorphism).

While this work is a step towards a better static typing of dynamic languages, several key features are still missing. First, the presence of side effects may invalidate our approach: if the $[\vee]$ rule in Figure 4.1 is applied to two different occurrences of an expression e' that is not pure, then the rule may type an expression that yields a run-time type error. This can be seen on the algorithmic system, where the transformation into a MSC form binds the two occurrences of e' to the same variable, thus wrongly assuming that they both yield the same result. Strictly speaking, our algorithmic approach does not require expressions to be pure; it just needs that when two occurrences of an expression may produce two distinct values, then these two occurrences must be bound by two different binds. Having only pure expressions is a straightforward way to satisfy this property. Having each subexpression bound to a distinct variable (i.e., no sharing, that is, a less precise system, in which the union rule is never used) is a way to retain safety in the presence of side effects. But between these two extrema, there is a whole palette of less coarse solutions that make it possible to apply our approach in the presence of side effects. This poses two main challenges: (*i*) how to separate problematic expressions from non-problematic ones which, in terms of the type system, corresponds to characterizing a class of subexpressions e' that can be safely used in rule $[\vee]$; and (*ii*) how to do so *before* our type inference, namely when putting expressions into their MSC form, at a point when type information is not available, yet.

Second, one may want to add gradual typing (Siek et al., 2015) to our type system. Indeed, even though we have (an incomplete) type inference, the programmer still has to write type-safe code and to insert, sometimes, type annotations. During

a phase of experimentation, the programmer might prefer to locally drop the guarantees of static typing in favor of a more hassle-free programming style. Also, gradual typing could be of help in the presence of language features that are notoriously difficult to type, such as the `eval` function: if we are able to estimate the variables in the context that may be impacted by a call to `eval`, we may use gradual types to express uncertainty about the type of these variables after the call to `eval`. In addition, gradual typing is necessary if we want to apply this type system to a language with an already large codebase, as this codebase (and in particular, libraries) may not be typeable out of the box and thus, may require consequent work over time to be annotated. A successful use of gradual typing to this purpose has been achieved in TypeScript (Microsoft), where typed codebases can freely interoperate with untyped ones, making the transition from JavaScript to TypeScript a lot easier but at the expense of type safety. The objective of TypeScript is to add a layer of static types on top of a dynamic language without worrying about blame when a type inconsistency happens at run-time: this differs from the approach of Siek et al. (2015) that aims to preserve a form of type safety by adding casts at runtime in order to detect type inconsistencies as early as possible. The formalization of gradual typing in set-theoretic types has been studied by Castagna and Lanvin (2017) and Castagna et al. (2019). Roughly, it consists in adding a new type constructor `?` (sometimes called "unknown type" or "dynamic type") and a materialization rule that allows substituting any occurrence of `?` by any type. From an algorithmic perspective, the DNF of types must be extended in order to account for `?`, and new type operators have to be defined in order to get rid of the materialization rule (which is not syntax-directed nor analytic) by "embedding" it in other rules whenever needed.

Third, depending on the targeted language, one has to consider adding support for abstract data types with nominal subtyping. Such an extension is required, for instance, to type the language Julia, where the user can define abstract data types and write subtyping relations between them (for instance, `abstract type SomeType <: SuperType end`). Abstract data types can be encoded in our current type algebra using type variables: each abstract data type can be associated to a fresh monomorphic type variable, and we can encode nominal subtyping between two abstract data types by using an intersection. For instance, two abstract types `T1 <: T2` can be represented by the types $\boldsymbol{\alpha}_1$ and $\boldsymbol{\alpha}_1 \wedge \boldsymbol{\alpha}_2$ respectively (with $\boldsymbol{\alpha}_1$ and $\boldsymbol{\alpha}_2$ two fresh monomorphic type variables). Unfortunately, this encoding does not suffice anymore in the presence of parametric data types (or equivalently, generic types). The way to implement them may vary depending on the targeted language. In some languages, parameters of generic types can be qualified with a variance: they can be bivariant, covariant, contravariant, or invariant. This is the case, for instance, of Python and OCaml. In other languages, such as Java and Julia, parameters of generic types cannot be qualified with a variance (the type system treat them as if they were invariant), but at the level of methods, bounded polymorphism can be used in place of a built-in notion of variance: for instance, in the Julia function definition `f{T<:Real}(a::Array{T})`, the type `Array{T}` is incomparable with every type `Array{S}` where `S` is distinct from `T` (invariance), but `T` can be arbitrarily

instantiated by any type smaller than `Real`.

Fourth, an important missing feature is the support of row-polymorphism: while the present work already supports records (cf. Chapter 7), the precise typing of functions operating over records requires row-polymorphism. This is especially important for dynamic languages where records are seamlessly used to encode both objects and dictionaries. A first step in that direction may be to integrate the work by Castagna (2023), which unifies dictionaries and records.

Fifth, the performance of the type reconstruction can certainly be improved by using more sophisticated implementation techniques and heuristics on the lines we outlined at the end of Chapter 9. Before being applicable to a real-world language, our type-checker should ensure uniform performance. While this requirement seems hard to reach for a fully unannotated codebase where the type of each function has to be inferred each time it is typed, it seems a lot more realistic to achieve it for an annotated codebase, where type inference would only be a tool for the programmers to annotate their functions.

Lastly, an important aspect that has not been discussed is the pretty printing of types and the generation of error messages. Finding a safety error through static typing is an interesting feature, but it is pointless if we cannot locate it precisely and display a clear error message to the programmer. Generating intuitive error messages is complicated when dealing with set-theoretic types: types can be very long, in particular recursive ones, and it may be difficult to recognize a user-defined type alias inside a more complex type (especially for parametric type aliases). Still, heuristics could be used to improve the readability of types. For instance, when the programmer annotates a definition using a type alias, a special care should be taken to recognize this type alias when pretty-printing the types within this definition. Furthermore, this pretty-printing information should be forwarded to later code that uses this definition. In other words, even though we use semantic subtyping, it may be worth storing some syntactic information about the types and use it for pretty printing. For what concerns the localization of errors, in the absence of user type annotations, we must also rely on heuristics to prioritize the different possible origins of errors. Even when a definition is well-typed, it might be worth raising an error: for instance, $\lambda x.\ (x{\in}\texttt{Int})\,?\,x\,:\,(42\ 42)$ is well-typed (the type inferred is $\texttt{Int} \wedge \alpha \to \texttt{Int} \wedge \alpha$), but it is highly improbable for the expected domain of this λ-abstraction to be `Int` as it makes the second branch of the type-case unreachable. These practical considerations are independent of the core type system presented in this manuscript, and their answers may depend on the language we are targeting.

To conclude, set-theoretic types are starting to be integrated into real-world languages, for instance by Schimpf et al. (2023) for Erlang, by Jeffrey (2022) for Luau, and by Castagna et al. (2023) for Elixir. We believe that, in the future, our work could be used in these systems in order to benefit from a more precise typing of type-cases and pattern-matching, as well as by providing an optional type inference that can be used in conjunction with explicit type annotations.

Bibliography

Pedro Ângelo and Mário Florido. 2022. Type Inference For Rank-2 Intersection Types Using Set Unification. In *Theoretical Aspects of Computing – ICTAC 2022: 19th International Colloquium, Tbilisi, Georgia, September 27–29, 2022, Proceedings* (Tbilisi, Georgia). Springer-Verlag, Berlin, Heidelberg, 462–480. https://doi.org/10.1007/978-3-031-17715-6_29 (Cited in page 217.)

Franco Barbanera, Mariangiola Dezani-Ciancaglini, and Ugo de'Liguoro. 1995. Intersection and Union Types. *Inf. Comput.* 119, 2 (June 1995), 202–230. https://doi.org/10.1006/inco.1995.1086 (Cited in pages vi, 10, 12 and 214.)

Viviana Bono, Betti Venneri, and Lorenzo Bettini. 2008. A typed lambda calculus with intersection types. *Theor. Comput. Sci.* 398, 1-3 (2008), 95–113. https://doi.org/10.1016/j.tcs.2008.01.046 (Cited in page 215.)

Giuseppe Castagna. 2020. Covariance and Controvariance: a fresh look at an old issue (a primer in advanced type systems for learning functional programmers). *Logical Methods in Computer Science* 16, 1 (2020), 15:1–15:58. https://doi.org/10.23638/LMCS-16(1:15)2020 (Cited in pages 22 and 168.)

Giuseppe Castagna. 2023. Typing Records, Maps, and Structs. *Proc. ACM Program. Lang.* 7, ICFP, Article 196 (Sept. 2023), 45 pages. https://doi.org/10.1145/3607838 (Cited in page 221.)

Giuseppe Castagna. 2024. *Programming with Union, Intersection, and Negation Types.* Springer International Publishing, Cham, 309–378. https://doi.org/10.1007/978-3-031-34518-0_12 (Cited in pages 7 and 20.)

Giuseppe Castagna, Guillaume Duboc, and José Valim. 2023. The Design Principles of the Elixir Type System. *The Art, Science, and Engineering of Programming* 8, 2 (Oct. 2023). https://doi.org/10.22152/programming-journal.org/2024/8/4 (Cited in pages 28, 213 and 221.)

Giuseppe Castagna and Alain Frisch. 2005. A Gentle Introduction to Semantic Subtyping. In *Proceedings of the 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming* (Lisbon, Portugal) *(PPDP '05)*. Association for Computing Machinery, New York, NY, USA, 198–208. https://doi.org/10.1145/1069774.1069793 (Cited in page 20.)

Giuseppe Castagna and Victor Lanvin. 2017. Gradual Typing with Union and Intersection Types. *Proc. ACM Program. Lang.* 1, ICFP, Article 41 (Aug. 2017), 28 pages. https://doi.org/10.1145/3110285 (Cited in page 220.)

Giuseppe Castagna, Victor Lanvin, Mickaël Laurent, and Kim Nguyen. 2022a. Revisiting Occurrence Typing. *Science of Computer Programming* 217 (mar 2022),

102781. https://doi.org/10.1016/j.scico.2022.102781 arXiv:1907.05590 (Cited in pages 10, 24 and 100.)

Giuseppe Castagna, Victor Lanvin, Tommaso Petrucciani, and Jeremy G. Siek. 2019. Gradual typing: a new perspective. *Proc. ACM Program. Lang.* 3, POPL, Article 16 (jan 2019), 32 pages. https://doi.org/10.1145/3290329 (Cited in page 220.)

Giuseppe Castagna, Mickaël Laurent, and Kim Nguyen. 2024a. Polymorphic Type Inference for Dynamic Languages. *Proceedings of the ACM on Programming Languages* 8, POPL (2024), 40. https://doi.org/10.1145/3632882 (Cited in page 10.)

Giuseppe Castagna, Mickaël Laurent, and Kim Nguyen. 2024b. *Prototype: Polymorphic Type Inference for Dynamic Languages.* https://doi.org/10.5281/zenodo.11203176 (Cited in pages 9, 185 and 194.)

Giuseppe Castagna, Mickaël Laurent, Kim Nguyen, and Matthew Lutze. 2022b. On Type-Cases, Union Elimination, and Occurrence Typing. *Proc. ACM Program. Lang.* 6, POPL, Article 13 (jan 2022), 31 pages. https://doi.org/10.1145/3498674 (Cited in pages 10 and 213.)

Giuseppe Castagna, Kim Nguyen, Zhiwu Xu, and Pietro Abate. 2015. Polymorphic functions with set-theoretic types. Part 2: local type inference and type reconstruction. In *Proceedings of the 42nd Annual ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '15)*. 289–302. https://doi.org/10.1145/2676726.2676991 (Cited in pages 8, 11, 24, 25, 33, 118, 144, 168, 211 and 213.)

Giuseppe Castagna, Kim Nguyen, Zhiwu Xu, Hyeonseung Im, Sergueï Lenglet, and Luca Padovani. 2014. Polymorphic Functions with Set-Theoretic Types. Part 1: Syntax, Semantics, and Evaluation. In *Proceedings of the 41st Annual ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '14)*. 5–17. https://doi.org/10.1145/2676726.2676991 (Cited in pages 8, 11, 33, 154 and 213.)

Giuseppe Castagna, Tommaso Petrucciani, and Kim Nguyễn. 2016a. Set-Theoretic Types for Polymorphic Variants. *SIGPLAN Not.* 51, 9 (sep 2016), 378–391. https://doi.org/10.1145/3022670.2951928 (Cited in page 22.)

Giuseppe Castagna, Tommaso Petrucciani, and Kim Nguyen. 2016b. Set-Theoretic Types for Polymorphic Variants. In *ICFP '16, 21st ACM SIGPLAN International Conference on Functional Programming*. 378–391. https://doi.org/10.1145/2951913.2951928 (Cited in pages 214 and 217.)

Giuseppe Castagna and Zhiwu Xu. 2011. Set-theoretic Foundation of Parametric Polymorphism and Subtyping. In *ICFP '11: 16th ACM-SIGPLAN International*

Conference on Functional Programming. 94–106. https://doi.org/10.1145/2034773.2034788 (Cited in pages 8, 17, 22 and 213.)

CDuce . *The CDuce Compiler*. CDuce. https://www.cduce.org (Cited in pages 28, 168 and 194.)

Mario Coppo, Mariangiola Dezani-Ciancaglini, and Betti Venneri. 1981. Functional Characters of Solvable Terms. *Mathematical Logic Quarterly* 27, 2-6 (1981), 45–58. https://doi.org/10.1002/malq.19810270205 (Cited in pages vi and 12.)

Luis Damas and Robin Milner. 1982. Principal Type-Schemes for Functional Programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)* (Albuquerque, New Mexico). Association for Computing Machinery, New York, NY, USA, 207–212. https://doi.org/10.1145/582153.582176 (Cited in pages ix, 10, 15, 24, 118 and 216.)

Stephen Dolan and Alan Mycroft. 2017. Polymorphism, Subtyping, and Type Inference in MLsub. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL '17)*, Giuseppe Castagna and Andrew D. Gordon (Eds.). Association for Computing Machinery, New York, NY, USA, 60–72. https://doi.org/10.1145/3009837.3009882 (Cited in page 216.)

Ecma. 2021. ECMAScript® 2021 Language Specification. https://262.ecma-international.org/12.0/ (Cited in page 7.)

Facebook . *Flow*. Facebook. https://flow.org/ (Cited in page 6.)

Alain Frisch. 2004. *Théorie, conception et réalisation d'un langage de programmation adapté à XML*. Ph. D. Dissertation. Université Paris Diderot. (Cited in pages v, 8, 11, 17, 22, 33, 148, 149, 164 and 213.)

Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. 2008. Semantic subtyping: dealing set-theoretically with function, union, intersection, and negation types. *Journal of the ACM* 55, 4 (Sept. 2008), 19:1–19:64. http://doi.acm.org/10.1145/1391289.1391293 (Cited in pages 3, 8, 20, 24 and 70.)

Nils Gesbert, Pierre Genevès, and Nabil Layaïda. 2015. A logical approach to deciding semantic subtyping. *ACM Transactions on Programming Languages and Systems* 38, 1 (2015), 3. https://doi.org/10.1145/2812805 (Cited in page 20.)

Aviral Goel, Pierre Donat-Bouillud, Filip Křikava, Christoph M. Kirsch, and Jan Vitek. 2021. What we eval in the shadows: a large-scale study of eval in R programs. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 125 (oct 2021), 23 pages. https://doi.org/10.1145/3485502 (Cited in page 5.)

Michael Greenberg. 2019. The Dynamic Practice and Static Theory of Gradual Typing. In *3rd Summit on Advances in Programming Languages (SNAPL 2019)*

(Leibniz International Proceedings in Informatics (LIPIcs), Vol. 136). 6:1–6:20. https://doi.org/10.4230/LIPIcs.SNAPL.2019.6 (Cited in page 200.)

Fritz Henglein. 1993. Type Inference with Polymorphic Recursion. *ACM Trans. Program. Lang. Syst.* 15, 2 (apr 1993), 253–289. https://doi.org/10.1145/169701.169692 (Cited in page 117.)

J. Roger Hindley. 1969. The Principal Type-Scheme of an Object in Combinatory Logic. *Trans. Amer. Math. Soc.* 146 (1969), 29–60. (Cited in pages vi and 12.)

Haruo Hosoya, Alain Frisch, and Giuseppe Castagna. 2005. Parametric Polymorphism for XML. *Conference Record of the Annual ACM Symposium on Principles of Programming Languages* 40, 50–62. https://doi.org/10.1145/1040305.1040310 (Cited in page 20.)

Alan Jeffrey. 2022. Semantic Subtyping in Luau. Blog post. https://blog.roblox.com/2022/11/semantic-subtyping-luau Accessed on May 6th 2023. (Cited in pages 205, 218 and 221.)

Trevor Jim. 2000. A Polar Type System. In *ICALP Workshops 2000, Proceedings of the Satelite Workshops of the 27th International Colloquium on Automata, Languages and Programming, Geneva, Switzerland, July 9-15, 2000*, José D. P. Rolim, Andrei Z. Broder, Andrea Corradini, Roberto Gorrieri, Reiko Heckel, Juraj Hromkovic, Ugo Vaccaro, and Joe B. Wells (Eds.). Carleton Scientific, Waterloo, Ontario, Canada, 323–338. (Cited in pages 216 and 217.)

Jukka Lehtosalo . *Mypy*. Jukka Lehtosalo. https://mypy.readthedocs.io/en/stable/ (Cited in pages 5 and 218.)

Julia. . *The Julia Programming Language*. https://docs.julialang.org/ (Cited in pages 217 and 220.)

A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. 1993. Type Reconstruction in the Presence of Polymorphic Recursion. *ACM Trans. Program. Lang. Syst.* 15, 2 (apr 1993), 290–311. https://doi.org/10.1145/169701.169687 (Cited in page 117.)

Laurie Kirby and Jeff Paris. 1982. Accessible Independence Results for Peano Arithmetic. *Bulletin of the London Mathematical Society* 14, 4 (1982), 285–293. https://doi.org/10.1112/blms/14.4.285 arXiv:https://londmathsoc.onlinelibrary.wiley.com/doi/pdf/10.1112/blms/14.4.285 (Cited in page 140.)

Luigi Liquori and Simona Ronchi Della Rocca. 2007. Intersection-types à la Church. *Inf. Comput.* 205, 9 (2007), 1371–1386. https://doi.org/10.1016/j.ic.2007.03.005 (Cited in page 215.)

Luau. . *Lua*u. https://luau-lang.org/ (Cited in pages 205 and 218.)

David MacQueen, Gordon Plotkin, and Ravi Sethi. 1986. An ideal model for recursive polymorphic types. *Information and Control* 71, 1 (1986), 95–130. https://doi.org/10.1016/S0019-9958(86)80019-5 (Cited in pages vi, 12 and 41.)

Per Martin-Löf. 1994. *Analytic and Synthetic Judgments in Type Theory*. Springer Netherlands, Dordrecht, 87–99. https://doi.org/10.1007/978-94-011-0834-8_5 (Cited in pages viii and 14.)

Microsoft a. *The Monaco Editor*. Microsoft. https://microsoft.github.io/monaco-editor/ (Cited in page 194.)

Microsoft b. *TypeScript*. Microsoft. https://www.typescriptlang.org/ (Cited in pages 6, 218 and 220.)

Robin Milner. 1978. A theory of type polymorphism in programming. *J. Comput. System Sci.* 17, 3 (1978), 348–375. https://doi.org/10.1016/0022-0000(78)90014-4 (Cited in pages vi, 12, 38, 63 and 189.)

OCaml. 2023. Standard Library: Map module. Github repository. https://github.com/ocaml/ocaml/blob/trunk/stdlib/map.ml (Cited in page 203.)

Bruno C. d. S. Oliveira, Zhiyuan Shi, and João Alpuim. 2016. Disjoint Intersection Types. *SIGPLAN Not.* 51, 9 (sep 2016), 364–377. https://doi.org/10.1145/3022670.2951945 (Cited in page 215.)

Lionel Parreaux, Aleksander Boruch-Gruszecki, Andong Fan, and Chun Yin Chau. 2024. When Subtyping Constraints Liberate: A Novel Type Inference Approach for First-Class Polymorphism. *Proc. ACM Program. Lang.* 8, POPL, Article 48 (jan 2024), 33 pages. https://doi.org/10.1145/3632890 (Cited in page 217.)

Lionel Parreaux and Chun Yin Chau. 2022. MLstruct: Principal Type Inference in a Boolean Algebra of Structural Types. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 141 (oct 2022), 30 pages. https://doi.org/10.1145/3563304 (Cited in pages 207 and 216.)

Tommaso Petrucciani. 2019. *Polymorphic Set-Theoretic Types for Functional Languages*. Ph. D. Dissertation. Joint Ph.D. Thesis, Università di Genova and Université Paris Diderot. https://tel.archives-ouvertes.fr/tel-02119930 Available at https://tel.archives-ouvertes.fr/tel-02119930. (Cited in pages 33, 214 and 217.)

Tommaso Petrucciani, Giuseppe Castagna, Davide Ancona, and Elena Zucca. 2018. Semantic Subtyping for Non-Strict Languages. In *TYPES18: 24th International Conference on Types for Proofs and Programs (LIPIcs, Vol. 130)*, Peter Dybjer, José Espírito Santo, and Luís Pinto (Eds.). 4:1–4:24. (Cited in page 209.)

Python documentation. 2021. What's New In Python 3.10. https://docs.python.org/3/whatsnew/3.10.html (Cited in page 163.)

Nick Rioux, Xuejing Huang, Bruno C. d. S. Oliveira, and Steve Zdancewic. 2023. A Bowtie for a Beast: Overloading, Eta Expansion, and Extensible Data Types in F⋈. *Proc. ACM Program. Lang.* 7, POPL, Article 18 (jan 2023), 29 pages. https://doi.org/10.1145/3571211 (Cited in page 215.)

John Alan Robinson. 1965. A Machine-Oriented Logic Based on the Resolution Principle. *J. ACM* 12, 1 (jan 1965), 23–41. https://doi.org/10.1145/321250.321253 (Cited in page 118.)

Simona Ronchi Della Rocca. 2002. Intersection Typed lambda-calculus. *Electr. Notes Theor. Comput. Sci.* 70, 1 (2002), 163–181. https://doi.org/10.1016/S1571-0661(04)80496-1 (Cited in page 215.)

Amr Sabry and Matthias Felleisen. 1992. Reasoning about programs in continuation-passing style. *SIGPLAN Lisp Pointers* V, 1 (jan 1992), 288–298. https://doi.org/10.1145/141478.141563 (Cited in page 91.)

Albert Schimpf, Stefan Wehr, and Annette Bieniusa. 2023. Set-Theoretic Types for Erlang. In *Proceedings of the 34th Symposium on Implementation and Application of Functional Languages* (Copenhagen, Denmark) *(IFL '22)*. Association for Computing Machinery, New York, NY, USA, Article 4, 14 pages. https://doi.org/10.1145/3587216.3587220 (Cited in pages 213 and 221.)

Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. 2015. Refined Criteria for Gradual Typing. In *1st Summit on Advances in Programming Languages (SNAPL 2015) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 32)*. 274–293. https://doi.org/10.4230/LIPIcs.SNAPL.2015.274 (Cited in pages 219 and 220.)

Christopher Strachey. 1967. Fundamental concepts in programming languages. Lecture notes for International Summer School in Computer Programming, Copenhagen. (Cited in page 6.)

Sam Tobin-Hochstadt and Matthias Felleisen. 2008. The Design and Implementation of Typed Scheme. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '08)*. ACM, New York, NY, USA, 395–406. https://doi.org/10.1145/1328438.1328486 (Cited in pages 9 and 215.)

Sam Tobin-Hochstadt and Matthias Felleisen. 2010. Logical types for untyped languages. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming* (Baltimore, Maryland, USA) *(ICFP '10)*. ACM, New York, NY, USA, 117–128. https://doi.org/10.1145/1863543.1863561 (Cited in pages 7 and 215.)

Types 2019. What exactly should we call syntax-directed inference rules? Discussion on the Types mailing list. http://lists.seas.upenn.edu/pipermail/types-list/2019/002138.html. (Cited in pages viii and 14.)

Joseph Brian Wells, Allyn Dimock, Robert J Muller, and Franklyn Albin Turbak. 2002. A calculus with polymorphic and polyvariant flow types. *J. Funct. Program.* 12, 3 (2002), 183–227. https://doi.org/10.1017/S0956796801004245 (Cited in page 215.)

Joe B. Wells and Christian Haack. 2002. Branching Types. In *ESOP '02 (LNCS, Vol. 2305)*. Springer, 115–132. https://doi.org/10.1007/3-540-45927-8_9 (Cited in page 215.)

Weixin Zhang, Yaozhu Sun, and Bruno C. D. S. Oliveira. 2021. Compositional Programming. *ACM Trans. Program. Lang. Syst.* 43, 3, Article 9 (sep 2021), 61 pages. https://doi.org/10.1145/3460228 (Cited in page 216.)