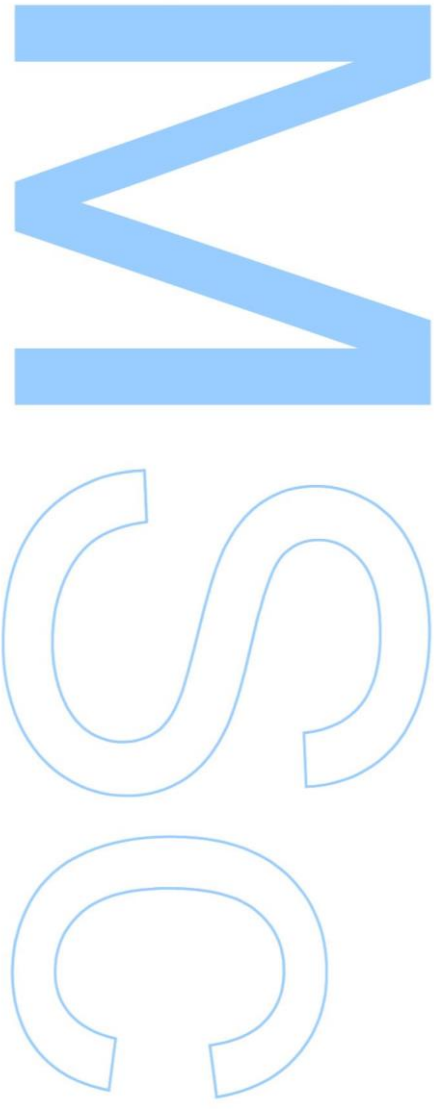
Subtyping: Study and Implementation

Rodrigo dos Reis Canedo Marques
Mestrado em Ciência de Computadores
Departamento de Ciência de Computadores
2022

Orientador
António Mário da Silva Marcos Florido, Professor Associado,
Faculdade de Ciências da Universidade do Porto

Coorientador
Pedro Baltazar Vasconcelos, Professor Auxiliar,
Faculdade de Ciências da Universidade do Porto



Resumo

Os sistemas de tipos são uma ferramenta poderosa para raciocinar sobre a correção de programas: a verificação estática de tipos pode chegar ao ponto de garantir a ausência de erros de execução. A inferência de tipos atribui automaticamente tipos aos programas, reduzindo os encargos mentais e sintácticos de um sistema de tipos e permitindo que os programadores se concentrem na lógica do programa, sendo assim mais produtivos. Alguns programas podem ser polimórficos, no sentido em que podem admitir vários tipos. Este polimorfismo é uma caraterística essencial para os programadores criarem abstracções eficazes. Uma forma muito útil de polimorfismo é a subtipagem, comummente encontrada nas linguagens orientadas para os objectos. Historicamente, a subtipagem não tem funcionado bem com a inferência de tipo completa.

Em 2016, Stephen Dolan apresentou MLsub, uma linguagem mínima com um sistema de tipos que combina polimorfismo paramétrico e subtipagem, com inferência de tipos, tipos principais compactos e subsunção decidível. A inferência de tipos baseia-se na biunificação, um análogo da unificação mas para restrições de subtipagem. Notando que a apresentação do MLsub era pesada em álgebra abstrata e conceitos avançados como a bisubstituição, em 2020 Lionel Parreux propôs que estes não são essenciais para a inferência de tipos e apresentou o Simple-sub, um algoritmo de inferência alternativo que é mais fácil de compreender. O Simple-sub assemelha-se mais à inferência de tipos tradicional de Hindley-Milner e é um algoritmo muito mais simples que pode ser implementado de forma eficiente.

Nesta dissertação, estudamos estes novos sistemas de subtipagem e reimplementamos Simple-sub em Haskell. Além disso, propomos uma extensão do Simple-sub que combina polimorfismo de linha com subtipagem. Para isso, adicionamos variáveis de linha à nossa linguagem de tipos e estendemos o método de resolução de restrições de tipo do nosso algoritmo de inferência de tipos em conformidade, mantendo a decidibilidade da inferência de tipos.

Abstract

Type systems are a powerful tool for reasoning about program correctness: static type checking can go as far as guaranteeing the absence of execution errors. Type inference automatically assigns types to programs, reducing the mental and syntactic burdens of a type system and allowing programmers to focus on the logic of the program, thus being more productive. Some programs can be polymorphic in the sense that they can admit multiple types. This polymorphism is an essential feature for programmers to create effective abstractions. One widely useful form of polymorphism is subtyping, commonly found in object-oriented languages. Subtyping has historically not played well with complete type inference.

In 2016, Stephen Dolan presented MLsub, a minimal language with a type system combining parametric polymorphism and subtyping, with type inference, compact principal types, and decidable subsumption. Type inference is based on biunification, an analog of unification but for subtyping constraints. Noting that MLsub's presentation was heavy on abstract algebra and advanced concepts such as bisubstitution, in 2020 Lionel Parreux proposed that these are not essential to type inference and presented Simple-sub, an alternative inference algorithm that is easier to understand. Simple-sub resembles traditional Hindley-Milner type inference more closely and is a much simpler algorithm that can be implemented efficiently.

In this dissertation, we study these novel subtyping systems and re-implement Simple-sub in Haskell. Furthermore, we propose an extension to Simple-sub combining row polymorphism with subtyping. For this we add row variables to our type language and extend the type constraint-solving method of our type inference algorithm accordingly, keeping the decidability of type inference.

Contents

Chapter 1

Introduction

Nearly all programming languages in use feature some form of *type system*. This is a formal system consisting of rules and axioms that assign *types* to expressions of the language. Types serve as a specification of how an expression is expected to be used, establishing the correct behavior of a program. Verifying that a program and the types of its expression adhere to the rules of the type system is referred to as *type checking*. It is in this verification step that programming languages diverge in their approach.

Some programming languages utilize *dynamic* type systems and conduct type checks during program execution. When an expression violates the rules of the type system, a *type error* will typically cause the program to crash. In contrast, other programming languages adopt *static* type systems in which type checking occurs prior to execution. This approach, however, requires additional information about types to be supplied to the type checker. This information is often conveyed in the form of *type annotations* that decorate the expressions of the program, much to the delight of some programmers and the frustration of many others.

To reduce the manual and visual burdens associated with type annotations, many languages employ *implicit typing*, in which a *type inference* algorithm deduces all or part of the types of a program from its syntax. This approach is especially common in *functional* programming languages, giving programmers of these languages the benefits of static type checking without the cost of inserting type annotations into the program. Type inference algorithms are mostly based on the Hindley-Milner algorithm [Mil78, DM82] which infers most general (principal) types given an entirely untyped program. These properties are a staple of the ML [MTHM97] family of languages, but they are hard to preserve even with simple extensions to the type system.

A famous property of the ML language is that *well-typed* programs do not generate errors during execution. However, there exist programs devoid of type errors that are not well-typed, since the type system is not *expressive* enough to convey suitable types for these programs. Thus, there is a constant endeavor to extend type systems with more expressive types that permit typing a greater number of programs. Because of the desirable properties of Hindley-Milner type inference, ML is a popular target for the study of more advanced type systems.

A feature that has long been considered difficult to integrate with the ML type system and Hindley-Milner type inference is *subtyping*, which imposes an ordering on types, creating a hierarchy of *subtypes* and *supertypes*. Recently, Dolan and Mycroft presented the MLsub type system [DM17], which successfully extends ML with subtyping, while solving many of the difficulties of subtyping and preserving the properties of Hindley-Milner type inference.

The presentation of MLsub, however, relies heavily on concepts from order theory and category theory that make it difficult to follow for those without a background in abstract algebra, and its type inference algorithm depends on novel results that depart from the traditional algorithm W for Hindley-Milner type inference [DM82]. More recently, Parreaux proposed a simpler algorithm for MLsub called Simple-sub [Par20], whose presentation more closely resembles the algorithm W.

1.1 Objectives and Contributions

With this dissertation, we set out to study novel algorithms for type inference in the presence of subtyping. While doing so we realized that previous research on the combination subtyping with row polymorphism could potentially be applied to these new systems and decided to explore this idea. The work on this dissertation has taken roughly the following route.

1. A review of type theory, from simple types and the different forms of polymorphism to type checking and inference. This served as the theoretical starting point for this work.
2. An implementation of early type inference algorithms, especially the algorithm W for Hindley-Milner type inference. This gave us a starting point from which to build the following implementations.
3. A study of Simple-sub and algebraic subtyping, and the implementation of the Simple-sub type inference algorithm in Haskell.
4. A study of type systems featuring polymorphic record operations. The focus of this study was the use of row variables to represent record polymorphism.
5. The development of a type system that combines algebraic subtyping with row polymorphism.
6. The implementation of a type inference algorithm for this type system.

The original contributions of this dissertation are as follows.

- The exploration of a type system combining algebraic subtyping with row polymorphism;
- A prototype implementation of this type system and inference algorithm.

A preliminary version of the extension outlined here was presented at the 2022 ML family workshop (co-located with ICFP 2022) [MFV22]. The source code of the developed prototype is available online on a GitHub repository at https://github.com/RodrigoMarques16/simple-sub-records.

1.2 Outline

The remainder of this dissertation is structured as follows:

Chapter 2 reviews background concepts necessary to understand this work. More precisely, it begins with a short introduction to the λ-calculus, as it and its extensions serve as the language used to present the systems studied here. Then types are introduced, based on Curry's Simple Type System, followed by an introduction to Hindley-Milner type inference. Finally, the chapter ends with a presentation of select extensions to the previous system, which will be used in the following chapters.

Chapter 3 reviews the state-of-the-art of type systems which include subtyping. The chapter starts with an informal introduction to subtyping, followed by a presentation of the MLsub type system, and ends with a presentation of the Simple-sub type inference algorithm for sub.

Chapter 4 proposes an extension to Simple-sub with extensible record operations, such as extending a record with a new field, and row variables to represent types for these operations. This chapter starts with a review of row polymorphism, followed by a presentation of the proposed extensions to the type system and a presentation of the adapted type inference algorithm, and ends with a review of alternative approaches to typing extensible records.

Chapter 5 concludes this dissertation with a discussion of the work thus far and proposes possible directions for future work in this area.

Chapter 2

Background

This chapter presents the theoretical basis necessary for the following chapters. The chapter begins with a presentation of the λ-Calculus as it is the base of the languages studied next. We begin by presenting the λ-Calculus and the simple type system. We then introduce types and the theory behind the simple type system followed by a presentation of Hindley-Milner type inference. Finally, we present some extensions to the earlier type systems that will be necessary in describing the systems in the following chapters.

2.1 The Lambda-Calculus

The λ-Calculus was first defined by Church [Chu33] in an effort to lay a foundation for logic and mathematics. It is based on function application and abstraction and is higher-order: functions can be used as arguments to other functions. While not successful as a foundation due to paradoxes [KR35] it was later shown useful as a formal model of computation [Tur37]. Since then it has been used in the study of programming language theory and served as a base for functional programming languages such as Haskell or ML.

In this section, we introduce the basics of the λ-Calculus needed throughout this dissertation. A more complete definition can be found in [Bar84].

Terms λ-terms, from now on just terms, are built from an infinite set of variables $\mathcal{V}$ using two fundamental functional operations: abstraction and application.

$$\begin{array}{rl} x & \text{(Variable)} \\ \lambda x.\ t & \text{(Abstraction)} \\ t_1\ t_2 & \text{(Application)} \end{array}$$

Variables are ranged over by x and terms by t, either with or without number subscripts.

An abstraction is a function with a parameter x and body t. An application represents applying a term t_2 as an argument to a term t_1.

The following abbreviations will be used:

$$t_1 t_2 \dots t_n \equiv ((t_1 t_2) \dots) t_n$$
$$\lambda x_1 \dots x_n.\ t \equiv \lambda x_1.\ (\lambda x_2.\ (\dots (\lambda x_n.\ t)))$$

Variable Scopes A variable x occurs bound when it appears in the body of an abstraction λx, otherwise, it occurs free. For example, in the term $((\lambda x.\ xy)x)$, inside the body of the abstraction x occurs bound and y occurs free, while outside x occurs free.

The set of free variables of a term is defined as follows.

$$\begin{aligned} \mathrm{fv}(x) &= \{x\} \\ \mathrm{fv}(\lambda x.\ t) &= \mathrm{fv}(t) \setminus \{x\} \\ \mathrm{fv}(t_1 t_2) &= \mathrm{fv}(t_1) \cup \mathrm{fv}(t_2) \end{aligned}$$

Terms are equivalent modulo the renaming of bound variables. For example, $\lambda x.\ x \equiv \lambda y.\ y$.

Substitution The result of applying a function $\lambda x.\ t_1$ to another term t_2 is obtained by substituting all free occurrences of x in t_1 by t_2. The definition of substitution follows.

$$x[y := u] = \begin{cases} u & \text{if } y \equiv x \\ x & \text{otherwise} \end{cases}$$
$$(\lambda x.\ t)[y := u] = \begin{cases} \lambda x.\ t & \text{if } y \equiv x \\ \lambda x.\ t[y := u] & \text{otherwise} \end{cases}$$
$$(t_1\ t_2)[y := u] = (t_1[y := u]\ t_2[y := u])$$

Substitution has to be applied carefully with respect to the bounds of variables. For instance, in the substitution $(\lambda x.\ y)[y := x]$ the variable x has two distinct occurrences, one bound and one free. Naively applying the substitution has the free occurrence become bound. To avoid this one of the occurrences has to be renamed. This is called α-conversion and is defined by:

$$\lambda x.\ t \to_\alpha \lambda y.\ t[x := y] \qquad \text{if } y \text{ does not occur in } t$$

Two terms are said to be α-congruent ($\equiv_\alpha$) if they can be obtained from each other by the renaming of bound variables.

Reduction A term can be evaluated by repeatedly replacing sub-terms with simpler terms, thus reducing them. The main computational axiom of the λ-Calculus is β-reduction, it represents function application by the substitution of the bound occurrences of the function's parameter by its input, in the function's body. The single step β-reduction is defined as follows. For the multistep reduction, the symbol $\twoheadrightarrow_\beta$ will be used.

$$(\lambda x.\ t_1)\ t_2\ \rightarrow_\beta\ t_1[x := t_2]$$

A term of the form $(\lambda x.\ t_1)\ t_2$ is called a *β-redex*, from reducible expression, while $t_1[x := t_2]$ is called its *contractum*.

The following example shows how the λ-Calculus can be used as a model of computation, through abstraction, application, reduction and some abuse of notation.

$$(\lambda x.\ x + 3)\ 2 = 2 + 3 = 5$$

Normal Form A term is said to be in *normal form* if it cannot be reduced further, meaning that a term in normal form does not contain any redexes. For example, the following terms are in normal form:

x	Variable
$x\ t$	Variable applied to a term t in normal form
$\lambda x.\ t$	Abstraction where t is in normal form

If there is a chain of reductions that lead a term to a normal form then that term is said to be *normalizable*. Not all terms are normalizable, for example, $(\lambda x.\ x\ x)(\lambda x.\ x\ x)$ reduces in a single step to itself, hence a sequence of reductions of this term will never reach a normal form. However, if a term is normalizable, then by the Church-Rosser theorem [CR36] it has only one normal form. The theorem states that if a term t has two distinct reduction sequences to terms t_1 and t_2 as follows:

$$t \twoheadrightarrow_\beta t_1 \qquad\qquad t \twoheadrightarrow_\beta t_2$$

Then there must exist a term t_3 to which t_1 and t_2 can be reduced to:

$$t_1 \twoheadrightarrow_\beta t_3 \qquad\qquad t_2 \twoheadrightarrow_\beta t_3$$

This is known as the property of *confluence*.

2.2 The Simple Type System

There are two main ways to introduce types to the λ-Calculus, one attributed to Alonzo Church [Chu40] and the other to Haskell Curry [Cur34, CFC+58]. In systems *à la Church*, types are built into the term language as annotations to terms and thus each term has a unique type. In systems *à la Curry*, types are kept separate from the term language and are instead assigned to terms through inference rules.

In this section, we review Curry's simple type system for the λ-Calculus. This simple system contains only two objects: variables and arrows; but it is at the core of all systems presented in later sections. An interesting property of this system is that terms can have an infinite number of types. An in-depth study of this system is available in [Hin97].

Types The language of types is defined inductively from an infinite supply of type variables and the arrow constructor composing two types.

$$\tau \ ::= \ \alpha \mid \tau \to \tau$$

Types are ranged over by τ and type variables by α, either with or without number subscripts. The type $\tau_1 \to \tau_2$ represents all abstractions that return a term of type τ_2 when given a term of type τ_1.

The arrow is right associative thus parenthesis can be omitted as in:

$$(\tau_1 \to (\tau_2 \to (\cdots \to (\tau_{n-1} \to \tau_n)))) \equiv \tau_1 \to \tau_2 \to \cdots \to \tau_n$$

This simple language of types is enough for the terms of the pure λ-Calculus, but practical programming languages will include more terms, which means that a more complex language of types will be necessary to describe those terms. Some common extensions are presented in section 2.5.

Type Assignments A type assignment is an expression of the form:

$$t : \tau$$

where the term t is called the subject and the type τ is called the predicate. This can be read informally as "t has type τ". The following are examples of type assignments:

$$x : \alpha$$
$$\lambda x.\ x : \alpha \to \alpha$$
$$\lambda f\ x.\ f\ (f\ x) : (\alpha \to \alpha) \to \alpha \to \alpha$$

Environments A type environment, also referred to as a typing context or basis, is a finite set of type assignments $\{x_1 : \tau_1, ..., x_n : \tau_n\}$. Environments are constructed recursively, starting from the empty set:

$$\Gamma \ ::= \ \varnothing \mid \Gamma,\ x : \tau$$

The comma operator in $\Gamma,\ x : \alpha$ is used to extend an environment with a new binding as an abbreviation for $\Gamma \cup \{x : \alpha\}$, with the restriction that x does not appear in Γ. The subjects of some environment Γ can be referred to using the following function:

$$\text{Subjects}(\Gamma) = \{x_1, \ldots, x_n\}$$

The removal of an assignment whose subject is x from an environment Γ is written as:

$$\Gamma - x$$

Typing Rules A typing judgment is an expression of the form:

$$\Gamma \vdash t : \tau$$

This expression means that $t : \tau$ is derivable from the assumptions in Γ, using the following deduction rules. When Γ is empty it is omitted ($\vdash t : \tau$).

$$\text{T-Var}\ \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \text{T-App}\ \frac{\Gamma \vdash t_1 : \tau_2 \to \tau_1 \qquad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash t_1 t_2 : \tau_1} \qquad \text{T-Abs}\ \frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda x.\ t : \tau_1 \to \tau_2}$$

The rule for abstractions (T-Abs) extends the environment with a new assumption about the type of its argument. It follows that the rule for variables (T-Var) states that a variable has the type that is assumed for it in the current environment. The rule for applications (T-App) states that if t_1 is a function with arguments of type τ_2 and results of type τ_1, and t_2 has type τ_2, then the result of applying t_1 to t_2 is a value of type τ_1.

A deduction is a tree with axioms at the top and each branch being derivable from the branches directly above by one of the typing rules above. For example, a possible deduction tree for the term $(\lambda x.\ x)\ (\lambda x.\ x)$ is:

$$\text{T-App}\ \frac{\text{T-Abs}\ \dfrac{\text{T-Var}\ \dfrac{}{x : \alpha \to \alpha \vdash x : \alpha \to \alpha}}{\vdash \lambda x.\ x : (\alpha \to \alpha) \to \alpha \to \alpha} \qquad \text{T-Abs}\ \dfrac{\text{T-Var}\ \dfrac{}{x : \alpha \vdash x : \alpha}}{\vdash \lambda x.\ x : \alpha \to \alpha}}{\vdash (\lambda x.\ x)\ (\lambda x.\ x) : \alpha \to \alpha}$$

Type Substitutions Type variables, as is their purpose, can be substituted by other types. A substitution $\mathbb{S}$ is an expression of the form $[\alpha_1 := \sigma_1, ..., \alpha_n := \sigma_n]$, where each α_i is a distinct type variable. $\mathbb{S}(\tau)$ is the result of applying a substitution $\mathbb{S}$ to a type τ, by simultaneously substituting all α_i in τ by the corresponding σ_i. $\mathbb{S}(\tau)$ is called an *instance* of τ. Substitution is defined formally as follows.

$$\mathbb{S}(\alpha) = \begin{cases} \sigma & \text{if } \alpha := \sigma \in \mathbb{S} \\ \alpha & \text{otherwise} \end{cases}$$

$$\mathbb{S}(\tau_1 \to \tau_2) = \mathbb{S}(\tau_1) \to \mathbb{S}(\tau_2)$$

Two substitutions $\mathbb{S}_1$ and $\mathbb{S}_2$ can be composed as in function composition.

$$\forall \alpha.\ (\mathbb{S}_1 \circ \mathbb{S}_2)(\alpha) = \mathbb{S}_2(\mathbb{S}_1(\alpha))$$

Substitutions are assumed to be idempotent, that is, for every substitution $\mathbb{S}$ it holds that $\mathbb{S} \circ \mathbb{S} = \mathbb{S}$. The notion of substitution is extended to environments as follows.

$$\mathbb{S}(\Gamma) = \{x_1 : \mathbb{S}(\tau_1),\ \ldots,\ x_n : \mathbb{S}(\tau_n)\} \qquad \text{if } \Gamma = \{x_1 : \tau_1,\ \ldots,\ x_n : \tau_n\}$$

Typability The simple type system divides terms into two classes: those which can be assigned types and those which cannot. We have seen a typable term in $(\lambda x.\ x)\ (\lambda x.\ x)$; an example of untypable term is the self application $\lambda x.\ x\ x$. Any deduction tree for this term has a structure similar to the following, where x is assigned a type τ_1 by T-Abs, and later $\tau_1 \to \tau_2$ by T-App. Since for any τ_1 and τ_2 it holds that $\tau_1 \not\equiv \tau_1 \to \tau_2$ this breaks the consistency of the environment.

$$\dfrac{\dfrac{x : \tau_1 \vdash x : \tau_1 \to \tau_2 \qquad x : \tau_1 \vdash x : \tau_1}{x : \tau_1 \vdash x\ x : \tau_2}\ \text{T-App}}{\vdash \lambda x.\ x\ x : \tau_1 \to \tau_2}\ \text{T-Abs}$$

A term t is said to be *typable* iff there exist Γ and τ such that:

$$\Gamma \vdash t : \tau$$

An important property of type systems, which holds for the Simple Type System is *subject reduction*. Subject reduction states that if $t_1 \twoheadrightarrow_\beta t_2$, then:

$$\Gamma \vdash t_1 : \tau \implies \Gamma \vdash t_2 : \tau$$

The opposite implication is called *subject expansion* and does not hold for the simple type system. By the inductive definition of typability, we have that if t is typable then $\lambda x.\ t$ is typable and that any sub-terms of t are also typable.

Principal Types In the simple type system, a typable term can be assigned multiple types. For example the term $\lambda x.\ x$ can be assigned any type of the form $\tau_1 \to \tau_1$. But all possible types for this term are instances of the more general $\alpha \to \alpha$. This type is called the *principal type* for $\lambda x.\ x$. The principal type of a term can be thought of as a finite representation for all admissible types for that term. Formally, a type τ is the principal type for a term t if

(i) $\Gamma \vdash t : \tau$, for some Γ
(ii) If $\Gamma' \vdash t : \sigma$, for some Γ' and σ, then σ is an instance of τ

A slightly different notion is that of *principal typing*, or principal pair. This is a pair $\langle \Gamma,\ \tau \rangle$ such that $\Gamma \vdash t : \tau$ is deducible and any other deducible $\Gamma' \vdash t : \sigma$ is an instance of $\Gamma \vdash t : \tau$ (Γ' is an instance of Γ and σ is an instance of τ).

2.3 Type Inference

A type inference algorithm decides whether a given term t is typable, and if it is, outputs its type. Type inference algorithms for the simple type system date back to Curry [Cur69] and Hindley [Hin69], developed for combinatory logic. But one of the most well-known is the algorithm W, developed by Milner [Mil78] as preliminary work for the ML programming language. Milner and Damas rewrite and extend the algorithm in [DM82] and correctness proofs are provided in Damas' Ph.D. thesis [Dam84]. This algorithm extends the type system and introduces a new term to the language, but we delay their introduction to the next section.

Type inference for the simple type system needs to handle only three cases: variables, abstraction and application. The first two are straightforward, but the case for application requires more work and turns out to be the core of the algorithm.

Suppose we have a typable term $t_1 t_2$ and we know

$$\vdash t_1 : \tau_1 \to \tau_2 \qquad\qquad \vdash t_2 : \tau_3$$

If we have a substitution $\mathbb{S}$ such that

$$\mathbb{S}(\tau_1) \equiv \mathbb{S}(\tau_3)$$

Then, we can deduce the type $\mathbb{S}(\tau_2)$ for $t_1 t_2$ by the T-App rule

$$\frac{\vdash t_1 : \mathbb{S}(\tau_1) \to \mathbb{S}(\tau_2) \qquad \vdash t_2 : \mathbb{S}(\tau_3)}{\vdash t_1 t_2 : \mathbb{S}(\tau_2)}\ \text{T-App}$$

This substitution that makes two types equal is called a *unifier*. Unification is central to type inference algorithms such as Hindley's and Milner's, which rely on external algorithms such as Robinson's [Rob65] unification method.

Unification A unifier between two types τ_1 and τ_2 is a substitution $\mathbb{S}$ such that $\mathbb{S}(\tau_1) = \mathbb{S}(\tau_2)$. If a unifier exists then τ_1 and τ_2 are said to be unifiable and $\mathbb{S}(\tau_1)$ (or $\mathbb{S}(\tau_2)$) is a unification of τ_1 and τ_2.

A unifier $\mathbb{S}$ is the most general unifier if for any other unifier $\mathbb{S}_1$ there exists a substitution $\mathbb{S}_2$ such that $\mathbb{S}_1 = \mathbb{S}_2 \circ \mathbb{S}$.

A presentation of Robinson's unification algorithm [Rob65] applied to types follows. The algorithm computes the most general unifier between two types or fails if no such unifier exists.

$$\begin{aligned}
\text{unify}(\alpha, \tau) &= \begin{cases} [\alpha := \tau] & \text{if } \alpha \neq \tau \text{ and } \alpha \notin ftv(t) \\ \text{Id} & \text{if } \alpha = \tau \\ \text{fail} & \text{otherwise} \end{cases} \\
\text{unify}(\tau, \alpha) &= \text{unify}(\alpha, \tau) \\
\text{unify}(\sigma_1 \to \sigma_2, \tau_1 \to \tau_2) &= \text{let } \mathbb{S} = \text{unify}(\sigma_2, \tau_2) \\
&\quad \text{in unify}(\mathbb{S}(\sigma_1), \mathbb{S}(\tau_1)) \circ \mathbb{S}
\end{aligned}$$

The function $ftv(\tau)$ retrieves the free type variables in a type and Id is the identity substitution.

A Type Inference Algorithm We now present a simplified recursive definition of the algorithm $\mathcal{W}$ [Mil78] for the Simple Type System, based on Damas' presentation as algorithm $\mathcal{T}$ in his PhD thesis [Dam84]. The algorithm takes as input any untyped λ-term t and outputs a principal typing (Γ, τ) for t if it exists. Otherwise, a statement that t is not typable.

Definition: $\mathcal{T}(t) = \langle \Gamma,\ \tau \rangle$, where:

$$\begin{aligned}
\mathcal{T}(x) &= \langle \{x : \alpha\},\ \alpha \rangle, \text{ where } \alpha \text{ is a new variable} \\
\mathcal{T}(t_1\ t_2) &= \text{let } \mathcal{T}(t_1) = \langle \Gamma_1,\ \tau_1 \rangle,\ \mathcal{T}(t_2) = \langle \Gamma_2,\ \tau_2 \rangle \\
&\quad \text{let } \mathbb{S}_1 = \text{unify}(\tau_1,\ \tau_2 \to \alpha), \text{ where } \alpha \text{ is a new variable} \\
&\quad \text{let } \mathbb{S}_2 = \text{unify}(\Gamma_1,\ \Gamma_2) \\
&\quad \text{let } \mathbb{S} = \mathbb{S}_1 \circ \mathbb{S}_2 \\
&\quad \mathcal{T}(t) = \langle \mathbb{S}(\Gamma_1 \cup \Gamma_2),\ \mathbb{S}(\alpha) \rangle \\
\mathcal{T}(\lambda x.\ t_1) &= \text{let } \mathcal{T}(t_1) = \langle \Gamma_1,\ \tau_1 \rangle \\
&\quad \text{if } \{x : \tau_2\} \in \Gamma_1 \\
&\quad \text{then } \mathcal{T}(t) = \langle \Gamma_1 - x,\ \tau_2 \to \tau_1 \rangle \\
&\quad \text{else } \mathcal{T}(t) = \langle \Gamma_1,\ \alpha \to \tau_1 \rangle \text{ where } \alpha \text{ is a new variable}
\end{aligned}$$

Remark: $\mathcal{T}$ fails if and only if unification fails.

The first case simply assigns a new type variable to the occurrence of a term variable. This requires an operation to generate a new type variable. To avoid burdening the presentation with details regarding type variable generation we assume that distinct invocations of the algorithm produce distinct sequences of type variables.

The second case, function application, starts by invoking the algorithm for each sub-term t_1 and t_2. We know that t_1 must have a functional type from τ_2 to some other type thus we generate a new type variable α to represent the return type and unify τ_1 with $\tau_2 \to \alpha$. All that is left then is to join the environments of each sub-term and apply the unifier to the result.

Finally, function abstraction outputs a function type. It starts by invoking the algorithm for the body t_1 to get the return type τ_1. The input type though depends on whether x already has a type τ_2 in the environment and if not a new type variable is generated.

2.4 Damas-Milner Type System

The Damas-Milner type system was developed as preliminary work for the ML programming language by Milner [Mil78] and later improved by Damas [DM82, Dam84]. The main feature of this system is the introduction of a restricted form of *parametric polymorphism*, that is the ability to abstract a type for a type variable using a universal quantifier $\forall$.

While parametric polymorphism exists in systems such as System F [Gir72], type inference in it is not decidable. Milner achieves decidability by restricting his system to *first order polymorphism*, where $\forall$ can only appear at the outermost layer of a type and not inside it.

In this section, we present the Damas-Milner type system, its extensions and the type inference algorithm W.

Parametric Polymorphism While in the Curry Type System presented a term can be assigned an infinite number of types, a variable cannot be assigned multiple types in the same expression. For example, the term

$$(\lambda i.\ i\ i)(\lambda x.\ x)$$

should be assigned type $\alpha \to \alpha$ since, semantically, $(\lambda x.\ x)(\lambda x.\ x)$ clearly has that type, but self-application is not typable as i needs to simultaneously have type $\alpha \to \beta$ and type α. By assigning a *type scheme* to i instead, of the form $\forall\alpha.\ \alpha \to \alpha$ we can *instantiate* α as needed in each occurrence of i. To know we have to assign a type scheme to i we rewrite the expression with a new *let* construct:

$$\text{let } i = \lambda x.\ x \text{ in } i\ i$$

Now the type scheme can be instantiated with $(\alpha \to \alpha) \to (\alpha \to \alpha)$ and $\alpha \to \alpha$ and self-application can be typed.

The polymorphism introduced here with quantified type variables is called *parametric polymorphism*. A type scheme for a term describes the set of all admissible types for that term that can obtained by instantiation. As in the previous example, the identity $\lambda x.\ x$ has the type scheme $\forall\alpha.\ \alpha \to \alpha$, meaning all function types with the same input and output.

Terms The term language is extended with a *let* construct used to factor multiple occurrences of a sub-term and enable polymorphism.

$$t \ ::= \ \ldots \mid \text{let } x = t \text{ in } t$$

Type Schemes Type schemes are introduced to the languages, forming a new class of *polymorphic types* of which the class of *monomorphic types* is part of. A type scheme σ is a type quantified by zero or more type variables.

$$\sigma \ ::= \ \tau \mid \forall\alpha.\ \sigma$$

The following abbreviation will be used:

$$\forall\alpha_1 \ldots \forall\alpha_n.\tau \equiv \forall\alpha_1 \ldots \alpha_n.\tau$$

The type variables $\alpha_1, \ldots, \alpha_n$ are called the *generic variables* of σ.

It now makes sense to consider the scopes of type variable occurrences. A type variable occurs in a type scheme $\forall\alpha_1,\ \ldots,\ \alpha_n.\ \tau$ if it occurs in τ and is said to occur bound if it is one of the generic variables, otherwise, it occurs free. This notion extends to type environments where the predicates are type schemes. The function $\text{ftv}(\cdot)$ receives a type or environment and returns the set of its free type variables.

Instantiation and Generalization Type schemes can be *instantiated* to types by substitution of their generic variables, and conversely, types can be *generalized* to type schemes by quantifying free variables.

A type τ' is a *generic instance* (or equivalently: is subsumed) of a type scheme $\forall\alpha_1,\ \ldots,\ \alpha_n.\ \tau$ iff τ' can be obtained from τ by substitution of the generic variables $\alpha_1,\ \ldots,\ \alpha_n$

$$\forall\alpha_1,\ \ldots,\ \alpha_n.\ \tau \sqsupseteq \tau' \qquad \text{iff} \qquad \mathbb{S} = [\alpha_1 : \tau_1, \ldots, \alpha_n : \tau_n] \qquad \text{and} \qquad \mathbb{S}(\tau) = \tau'$$

Instantiation is extended to type schemes so that a type scheme σ' is an instance of a type scheme σ if and only if every type subsumed by σ' is also subsumed by σ.

A type τ can be *generalized* to a type scheme, with respect to an environment Γ, by quantifying its free type variables that do not occur free in Γ. The definition of generalization, denoted by $\overline{\Gamma}(\tau)$, follows.

$$\overline{\Gamma}(\tau) = \forall \alpha_1,\ \ldots,\ \alpha_n.\ \tau \qquad \text{where } \alpha_1,\ \ldots,\ \alpha_n = \mathrm{ftv}(\tau) \setminus \mathrm{ftv}(\Gamma)$$

Substitution Substitution is easily extended to type schemes with respect to variable bounds and the renaming of generic variables.

Type System The previous type system remains unchanged. Two new rules are introduced for type schemes and one for let.

$$\text{T-Inst}\ \frac{\Gamma \vdash t : \sigma_1 \qquad \sigma_2 \sqsubseteq \sigma_1}{\Gamma \vdash t : \sigma_2} \qquad \text{T-Gen}\ \frac{\Gamma \vdash t : \sigma \qquad \alpha \notin \mathrm{ftv}(\Gamma)}{\Gamma \vdash t : \forall \alpha.\ \sigma}$$

$$\text{T-Let}\ \frac{\Gamma \vdash t_1 : \sigma \qquad \Gamma_x,\ x : \sigma \vdash t_2 : \tau}{\Gamma \vdash (\text{let } x = t_1 \text{ in } t_2) : \tau}$$

We can now deduce a type for the self-application example, using a let expression and type schemes. Note that without let, self-application is still untypable in this system.

$$\frac{\vdash \lambda x.\ x : \forall \alpha.\ \alpha \to \alpha \qquad i : \forall \alpha.\ \alpha \to \alpha \vdash i\ i : \alpha \to \alpha}{\vdash \text{let } i = \lambda x.\ x \text{ in } i\ i : \alpha \to \alpha}\ \text{T-Let}$$

On the left-hand side, we deduce $\alpha \to \alpha$ for $\lambda x.\ x$ as usual and generalize it.

$$\frac{\dfrac{x : \alpha \vdash x : \alpha}{\vdash \lambda x.\ x : \alpha \to \alpha}\ \text{T-Abs}}{\vdash \lambda x.\ x : \forall \alpha.\ \alpha \to \alpha}\ \text{T-Gen}$$

On the right-hand side, we can now instantiate the scheme as needed in each occurrence of i. Taking $\Gamma \equiv \{i : \forall \alpha.\ \alpha \to \alpha\}$.

$$\frac{\text{T-Inst}\ \dfrac{\Gamma \vdash i : \forall \alpha.\ \alpha \to \alpha}{\Gamma \vdash\!: i : (\alpha \to \alpha) \to \alpha \to \alpha} \qquad \dfrac{\Gamma \vdash i : \forall \alpha.\ \alpha \to \alpha}{\Gamma \vdash i : \alpha \to \alpha}\ \text{T-Inst}}{\Gamma \vdash i\ i : \alpha \to \alpha}\ \text{T-App}$$

The Algorithm W We now present the type inference algorithm $\mathcal{W}$ from Damas and Milner. This algorithm infers a principal type for a given term t and a given set of assumptions A. In contrast to the previous algorithm, $\mathcal{W}$ produces a type τ and a substitution $\mathbb{S}$ such that

$$\mathbb{S}(\Gamma) \vdash t : \tau$$

Where τ is the most general type of t, and $\mathbb{S}(\Gamma)$ is the most general of the instances of Γ which make the derivation possible. Before presenting $\mathcal{W}$, we define functions for instantiation and generalization.

$$\begin{aligned} \text{inst}(\forall\alpha_1,\ \ldots,\ \alpha_n.\ \tau) &= \forall\beta_1,\ \ldots,\ \beta_n.\ \tau[\overline{\alpha_i := \beta_i}] && \text{where } \beta_i \text{ are new variables} \\ \text{gen}(\Gamma, \tau) &= \forall\alpha_1,\ \ldots,\ \alpha_n.\ \tau && \text{where } \alpha_i = \text{ftv}(\tau) \setminus \text{ftv}(\Gamma) \end{aligned}$$

Definition: $\mathcal{W}(\Gamma, t) = \langle \mathbb{S},\ \tau \rangle$, where:

$$\begin{aligned} \mathcal{W}(\Gamma, x) &= \text{If } \{x : \forall\alpha_1,\ \ldots,\ \alpha_n.\ \tau'\} \in \Gamma, \\ &\quad \langle [],\ \text{inst}(\forall\alpha_1,\ \ldots,\ \alpha_n.\ \tau')\rangle \\ \mathcal{W}(\Gamma, t_1\ t_2) &= \text{Let } \langle \mathbb{S}_1,\ \tau_1\rangle = \mathcal{W}(\Gamma,\ t_1) \\ &\quad \text{and } \langle \mathbb{S}_2,\ \tau_2\rangle = \mathcal{W}(\mathbb{S}_1(\Gamma),\ t_2) \\ &\quad \text{and } \mathbb{S} = \text{unify}(\mathbb{S}_2(\tau_1),\ \tau_2 \to \alpha),\ \text{where } \alpha \text{ is new} \\ &\quad \langle \mathbb{S} \circ \mathbb{S}_2 \circ \mathbb{S}_1,\ \mathbb{S}(\alpha)\rangle \\ \mathcal{W}(\Gamma, \lambda x.\ t_1) &= \text{Let } \langle \mathbb{S},\ \tau\rangle = \mathcal{W}(\Gamma \cup \{x : \alpha\},\ t_1),\ \text{where } \alpha \text{ is new} \\ &\quad \langle \mathbb{S},\ \mathbb{S}(\alpha \to \tau)\rangle \\ \mathcal{W}(\Gamma, \text{let } x = t_1 \text{ in } t_2) &= \text{Let } \langle \mathbb{S}_1,\ \tau_1\rangle = \mathcal{W}(\Gamma,\ t_1) \\ &\quad \text{and } \sigma = \text{gen}(\mathbb{S}_1(\Gamma),\ \tau_1) \\ &\quad \text{and } \langle \mathbb{S}_2,\ \tau_2\rangle = \mathcal{W}(\mathbb{S}_1(\Gamma) \cup \{x : \sigma\},\ t_2) \\ &\quad \langle \mathbb{S}_2 \circ \mathbb{S}_1,\ \tau_2\rangle \end{aligned}$$

Remark: $\mathcal{W}$ fails when any of the above conditions are not met.

2.5 Extensions

In this section, we present some useful language extensions which will be used later. These extensions help bridge the gap between the λ-Calculus and more practical programming languages.

2.5.1 Base Types

In the pure λ-Calculus there is only one set of atomic values, variables. While variables are enough to study interesting theoretical properties, programming languages need more practical values and operations to be useful. Thus, the system is extended with *primitive types*, belonging to a set $\mathcal{P}$ of names and ranged over by p.

$$\tau \ ::= \ \ldots \mid p$$

Most examples presented in the following chapters will use numeric types such as int, nat or float but also bool and string to keep things interesting.

Typing primitive values is often obvious, for instance, `"hello"` has type `string` and `True` has type `bool`, thus we assume the typing rules for primitives to be implicit. The two previous examples are easy because they belong to disjoint sets of types. On the other hand, numeric types often overlap, for example, `1` has both `int` and `nat` as possible types. In this case, `int` is more general an `nat`. The precise mechanism to determine which base type is most general will be presented in detail in chapter 3.

2.5.2 Records

An essential feature for programming languages is *compound* data structures, allowing the programmer to aggregate data in a structured way. Common structures are *pairs*, *tuples* or *arrays*, but the focus here will be on *records*. Records are prominent features of object-oriented languages, where they relate to objects and record types to class types; and also in database programming where they relate to rows.

A record essentially associates names, or *labels*, with terms, allowing terms to be referenced by label. Formally, a record is a finite mapping of labels to terms, for example:

$$\{x = 1,\ y = 2\}$$

This record has two *fields*, with labels x and y and terms 1 and 2, respectively. All the labels in a record are required to be distinct. Fields are usually unordered, so the following records are semantically equivalent.

$$\{\mathrm{x} = 1,\ \mathrm{y} = 2\} \equiv \{\mathrm{y} = 2,\ \mathrm{x} = 1\}$$

The basic operation on records is selection, written with the dot (.) operator along with the label of the field being selected.

$$\{\mathrm{x} = 1,\ \mathrm{y} = 2\}.y \equiv 2$$

Terms The term language is thus expanded as follows, where l ranges over labels, obtained from some predetermined set of names $\mathcal{L}$.

$$t \ ::= \ \ldots \mid \{l_1 = t_1,\ \ldots,\ l_n = t_n\} \mid t.l$$

The term $\{l_1 = t_1,\ \ldots,\ l_n = t_n\}$ is a record constructor with n fields of the form $l_i = t_i$, where l_i is the label and t_i is the term. The term $t.l$ is the selection of the term in the field labeled by l.

Types The type language is expanded in the same way, with record types represented as a sequence of fields of labels and types.

$$\tau \ ::= \ \ldots \mid \{l_1 : \tau_1,\ \ldots,\ l_n : \tau_n\}$$

An example of a record type follows.

$$\{\mathrm{name} = \text{"John"}, \mathrm{age} = 42\} : \{\mathrm{name} : \mathsf{string}, \mathrm{age} : \mathsf{int}\}$$

Typing Rules The monomorphic typing rules for records are simple and defined structurally on the types of each field. From now on we will often write $\overline{E}^i$ as an abbreviation for a sequence of expressions indexed by i.

$$\text{T-Rcd}\ \frac{\overline{\Gamma \vdash t_i : \tau_i}^{\,i}}{\Gamma \vdash \{\overline{l_i = t_i}^{\,i}\} : \{\overline{l_i : \tau_i}^{\,i}\}} \qquad \text{T-Sel}\ \frac{\Gamma \vdash t : \{\overline{l_i : \tau_i}^{\,i}\} \qquad l : \tau \in \{\overline{l_i : \tau_i}^{\,i}\}}{\Gamma \vdash t.l : \tau}$$

The record rule T-Rcd states that a record type is constructed from the types of its subterms, while the T-Sel rule states that a selection has the type of the selected term. The following is a derivation tree using these rules.

$$\dfrac{\dfrac{\vdash 1 : \mathrm{int} \qquad \vdash \mathtt{true} : \mathrm{bool}}{\vdash \{\mathrm{x} = 1,\ \mathrm{y} = \mathtt{true}\} : \{\mathrm{x} = \mathrm{int},\ \mathrm{y} = \mathrm{bool}\}}\ \text{T-Rcd}}{\vdash \{\mathrm{x} = 1,\ \mathrm{y} = \mathtt{true}\}.\mathrm{x} : \mathrm{int}}\ \text{T-Sel}$$

Polymorphism The record extension described so far is straightforward but monomorphic. ML-style type inference proves difficult without some form of *record polymorphism*. For instance, the following inference for a record generic selection of the field with label a:

$$\vdash \lambda r.\ r.a : \{a : \alpha\} \to \alpha$$

This type is polymorphic in the type of the field but not in the shape of the record, making this abstraction limited to records of the form $\{name = \_\}$. For example, in the following applications, the first one is valid but the second is not.

$$(\lambda r.\ r.a)\ \{a = 1\} \qquad\qquad (\lambda r.\ r.a)\ \{a = 1,\ b = \texttt{true}\}$$

Some languages remediate this issue with type annotations, passing the burden of choosing a type to the user but still requiring a record type with a fixed set of labels. A better solution is some form of *record polymorphism* that allows field selection independent of the complete shape of the record. This way there can be record types that represent sets larger than just the labels explicitly specified.

Many systems that allow for record polymorphism have been studied over the years. One such will be studied in detail in chapter 3, and another will be studied in chapter 4. A third approach, not studied here but worth mentioning is Ohori's system of *kinds* [Oho95], based on *bounded quantification* [CW85].

2.5.3 Recursive Types

Recursive functions are another feature commonly found in programming languages, but in the λ-Calculus a function definition cannot refer to itself by name as is typically done in languages that support recursive definitions. Instead, encoding recursive definitions in the λ-Calculus requires the use of a combinator that receives a function as an argument and uses that argument to apply the recursive call. One such combinator for the λ-Calculus is Curry's $\mathbf{Y}$ combinator which calculates the *fixed point* of its functional argument. The fixed point of a function f is an argument x that is mapped to itself, or $x = f\ x$. Defined as follows, Curry's combinator calculates the fixed point of the function it receives as an argument.

$$\mathbf{Y} \equiv \lambda f.\ (\lambda x.\ f\ x\ x)(\lambda x.\ f\ x\ x)$$

Recursion can be encoded by using such a combinator to unfold f as many times as needed.

$$\mathbf{Y}\ f = f\ (\mathbf{Y}\ f) = f\ (f\ (\mathbf{Y}\ f)) = f(\dots f\ (\mathbf{Y}\ f)\dots)$$

Recursive Types The definition of the **Y** combinator is not typable in the simple type system since it contains the self-application $x\ x$ as a sub-term. This term is untypable because the type of x has to be a function type whose domain is the type of x, and the simple type system cannot express a type with this property. To deal with this limitation the system can be extended with *recursive types* which allow for expressing types whose definition depends on itself. A recursive type has the form that follows, where μ is the recursion operator.

$$\mu\alpha.\ \tau$$

The variable α may occur in τ and it represents the recursive definition, that is, a recursive type $\mu\alpha.\ \tau$ can be unfolded by applying the substitution $\tau[\alpha := \mu\alpha.\ \tau]$ and this unfolding is potentially infinite. For instance, the recursive type $\mu\alpha.\ \beta \to \alpha$ can be unfolded as follows.

$$\mu\alpha.\ \beta \to \alpha \equiv \beta \to \mu\alpha.\ \beta \to \alpha \equiv \beta \to \beta \to \mu\alpha.\ \beta \to \alpha \equiv \ldots$$

A term described by this type is, for example, $\mathbf{Y}(\lambda f.\ \lambda x.\ f)$ which is a function that ignores its argument and returns itself. To avoid having to include combinators with every recursive function it is common to extend the term language with explicit syntax for recursive definitions, such as the following:

$$t\ ::=\ \ldots \mid \text{let rec } x = t$$

With this syntax, the previous example can be written as follows.

$$\text{let rec foo} = \lambda f.\ \lambda x.\ f$$

Equi-recursive Types There are two distinct treatments of recursive types. In the *equi-recursive* approach types are equal to their unfoldings, that is, $\mu\alpha.\tau \equiv \tau[\alpha := \mu\alpha.\tau]$, thus terms of either type can be used in contexts requiring the other type. Each equi-recursive type denotes an infinite unfolding.

Equi-recursive types are conceptually simple and easily integrated into type systems, the only change needed being to allow infinite types. But while the intuition for equi-recursive types is simple, their implementation is more complicated, especially in the presence of other advanced typing features.

Iso-recursive Types In the *iso-recursive* approach types are different from their unfoldings, but related by isomorphism, in the sense that one can be obtained from the other by either folding or unfolding. In these systems, each recursive type introduces two operators, fold and unfold, which must be placed accordingly, increasing the notational burden of the system.

While equi-recursive types result in simple systems, iso-recursive types complicate their presentation with the fold operations. The inconvenience of folding operations can be removed in practice though, by hiding them behind other language constructs. For example, in Haskell, iso-recursive types are given by the `data` constructor, and folds and unfolds are implicit through constructors and pattern matching. Iso-recursive types are however easier to implement since types remain finite.

Chapter 3

Subtyping

This chapter studies recent developments for type inference in the presence of *subtyping*, namely the MLsub type inference algorithm by Dolan [DM17] and Simple-sub by Parreaux [Par20]. Subtyping, also known as *inclusion polymorphism* [CW85], is a form of polymorphism that organizes types into a hierarchy of *subtypes* and *supertypes* under the assumption that subtypes are compatible with their supertypes, meaning that a value of a subtype can be safely used in contexts expecting values of its supertypes. In practice, subtyping allows a programmer to write programs that are polymorphic over a finite set of types defined by a *subtype relation*.

The remainder of this chapter is structured as follows. The first section introduces subtyping informally. The second section introduces the MLsub type system. The final section introduces the Simple-sub type inference algorithm. A re-implementation of Simple-sub is available at placeholder.

3.1 Subtyping

The rules for the simply typed λ-Calculus are very strict, leading to programs that are safe to run being rejected because the type system enforces equality constraints where more relaxed constraints are enough to guarantee safety. This makes essential extensions such as record data structures annoying to use in practice because equality constraints force functions to accept only one set of fields, so although the application in the following example is safe, it is rejected because the type of the record is not equal to the type expected by the function.

$$\text{read} : \{x : \alpha\} \to \alpha \qquad\qquad \text{read } \{x = 1,\ y = 2\}$$

*Subtyping*is a form of polymorphism that aims to solve such annoyances by replacing equality with a *subtype relation*, written as $\tau_1 \leq \tau_2$, which organizes types into hierarchies of *subtypes* and *supertypes*. How the subtype relation is defined varies between systems, but in general, it expresses a notion of *precision* in the sense that a subtype is more precise in describing values

and so describes only values that can also be described by its supertypes, and also a notion of *compatiblity*, which is to say that it is always safe to use values of a subtype in contexts expecting values of any of its supertypes. A subtype relation can be introduced to a type system with a rule for subtyping that usually takes a form such as the following.

$$\frac{t : \tau_1 \qquad \tau_1 \leq \tau_2}{t : \tau_2}$$

The rule for subtyping states that if a term is described by a type τ_1 then it can also be described by all types τ_2 which are supertypes of τ_1. This rule can be interpreted as *promoting* the type of a term to one of its supertypes, weakening its precision by *forgetting* the extra information the subtype carries. This rule allows terms of some type τ to be used in contexts that expect terms of a type less restrictive than τ. For instance, considering types int and real for integers and real numbers with the relation $\mathsf{int} \leq \mathsf{real}$, then any term of type int is also a term of type real and the type checker can allow integer values to be passed as arguments to functions over real numbers.

Subtyping is powerful and interacts with most other features of a type system, sometimes in non-trivial ways, so the subtype relation has to be carefully constructed with consideration for its interactions with the rest of the system. When subtyping is limited to base types it is usually called *atomic subtyping* [Mit91] and although limited it has practical uses in languages with implicit *coercion semantics* as studied by Reynolds [Rey80] and Mitchell [Mit84], using subtyping to introduce run-time conversions. The earlier example with $\mathsf{int} \leq \mathsf{real}$ can be problematic, since in practice integers and floating-points are represented differently so it is not sound to apply their respective operations interchangeably without first converting between representations.

Atomic subtyping can be extended to composite types, for instance, for pairs it can be defined that $\langle \tau_1,\ \tau_1 \rangle$ is a subtype of $\langle \tau_2,\ \tau_2 \rangle$ if $\tau_1 \leq \tau_2$ holds. This form of subtyping is usually called *structural subtyping* and is often studied in the context of type inference [HM95, FM88, Sim03]. In structural subtyping, if two types are related then they necessarily follow the same structure, that is, their syntax trees differ only at the leaves, so inference reduces to constraints between atomic types which are simple to solve. Subtyping, however, has gained its popularity because it can be applied to type systems for *object-oriented* programming languages such as Simula [DN66] or Smalltalk [GR83], where the subtype relation is not atomic.

In object-oriented languages, there are *objects* that encapsulate data or computations and can answer *messages* meant to access those. A common encoding for objects in type systems is to represent them using records where each field represents a message the object can answer. The classical example is that an object representing a point has type $\{x : \mathsf{int}\}$ and an object representing a colored point has type $\{x : \mathsf{int},\ c : \mathsf{color}\}$. The essence of object-oriented languages is that a context that expects objects to be points should treat simple points and colored points uniformly, which is similar to the motivating example for subtyping between records at the beginning of this section.

To use records representing points uniformly the subtype relation can define that a point is a subtype of a colored point with $\{x : \mathsf{int},\ c : \mathsf{color}\} \leq \{x : \mathsf{int}\}$, but this relation cannot be structural: the tree for point has a single leaf and the tree for the colored point is a branch with two leaves. To allow a relation between records with different numbers of fields, or define interesting types such as the supertype of all types, subtyping has necessarily to be *non-structural*, that is, it needs to be able to compare types built with different constructors and which have different structures.

An influential example of non-structural subtyping is Amadio and Cardelli's [AC93] subtype relation over the syntax:

$$\tau \ ::= \ \bot \mid \top \mid \alpha \mid \tau \to \tau \mid \mu\alpha.\ \tau$$

The properties of this subtype relation are shared by many other non-structural subtyping orders. It is a *partial order* $\leq$, so it is reflexive, transitive, and anti-symmetric. It is bounded by greatest ($\top$) and least ($\bot$) types as to form a *lattice* [DP02]. By including function types the relation has to deal with a type constructor that introduces a notion of *variance* [Cas95], with functions being *contra-variant* on their input covariant on outputs. Recursive types are *equi-recursive* [Pie02, Chapter 20], instead of the more common *iso-recursive*.

Non-structural subtype relations offer more expressiveness at the expense of introducing many complexities [Reh98] to the technical treatment of subtyping and so have received the most attention, in particular in combination with ML-style polymorphism. One problem in these systems is that usually the typability of a term is decided by collecting constraints between types and type variables and using unification to find a solution that satisfies this set of constraints, but subtyping constraints $\tau_1 \leq \tau_2$ cannot be eliminated with unification. Because of this type inference algorithms for subtyping, starting with Mitchell [Mit91] and Fuh and Mishra [FM88], have produced *constrained types*, which are types with a set of associated constraints attached. For example, in Fuh and Mishra's system [FM88], the function twice ($\lambda f.\ \lambda x.\ f\ (f\ x)$) can be assigned the following constrained type.

$$\{\alpha \leq \beta\} \vdash (\beta \to \alpha) \to \beta \to \alpha$$

This type means that the first argument of twice f is a function that receives a value of some type β and produces a value of a type α that contains more information than β. While this type is easy to read, the number of constraints generated by type inference grows with the size of the program [HM95] and constrained types quickly become unwieldy for programmers to reason about and for algorithms to deal with efficiently. The issues with constrained types have led to a long line of work on constraint *simplification* [FM89, EST95, AWP97, Pot98a, Pot01], culminating in Pottier's seminal PhD dissertation [Pot98b]. Simplification leads to one more problem: to check if a simplification step is correct one has to decide whether two constraint sets are equivalent, a problem known as *entailment*. The complexity of entailment and many algorithmic bounds are studied in depth in Rehof's PhD dissertation [Reh98].

Recently, Dolan and Mycroft [DM17] proposed that many of these problems derive from the focus of previous approaches on the *syntax* of types, leading to ill-behaved and unwieldy algebras and that these problems can be avoided by focusing instead on the *algebra* of types. Including their new approach, three main approaches to formalizing systems with subtyping can be distinguished: the *syntactic* approach, the *semantic* approach, and the recent *algebraic* approach.

The most commonly found is the *syntactic approach* in which the type system and subtype relation are defined in a formal system, usually a set of axioms and inference rules, that follows the syntax of types. The syntactic approach has a simple presentation and extracting a syntax-directed algorithm for subtyping from the rules is also simple, however, these rules have to be carefully designed so that they form a subtype relation with the desired properties. To subsequentially prove that the algebra formed by the subtyping rules is well-behaved can be a tedious process and prone to errors.

In the *semantic approach* [FCB02, FCB08] a type is denoted as the set of values it describes and the subtype relation is defined as set inclusion between denotations of types. Interpreting types as sets is intuitive, and many proofs come for free from set theory, but the technical details of this approach are difficult. For example, the subtype relation is defined in terms of sets of values, and because of the subtyping rule, these sets of values are defined in terms of the subtype relation, introducing a circularity. A good introduction to semantic subtyping is given by Castagna and Frisch [FC05] where they describe and solve this and other technical difficulties of semantic subtyping.

The most recent approach is the *algebraic approach* introduced by Dolan and Mycroft [DM17] with the MLsub type system, which combines parametric polymorphism with subtyping. In contrast to previous approaches that start with a syntax of types and build an algebra by proving facts about this syntax, the algebraic approach is to start by building an algebra with the desired properties and then extract a syntax for types from the equations of this algebra. By focusing on the algebraic properties of the system, they successfully develop a type inference algorithm for their system that is decidable, has principal types, and produces compact type schemes with no constraint sets attached to them.

3.2 Algebraic Subtyping

Algebraic subtyping is an approach to the definition of type systems with subtyping introduced by Dolan and Mycroft with a paper [DM17] defining the MLsub language and type system. MLsub is a minimal extension to the calculus of Damas and Milner [DM82] with booleans, records, and subtyping. The MLsub type system is distinguished from previous approaches to combine subtyping with ML-style parametric polymorphism because it has decidable subsumption and a type inference algorithm that infers principal, compact types without any constraint sets attached.

The novelty in MLsub is its algebra-first approach to start by finding the simplest algebra of types that satisfies all the desirable properties for subtyping, and from that extract some syntax to describe types. It is this focus on algebra that allows MLsub to find simpler solutions to the problematic details of previous approaches to subtyping such as constraint elimination, subsumption between type schemes, and simplification. However, finding the simplest algebra for MLsub, that is, a subtype relation that forms a *profinite distributive lattice*, requires a strong mathematical background in order theory and category theory and its presentation is difficult to follow for most.

These difficulties with algebraic subtyping and the more intricate details of type inference are noted in Parreaux's paper [Par20] which gives a syntactic presentation of the MLsub type system and proposes the much simpler type inference algorithm Simple-sub which is much closer to traditional Hindley-Milner type inference. The remainder of this section presents the MLsub type system syntactically, following Parreaux, and describes the main properties of the subtype relation. The field of algebraic subtyping is presented in depth in Dolan's PhD dissertation [Dol17] of the same name.

3.2.1 The MLsub Language

The MLsub language may be minimal, but its inclusion of functions, records, and parametric polymorphism is enough to make the subtype relation challenging to define. The term language follows, where the meta-variable t ranges over terms, the meta-variable x ranges over an infinite supply of variables, and the meta-variable l rangers over labels from some finite set of record labels $\mathcal{L}$.

$$t \ ::= \ x \mid \lambda x.\ t \mid t\ t \mid \text{let [rec] } x = t \text{ in } t \mid \{l_0 = t,\ \ldots,\ l_n = t\} \mid t.l \mid \texttt{true} \mid \texttt{false}$$

MLsub simply extends ML with records and boolean literals. Records are defined as a possibly empty sequence of fields, where a field $l = t$ associates a label l with term t. Two operations on records are included: record construction ($\{\ldots\}$), where all fields are assumed to contain distinct labels, and field selection ($t.l$) extracts the value t of a field with label l.

The language of types is similarly minimal. The set of simple types in MLsub is given by the following definition, where the meta-variable τ ranges over types, and the meta-variable l ranges over labels.

$$\tau \ ::= \ \alpha \mid \tau \to \tau \mid \{l_0 : \tau,\ \ldots,\ l_n : \tau\} \mid \mathsf{Bool} \mid \mathsf{True} \mid \mathsf{False}$$

The two additional base types True and False are not in MLsub but are introduced here to make the subtype relation slightly more interesting. While this definition of types is standard, it is important to note that MLsub diverges from previous approaches by including type variables in the definition of types, instead of viewing them as ranging over possible substitutions of ground

types. This change is important because it disallows case analyses over types when reasoning about the subtype relation, making the type system *extensible*, in the sense that adding new types to the language does not make previous terms untypable, and also because this changed allowed Dolan to give a complete algorithm to decide entailment.

The typing rules for this system are entirely standard and the following presentation should hold no surprises.

$$\textsc{T-Var}\ \frac{\Gamma(x) = \overline{\forall\alpha_i.}^{\,i}\,\tau}{\Gamma \vdash x : \overline{[\tau_i/\alpha_i]}^{\,i}\,\tau} \qquad \textsc{T-Abs}\ \frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda x.t : \tau_1 \to \tau_2} \qquad \textsc{T-App}\ \frac{\Gamma \vdash t_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash t_1\ t_2 : \tau_2}$$

$$\textsc{T-Rcd}\ \frac{\Gamma \vdash t_1 : \tau_1 \qquad \ldots \qquad \Gamma \vdash t_n : \tau_n}{\Gamma \vdash \{l_1 = t_1,\ \ldots,\ l_n = t_n\} : \{l_1 : \tau_1,\ \ldots,\ l_n : \tau_n\}} \qquad \textsc{T-Proj}\ \frac{\Gamma \vdash t : \{l : \tau\}}{\Gamma \vdash t.l : \tau}$$

$$\textsc{T-Let}\ \frac{\Gamma, x : \tau_1 \vdash t_1 : \tau_1 \qquad \Gamma, x : \overline{\forall\alpha_i.}^{\,i}\,\tau_1 \vdash t_2 : \tau_2}{\Gamma \vdash \text{let rec } x = t_1 \text{ in } t_2 : \tau_2}\ (\alpha_i \text{ not free in } \Gamma)$$

The essential change in MLsub is the addition of the subtyping rule, which is also standard.

$$\textsc{T-Sub}\ \frac{\Gamma \vdash t : \tau_1 \qquad \tau_1 \leq \tau_2}{\Gamma \vdash t : \tau_2}$$

This starting set of simple types drives the construction of the subtype relation:

- Function types give a constructor with two type parameters, forcing the subtype relation to consider the *variance* of each parameter.
- Record types introduce the need to compare constructors of different arities, forcing the relation to be *non-structural*.
- Base types give constructors incomparable with any others, except by direct specification in the rules of the type system, a practice known as *nominal* subtyping.

The remainder of this section describes the subtype relation ($\leq$) and increments the syntax of types as necessitated by the subtype relation.

3.2.2 Basic Properties

The subtype relation is a *preorder*, meaning that it is reflexive and transitive.

$$\text{S-Refl}\ \frac{}{\tau \leq \tau} \qquad \text{S-Trans}\ \frac{\tau_1 \leq \tau_2 \qquad \tau_2 \leq \tau_3}{\tau_1 \leq \tau_3}$$

3.2.3 Base Types

The subtype relation between base types is given nominally as follows.

$$\text{S-True}\ \frac{}{\mathsf{True} \leq \mathsf{Bool}} \qquad \text{S-False}\ \frac{}{\mathsf{False} \leq \mathsf{Bool}}$$

The types True and False are *incomparable*, but each one is a subtype of Bool.

3.2.4 Functions and Variance

A function type $\tau_1 \to \tau_2$ is a type constructor that receives two arguments τ_1 andτ_2, respectively the domain and range. Informally, a function g is said to be a subtype of a function f if g accepts as input at least the same values as f, and if g produces at most the same values as f. To express this the subtype relation for functions has to be *contravariant* when comparing domains and *covariant* when comparing ranges, meaning that the direction of the subtype relation is inversed when comparing domains and preserved when comparing ranges. Subtyping between functions is formalized as follows.

$$\text{S-Fun}\ \frac{\tau_3 \leq \tau_1 \qquad \tau_2 \leq \tau_4}{\tau_1 \to \tau_2 \leq \tau_3 \to \tau_4}$$

In practice, if a context requires functions of a type $\tau_3 \to \tau_4$, then it requires functions that expect values of type τ_3 and produce values of type τ_4. It is safe to pass instead functions of type $\tau_1 \to \tau_2$, because those accept all the expected values ($\tau_3 \leq \tau_1$), and do not produce any unexpected values ($\tau_2 \leq \tau_4$).

3.2.5 Records in Depth and Width

To compare record types the subtype relation has to compare type constructors that might have a different number of arguments, so it has to consider two dimensions: how the relation should compare record types with different lengths and how the relation should compare the types at each common label. The subtype relation between records in MLsub is covariant and describes that a record type r_1 is a subtype of a record type r_2, if for each field $(l : \tau_2)$ that is present in r_2, a field $(l : \tau_1)$ is present in r_1 and $\tau_1 \leq \tau_2$. This relation has the desirable property that a record with fewer fields is a subtype of a record with more, allowing, for instance, a record of

type {x : int, y : int} to be used as the argument for functions expecting records of type {x : int}.

The relation for records is formalized with two rules. The first rule, for *depth subtyping*, describes that the relation is covariant in common fields and is formalized as follows.

$$\text{S-RcdDepth}\quad \frac{\tau_1 \leq \tau_1' \quad \ldots \quad \tau_n \leq \tau_n'}{\{l_1 : \tau_1,\ \ldots,\ l_n : \tau_n\} \leq \{l_1 : \tau_1',\ \ldots,\ l_n : \tau_n'\}}$$

A second rule, for *width subtyping*, describes that smaller records are subtypes of bigger records as follows.

$$\text{S-RcdWidth}\quad \frac{}{\{l_1 : \tau_1,\ \ldots,\ l_n : \tau_n,\ \ldots,\ l_{n+m} : \tau_{n+m}\} \leq \{l_1 : \tau_1,\ \ldots,\ l_n : \tau_n\}}$$

3.2.6 Least and Greatest Types

The subtype relation in MLsub, as many others, forms a *lattice* [DP02]. A property of this lattice is that it is *bounded*, meaning it has the least and greatest elements, so new types have to be introduced to represent those elements. These types are, respectively, the type bottom ($\bot$), the subtype of all types; and the type top ($\top$), the supertype of all types. The subtyping rules for these types are as follows.

$$\text{S-Bot}\quad \frac{}{\bot \leq \tau} \qquad\qquad \text{S-Top}\quad \frac{}{\tau \leq \top}$$

3.2.7 Meets and Joins

Since the subtype relation forms a lattice, it is equipped with a *meet* operation ($\sqcap$) that represents the *greast lower bound* of two types, and a *join* operation ($\sqcup$) that represents the *least upper bound* of two types. Having the relation form a lattice is desirable because meets and joins are necessary to express principal types, for instance, the type of an expression that can be assigned either type τ_1 or τ_2 is the least upper bound of both types, represented by $\tau_1 \sqcup \tau_2$. The *intersection* $\tau_1 \sqcap \tau_2$ is the type that describes values that are described by *both* τ_1 and by τ_2, and the *union* $\tau_1 \sqcup \tau_2$ is the type that describes values that are described *either* by τ_1 or by τ_2. The subtype relation for these type constructors is given by the following rules.

$$\text{S-Or}\quad \frac{\forall i, \exists j, \Sigma \vdash \tau_i \leq \tau_j'}{\Sigma \vdash \bigsqcup_i \tau_i \leq \bigsqcup_j \tau_j'} \qquad\qquad \text{S-And}\quad \frac{\forall j, \exists i, \Sigma \vdash \tau_i \leq \tau_j'}{\Sigma \vdash \bigsqcap_i \tau_i \leq \bigsqcap_j \tau_j'}$$

Unions and intersections not only help make principal type inference possible but they can be used to eliminate constraints between type variables and types, which helps to simplify inferred constrained types. For example, if the algorithm generated a constrained type such as the following.

$$\{\alpha \leq \tau_1,\ \tau_2 \leq \beta\} \vdash \alpha \to \beta$$

MLsub represents the constraints directly in the type using union and intersection as follows.

$$\alpha \sqcap \tau_1 \to \beta \sqcup \tau_2$$

If α is a type variable that can only be instantiated by subtypes of τ_1, then it represents the types that describe *at most* as many values as τ_1 does, which are exactly the types described by the intersection $\alpha \sqcap \tau_1$.

3.2.8 Recursive Types

Subtyping allows MLsub to type more types than ML. One example given by Dolan [DM17] is the following term, where $\mathbf{Y}$ is the call-by-value Y-combinator:

$$\mathbf{Y}\ (\lambda f.\ \lambda x.\ f)$$

This term represents the function that ignores its parameters and returns itself. In MLsub, this term can be assigned infinitely many types, including the following:

$$\top \to \top$$
$$\top \to (\top \to \top)$$
$$\top \to (\top \to (\top \to \top))$$

Recursive types are needed to express the principal type for this function, which is the following:

$$\mu\alpha.\ \top \to \alpha$$

3.3 Type Inference

Type inference is significantly trickier when subtyping is involved, as the traditional Hindley-Milner approach of solving constraints with unification is not feasible. As a result, type inference algorithms for subtyping tend to produce constrained type schemes that have unwieldy constraint sets that render them hard for programmers to understand. Consequentially, considerable effort has been made to study techniques to simplify inferred type schemes and constraint sets. Pottier provides a comprehensive analysis of constrained types and their simplification in his seminal PhD dissertation [Pot98b].

A key insight in Pottier's dissertation is that a strict separation of inputs and outputs simplifies the analysis of constraint graphs. Building on this, Dolan and Mycroft [DM17] apply this same insight to avoid constraint graphs altogether and develop a type inference algorithm that enjoys the same properties as Hindley-Milner. The type inference algorithm for MLsub is the first to be decidable, infers principal types and the types it generates are *compact*, in the sense that no constraint sets are attached to them.

The presentation of MLsub's type inference algorithm, however, departs from the traditional Hindley-Milner style. The separation of inputs and outputs in the type system leads to a theory of *polar types* and the associated concepts of *biunification* and *bisubstitution*, the analogs of unification and substitution for polar types. While these concepts unlock type inference, they make the specification hard to follow for those without a strong background in abstract algebra.

These difficulties are noted by Parreaux [Par20], who proposes that the essence of algebraic subtyping does not depend on an extended language of types or the introduction of polar types and biunification, but instead depends on:

- a construction of types that allows inferred subtyping constraints to be reduced to constraints on type variables;
- the use of intersections and unions to indirectly constrain type variables and avoid separate constraint sets.

Parreaux concludes that the set-theoretic types of MLsub and the notion of polarity that divides the syntax of types into positive and negative types are not necessary to infer principal types, but are essential to give a *compact representation* to the inferred types. With a simplified syntax of types and without biunification, type inference in Simple-sub proceeds along the lines of Hindley-Milner but produces a constrained type [FM88] since it does not attempt to solve constraints. To present a *compact type* as its final output, the Simple-sub algorithm includes the application of several transformations and simplification steps to the produced type and constraint set. The remainder of this section presents the algorithms that constitute Simple-sub type inference.

3.3.1 Simple Type Inference

As pointed out at the beginning of this section, the types that arise from the algebraic construction of the subtype relation are not essential for the type inference algorithm. Their main purpose is to present compact representations of the inferred constrained types, and so the language of types of Simple-sub includes only the simple types of MLsub:

$$\tau \ ::= \ \alpha \mid \tau \to \tau \mid \{l_0 : \tau, \ \ldots, \ l_n : \tau\} \mid \mathsf{Bool} \mid \mathsf{True} \mid \mathsf{False}$$

A presentation of the type inference algorithm of Simple-sub (here called $\mathcal{S}$) follows. The algorithm receives as arguments an initial environment Γ and a term t and infers a constrained type $\langle \tau, \ C \rangle$ where τ is the type and C its associated set of constraints.

Definition: $\mathcal{S}(\Gamma, \ t) = \langle \tau, \ C \rangle$, where:

$$\begin{array}{rcll}
\mathcal{S}(\Gamma, \ \mathsf{true}) & = & \langle \mathsf{True}, \ \emptyset \rangle & \\
\mathcal{S}(\Gamma, \ \mathsf{false}) & = & \langle \mathsf{False}, \ \emptyset \rangle & \\
\mathcal{S}(\Gamma \cup \{x : \sigma\}, \ x) & = & \mathrm{inst}(\sigma) & \\
\mathcal{S}(\Gamma, \ \lambda x. \ t_1) & = & \text{Let } \langle \tau_1, \ C_1 \rangle = \mathcal{S}(\Gamma \cup \{x : \alpha\}, \ t_1) & \text{where } \alpha \text{ is new} \\
 & & \text{in } \langle \alpha \to \tau_1, \ C_1 \rangle & \\
\mathcal{S}(\Gamma, \ t_1 \ t_2) & = & \text{Let } \langle \tau_1, \ C_1 \rangle = \mathcal{S}(\Gamma, \ t_1) & \\
 & & \quad \langle \tau_2, \ C_2 \rangle = \mathcal{S}(\Gamma, \ t_2) & \\
 & & \text{in } \langle \alpha, \ C_1 \cup C_2 \cup \{\tau_1 \leq \tau_2 \to \alpha\} \rangle & \text{where } \alpha \text{ is new} \\
\mathcal{S}(\Gamma, \ \{\}) & = & \langle \{\}, \ \emptyset \rangle & \\
\mathcal{S}(\Gamma, \ \{l = t\} \cup r) & = & \text{Let } \langle \tau_1, \ C_1 \rangle = \mathcal{S}(\Gamma, \ t) & \\
 & & \quad \langle \tau_2, \ C_2 \rangle = \mathcal{S}(\Gamma, \ r) & \\
 & & \text{in } \langle \{l : \tau_1\} \cup \tau_2, \ C_1 \cup C_2 \rangle & \\
\mathcal{S}(\Gamma, \ t.l) & = & \text{Let } \langle \tau, \ C \rangle = \mathcal{S}(\Gamma, \ t) & \\
 & & \text{in } \langle \alpha, \ \tau \leq \{l : \alpha\} \cup C \rangle & \text{where } \alpha \text{ is new} \\
\mathcal{S}(\Gamma, \ \text{let } x = t_1 \text{ in } t_2) & = & \text{Let } \langle \tau_1, \ C_1 \rangle = \mathcal{S}(\Gamma, \ t_1) & \\
 & & \text{in } \mathcal{S}(\Gamma \cup \{x : \mathrm{gen}(\Gamma, \ \tau_1)\}, \ t_2) &
\end{array}$$

Where inst and gen are functions for instantiation and generalization defined as follows. Instantiation not only duplicates type variables but also their constraints.

$$\begin{array}{rcl}
\mathrm{inst}(\forall \alpha_1, \ \ldots, \ \alpha_n. \ \tau, \ C) & = & \langle \forall \beta_1, \ \ldots, \ \beta_n. \ \tau[\alpha_i := \beta_i], \\
 & & \quad \{ \ \beta_i \leq \mathrm{inst}(\tau_i, \ C) \mid \alpha_i \leq \tau_i \in C \ \} \\
 & & \cup \{ \ \mathrm{inst}(\tau_i, \ C) \leq \beta_i \mid \tau_i \leq \alpha_i \in C \ \} \\
 & & \cup \ C \rangle
\end{array}$$

$$\text{gen}(\Gamma,\ \tau) \quad = \quad \forall\alpha_1,\ \ldots,\ \alpha_n.\ \tau \qquad\qquad \text{where } \alpha_i = \text{ftv}(\tau) \setminus \text{ftv}(\Gamma)$$

The type inference algorithm diverges from algorithm W in the cases of application and selection in which subtyping constraints are generated. Since neither unification nor its analog of biunification can be used to find most general unifiers, the type inference algorithm produces a constrained type $\langle \tau,\ C \rangle$. The set of constraints C is composed of constraints over the type variables that were created during inference. These constraints define the *upper bounds*, of the form $\alpha \leq \tau$, and the *lower bounds*, of the form $\tau \leq \alpha$, of the type variables that occur in τ. For type inference to produce constraints of that form a rewriting step is performed where before unification would be called.

The constraint rewriting algorithm proceeds by decomposing constraints until they are either

(i) Constraints between primitive types
(ii) Constraints in which a type variable appears on either the left or right-hand side

In the first case, the constraint can be solved by checking whether it is consistent with the subtype relation. If it is, the constraint can be eliminated, otherwise, a type error is found and type inference fails. In the second case, if the constrain is of the form $\alpha \leq \tau$ then the constraint is propagated to the lower bounds of the type variable α, or to the upper bounds of the type variable if the constraint is of the form $\tau \leq \alpha$. A presentation of the rewriting rules follows.

Definition: While $(\tau_1 \leq \tau_2 \cup C)$ can be rewritten:

$$\begin{aligned}
\tau \leq \tau \cup C &\longrightarrow C \\
\mathsf{True} \leq \mathsf{Bool} \cup C &\longrightarrow C \\
\mathsf{False} \leq \mathsf{Bool} \cup C &\longrightarrow C \\
\tau_1 \to \tau_2 \leq \tau_3 \to \tau_4 \cup C &\longrightarrow \{\tau_3 \leq \tau_1\} \cup \{\tau_2 \leq \tau_4\} \cup C \\
\{r\} \leq \{s\} \cup C &\longrightarrow \{\ r(l) \leq s(l) \mid l \in \text{dom}(s)\ \} \cup C \\
\alpha \leq \tau \cup C &\longrightarrow \{\alpha \leq \tau\} \cup \{\ \tau_2 \leq \tau \mid \tau_2 \leq \alpha \in C\ \} \cup C \\
\tau \leq \alpha \cup C &\longrightarrow \{\tau \leq \alpha\} \cup \{\ \tau \leq \tau_2 \mid \alpha \leq \tau_2 \in C\ \} \cup C
\end{aligned}$$

Remark: Constrain resolution fails for any case not defined above.

3.3.2 Type Coalescence

The type inference algorithm generates a constrained type, comprised of type variables, booleans, record types or function types, and the accumulated set of constraints on type variables recorded on the global store. The remaining types of MLsub, those that arose during the algebraic construction of the subtype relation, the union types, intersection types, top and bottom types, and recursive types, can be used to express the accumulated constraints as part of the syntax of types.

This is done through the *type coalescence* [Par20] pass of Simple-sub, an algorithm that performs a *polarity* analysis of the type variables that occur in the inferred type. The polarity of a type variable represents whether it appears in an *input* position or an *output* position, respectively negative and positive positions. For example, in the following type, the type variables are annotated with their polarities, with $(-)$ representing inputs and $(+)$ representing outputs.

$$(\alpha^{+} \to \beta^{-}) \to (\alpha^{-} \to \beta^{+})$$

The coalescence algorithm is defined as a recursive descent through the syntactic structure of the type that results from type inference. The algorithm keeps track of the polarity of the position it is analyzing, which starts positive, and also because the algorithm is recursive, keeps track of the set of type variables that have been visited at each polarity.

The cases for booleans, functions and records are simple. Booleans are already as simple as possible. Coalescing a record just recurses into its fields. Coalescing a function flips the polarity when recursing on the left-hand side. The interesting cases are for type variables.

The coalescence algorithm replaces *positive occurrences* of a type variable with the *union of its lower bounds*, and the *negative occurences* of a type variable with the *intersection of its upper bounds*. In the case where these bounds are empty, the $\top$ type replaces unbounded negative type variables and the $\bot$ type replaces the unbounded positive type variables.

The coalescence algorithm is applied recursively to the bounds of the type variables it encounters. When a type variable that was already visited is found in its bounds, the algorithm introduces a recursive type.

Chapter 4

Extensible Records

Subtyping is an expressive mechanism of polymorphism and is especially useful for structured types such as records, but it is not without its drawbacks. When subtyping promotes a value of some type to one of its supertypes the extra information expressed by the original type is lost. In the case of record types, this means *forgetting* the presence of fields, which complicates the use of subtyping in systems for *extensibility* of records, that is, systems with operations that depend on and alter the structure of records, such as adding, removing or updating fields.

The study of systems with extensible records became a popular area of research not only because of their intrinsic interest but also because they lay the foundations for more complex systems for object-oriented programming and database languages. At the core of these systems is the idea of using type variables to represent the fields of a record that are not known or used, variables that can be instantiated or constrained to preserve those fields. When used this way these variables are usually referred to as *rows* or *row variables* and in some systems kept separate from traditional type variables. A row variable represents not necessarily a single type, like a type variable does, but possibly many as a sequence of fields. Rows can be used to design more expressive type systems for records, variants or even objects, and their operations.

The notions of extensibility and row are first introduced by Wand [Wan87] to design a type inference algorithm for an object calculus, and which became the starting point of a long line of research on *row-polymorphic* calculi. While the original type inference algorithm proved to be incomplete [Wan88], alternatives were found for restricted versions of the system by Jategaonkar [JM88], Stansifer [Sta88] and Ohori [OB88]. An important improvement to this approach is developed by Rémy [Rém89], who refines the notion of field, such that they can be used to refer not only to labels that are present but also those that must be absent. This line is further extended by Wand, Rémy and Pottier [Wan91, Rém93b, Rém93a, PR05].

Rows and presence tags have become the building blocks for many researchers developing systems for extensible records. Many alternative systems for polymorphic records deal with extensibility by integrating these notions. These include systems based on predicates and qualified types [HP91, GJ96, Gas98, MM19], kinds [Oho95, Sul97, AR21] and scoped rows [BDS95, Lei05].

Solutions involving subtyping have been studied by Cardelli [CM89, Car94] and also by Rémy [Rém95a, Rém98], but their approaches do not support type inference. Although alternatives based on row polymorphism have been more popular due to their practicality, subtyping remains the more general approach to polymorphism. Rémy notes that the presence of both features results in a strictly richer type system [Rém95a], with row polymorphism being used to preserve fields and subtyping for its expressiveness. Pottier follows this idea in his PhD [Pot98b, Sections 14.5-14.7] where he sketches an extension of his system with rows *à la Rémy*, combining subtyping and row polymorphism in a system with type inference. He further studies the interaction between rows and subtyping in [Pot00, Pot03].

This chapter shows that row polymorphism and algebraic subtyping also integrate effortlessly, by extending the system and type inference algorithm of Simple-sub with record extension and row variables, following Pottier's approach. The first section recalls some definitions of records, presents the concept of extensibility and motivates the need for an extension to Simple-sub. The second section gives a background for row systems and the associated concepts used in our extension. The third and fourth sections present the extended type system and type inference algorithm, respectively.

4.1 Extensible Records

Records are an essential data structure of any programming language for their ability to associate names with values in a structured way. Type systems for records are naturally interesting to study because of the practical use of records, but also because these systems lay the foundations for more complex type systems, for example, for object-oriented programming [Car84] or database programming [OB88]. The systems studied so far in this dissertation have used the following definitions for records:

$$\begin{array}{rl} \text{values} & \{l_0 = t_0,\ \ldots,\ l_n = t_n\} \\ \text{types} & \{l_0 : \tau_0,\ \ldots,\ l_n : \tau_n\} \\ \text{selection} & -.l \end{array}$$

This definition is useful to study polymorphism, but it is static: a record cannot change after being constructed, only be read from through selection. A more powerful definition would include operations to manipulate records dynamically so that fields could be inserted on a record that has already been constructed or their contents altered. This idea is formalized by Wand in a simple calculus [Wan87] where records can be constructed from the empty record using an extension operation. Wand's work introduces the notion of *extensible records*, the name for records that can be altered after construction, and has since inspired a variety of studies [Rém89, CM89, HP91, GJ96, Sul97, Lei05, AR21] of systems dealing with extensible records and operations on them.

The aim of this section is twofold: first, it is to present, informally and by example, a set of fundamental extensible operations to give the reader an intuition on how they can be used; second it is to show the trouble with finding useful types for these operations and motivate the extension to Simple-sub that follows.

4.1.1 Extensible Operations

Following Cardelli and Mitchell [CM89], these are a set of primitive extensible operations on records from which other more complex operations can be constructed.

Extension A record r is extended with a label l and value t using the following syntax:

$$r \mid \{l = t\} \qquad (strict)$$
$$r \text{ with } \{l = t\} \qquad (free)$$

The extension operation is said to be *strict* if it is defined to fail when the label l is already present in the record r, in which case the operator | is used. Otherwise, if the extension operation is defined such that when l is present then it overwrites the previously associated value with t then it is said to be *free* and the operator with is used, as it was used by Wand [Wan87]. To simplify the notation sequences of extensions will be abbreviated to a list of comma-separated fields as follows.

$$((r \text{ with } \{l_1 = t_1\}) \text{ with } \{l_2 = t_2\}) \ldots) \equiv r \text{ with } \{l_1 = t_2,\ l_2 = t_2\ , \ldots\}$$

As for examples, extension can be used directly to construct records from literals or to build on previously defined records in variables:

$$\mathsf{let}\ \text{point} = \{x = 1\} \text{ with } \{\text{y} = 2\}\ \mathsf{in}\ \text{point with } \{\text{color} = \text{blue}\}$$
$$\rightarrow \quad \{\text{x} = 1,\ \text{y} = 2,\ \text{color} = \text{blue}\}$$

But the most interesting use of extension is in the definition of generic functions to introduce new fields as in the following paint function:

$$\mathsf{let}\ \text{paint}\ r\ c = r \text{ with } \{\text{color} = c\}$$
$$\text{paint } \{x = 1,\ y = 2\} \text{ blue} \quad \rightarrow \quad \{\text{x} = 1,\ \text{y} = 2,\ \text{color} = \text{blue}\}$$

Restriction The counterpart to extension is *restriction*, which removes a field labeled l from a record r and is defined with the following syntax:

$$r \setminus l$$

Restriction can similarly be defined as strict and fail if the label is not present, or free and do nothing in that case. For a simple example of restriction, a field labeled y can be removed as follows.

$$\{x = 1,\ y = 2\} \setminus \mathrm{y} \quad \rightarrow \quad \{x = 1\}$$

Using extension and restriction as primitives, other common extensible operations can defined such as *update* and *rename*.

$$\begin{array}{lcll} r \text{ where } \{l := t\} & \equiv & (r \setminus l) \mid \{l = t\} & \text{(update)} \\ r \text{ where } \{l \leftarrow m\} & \equiv & (r \setminus m) \mid \{l = r.m\} & \text{(rename)} \end{array}$$

The behavior of the above operations naturally depends on whether the primitives are defined to be strict or free. While for the above examples, it seems to make more sense to consider both primitives to be strict, other combinations can be interesting. For example, in Cardelli and Mitchell's system [CM89] restriction is free while extension is strict. With that combination they define an *override* operation, identical in definition to the above update example, but which instead behaves as the free extension of Wand [Wan87].

Concatenation Another extensible operation that in some systems [HP91] is used as a primitive is *concatenation* but it is usually defined in terms of extension because an encoding was developed by Rémy [Rém93b]. The concatenation operator takes two records and returns a new record containing all the fields from both of its arguments and is written as follows.

$$r_1 \parallel r_2$$

Different semantics can be given to concatenation in the case the records contain fields with the same label: *symmetric* concatenation [HP91] requires its arguments to have disjoint sets of fields, failing otherwise; *asymmetric* concatenation [Wan91] chooses one field in case of conflict, usually the right-hand record; *recursive* [OB88] concatenation attempts to concatenate the values of common fields recursively. Thus, depending on the selected behavior, the concatenation

$$\{x = 1,\ y = \{a = 1,\ b = 2\}\} \parallel \{z = 2,\ y = \{c = 3,\ d = 4\}\}$$

can result in either:

$$\begin{array}{rr} \text{fail} & \text{(symmetric)} \\ \{x = 1,\ y = \{c = 3,\ d = 4\},\ z = 2\} & \text{(asymmetric)} \\ \{x = 1,\ y = \{a = 1,\ b = 2,\ c = 3,\ d = 4\},\ z = 2\} & \text{(recursive)} \end{array}$$

4.1.2 Typing Record Operations

Extending the polymorphic records of MLsub, described in chapter 3 to support extensibility is not straightforward. Because subtyping is used for record polymorphism, function types in MLsub lose information about fields that are not specified in their signature. This is enough for selection but to assign meaningful types to the extensible operations it is necessary to preserve the types of unspecified fields also. It is easy to see that subtyping fails to preserve the types of fields by looking at a type such as:

$$\{l : \beta\} \to \beta$$

Information about the presence of fields other than l in a record passed to a function of this type is lost as a consequence of going up in the type hierarchy and so these fields cannot be referred to. For example, the following type is sound for free extension, but any fields other than l become inaccessible.

$$- \text{ with } \{l = -\} \quad : \quad \{\} \to \beta \to \{l : \beta\}$$

Fields can be preserved using parametric polymorphism instead, but the interaction between subtyping and extensibility is more subtle than losing fields. Consider the strict update operation, which updates the value of an existing field while preserving its type. A possible simplified type scheme in MLsub for this operation could be:

$$- \text{ where } \{l := -\} \quad : \quad \alpha \sqcap \{l : \beta\} \to \beta \to \alpha$$

Even though here fields are preserved through α, the use of β to constrain two values, in the first and second arguments, only means that β represents their greatest common upper bound. As there are no constraints on β that type is $\top$ and the scheme can be simplified further:

$$\alpha \sqcap \{l : \top\} \to \top \to \alpha$$

Requiring the two values to have the exact same type has no effect here because values of different and incompatible subtypes of that type can be used and thus it is incorrect to return α. To force the requirement to hold it is enough to make the subtype relation over record fields *invariant*, but in doing so depth subtyping is lost. It is better to define variance field by field by introducing type constructors for fields and defining the subtype relation between these constructors in a way that achieves the desirable effects. The strict update operation can then be assigned a sound type using an invariant constructor on its signature.

Field types also deal with another limitation of subtyping. Without polymorphism, a record type that omits some field represents records without that field, whereas with subtyping it

represents records that might have that field. Some operations need the certainty of the former, so to get it back a field type can be introduced to explicitly express the absence of a field.

In conclusion, to give the extensible operations sound and useful types the structure of record types needs to be enriched in two ways:

1. To allow for quantification over fields to be able to refer to the unknown fields as a whole, so that they are not lost through subtyping.
2. With more expressive types for fields that allow for a more refined subtype relation to express stronger constraints in function signatures, such as field absence or invariance;

The approach taken here to integrate records, polymorphism, subtyping and extensibility will follow Pottier [Pot98b, Sections 14.5-14.7], who in turn followed Rémy [Rém93a].

4.2 Row Polymorphism

Extensible records were introduced by Wand while studying type inference for a polymorphic record calculus [Wan87] that included free extension. In this calculus Wand applies ML-style parametric polymorphism to records, generalizing over *rows* (i.e. sequences) of fields, as opposed to using subtyping to ignore fields. This approach was motivated by the undecidability of type inference for the subtyping [Car84], but the given algorithm proved to be incomplete [Wan88]. Nevertheless, this system remains an important reference as the first proposal dealing with type inference and polymorphism for extensible records and the first of many calculi for *row polymorphism*. An important refinement of the theory of rows is due to Rémy [Rém89, Rém93a], in which rows associate labels with *field types* instead of types, increasing the expressiveness of record types and allowing him to present a type inference algorithm which is complete and produces principal types.

This section introduces Rémy's variant of row polymorphism, which includes row variables and field types, and presents some examples of the types that can be assigned to the polymorphic record operations using this discipline, absent subtyping. A complete introduction to the theory of rows is presented in [PR05].

Rows To assign meaningful types to polymorphic record operations, Wand and Rémy [Wan87, Rém93a] introduce a notation to express enumerable collections of types indexed by labels, called *rows*. A row denotes a function from labels to types and is said to be *complete* if it associates a type with every label of its domain (i.e. it denotes a *total function*), otherwise, it is said to be *incomplete*. Because the set of labels can be infinite a row might represent an infinite type, to give rows a finite representation they are built from two constructs: a constant that associates with every label the same type; and the strict extension of an incomplete row with a

new association. The definition of rows follows, where ρ ranges over rows, and τ and α range respectively over types and variables as before.

$$\rho \ ::= \ \alpha \mid l : \tau,\ \rho \mid \partial\tau$$

A *row variable* α can be used to represent an unknown row, just as a type variable can be used to represent an unknown type. The term $l : \tau,\ \rho$ represents a row that associates the label l with type τ, and whose other fields are given by ρ. A row of this form is only defined if l does not appear in ρ. Finally, the term $\partial\tau$ represents a constant row that associates every label with the type τ.

Two rows can be syntactically different but denote the same type if they associate the same labels with the same types but in different orders. This is represented by the following equivalences between rows:

$$\begin{aligned} l_1 : \tau_1,\ l_2 : \tau_2,\ \rho \ &= \ l_2 : \tau_2,\ l_1 : \tau_1,\ \rho \\ \partial\tau \ &= \ l : \tau,\ \partial\tau \end{aligned}$$

Having defined rows they can now be used to build record types. The syntax of types is changed so that the record type constructor $\{\}$ now takes a row as its parameter.

$$\tau \ ::= \ \ldots \mid \{\rho\}$$

Some examples of record types constructed using rows follow.

$\{x : \mathsf{int},\ y : \mathsf{int},\ \alpha\}$	denotes records containing at least the labels x and y with type int
$\{\partial\mathsf{int}\}$	denotes records where all fields are of type int
$\{a : \tau_1,\ a : \tau_2,\ \alpha\}$	is invalid because a label may not appear twice in a row

The ability to quantify via row variables adds a great deal of expressiveness to the type system, allowing it to express operations that treat all fields uniformly, except those that appear explicitly in the row. For example, the following type denotes the functions that first take a row where the label l is present and change the value associated with it to some value of type β. The row variable φ is used to express that only the label l is altered and the remaining fields, if any, are left unchanged.

$$\{l : \alpha,\ \varphi\} \to \beta \to \{l : \beta,\ \varphi\}$$

The use of rows here gives a finite representation to the infinite type which expresses all possible extensions of $l : \alpha$, as a rule for width subtyping would, but while using subtyping here would discard the fields other than l, rows make use of parametric polymorphism to instantiate the row variable φ as needed to preserve those fields which would be lost.

Fields Types The types for records so far can only express *positive* information, that is, these types can only express that some labels are known to be present in the values they represent. To further express *negative* information, that is, which labels cannot be present, Rémy [Rém93a] introduces two type constructors Abs and Pre τ to be used in fields and expressing, respectively, that a label is *absent* or *present* with some type. This separation in the types of fields grants more expressiveness to record types: being able to express that a label cannot be present is necessary to assign sound types to operations such as *strict* extension.

Field types, ranged over by θ, are introduced as follows, creating a distinction between "normal types" τ and the types built with the new type constructors to be used in fields.

$$\begin{array}{lll} \tau & ::= & \ldots \mid \{\rho\} & \text{types} \\ \rho & ::= & \alpha \mid l : \theta,\ \rho \mid \partial\theta & \text{rows} \\ \theta & ::= & \alpha \mid \mathsf{Pre}\ \tau \mid \mathsf{Abs} & \text{fields} \end{array}$$

Rows are changed to associate labels with field types, which are built with either the unary type constructor Pre to indicate a label is present with some type τ or with constant type constructor Abs to indicate a label is absent or be a type variable in the case the presence is unknown. For example, the following record type represents records where the label a is present and the label b is absent:

$$\{a : \mathsf{Pre}\ \alpha,\ b : \mathsf{Abs},\ \beta\}$$

An important use case for Abs is that in combination with the constant row, it confers to the type system the ability to express monomorphic record types again. A sequence of rows ending in ∂Abs marks all labels as absent except the ones that appear in the sequence. For example, the following type expresses records with only the two fields labeled x and y.

$$\{x : \mathsf{Pre}\ \mathsf{int},\ y : \mathsf{Pre}\ \mathsf{int},\ \partial\mathsf{Abs}\}$$

The notion of field type is more general than the two constructors focused on here, but while more constructors could be introduced now, Abs and row polymorphism confer enough expressiveness to assign meaningful types to the record operations considered so far. This changes in the presence of subtyping, however, other constructors become more useful as a means to refine the subtype relation. Cardelli, for example, introduces an invariant constructor Var τ, to mark fields as mutable, solving the problem arising from variance and mutability [CM89]. Rémy takes the notion of field types much further in the context of object subtyping [Rém98], using a wide array of constructors to build an intricate subtype relation between objects to model class inheritance.

Typing Record Operations The remainder of this section illustrates the use of row polymorphism and field types in the type checking of records and the primitive record operations.

The construction of record terms follows from keeping presence information explicit. The types of the present labels are tagged with Pre while all other labels are marked as absent through ∂Abs.

$$\begin{array}{rcl} \{\} & : & \{\partial\mathsf{Abs}\} \\ \{l_1 = t_1,\ \ldots,\ l_n = t_n\} & : & \{l_1 : \mathsf{Pre}\ \tau_1,\ \ldots,\ l_n : \mathsf{Pre}\ \tau_n,\ \partial\mathsf{Abs}\} \end{array}$$

Assigning polymorphic type schemes to the primitive record operations is straightforward using rows:

$$\begin{array}{rcll} -.l & : & \{l : \mathsf{Pre}\ \alpha,\ \varphi\} \to \alpha & \text{selection} \\ r \text{ with } \{l = t\} & : & \{l : \alpha,\ \varphi\} \to \beta \to \{l : \mathsf{Pre}\ \beta,\ \varphi\} & \text{extension} \\ r \setminus l & : & \{l : \mathsf{Pre}\ \alpha,\ \varphi\} \to \{l : \mathsf{Abs},\ \varphi\} & \text{restriction} \end{array}$$

Selection requires the label to be present as expected and is polymorphic on the presence of any other labels. The free extension operation is polymorphic on the presence of the label to be added, and any other remaining fields are preserved in the result by the row variable φ. Strict restriction requires a label to be present in its output and marks it as absent in its output. While row variables make it easy to preserve fields from input to output, field types make it easy to deal with strictness. Record extension can be made strict by using Abs instead of a type variable, and conversely, restriction can be made free by using a type variable.

Assigning a type to record concatenation, however, is much more challenging than the other operations because its semantics depend on the presence or absence of each label in either argument. For example, in the case of symmetric concatenation, the type assigned to the operation has to ensure that a label present in one argument is absent in the other, and in the case of asymmetric concatenation, the type of the operation has to express a choice between types when a label is present in both arguments. Because there is no simple way to formulate the necessary constraints to express meaningful types for concatenation, many authors have developed different approaches to this problem.

Wand introduces disjunctions of constraints to solve type inference in [Wan91], however, this gives the system exponential complexity. Harper and Pierce [HP91] study concatenation in a higher order explicitly typed system using constraints of the form $\forall\alpha\#r$, restricting instantiations of α to records which do not contain the fields in r. Rémy [Rém93b] presents a translation from concatenation to extension, where records are encoded as functions from rows to rows and concatenation is given by composition. Cardelli adapts this approach in [Car94]. Sulzmann [Sul97] gives an instance of HM(X) for concatenation. Rémy presents a more direct approach in [Rém95b], which is later built on by Pottier[Pot00, Pot03].

Concatenation will not be considered in the following sections to not introduce more complexity to the system, but it should be straightforward to adopt Pottier's approach [Pot03].

4.3 Extending MLsub

This section aims to show that subtyping and row polymorphism integrate easily and complement each other well, as claimed by Rémy [Rém95a] and Pottier [Pot98b, Chapter 14.7], through the formal presentation of an extension of Simple-sub with extensible record operations, row polymorphism, and field types.

Terms The syntax of terms is extended with operations for *free* extension and *strict* restriction as follows.

$$
\begin{array}{rcll}
t & ::= & \ldots \mid t \text{ with } \{l = t\} & \text{extension} \\
& & \mid t \setminus l & \text{restriction}
\end{array}
$$

Since record extension is free it can double as field update when the label is already present. On the other hand, the restriction operation is strict and fails if the label is absent.

Types The language of types is extended, almost exactly as in section 4.2, with rows and field types, to confer to the type system the ability to express polymorphic type schemes for extensible record operations. For now assume that if a row is of the form $l : \tau,\ \rho$ then the label l does not appear in ρ.

$$
\begin{array}{rcll}
\tau & ::= & \ldots \mid \{\rho\} & \text{types} \\
\rho & ::= & \alpha \mid l : \theta,\ \rho \mid \partial\theta & \text{rows} \\
\theta & ::= & \alpha \mid \mathsf{Pre}\ \tau \mid \mathsf{Abs} \mid \mathsf{Bot} \mid \mathsf{Any} & \text{fields}
\end{array}
$$

The difference is the introduction of the Bot and Any constructors to represent, respectively, the least and greatest elements of field types. Note that Dolan already introduces a top element for fields in MLsub [Dol17, 3.1.2], which is implicitly associated with absent labels and that is why the system can only ignore labels and not express their absence. The addition of rows and fields offers a more expressive type system with the ability to express strictly positive (Pre) or negative (Abs) information, and the ability to choose when to ignore (Any) labels. Some examples of the record types that can be expressed follow.

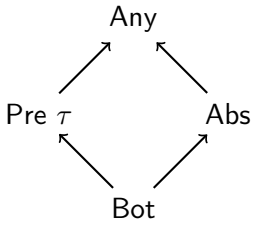
Records with the label x	$\{x : \mathsf{Pre}\ \alpha,\ \partial\mathsf{Any}\}$
Records with the label x, other labels quantified	$\{x : \mathsf{Pre}\ \alpha,\ \varphi\}$
Records with only the label x	$\{x : \mathsf{Pre}\ \alpha,\ \partial\mathsf{Abs}\}$
Records without the label x	$\{x : \mathsf{Abs},\ \partial\mathsf{Any}\}$

Record types are equivalent if the rows they are built with are equivalent. The row equivalence rules from section 4.2 for field commutativity and constant expansion are adapted to the presence of fields as follows, the only difference being the use of fields instead of types.

$$l_1 : \theta_1,\ l_2 : \theta_2,\ \rho \ \equiv\ l_2 : \theta_2,\ l_1 : \theta_1,\ \rho \qquad \text{(commutativity)}$$

$$\partial\theta \ \equiv\ l : \theta,\ \partial\theta \qquad \text{(expansion)}$$

Subtype Relation The ordering of field types chosen here follows the one by Pottier in [Pot00], which in turn is based on Rémy's in [Rém98]. The constructors Abs and Pre are kept incomparable so that they represent, respectively, strictly positive and strictly negative information about the presence of labels. Using Bot and Any for the bottom and top element of fields, the subtype relation between these constructors forms the following lattice.



Some systems, for example, [Pot98b, 14.5], define Abs as greater than Pre and this defines the usual width subtyping relation which allows fields to be forgotten because the type associated with a label can always be promoted to Abs. This promotion allows records to pretend that labels are absent, but this complicates typing for record operations that rely on precise information about the absence of labels, such as strict extension or concatenation. This problem is described but unsolved by Cardelli and Mitchell in [CM89, 4.4]. Pottier's solution in [Pot00] is to place Pre and Abs at the same level of the type hierarchy and introduce the upper bound Any for width subtyping. His approach is based on ideas described by Rémy in the case of objects in [Rém98] which includes a much more intricate subtype relation.

Following these ideas, the subtype relation of Simple-sub defined in section 3.1 is extended with inference rules for records, rows and fields. The extension is straightforward, starting with the following axioms for the lattice of field constructors and a rule for the co-variance of the Pre constructor on its argument.

$$\text{S-Any}\ \frac{}{\theta \le \mathsf{Any}} \qquad \text{S-Bot}\ \frac{}{\mathsf{Bot} \le \theta} \qquad \text{S-Pre}\ \frac{\Sigma \vdash \tau_1 \le \tau_2}{\Sigma \vdash \mathsf{Pre}\ \tau_1 \le \mathsf{Pre}\ \tau_2}$$

The extension to records is less direct because absent labels are not ignored implicitly anymore, but is still simple. In Simple-sub the width and depth subtype relation on records is written in terms of single-field records using the following equivalence.

$$\{\overline{l_i : \tau_i}^{\,i}\} \equiv \sqcap_i \{l_i : \tau_i\}$$

However, this equivalence is only true when viewing absent labels as associated with a $\top$ type, which in this extension is the case for records built with $\partial\mathsf{Any}$, but not for those built with any other field constructor. Using instead the equivalences for row commutativity and expansion, the following inference rules are proposed for record subtyping written in terms of rows:

$$\textsc{S-Rcd}\ \frac{\Sigma \vdash \rho_1 \leq \rho_2}{\Sigma \vdash \{\rho_1\} \leq \{\rho_2\}} \qquad \textsc{S-Row}\ \frac{\Sigma \vdash \theta_1 \leq \theta_2 \qquad \Sigma \vdash \rho_1 \leq \rho_2}{\Sigma \vdash l : \theta_1,\ \rho_1 \leq l : \theta_2,\ \rho_2} \qquad \textsc{S-Const}\ \frac{\Sigma \vdash \theta_1 \leq \theta_2}{\Sigma \vdash \partial\theta_1 \leq \partial\theta_2}$$

$$\textsc{S-Expand}\ \frac{}{\Sigma \vdash l : \theta,\ \partial\theta \equiv \partial\theta} \qquad \textsc{S-Comm}\ \frac{}{l_1 : \theta_1,\ l_2 : \theta_2,\ \rho\ \equiv\ l_2 : \theta_2,\ l_1 : \theta_1,\ \rho}$$

The rules for records (S-Rcd) and constant rows (S-Const) follow directly from their structure. The rule S-Row is essentially the rule for depth subtyping written recursively in terms of rows, but width subtyping can still be achieved in the case the right-hand side row is built with $\partial\mathsf{Any}$.

Typing Rules It is straightforward to assign types to the primitive record operations on the previous definitions.

$$\textsc{Cons}\ \frac{\Gamma \vdash t_1 : \tau_1 \qquad \ldots \qquad \Gamma \vdash t_n : \tau_n}{\Gamma \vdash \{l_1 = t_1,\ \ldots,\ l_n = t_n\} : \{l_1 : \mathsf{Pre}\ \tau_1,\ \ldots,\ l_n : \mathsf{Pre}\ \tau_n,\ \partial\mathsf{Abs}\}}$$

$$\textsc{Select}\ \frac{\Gamma \vdash t : \{l : \mathsf{Pre}\ \tau,\ \partial\mathsf{Any}\}}{\Gamma \vdash t.l : \tau}$$

$$\textsc{Restrict}\ \frac{\Gamma \vdash t : \{l : \mathsf{Pre}\ \top,\ \varphi\}}{\Gamma \vdash t \setminus l : \{l : \mathsf{Abs},\ \varphi\}}$$

$$\textsc{Extend}\ \frac{\Gamma \vdash t_1 : \{l : \mathsf{Any},\ \varphi\} \qquad \Gamma \vdash t_2 : \tau}{\Gamma \vdash t_1\ \text{with}\ \{l = t_2\} : \{l : \mathsf{Pre}\ \tau,\ \varphi\}}$$

The types assigned to record literals here are more accurate since they denote exactly the presence of only the fields that appear in the type. Selection is assigned a type scheme that is explicitly polymorphic on record width. Restriction requires the label to be present in the input and marks it absent in the output, while extension has no requirements on the presence and marks the label present. The opposite semantics for either operation can easily be obtained by replacing the field constructors in their requirements. Both extensible operations are assigned type schemes that quantify over the remaining fields with a row variable so that they can be referred to in the type of the result. The type for extension explicitly ignores the presence of the label so that the row variable φ cannot be instantiated with a row that contains the label l.

Examples The remainder of this section consists of example derivations using the typing and subtyping rules. The first example is presented in figure 4.1 and is a derivation for the type of a selection from a record with two fields. This example is meant to show how row expansion combined with the subtype relation on fields permits width polymorphism.

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash 1 : \mathsf{int} \qquad \Gamma \vdash 2 : \mathsf{int}}{\Gamma \vdash \{x = 1,\ y = 2\} : \{x : \mathsf{int},\ y : \mathsf{int},\ \partial\mathsf{Abs}\}}\ \textsc{Cons} \qquad \star}{\Gamma \vdash \{x = 1,\ y = 2\} : \{x : \mathsf{int},\ \partial\mathsf{Any}\}}\ \textsc{Sub}}{\Gamma \vdash \{x = 1,\ y = 2\}.x : \mathsf{int}}\ \textsc{Sel}$$

$$\star \dfrac{\textsc{Refl}\ \dfrac{}{\mathsf{int} \le \mathsf{int}} \qquad \dfrac{\textsc{Any}\ \dfrac{}{\mathsf{int} \le \mathsf{Any}} \qquad \textsc{refl}\ \dfrac{}{\partial\mathsf{Abs} \le \partial\mathsf{Any}}}{y : \mathsf{int},\ \partial\mathsf{Abs} \le y : \mathsf{Any},\ \partial\mathsf{Any}}\ \textsc{Row}}{\{x : \mathsf{int},\ y : \mathsf{int},\ \partial\mathsf{Abs}\} \le \{x : \mathsf{int},\ \partial\mathsf{Any}\}}\ \textsc{S-Row}$$

Figure 4.1: Selection from a record with multiple fields.

The second example (figure 4.2) shows how fields are preserved through a generic operation that extends a record with a field y.

$$\Gamma = \{r : \{x : \mathsf{int},\ \partial\mathsf{Abs}\}\}$$

$$\textsc{App}\ \dfrac{\textsc{Fun}\ \dfrac{\textsc{Ext}\ \dfrac{\textsc{Sub}\ \dfrac{\textsc{Var}\ \dfrac{}{\Gamma \vdash r : \{x : \mathsf{int},\ \partial\mathsf{Abs}\}} \qquad \star}{\Gamma \vdash r : \{y : \mathsf{Any},\ x : \mathsf{int},\ \partial\mathsf{Abs}\}} \qquad \vdots}{r : \{x : \mathsf{int},\ \partial\mathsf{Abs}\} \vdash r \text{ with } y = 2 : \{x : \mathsf{int},\ y : \mathsf{int},\ \partial\mathsf{Abs}\}}}{\vdash \lambda r.\ r \text{ with } y = 2 : \{x : \mathsf{int},\ \partial\mathsf{Abs}\} \to \{x : \mathsf{int},\ y : \mathsf{int},\ \partial\mathsf{Abs}\}} \qquad \vdots}{\vdash (\lambda r.\ r \text{ with } y = 2)\ \{x = 1\} : \{x : \mathsf{int},\ y : \mathsf{int},\ \partial\mathsf{Abs}\}}$$

$$\star \dfrac{\textsc{Refl}\ \dfrac{}{\mathsf{int} \le \mathsf{int}} \qquad \dfrac{\textsc{Any}\ \dfrac{}{\mathsf{Abs} \le \mathsf{Any}} \qquad \textsc{Refl}\ \dfrac{}{\partial\mathsf{Abs} \le \partial\mathsf{Abs}}}{\partial\mathsf{Abs} \le \{y : \mathsf{Any},\ \partial\mathsf{Abs}\}}\ \textsc{row}}{\{x : \mathsf{int},\ \partial\mathsf{Abs}\} \le \{y : \mathsf{Any},\ x : \mathsf{int},\ \partial\mathsf{Abs}\}}\ \textsc{Row}$$

Figure 4.2: Example 2

The third example (figure 4.3) shows how the system avoids the problem with record updates given by Cardelli in [CM89].

$$\Gamma = \{\text{not} : \mathsf{Bool} \to \mathsf{Bool},\ \text{update} : \alpha \sqcap \{b : \mathsf{Bool}\} \to \alpha\} \qquad \Sigma = \{\mathsf{True} \leq \mathsf{Bool}\}$$

$$\frac{r : \alpha \sqcap \{b : \mathsf{Bool}\} \vdash r \text{ with } b = \text{not } r.b : \alpha}{\vdash \lambda r.\ r \text{ with } b = \text{not } r.b : \forall \alpha.\ \alpha \sqcap \{b : \mathsf{Bool}\} \to \alpha}$$

$$\frac{\begin{array}{c}\Gamma \vdash \text{update} : \{b : \mathsf{Pre}\ \mathsf{True},\ \partial\mathsf{Abs}\} \to \{b : \mathsf{Pre}\ \mathsf{False},\ \partial\mathsf{Abs}\} \\ \Gamma \vdash \{b = \text{true}\} : \{b : \mathsf{Pre}\ \mathsf{True},\ \partial\mathsf{Abs}\}\end{array}}{\Gamma \vdash \text{update}\ \{b = \text{true}\} : \{b : \mathsf{Pre}\ \mathsf{False},\ \partial\mathsf{Abs}\}}$$

Figure 4.3: Example 3

4.4 Type Inference

This section presents the extensions to the Simple-sub type inference algorithm to accommodate extensible record operations. The following presentation builds on the definition given in chapter 3.

Definition: $\mathcal{P}(\Gamma,\ \Theta,\ t) = \tau$, where:

$$\begin{array}{rcll}
\mathcal{S}(\Gamma,\ \Theta,\ \{\}) & = & \langle \{\partial\mathsf{Abs}\},\ \emptyset \rangle & \\
\mathcal{S}(\Gamma,\ \Theta,\ \{l = t\} \cup r) & = & \text{Let } \langle \tau_1,\ C_1 \rangle = \mathcal{S}(\Gamma,\ \Theta,\ t) & \\
 & & \quad \langle \tau_2,\ C_2 \rangle = \mathcal{S}(\Gamma,\ \Theta,\ r) & \\
 & & \text{in } \langle \{l : \mathsf{Pre}\ \tau_1\} \cup \tau_2,\ C_1 \cup C_2 \rangle & \\
 & & & \\
\mathcal{S}(\Gamma,\ \Theta,\ t.l) & = & \text{Let } \langle \tau,\ C \rangle = \mathcal{S}(\Gamma,\ \Theta,\ t) & \\
 & & \text{in } \langle \beta,\ \tau \leq \{l : \mathsf{Pre}\ \beta,\ \partial\mathsf{Any}\} \cup C \rangle & \text{where } \beta \text{ is new} \\
 & & & \\
\mathcal{S}(\Gamma,\ \Theta,\ t \setminus l) & = & \text{Let } \langle \tau,\ C \rangle = \mathcal{S}(\Gamma,\ \Theta,\ t) & \\
 & & \text{in } \langle \{l : \mathsf{Abs},\ \varphi\}, & \text{where } \varphi \text{ is new} \\
 & & \quad \tau \leq \{l : \mathsf{Pre}\ \top,\ \varphi\} \cup C \rangle & \\
 & & & \\
\mathcal{S}(\Gamma,\ \Theta,\ t_1 \text{ with } l = t_2) & = & \text{Let } \langle \tau_1,\ C_1 \rangle = \mathcal{S}(\Gamma,\ \Theta,\ t_1) & \\
 & & \quad \langle \tau_2,\ C_2 \rangle = \mathcal{S}(\Gamma,\ \Theta,\ t_2) & \\
 & & \text{in } \langle \{l : \mathsf{Pre}\ \tau_2,\ \varphi\}, & \text{where } \varphi \text{ is new} \\
 & & \quad \tau_1 \leq \{l : \mathsf{Any},\ \varphi\} \cup C_1 \cup C_2 \rangle &
\end{array}$$

For record literals inference proceeds as before but the record type is closed with $\partial\mathsf{Abs}$. The only change for selection is similar, with $\partial\mathsf{Any}$ used to make width subtyping explicit. In the case of extension, the constraint $\tau_1 \leq \{l : \mathsf{Any},\ \varphi\}$ is important because even though the

presence of the label is explicitly ignored with Any, it being present in the constraint prevents it from occurring in the row represented by φ. The inferred type for extension represents the new label being present with the new type and that the remaining fields are preserved. In the case of restriction, the constraint makes sure the label l is present in the input record, and the inferred type for the result sets it to absent while preserving the remaining fields. The following presentation of the constraint rewriting algorithm builds on the definition in chapter 3.

Definition: While $(\tau_1 \leq \tau_2 \cup C)$ can be rewritten:

$$\begin{aligned} \{\mathsf{Pre}\ \tau \leq \mathsf{Any}\} \cup C &\longrightarrow C \\ \{\mathsf{Abs} \leq \mathsf{Any}\} \cup C &\longrightarrow C \\ \{\mathsf{Pre}\ \tau_1 \leq \mathsf{Pre}\ \tau_2\} \cup C &\longrightarrow \{\tau_1 \leq \tau_2\} \cup C \\ \{r\} \leq \{s\} & \end{aligned}$$

Remark: Constrain resolution fails for any case not defined above.

Solving constraints between field types is simple, the relation is trivial except between Pre τ constructors for which it is covariant in the argument τ. Subtyping between records proceeds very differently though. Consider the following constraint:

$$\{a : \alpha,\ b : \beta,\ \varphi\} \leq \{a : \alpha',\ \psi\}$$

The label b is not present on the right-hand side, but it could be part of the row ψ. Pottier introduces *demand-driver expansion* [Pot98b, Chapter 14.5] to deal with constraints of this form. When the type inference algorithm encounters such a constraint, it creates an *expansion*:

$$\psi \longleftarrow \{b : \beta',\ \psi'\}$$

And then decomposes the constraint into the following.

$$\alpha \leq \alpha' \qquad \beta \leq \beta' \qquad \varphi \leq \psi'$$

4.5 Other approaches

Subtyping The problem with information loss is tackled by Cardelli and Mitchell [CM89] in a full second-order type system with subtyping by introducing specialized type operators that allow for record types to be constrained by extension and restriction, mirroring the operators on record terms. Through the combination of subtyping and record type variables and operators, the system can express satisfactory types for polymorphic functions while avoiding information loss across applications.

However, by choosing a higher-order system they give up on type inference and require explicit type parameters to type polymorphic functions. These parameters interact with subtyping subtly when dealing with record updates and more type operators are added to the system to preserve soundness. As an example consider a function that flips the value of a boolean field. In their system, it is defined as follows.

$$\begin{aligned}\mathsf{let}\ \mathrm{update}(R \leq \{\mathrm{b} : \mathsf{bool}\})(r : R) : R \\ = r\ \text{with}\ \{b := \text{not}\ r.b\}\end{aligned}$$

Where R is the type parameter that preserves information. The only sound instantiation for R is $\{b : \mathsf{bool}\}$. Passing a subtype as the type parameter, e.g. $\{b : \mathsf{true}\}$, leads to an incorrect result type since the result type is not updated to reflect the value change. To solve this problem, Cardelli and Mitchell introduce an overriding operator $\leftarrow$, standing for restriction followed by extension, which properly changes the type of the field. The correct definition is then

$$\mathsf{let}\ \mathrm{update}(R \leq \{\mathrm{b} : \mathsf{bool}\})(r : R) : \{R \leftarrow b : \mathsf{bool}\}$$

A further problem arises from updating fields of records nested inside other records. To preserve field information across multiple levels a function needs to receive a type parameter for each level of depth, which is quite awkward in practice. To avoid this one last operator is introduced, type extraction, which extracts the type of a field by label, as selection does values.

Although sometimes clumsy this system is interesting to us because it combines subtyping and extensibility. The fact it is based on a higher-order theory means we cannot use it, but it can be seen as generalizing many concepts.

Kinds An alternative approach to record polymorphism is that of Ohori's [Oho95] whose goal was a type system for records and variants which supported an efficient compilation method. He achieved this with an extension of ML's type system [DM82] with *kinded quantification*, that is, polymorphic types of the form $\forall\alpha :: \kappa.\theta$, where a type variable α is constrained to range only over the set of types described by a *kind* κ. A kind κ is either the universal kind U, describing the set of all types or of the form $\{l_1 : \tau_1,\ \ldots,\ l_n : \tau_n\}$, describing the set of record types which contain at least those fields.

Ohori's kind system refines ML style polymorphism with a mechanism analogous to bounded quantification, while preserving the completeness of type inference, achieving an efficient compilation algorithm and avoiding the problem of information loss in the subtyping approach. As an example, field update has the following type in Ohori's system:

$$\forall\tau_1 :: U.\ \forall\tau_2 :: \{l : \tau_1\}.\ \tau_1 \to \tau_2 \to \tau_2$$

The main limitation of Ohori's system is that it did not deal with extensibility but this has

since been addressed. One approach by Alves and Ramos [AR21] refines the notion of record kind to include negative information, taking the form:

$$\{l_1 : \tau_1,\ \ldots,\ l_n : \tau_n \parallel l'_1 : \tau'_1,\ \ldots,\ l'_m : \tau'_m\}$$

This kind describes records that contain the fields before $\parallel$ but must not contain the ones after. Extension and restriction are given the types $\alpha + \{l : \tau\}$ and $\alpha - \{l : \tau\}$, respectively, where α is the type of the record being extended. For example, the type for polymorphic strict extension is:

$$\forall \tau_1 :: U.\ \forall \tau_2 :: \{\parallel l : \tau_1\}.\ \tau_1 \to \tau_2 \to \tau_2 + \{l : \tau_1\}$$

In contrast, another approach by Sulzmann [Sul97] reframes Ohori's calculus as an instance of the HM(X) framework [Sul01] and then, building on that instance, defines others for extension, concatenation and polymorphic labels. Extension in Sulzmann's system is given type $\forall \alpha\beta\gamma.\ \text{extend}_l(\alpha, \beta, \gamma) \Rightarrow \alpha \to \beta \to \gamma$ where extend is a constraint defined by a large set of rules, of which we present only some:

R6	$\text{extend}_l(\alpha, \beta, \gamma) \vdash (\gamma :: \{l : \beta\})$
R7	$\text{extend}_l(\alpha, \beta, \gamma) \wedge (\alpha :: \{l' : \tau\}) \vdash (\gamma :: \{l' : \tau\})$
R8	$\text{extend}_l(\alpha, \beta, \gamma) \wedge (\gamma :: \{l' : \tau\}) \vdash (\alpha :: \{l' : \tau\})$ for $l \neq l'$
R11	$\text{extend}_l(\alpha, \beta, \gamma) \wedge (\alpha :: \{l : \tau\}) \vdash \mathsf{false}$

The interesting rules are R6, which handles extension, R7-8 which preserves field information, and R11 which guarantees strictness. The remaining rules handle trivial or unsound cases.

Predicates and Qualified Types A type system for extensible records based on *predicates* is presented by Harper and Pierce [HP91]. A predicate is a logical formula over types that must be satisfied to use a value of that type. Predicates are used to capture the presence and absence of fields in record types to check operations. For example, given a predicate $r_1 \# r_2$ for the assertion that two records have disjoint sets of labels, and having row concatenation $r_1 \parallel r_2$ defined only when $r_1 \# r_2$, polymorphic selection $(-.l)$ is given the type:

$$\forall \alpha \forall r.(r \# \{l : \alpha\}) \Rightarrow \{r \parallel l : \alpha\} \to \alpha$$

Meaning that l is a label that appears only once in the record r. Harper and Pierce's approach to typing records dealt only with type checking but is one of the motivations for Jones' general theory of *qualified types* [Jon92] which also deals with type inference and compilation. Combining the notions of rows, kinds, qualified types, and constructors [Jon93] Gaster and Jones [GJ96, Gas98] present a higher-order polymorphic type system for extensible records and

variants with principal type inference and efficient compilation. Here row extension is used to capture positive information, that is, which labels are used, of a given row, while predicates are used to reflect negative information, that is, which labels are not used.

Scoped Labels All systems thus far have required that each label in a record be unique, so the extension operation would either fail or replace an existing label. However, there are models which allow repetition such as the tagged types of Berthomieu de le Moniés de Segazan [BDS95] and the scoped labels of Leijen [Lei05] used in the Koka programming language [Lei14]. In Leijen's system, it is not required that the set of labels of a record contain unique labels. Instead, when a record is extended with a value for a label that is already present, the new value takes priority over the older values already associated with that label. That is, when that label is next selected, the returned value will be the one most recently associated with that label. Older values for a label can be recovered by removing the newer one, introducing an elegant form of scoping over labels.

Chapter 5

Conclusion

The initial goal of this dissertation was to understand the novel algorithms for type inference in the presence of subtyping that are MLsub and Simple-sub. The first step was the review of literature on type systems and type theory from the beginning up to the Hindley-Milner type system that MLsub extends. This was followed by the study of MLsub and Simple-sub and the work they build on. A re-implementation of Simple-sub in Haskell was developed to help understand the details not covered by the original paper, such as type simplification.

While the original goal was to stop at this point and write this document, a review of alternatives to subtyping led to the work of Rémy on type systems for row polymorphism and his and Pottier's subsequent work on the combination of those systems with subtyping. It seemed possible to integrate row polymorphism into the already developed re-implementation of Simple-sub, and so this discovery led to the extension to the type system proposed here and the associated prototype.

5.1 Future Work

It is important to note that the extension delineated here is not yet complete. An important next step is to pursue formal proofs that completeness, soundness, and principality and preserved by the extension, that we believe should hold. Another is to continue the development of the type inference algorithm by fully considering the application of type simplification to row variables, which we believe to be straightforward but have not yet considered.

Although the developed prototype is only in an initial phase, it has been applied successfully to a few examples and the results are promising. The logical next step is to continue thoroughly testing the prototype, by applying it to more complex examples and by applying property testing techniques.

The type system for extensible records is relatively simple and based on Rémy and Pottier's early work [Rém95a, Pot98b]. Applying more of their ideas to the system would be an interesting

line of work, some of them include:

- Record concatenation as a primitive operation [Rém95b, Pot00];
- Conditional constraints [Pot00];
- Filters that generalize rows [Pot03];
- Lifting type constructors to the level of rows, for instance, applying a record of functions to a record of values [Pie04, Chapter 10].

Diverging from rows, a potentially interesting line of work to follow is to consider an extension of MLsub with proper intersection types to reinforce Dolan's notion that types should represent data flow. Consider the following definitions for the function twice and the polymorphic field selection.

$$\text{twice} = \lambda f.\ \lambda x.\ f\ (f\ x) \qquad \text{get} = \lambda r.\ r.l$$

Their types in MLsub are as follows.

$$\text{twice} : (\alpha \to \beta \sqcap \alpha) \to \alpha \to \beta \qquad \text{getX} : \{l : \alpha\} \to \alpha$$

The type inferred for the currying of twice and get is the following recursive type.

$$\text{twice get} : \mu\alpha.\ (\{l : \alpha\} \sqcap \beta) \to \beta$$

This is a function that can only be applied to infinite records $\{l = \{l = \{x = \ldots\}\}\}$, although it is safe to apply to finite records such as $\{l = \{l = 1\}\}$. The type assigned to the function twice constrains its parameter f to be a function that returns a value that is a possible specialization of its argument. This constraint represents that f can be applied to its output, however, it does not represent that f only has to be applied to its output *once*. An exact solution to this problem that integrates well with algebraic subtyping remains an open problem.

Bibliography

[AC93] R. M Amadio and L. Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15:575–631, 1993.

[AR21] S. Alves and M. Ramos. An ML-style record calculus with extensible records. *Electronic Proceedings in Theoretical Computer Science*, 351:1–17, 2021.

[AWP97] A. Aiken, E. L. Wimmers, and J. Palsberg. Optimal representations of polymorphic types with subtyping. In *Theoretical Aspects of Computer Software: Third International Symposium, TACS'97 Sendai, Japan, September 23–26, 1997 Proceedings 3*, pages 47–76. Springer, 1997.

[Bar84] H. P. Barendregt. The lambda calculus. In *Studies in Logic and the Foundations of Mathematics*, volume 103. North-Holland, 1984.

[BDS95] B. Berthomieu and C. De Sagazan. A calculus of tagged types, with applications to process languages. *Types for Program Analysis*, page 1, 1995.

[Car84] L. Cardelli. A semantics of multiple inheritance. In *International symposium on semantics of data types*, pages 51–67. Springer, 1984.

[Car94] L. Cardelli. Extensible records in a pure calculus of subtyping. In *Theoretical aspects of object-oriented programming: types, semantics, and language design*, pages 373–425. MIT press, 1994.

[Cas95] G. Castagna. Covariance and contravariance: conflict without a cause. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 17:431–447, 1995.

[CFC+58] H.B. Curry, R. Feys, W. Craig, J. R.. Hindley, and J. P. Seldin. *Combinatory logic*, volume 1. North-Holland Amsterdam, 1958.

[Chu40] A. Church. A formulation of the simple theory of types. *The journal of symbolic logic*, 5:56–68, 1940.

[Chu33] A. Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, 33–34:346–366 and 839–864, 1932/33.

[CM89] L. Cardelli and J. C. Mitchell. Operations on records. In *International Conference on Mathematical Foundations of Programming Semantics*, pages 22–52. Springer, 1989.

[CR36] A. Church and J. B. Rosser. Some properties of conversion. *Transactions of the American Mathematical Society*, 39(3):472–482, 1936.

[Cur34] H. B. Curry. Functionality in combinatory logic. *Proceedings of the National Academy of Sciences*, 20:584–590, 1934.

[Cur69] H.B. Curry. Modified basic functionality in combinatory logic. *Dialectica*, 23:83–92, 1969.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17:471–523, 1985.

[Dam84] L. Damas. *Type assignment in programming languages.* PhD thesis, The University of Edinburgh, 1984.

[DM82] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '82, pages 207–212, 1982.

[DM17] S. Dolan and A. Mycroft. Polymorphism, subtyping, and type inference in MLSub. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 60–72, 2017.

[DN66] O. Dahl and K. Nygaard. Simula: an algol-based simulation language. *Communications of the ACM*, 9(9):671–678, 1966.

[Dol17] S. Dolan. *Algebraic subtyping.* PhD thesis, University of Cambridge, 2017.

[DP02] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order.* Cambridge University Press, 2nd ed edition, 2002.

[EST95] J. Eifrig, S. Smith, and V. Trifonov. Sound polymorphic type inference for objects. In *Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications*, pages 169–184, 1995.

[FC05] A. Frisch and G. Castagna. A gentle introduction to semantic subtyping. In *Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 198–208, 2005.

[FCB02] A. Frisch, G. Castagna, and V. Benzaken. Semantic subtyping. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pages 137–146. IEEE, 2002.

[FCB08] A. Frisch, G. Castagna, and V. Benzaken. Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types. *Journal of the ACM (JACM)*, 55(4):1–64, 2008.

[FM88] Y. Fuh and P. Mishra. Type inference with subtypes. In *European Symposium on Programming*, pages 94–114, 1988.

[FM89] Y. Fuh and P. Mishra. Polymorphic subtype inference: Closing the theory-practice gap. In *TAPSOFT'89: Proceedings of the International Joint Conference on Theory and Practice of Software Development Barcelona, Spain, March 13–17, 1989 3*, pages 167–183. Springer, 1989.

[Gas98] B. R. Gaster. *Records, variants and qualified types.* PhD thesis, University of Nottingham, 1998.

[Gir72] J. Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur.* PhD thesis, Université Paris 7, 1972.

[GJ96] B. R. Gaster and M. P. Jones. A polymorphic type system for extensible records and variants. Technical report, Technical Report NOTTCS-TR-96-3, Department of Computer Science, University . . . , 1996.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: the language and its implementation.* Addison-Wesley Longman Publishing Co., Inc., 1983.

[Hin69] J. R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the american mathematical society*, 146:29–60, 1969.

[Hin97] J. R. Hindley. *Basic Simple Type Theory.* Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1997.

[HM95] M. Hoang and J. C. Mitchell. Lower bounds on type inference with subtypes. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 176–185, 1995.

[HP91] R. Harper and B. Pierce. A record calculus based on symmetric concatenation. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 131–142, 1991.

[JM88] L. Jategaonkar and J. Mitchell. ML with extended pattern matching and subtypes. In *Proceedings of the 1988 ACM conference on LISP and functional programming*, pages 198–211, 1988.

[Jon92] M. P. Jones. A theory of qualified types. In *ESOP*, volume 92, pages 287–306, 1992.

[Jon93] M. P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. In *Proceedings of the conference on Functional programming languages and computer architecture*, pages 52–61, 1993.

[KR35] S. C. Kleene and J. B. Rosser. The inconsistency of certain formal logics. *Annals of Mathematics*, pages 630–636, 1935.

[Lei05] D. Leijen. Extensible records with scoped labels. *Trends in Functional Programming*, 6:179–194, 2005.

[Lei14] D. Leijen. Koka: Programming with row polymorphic effect types. *arXiv preprint arXiv:1406.2061*, 2014.

[MFV22] R. Marques, M. Florido, and P. Vasconcelos. Towards Algebraic Subtyping for Extensible Records. In *2022 ML Family Workshop, ML 2022, Ljubjana, Slovenia (in affiliation with ICFP)*, 2022.

[Mil78] R. Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17:348–375, 1978.

[Mit84] John C. Mitchell. Coercion and type inference. In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 175–185, 1984.

[Mit91] J. C. Mitchell. Type inference with simple subtypes. *Journal of functional programming*, 1:245–285, 1991.

[MM19] J. G. Morris and J. McKinna. Abstracting extensible data types: or, rows by any other name. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–28, 2019.

[MTHM97] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The definition of standard ML: revised*. MIT press, 1997.

[OB88] A. Ohori and P. Buneman. Type inference in a database programming language. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming*, pages 174—183, 1988.

[Oho95] A. Ohori. A polymorphic record calculus and its compilation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 17(6):844–895, 1995.

[Par20] L. Parreaux. The simple essence of algebraic subtyping: principal type inference with subtyping made easy. *Proceedings of the ACM on Programming Languages*, 4:1–28, 2020.

[Pie02] B. C. Pierce. *Types and programming languages*. MIT press, 2002.

[Pie04] B. C. Pierce. *Advanced topics in types and programming languages*. MIT press, 2004.

[Pot98a] F. Pottier. A framework for type inference with subtyping. *ACM SIGPLAN Notices*, 34:228–238, 1998.

[Pot98b] F. Pottier. *Type inference in the presence of subtyping: from theory to practice.* PhD thesis, INRIA, 1998.

[Pot00] F. Pottier. A versatile constraint-based type inference system. *Nordic Journal of Computing*, 7(4):312–347, 2000.

[Pot01] F. Pottier. Simplifying subtyping constraints: a theory. *Information and computation*, 170:153–183, 2001.

[Pot03] F. Pottier. A constraint-based presentation and generalization of rows. In *18th Annual IEEE Symposium of Logic in Computer Science, 2003. Proceedings.*, pages 331–340. IEEE, 2003.

[PR05] F Pottier and R. Rémy. The essence of ML type inference. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 10, pages 389–489. MIT Press, 2005.

[Reh98] J. Rehof. *The complexity of simple subtyping systems.* PhD thesis, Citeseer, 1998.

[Rém89] D. Rémy. Typechecking records and variants as a natural extension of ML. In *Sixteenth Annual Symposium on Principles Of Programming Languages*, 1989.

[Rém93a] D. Rémy. Type inference for records in a natural extension of ML. In *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design.* MIT Press, 1993.

[Rém93b] D. Rémy. Typing record concatenation for free. In *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design.* MIT Press, 1993.

[Rém95a] D. Rémy. Better subtypes and row variables for record types. Presented at the workshop on Advances in types for computer science at the Newton Institute, Cambridge, UK, 1995.

[Rém95b] D. Rémy. A case study of typechecking with constrained types: Typing record concatenation. Presented at the workshop on Advances in types for computer science at the Newton Institute, Cambridge, UK, 1995.

[Rém98] D. Rémy. From classes to objects via subtyping. In *European Symposium On Programming*, volume 1381 of *Lecture Notes in Computer Science*. Springer, March 1998.

[Rey80] J. C. Reynolds. Using category theory to design implicit conversions and generic operators. In *International Workshop on Semantics-Directed Compiler Generation*, pages 211–258, 1980.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12:23—41, 1965.

[Sim03] V. Simonet. Type inference with structural subtyping: A faithful formalization of an efficient constraint solver. In *Asian Symposium on Programming Languages and Systems*, pages 283–302. Springer, 2003.

[Sta88] R: Stansifer. Type inference with subtypes. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 88–97, 1988.

[Sul97] M. Sulzmann. Designing record systems. Technical report, Research Report YALEU/DCS/RR-1128, Department of Computer Science, Yale University, 1997.

[Sul01] M. Sulzmann. A general type inference framework for hindley/milner style systems. In *International Symposium on Functional and Logic Programming*, FLOPS '01, pages 248–263, 2001.

[Tur37] A. M. Turing. Computability and λ-definability. *The Journal of Symbolic Logic*, 2:153–163, 1937.

[Wan87] M. Wand. Complete type inference for simple objects. In *IEEE Symposium on Logic in Computer Science*, pages 37–44, 1987.

[Wan88] M. Wand. Corrigendum: Complete type inference for simple objects. In *Proceedings. Third Annual Symposium on Logic in Computer Science*, page 132, 1988.

[Wan91] M. Wand. Type inference for record concatenation and multiple iheritance. *Information and Computation*, 93:1–15, 1991.