# Polymorphism, Subtyping, and Type Inference in MLsub

Stephen Dolan Alan Mycroft
University of Cambridge, UK
stephen.dolan@cl.cam.ac.uk alan.mycroft@cl.cam.ac.uk


## Abstract

We present a type system combining subtyping and ML-style parametric polymorphism. Unlike previous work, our system supports type inference and has compact principal types. We demonstrate this system in the minimal language MLsub, which types a strict superset of core ML programs.

This is made possible by keeping a strict separation between the types used to describe inputs and those used to describe outputs, and extending the classical unification algorithm to handle subtyping constraints between these input and output types. Principal types are kept compact by type simplification, which exploits deep connections between subtyping and the algebra of regular languages. An implementation is available online.

***Categories and Subject Descriptors*** D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Type structure

***Keywords*** Subtyping, Polymorphism, Type Inference, Algebra

## 1. Introduction

The Hindley-Milner type system of ML and its descendants is popular and practical, sporting decidable type inference and principal types. However, extending the system to handle subtyping while preserving these properties has been problematic.

Subtyping is useful to encode extensible records, polymorphic variants, and object-oriented programs, but also allows us to expose more polymorphism even in core ML programs that do not use such features. For instance, given the definition $\text{twice} = \lambda f.\lambda x.f(fx)$, ML gives $\text{twice}\ (\lambda x.\text{true})$ the principal type $\text{bool} \to \text{bool}$, allowing only arguments of type bool, while MLsub gives it type $\top \to \text{bool}$, allowing arguments of any type (see Section 2.4 for details).

Supporting subtyping requires thinking carefully about data flow. Consider the `select` function, which takes three arguments: a predicate $p$, a value $v$ and a default $d$, and returns the value if the predicate holds of it, and the default otherwise:

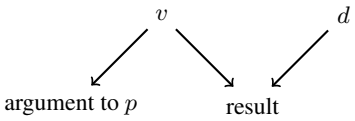
$$\texttt{select}\ p\ v\ d = \texttt{if}\ (p\ v)\ \texttt{then}\ v\ \texttt{else}\ d$$

In ML and related languages, `select` has type scheme

$$\forall\alpha.\,(\alpha \to \texttt{bool}) \to \alpha \to \alpha \to \alpha$$

This scheme is quite strange, in that it demands that whatever we pass as the default $d$ be acceptable to the predicate $p$. But this constraint does not arise from the behaviour of the program: at no point does our function pass $d$ to $p$.

Let's examine the actual data flow of this function:




Only by ignoring the orientation of the edges above could we conclude that $d$ flows to the argument of $p$. Indeed, this is exactly what ML does: by turning data flow into equality constraints between types, information about the *direction* of data flow is ignored. Since equality is symmetric, data flow is treated as undirected.

To support subtyping is to care about the direction of data flow. With subtyping, a source of data must provide at least the guarantees that the destination requires, but is free to provide more.

The data flow graph of a program gives rise to constraints between types, which can be represented as a *constraint graph*. Once we add subtyping, we must treat the constraint graph as directed rather than undirected. This directed constraint graph complicates matters: instead of using unification to find connected components in the constraint graph, we need to deal with reachability and cycles and strong connectivity.

One way to sidestep these difficulties is to widen the definition of "type" to include an explicit constraint graph. This gives the notion of a *constrained type* used in previous work [7, 22], where the principal type of an expression is given as a type with an attached explicit graph of constraints. For instance, such a system might represent the type of `select` as follows:

$$(\alpha \to \texttt{bool}) \to \alpha \to \beta \to \gamma \mid \alpha \le \gamma,\ \beta \le \gamma$$

Unfortunately, this just moves the burden of dealing with constraint graphs onto the programmer. Constrained types are often large, and are difficult to compare.

In his PhD thesis, Pottier[1] noticed that constraint graphs can be simplified due to the structure of data flow. By keeping a strict separation between inputs and outputs, we can always represent the constraint graph as a *bipartite* graph: data flows from inputs to outputs. With edges only from inputs to outputs, such graphs have no cycles (or even paths of more than one edge), simplifying analysis.

We take this insight a step further, and show that by keeping the same religious distinction between input and output we can avoid using explicit constraint graphs altogether, giving a much simpler type system. In MLsub, `select` has this type scheme:

$$\forall\alpha, \beta.\,(\alpha \to \texttt{bool}) \to \alpha \to \beta \to (\alpha \sqcup \beta)$$



[1] The *mono-polarity invariant* [18, ch. 12]

This represents the data flow graph above: the predicate $p$ must accept the value $v$ (of type $\alpha$), but the default $d$ may be of a different type $\beta$. The output, being of type $\alpha \sqcup \beta$, can only be used in contexts that accept either an $\alpha$ or a $\beta$.

Our contributions are as follows:

***The MLsub type system*** We extend the ML type system with subtyping (Section 2). This amounts to a trivial change to the typing rules (the addition of a subtyping rule), but we take some care in constructing the lattice of types. The distinction between inputs and outputs uses an unusual (but equivalent) formulation of the typing rules (Section 3), which simplifies type inference.

***Type inference*** We show how MLsub preserves the decidability and principality of ML type inference (Section 4). As part of this, we introduce *biunification*, a variant of the unification algorithm which handles subtype rather than equality constraints.

***Type simplification*** MLsub types have many equivalent representations (e.g. bool $\sqcup$ bool $\sqcup \bot$ is equivalent to just bool), and a naive implementation of inference may generate a complicated one. Like previous work, we use automata to simplify types (Section 5). Unlike previous work, we do not specify a particular simplification algorithm. Instead, we prove the general theorem that two automata represent equivalent types iff they accept equal languages (Section 5.3). This allows any optimisation algorithm from the literature on finite automata to be used to simplify types.

### 1.1 Focus of this Paper

This paper focuses on the core problem of inferring types for pure programs in a minimal ML-style calculus (booleans, records, functions and `let`). A full-scale programming language requires many more features, some of which are described in the first author's thesis [6], along with proofs of this paper's results.

Two important topics not discussed here are *deciding subsumption* and *mutable reference types*. Subsumption, written $\leq^\forall$ and discussed in Section 3.2, describes whether one polymorphic type scheme is more general than another. While unnecessary for type inference, it is necessary to decide subsumption in order to check modules against signatures, or to compare an inferred type scheme with a user-provided annotation. Unlike much previous work, the algebraic construction of types in MLsub makes subsumption decidable [6, Ch. 8].

Mutable references are usually typed with *invariant* parameters (neither co- nor contravariant), which would appear to pose a problem for MLsub since inference relies on each type parameter being either co- or contravariant. However, the issue is easily solved: the reference type can be given two type parameters, a contravariant parameter for what goes in and a covariant parameter for what comes out (see e.g. Pottier [18, p.165]). In fact, this encoding gives more precise types than invariance [6, Sec. 9.1].

A prototype implementation of MLsub is available on the first author's website:

`http://www.cl.cam.ac.uk/~sd601/mlsub`

This prototype includes features not discussed here for space reasons, such as type definitions, labelled and optional function arguments, type annotations, and others. When tested against OCaml's standard `List` module (with some syntactic changes to satisfy our rather primitive parser), the types inferred by MLsub were no larger than the types inferred by OCaml, despite being more general. For instance, `List.map` is assigned a type syntactically identical to that assigned by OCaml, but in MLsub may be passed a list containing records with different sets of fields, and a function that accesses only the common ones.

### 1.2 An Algebraic Approach

A vital aspect of this work is to adopt a more algebraic approach to the construction of types and the subtyping relation. We distinguish two aspects of types: the *model* of types (the concrete construction of types themselves, usually some flavour of syntax tree), and their *algebra* or *theory* of types (the equations and relations true about them).

Traditionally, we start by picking a model of types, by giving them a syntax, and only later do we explore the algebra, by proving facts about this syntax. Unfortunately, with subtyping this leads to a rather ill-behaved algebra of types. We give an example here, but leave detailed discussion of related work to Section 6.

A standard definition of types with function types, top and bottom might start with the following syntax for ground types:

$$\tau ::= \bot \mid \tau \to \tau \mid \top$$

We give these a subtyping order making functions contravariant in their domain and covariant in their range, and making $\bot$ and $\top$ the least and greatest types. With this ordering, these types form a lattice, so for any two types $\tau_1, \tau_2$ we have their least upper bound $\tau_1 \sqcup \tau_2$ and greatest lower bound $\tau_1 \sqcap \tau_2$. We introduce type variables $\alpha, \beta, \ldots$ by quantifying over these ground types.

Despite the succinctness of this definition, these types have a remarkably complicated and unwieldy algebra. Pottier gave the following example illustrating the incompleteness of a particular subtyping algorithm [18, p. 86]. We reuse his example, arguing that it demonstrates a flaw in the underlying definition rather than the incompleteness of one algorithm:

$$(\bot \to \top) \to \bot \;\leq\; (\alpha \to \bot) \sqcup \alpha \tag{E}$$

With the above definition of subtyping, (E) holds. We can show this by case analysis on $\alpha$: if $\alpha = \top$, then it holds trivially. Otherwise, $\alpha \leq \bot \to \top$ (the largest function type), and so $(\bot \to \top) \to \bot \leq \alpha \to \bot$ by contravariance.

However, this example is rather fragile. Extending the type system with a more general function type $\tau_1 \overset{\circ}{\to} \tau_2$ (a supertype of $\tau_1 \to \tau_2$, say "function that may have side effects") gives a counterexample $\alpha = (\top \overset{\circ}{\to} \bot) \overset{\circ}{\to} \bot$. This counterexample arises even though the extension did not affect the subtyping relationship between existing types, and only added new ones. Introducing type variables by quantification over ground types permits reasoning by case analysis on types, which is invalidated by language extensions.

In order to make an extensible type system, we focus on building a well-behaved algebra of types, rather than choosing for minimality of syntax. Two changes are particularly important: first, we add type variables directly to the definition of types, rather than first defining ground types and then quantifying over them. This disallows case analysis over types, and ensures that the awkward example (E) does not hold in MLsub.

Secondly, and more subtly, we construct types as a *distributive lattice*. Frequently, the lattice of types is constructed such that different type constructors have no common subtypes other than $\bot$. That is, given a function type $\tau_f$ and a record type $\tau_r$, then $\tau_f \sqcap \tau_r = \bot$. While this seems intuitive, it also implies some odd relations, such as:

$$\tau_f \sqcap \tau_r \leq \text{bool}$$

This is counterintuitive, and makes type inference more difficult: the compiler must accept as a boolean any value used as both a function and a record. The vacuous reasoning that justifies it (reasoning that the left-hand side is empty, and thus a subtype of anything) is not extensible, since adding values which are usable both as functions and records falsifies it.

To preclude such examples, we demand that all subtyping relationships between the meet of two types and another type follow

$$\text{(VAR-}\lambda\text{)}\quad \frac{}{\Gamma \vdash x : \tau}\ \Gamma(x) = \tau$$

$$\text{(VAR-}\forall\text{)}\quad \frac{}{\Gamma \vdash \hat{\mathbf{x}} : \tau[\vec{\tau}/\vec{\alpha}]}\ \Gamma(\hat{\mathbf{x}}) = \forall\vec{\alpha}.\tau$$

$$\text{(ABS)}\quad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2}$$

$$\text{(APP)}\quad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 e_2 : \tau_2}$$

$$\text{(LET)}\quad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \forall\vec{\alpha}.\tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \text{let } \hat{\mathbf{x}} = e_1 \text{ in } e_2 : \tau_2}\ \vec{\alpha} \text{ not free in } \Gamma$$

$$\text{(TRUE)}\quad \frac{}{\Gamma \vdash \text{true} : \text{bool}}$$

$$\text{(FALSE)}\quad \frac{}{\Gamma \vdash \text{false} : \text{bool}}$$

$$\text{(IF)}\quad \frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau}$$

$$\text{(CONS)}\quad \frac{\Gamma \vdash e_1 : \tau_1 \quad \ldots \quad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash \{\ell_1 = e_1, \ldots, \ell_n = e_n\} : \{\ell_1 : \tau_1, \ldots, \ell_n : \tau_n\}}$$

$$\text{(PROJ)}\quad \frac{\Gamma \vdash e : \{\ell : \tau\}}{\Gamma \vdash e.\ell : \tau}$$

$$\text{(SUB)}\quad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash e : \tau'}\ \tau \le \tau'$$

Figure 1: MLsub typing rules

from some property of the first two. Formally,

If $\tau_1 \sqcap \tau_2 \le \tau_3$, then $\tau_1' \sqcap \tau_2' = \tau_3$ for some $\tau_1' \le \tau_3, \tau_2' \le \tau_3$

This condition is equivalent to distributivity. Its effect on the type system is to require the existence of more types, allowing us to distinguish $\tau_f \sqcap \tau_r$ from $\bot$.

The construction of types in Section 2.1 follows these principles. We do build an explicit model of types, but we do so by starting from the algebra: types are defined by quotienting syntax by the equations of the algebra.

## 2. The MLsub Language

Presentations of the Hindley-Milner calculus often limit themselves to function types and a base type, since all of the difficulties encountered in type inference arise in this simpler setting. With subtyping, we need a slightly richer type language before we hit the difficult cases. So, we include a base type (booleans), functions (so that we must handle co- and contravariance in the same type constructor), and records (so that we must handle subtyping between type constructors of different arity).

In all variants of ML, the typing of let-bound and $\lambda$-bound variables is quite different, allowing polymorphism for let-bound variables and allowing the type of $\lambda$-bound variables to be inferred from their uses. This separation is important in MLsub, so we include it in the syntax, distinguishing $\lambda$-bound variables $x$ from let-bound variables $\hat{\mathbf{x}}$, giving the following:

$$\begin{aligned} e ::= \quad & x \mid \lambda x.e \mid e_1\, e_2 \mid \\ & \{\ell_1 = e_1, \ldots, \ell_n = e_n\} \mid e.\ell \mid \\ & \text{true} \mid \text{false} \mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \mid \\ & \hat{\mathbf{x}} \mid \text{let } \hat{\mathbf{x}} = e_1 \text{ in } e_2 \end{aligned}$$

In the record construction syntax $\{\ldots\}$, the labels $\ell_i$ are drawn from some collection $\mathcal{L}$ and assumed distinct. We take the order of labels to be significant in expressions (so that a simple, deterministic, left-to-right evaluation order can be defined), but insignificant in types.

Type environments $\Gamma$ likewise distinguish let and $\lambda$-bound variables, by mapping $\lambda$-bound variables to monomorphic types $\tau$, and let-bound variables to *type schemes* $\forall\vec{\alpha}.\tau$:

$$\Gamma ::= \epsilon \mid \Gamma, x : \tau \mid \Gamma, \hat{\mathbf{x}} : \forall\vec{\alpha}.\tau$$

The typing rules for MLsub (Fig. 1) are entirely standard: the Hindley-Milner calculus, extended with records, booleans and the standard subtyping rule. The only novelty here lies in the construction of the monotypes $\tau$.

### 2.1 Defining Types

As well as standard boolean, function and record types, MLsub has least-upper-bound and greatest-lower-bound type operators (written $\sqcup$ and $\sqcap$), and least and greatest types (written $\bot$ and $\top$), which are needed to assign principal types.

Due to subtyping, recursive types are also necessary for principality. With the $\top$ and $\bot$ types, there are expressions typeable in MLsub not typeable in ML, for instance:

$$Y(\lambda f.\lambda x.f)$$

where $Y$ is the call-by-value Y-combinator. This expression gives a function which ignores its argument and returns itself. In MLsub, this expression is typeable with any of the types $\top \to \top$, $\top \to (\top \to \top)$, $\top \to (\top \to (\top \to \top))$, ..., and the recursive type $\mu\alpha.\top \to \alpha$ is needed to express its principal type.

So, we define MLsub types using the following syntax:

$$\begin{aligned} \tau \ ::= \ & \text{bool} \mid \tau_1 \to \tau_2 \mid \{\ell_1 : \tau_1, \ \ldots, \ \ell_n : \tau_n\} \mid \\ & \alpha \mid \top \mid \bot \mid \tau \sqcup \tau \mid \tau \sqcap \tau \end{aligned}$$

where $\alpha$ ranges over type variables. Rather than using a syntactic list, record types are defined as a partial function from record labels to types, making the order of label-type pairs in record types irrelevant. Occasionally, we explicitly denote this: $\{\ell_1 : \tau_1, \ell_2 : \tau_2\}$ is the same as $\{f\}$ where $\operatorname{dom} f = \{\ell_1, \ell_2\}$, $f(\ell_1) = \tau_1$, $f(\ell_2) = \tau_2$. We assume the set $\mathcal{L}$ of record labels and the set $\mathcal{A}$ of type variables to be finite, but arbitrary.

We allow the syntax trees of types to be infinite, except that we require that the syntax tree of any type contain no infinite path of $\sqcup$, $\sqcap$ without an intervening type constructor (bool, $\to$, or $\{\ldots\}$). These syntax trees are quotiented by an equivalent relation making $\sqcup$, $\sqcap$, $\bot$, $\top$ a distributive lattice. Thus, we allow the infinite type:

$$\text{bool} \to (\text{bool} \to (\text{bool} \to \ldots))$$

but not:

$$\text{bool} \sqcup (\text{bool} \sqcup (\text{bool} \sqcup \ldots))$$

Rather than directly operating on these infinite syntax trees, the concrete algorithms of Section 4 operate on *polar types* (defined in Section 3.3), a restricted subset of types which can be represented finitely and which suffice for type inference.

More precisely, we define for each $i \in \mathbb{N}$ the *$i$-types* to be terms of the above syntax, in which type constructors (bool, $\to$ and $\{\ldots\}$) are nested to a depth of at most $i$. We define an equivalence relation $\equiv_i$ on $i$-types as the least relation closed under the equations of distributive lattices (Fig. 2, or see e.g. Davey and Priestley [5]) and the equations of Fig. 3. The distributive lattice $\mathcal{T}_i$ is then given by the equivalence classes of $\equiv_i$.

The distributive lattice $\mathcal{T}$ of MLsub types is the inverse limit:

$$\mathcal{T} = \varprojlim_i \mathcal{T}_i$$

The equations of Fig. 3 imply the usual subtyping rules (co- and contravariance for functions, and width and depth subtyping for records). They go a little further by directly specifying that e.g. the upper bound of two record types has the intersection of their fields. We recover a partial order (up to $\equiv$) from this structure by defining

$$\tau \sqcup \tau \equiv \tau \qquad \tau \sqcap \tau \equiv \tau$$
$$\tau_1 \sqcup \tau_2 \equiv \tau_2 \sqcup \tau_1 \qquad \tau_1 \sqcap \tau_2 \equiv \tau_2 \sqcap \tau_1$$
$$\tau_1 \sqcup (\tau_2 \sqcup \tau_3) \equiv (\tau_1 \sqcup \tau_2) \sqcup \tau_3 \qquad \tau_1 \sqcap (\tau_2 \sqcap \tau_3) \equiv (\tau_1 \sqcap \tau_2) \sqcap \tau_3$$
$$\tau_1 \sqcup (\tau_1 \sqcap \tau_2) \equiv \tau_1 \qquad \tau_1 \sqcap (\tau_1 \sqcup \tau_2) \equiv \tau_1$$
$$\bot \sqcup \tau \equiv \tau \qquad \bot \sqcap \tau \equiv \bot$$
$$\top \sqcup \tau \equiv \top \qquad \top \sqcap \tau \equiv \tau$$
$$\tau_1 \sqcup (\tau_2 \sqcap \tau_3) \equiv (\tau_1 \sqcup \tau_2) \sqcap (\tau_1 \sqcup \tau_3)$$
$$\tau_1 \sqcap (\tau_2 \sqcup \tau_3) \equiv (\tau_1 \sqcap \tau_2) \sqcup (\tau_1 \sqcap \tau_3)$$

Figure 2: Equations of distributive lattices

$$(\tau_1 \to \tau_2) \sqcup (\tau_1' \to \tau_2') \equiv (\tau_1 \sqcap \tau_1') \to (\tau_2 \sqcup \tau_2')$$
$$(\tau_1 \to \tau_2) \sqcap (\tau_1' \to \tau_2') \equiv (\tau_1 \sqcup \tau_1') \to (\tau_2 \sqcap \tau_2')$$

$$\{f\} \sqcup \{g\} \equiv \{h\} \quad \text{where } \operatorname{dom} h = \operatorname{dom} f \cap \operatorname{dom} g \text{ and } h(\ell) = f(\ell) \sqcup g(\ell)$$

$$\{f\} \sqcap \{g\} \equiv \{h\} \quad \text{where } \operatorname{dom} h = \operatorname{dom} f \cup \operatorname{dom} g \text{ and } h(\ell) = \begin{cases} f(\ell) \sqcap g(\ell) & \text{if } \ell \in \operatorname{dom} f \cap \operatorname{dom} g \\ f(\ell) & \text{if } \ell \in \operatorname{dom} f - \operatorname{dom} g \\ g(\ell) & \text{if } \ell \in \operatorname{dom} g - \operatorname{dom} f \end{cases}$$

Figure 3: Equations for function and record type constructors

$\tau_1 \leq \tau_2$ as $\tau_1 \sqcup \tau_2 \equiv \tau_2$, or equivalently $\tau_1 \equiv \tau_1 \sqcap \tau_2$. In this order, $\sqcup$ gives least-upper-bounds, $\sqcap$ gives greatest-lower-bounds, and $\bot$ and $\top$ are the least and greatest elements.

Since $\mathcal{L}$ and $\mathcal{A}$ are assumed finite, $\mathcal{T}$ is a *profinite distributive lattice* (an inverse limit of finite lattices), which is an algebraic class with many pleasant properties. For an algebraic derivation of this construction, see the first author's thesis [6, Ch. 3], which also contains the technical details of the construction of $\mathcal{T}$.

### 2.2 Recursive Types

The treatment of recursive types in MLsub deserves special comment. Traditionally, a recursive type $\mu\alpha.\tau$ is treated as the unique fixed point of a map $\phi(\tau') = \tau[\tau'/\alpha]$. This justifies the $\alpha$-renaming of the variable bound by $\mu$, as well as giving us the following reasoning principle:

$$\tau[\tau_0/\alpha] = \tau_0 \implies \mu\alpha.\tau = \tau_0$$

This is not strong enough in the presence of subtyping, since the knowledge that $\tau[\tau_0/\alpha] = \tau_0$ is hard to come by. Instead, we need a reasoning principle that uses subtyping information:

$$\tau[\tau_0/\alpha] \leq \tau_0 \implies \mu\alpha.\tau \leq \tau_0$$

This requires that $\mu\alpha.\tau$ be a *least pre-fixed point* of the map $\phi$, a condition not implied by being the unique fixed point. (For example, the function $f(x) = 2x$ on $\mathbb{R}$ has a unique fixed point at 0, but no least pre-fixed point).

The algebraic structure of $\mathcal{T}$ allows us to construct least pre-fixed points. We say that $\alpha$ is *covariant* in $\tau$ if all occurrences of $\alpha$ in the syntax of $\tau$ are to the left of an even number of $\to$. More concisely, $\alpha$ is covariant in $\tau$ when the map $\phi(\tau') = \tau[\tau'/\alpha]$ is monotone.

**Lemma 1.** *If $\alpha$ is covariant in $\tau$, then $\phi(\tau') = \tau[\tau'/\alpha]$ has a least pre-fixed point, written $\mu^+\alpha.\tau$, and a greatest post-fixed point written $\mu^-\alpha.\tau$.*

The $\mu$ syntax binds loosely: $\mu^+\alpha.\,\tau_1 \to \tau_2$ is $\mu^+\alpha.(\tau_1 \to \tau_2)$, not $(\mu^+\alpha.\tau_1) \to \tau_2$. This lemma provides both least pre-fixed and greatest post-fixed points, but they need not agree in general: $\alpha$ is covariant in $\alpha$, but:

$$\mu^+\alpha.\,\alpha = \bot$$
$$\mu^-\alpha.\,\alpha = \top$$

To remedy this, we introduce another condition. We say that $\alpha$ is *guarded* in $\tau$ if all occurrences of $\alpha$ occur under at least one type constructor (that is, are guarded by a function or record type). Note that the operators $\sqcup$ and $\sqcap$ are not, by themselves, sufficient for guardedness: $\alpha$ is guarded in $\bot \to \alpha$ and $\bot \to (\alpha \sqcup \text{bool})$ but not in $\alpha \sqcup \text{bool}$.

**Lemma 2.** *If $\alpha$ is covariant and guarded in $\tau$, then*

$$\mu^+\alpha.\tau = \mu^-\alpha.\tau$$

We write this type as $\mu\alpha.\tau$. Note that $\alpha$ is vacuously covariant and guarded in any $\tau$ in which it does not occur, but this causes no issues: if $\alpha$ does not appear in $\tau$, then $\mu\alpha.\tau = \tau$.

At first glance, it seems that the covariance condition reduces expressiveness since it excludes, for example, $\mu\alpha.\alpha \to \alpha$. However, a result of Bekić [2] allows us to find a unique type $\tau$ such that $\tau \equiv \tau \to \tau$. The trick is to define *two* types $\tau_1, \tau_2$ in a mutually recursive fashion:

$$\tau_1 = \tau_2 \to \tau_1 \qquad \tau_2 = \tau_1 \to \tau_2$$

Here, $\tau_1$ depends on itself in a covariant and guarded way, as does $\tau_2$. We can thus introduce well-formed $\mu$-types:

$$\tau_1 = \mu\alpha.\,\tau_2 \to \alpha \qquad \tau_2 = \mu\beta.\,\tau_1 \to \beta$$

Substitution gives:

$$\tau_1 = \mu\alpha.\,(\mu\beta.\,\alpha \to \beta) \to \alpha \quad \tau_2 = \mu\beta.\,(\mu\alpha.\,\beta \to \alpha) \to \beta$$

These types are $\alpha$-equivalent, so we conclude that $\tau_1 \equiv \tau_2$, giving us a type $\tau \equiv \tau \to \tau$.

This technique works in general, allowing us to express arbitrary recursive types by encoding them with covariant ones[2]—any map $\phi(\tau_1, \tau_2)$ (contravariant in $\tau_1$ and covariant in $\tau_2$) has a fixpoint given by:

$$\mu\alpha.\,\phi(\mu\beta.\,\phi(\alpha, \beta), \alpha)$$

### 2.3 Soundness

We equip MLsub with a small-step operational semantics $e \longrightarrow e'$ (elided for space), which evaluates terms left-to-right using call-by-value. We carry out a standard soundness proof by proving progress and preservation [23], but a wrinkle arises at the point where a standard soundness proof attempts inversion on the subtyping relation.

Our subtyping relation is not defined by cases, and inversion by case analysis is not a meaningful operation. Indeed, given merely that $\tau_1 \leq \tau_2 \to \tau_3$, it is not the case that $\tau_1$ is a function type: we could have $\tau_1 = \bot$ or $\tau_1 = (\alpha \to \beta) \sqcap \alpha$. However, a weaker form of inversion does hold in $\mathcal{T}$, when we know the head type constructor on both sides of $\leq$:

[2] This works in MLsub because positive recursive types (e.g. $\mu\alpha.\ (\alpha \to \bot) \to \bot$) are the unique solutions of their defining equations (e.g. $\tau = (\tau \to \bot) \to \bot$). Positive recursive types can also be encoded in say, System F, but lack uniqueness: $\tau = (\tau \to \bot) \to \bot$ has distinct least (inductive) and greatest (coinductive) solutions, which prevents the use of Bekić's theorem to find $\tau$ where $\tau = \tau \to \tau$. If we try to carry through the above construction in System F, we end up defining $\tau_1$ inductively and $\tau_2$ coinductively (or vice versa), and cannot then conclude that $\tau_1 = \tau_2$.

**Lemma 3** (Inversion). *If $\tau_1 \to \tau_2 \leq \tau_1' \to \tau_2'$, then $\tau_1' \leq \tau_1$ and $\tau_2 \leq \tau_2'$.*
*If $\{f\} \leq \{g\}$, then $f(\ell) \leq g(\ell)$ for $\ell \in \operatorname{dom} g$.*

Happily, this weak inversion principle suffices to carry out the standard proof.

**Theorem 4** (Progress). *If $\vdash e : \tau$ then either $e$ is a value, or $e \longrightarrow e'$ for some $e'$.*

**Theorem 5** (Preservation). *If $\vdash e : \tau$ and $e \longrightarrow e'$, then $\vdash e' : \tau$.*

### 2.4 MLsub Subsumes ML

Since the typing rules for MLsub are just those of ML with an extra (SUB) rule, any expression typeable in ML is trivially typeable in MLsub with the same type. Furthermore, since recursive types can be expressed in MLsub (Section 2.2), MLsub also subsumes MLrec (ML extended with recursive types). We have the (strict) inclusions:

$$\text{ML} \subset \text{MLrec} \subset \text{MLsub}$$

However, some ML expressions can be given a more general type in MLsub. Consider the function twice, defined as:

$$\lambda f.\lambda x.f(fx)$$

This is typeable in ML with type scheme $\forall\alpha.\,(\alpha \to \alpha) \to \alpha \to \alpha$. The same type scheme works in MLsub, but the following principal type is more general:

$$\forall\alpha, \beta.\,((\alpha \sqcup \beta) \to \beta) \to (\alpha \to \beta)$$

This type scheme states that any result of $f$ (of type $\beta$) must be usable as an input of $f$ (since $\beta \leq \alpha \sqcup \beta$), and that twice $f$ must be passed something (of type $\alpha$) usable as an input of $f$ (since $\alpha \leq \alpha \sqcup \beta$), but these types need not be identical.

For instance, in ML, twice $(\lambda x.\text{true})$ has type bool $\to$ bool since the input type of $\lambda x.$true is unified with its output type. In MLsub, the same expression has type $\top \to$ bool and may be passed anything, since the above type scheme for twice can be instantiated by $\alpha \mapsto \top, \beta \mapsto$ bool, giving

$$(\top \to \text{bool}) \to \top \to \text{bool}$$

Note that the operator $\sqcup$ is instrumental in defining the above principal typing scheme for twice. In fact, Hoang and Mitchell showed that a type system with subtyping using only ML types (i.e. types lacking $\sqcup$ and $\sqcap$) cannot express the principal type for twice without explicit constraints [11].

Other expressions are not typeable at all in ML, but are in MLsub. For instance, the self-application function $\lambda x.\,xx$ is not typeable in ML while in MLsub it can be given the following (principal) type scheme:

$$\forall\alpha, \beta.\,((\alpha \to \beta) \sqcap \alpha) \to \beta$$

Applying this function to $\lambda x.$true gives a term of type bool.

## 3. Reformulated Typing Rules

In this section, we reformulate the typing rules to more clearly separate inputs and outputs, which simplifies type inference.

The strength of ML is that expressions can be written once and re-used at different types using the let construct. For instance, here the identity function is bound to $g$ and used twice:

$$\text{let } g = \lambda x.x \text{ in } (\dots g \dots g \dots)$$

The function $g$ has type scheme $\forall\alpha.\,\alpha \to \alpha$, and then $g$ can be used at distinct types $\tau_1 \to \tau_1$ and $\tau_2 \to \tau_2$.

The tricky part of the ML type system is when the type of a let-bound expression cannot be fully generalised because it depends on $\lambda$-bound variables, as in the following example.

$$\lambda f.\,\text{let } g = \lambda x.\text{if } f\,x \text{ then } x \text{ else } x \text{ in } (\dots g \dots g \dots)$$

If $\Gamma(f) = \alpha \to$ bool, the type of $g$ is $\alpha \to \alpha$, which cannot be generalised because of the dependency on $f$. The two uses of $g$ must then both be at the same type.

Adding subtyping complicates this further. Suppose $\Gamma(f) = \tau_0 \to$ bool and $\tau_1$ and $\tau_2$ are both subtypes of $\tau_0$. In order to maintain the let-expansion property that let $x = e_1$ in $e_2$ is typeable if $e_2[e_1/x]$ is (again assuming $x$ occurs in $e_2$), we must allow the above example with $g$ used at types $\tau_1 \to \tau_1$ and $\tau_2 \to \tau_2$.

Giving $g$ a monomorphic type does not suffice. We need a type like "$\forall\alpha$ where $\alpha \leq \tau_0.\,\alpha \to \alpha$", but we promised to avoid types with constraints. Instead, we reformulate the typing rules to make explicit such partially generalised types.

### 3.1 Typing Schemes

Following Trifonov and Smith [22] and Pottier [18], our reformulated typing rules use the so-called *lambda-lifted* style. Instead of type schemes $\sigma$, we use *typing schemes* $[\Delta]\tau$, where $\Delta$ is as before a finite map from $\lambda$-bound variables to types, and $\tau$ is a type. Typing schemes have no free type variables: *all* of their type variables are implicitly generalised, and so we omit the $\forall$. Every typing scheme explicitly carries around its dependencies on $\lambda$-bound *program variables*, rather than relying on a coincidence of *type variables* (which happens to work for ML, but is trickier with subtyping).

Thus, in the previous example, $g$ (being defined in terms of the $\lambda$-bound variable $f$) gets the following typing scheme:

$$[f : \alpha \to \beta]\alpha \to \alpha$$

For the reformulated rules, we assume the expression being typed has undergone enough $\alpha$-renaming that bound variables are distinct, avoiding issues with shadowing.

We split the type environment $\Gamma$ into a monomorphic part $\Delta$ and a polymorphic part $\Pi$, where $\Delta$ maps $\lambda$-bound variables to monomorphic types and $\Pi$ maps let-bound variables to typing schemes:

$$\Delta ::= \epsilon \mid \Delta,\, x : \tau$$
$$\Pi ::= \epsilon \mid \Pi,\, \hat{\mathbf{x}} : [\Delta]\tau$$

Writing the ML typing rules in terms of typing schemes leads to the rules in Figure 4, with judgements of the form $\Pi \Vdash e : [\Delta]\tau$. We write $\Delta_1 \sqcap \Delta_2$ to denote that $\Delta$ such that $\operatorname{dom}\Delta = \operatorname{dom}\Delta_1 \cup \operatorname{dom}\Delta_2$, and $\Delta(x) = \Delta_1(x) \sqcap \Delta_2(x)$ when both are defined, and equal to whichever is defined in other cases.

Mostly, the translation consists of bringing $\Delta$ to the other side of the turnstile, except that the side-condition of the new (SUB) rule uses the subsumption relation $\leq^\forall$, combining subtyping and instantiation of type variables. In particular, note that if the subtyping relation were trivial (i.e. all $\leq$ replaced with $=$), then the relation $\leq^\forall$ reduces to Hindley-Milner instantiation.

Unlike previous work in which explicit constraints complicated the presentation, the reformulated typing rules can be related directly to the original, ML-style typing rules:

**Theorem 6.** *$\vdash e : \tau$ iff $\Vdash e : [\,]\tau$*

The proof is by proving a stronger theorem, which works in nonempty typing contexts. The proof is straightforward induction, although the statement of the theorem is somewhat messy due to the need to convert between $\Gamma$ and $\Pi/\Delta$-style environments.

$$[\Delta]\tau \leq^{\forall} [\Delta']\tau' \text{ iff } \mathrm{dom}(\Delta) \subseteq \mathrm{dom}(\Delta')$$
$$\text{and } \Delta'(x) \leq \rho(\Delta(x)) \text{ for all } x \in \mathrm{dom}(\Delta)$$
$$\text{and } \rho(\tau) \leq \tau', \text{ for some substitution } \rho$$

$$\text{(VAR-}\Pi\text{)} \quad \frac{}{\Pi \Vdash \hat{\mathbf{x}} : [\Delta]\tau}\ \Pi(\hat{\mathbf{x}}) = [\Delta]\tau$$

$$\text{(VAR-}\Delta\text{)} \quad \frac{}{\Pi \Vdash x : [x : \tau]\tau}$$

$$\text{(ABS)} \quad \frac{\Pi \Vdash e : [\Delta, x : \tau]\tau'}{\Pi \Vdash \lambda x.\, e : [\Delta]\tau \to \tau'}$$

$$\text{(APP)} \quad \frac{\Pi \Vdash e_1 : [\Delta]\tau \to \tau' \quad \Pi \Vdash e_2 : [\Delta]\tau}{\Pi \Vdash e_1\, e_2 : [\Delta]\tau'}$$

$$\text{(LET)} \quad \frac{\Pi \Vdash e_1 : [\Delta_1]\tau_1 \quad \Pi, \hat{\mathbf{x}} : [\Delta_1]\tau_1 \Vdash e_2 : [\Delta_2]\tau_2}{\Pi \Vdash \text{let } \hat{\mathbf{x}} = e_1 \text{ in } e_2 : [\Delta_1 \sqcap \Delta_2]\tau_2}$$

$$\text{(TRUE)} \quad \frac{}{\Pi \Vdash \text{true} : [\,]\text{bool}}$$

$$\text{(FALSE)} \quad \frac{}{\Pi \Vdash \text{false} : [\,]\text{bool}}$$

$$\text{(IF)} \quad \frac{\Pi \Vdash e_1 : [\Delta]\text{bool} \quad \Pi \Vdash e_2 : [\Delta]\tau \quad \Pi \Vdash e_3 : [\Delta]\tau}{\Pi \Vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : [\Delta]\tau}$$

$$\text{(CONS)} \quad \frac{\Pi \Vdash e_1 : [\Delta]\tau_1 \quad \ldots \quad \Pi \Vdash e_n : [\Delta]\tau_n}{\Pi \Vdash \{\ell_1 = e_1, \ldots, \ell_n = e_n\} : [\Delta]\{\ell_1 : \tau_1, \ldots, \ell_n : \tau_n\}}$$

$$\text{(PROJ)} \quad \frac{\Pi \Vdash e : [\Delta]\{\ell : \tau\}}{\Pi \Vdash e.\ell : [\Delta]\tau}$$

$$\text{(SUB)} \quad \frac{\Pi \Vdash e : [\Delta]\tau}{\Pi \Vdash e : [\Delta']\tau'}\ [\Delta]\tau \leq^{\forall} [\Delta']\tau'$$

Figure 4: Reformulated typing rules

### 3.2 Type Variables in MLsub

The reformulated (SUB) rule uses the relation $\leq^{\forall}$, where $[\Delta]\tau \leq^{\forall} [\Delta']\tau'$ iff there is some substitution $\rho$ making $\rho(\tau)$ a subtype of $\tau'$, and $\rho(\Delta(x))$ a supertype of $\Delta'(x)$ (for $x \in \mathrm{dom}\,\Delta$). This relation thus combines the two forms of polymorphism in MLsub: subtyping and the instantiation of type variables.

We write $\mathsf{inst}([\Delta]\tau)$ for the *instances* of $[\Delta]\tau$, that is, the set of $[\Delta']\tau'$ such that $[\Delta]\tau \leq^{\forall} [\Delta']\tau'$. The instances of a typing scheme describe the ways it can be used, using any combination of subtyping and instantiation.

This subsumption relation is simpler than those of previous work. Pottier's relation $\leq^{\forall}$ requires that, for every $\rho'$, there exists some $\rho$ making $\rho(\tau)$ a subtype of $\rho'(\tau')$ (and conversely for $\Delta, \Delta'$). This more complex condition seems more general, but it is a consequence of the definition of $\mathcal{T}$ (in particular, the treatment of type variables as indeterminates rather than quantification over ground types) that in the current setting they coincide.

The presence of subtyping causes type variables to behave differently in MLsub as compared to ML. We say two typing schemes $[\Delta_1]\tau_1$ and $[\Delta_2]\tau_2$ are *equivalent* when each subsumes the other. In ML two typing schemes are equivalent iff they are syntactically equal up to renamings of type variables. Equivalence in MLsub is more subtle, since it is possible in MLsub to have two equivalent typing schemes with different numbers of type variables.

An example is the function *choose*, which takes two (curried) arguments and randomly returns one or the other. In MLsub, it can be given either of the following two typing schemes:

$$[\,]\alpha \to \alpha \to \alpha$$
$$[\,]\beta \to \gamma \to (\beta \sqcup \gamma)$$

The first corresponds to the usual ML type scheme $\forall\alpha.\alpha \to \alpha \to \alpha$, while the second reflects the behaviour of subtyping: the function takes two arguments, possibly of different types, and returns something at the upper bound of those types. Superficially, the second appears more general, and indeed

$$[\,]\beta \to \gamma \to (\beta \sqcup \gamma) \leq^{\forall} [\,]\alpha \to \alpha \to \alpha$$

since we can take $\beta = \gamma = \alpha$. Less obviously, we also have that

$$[\,]\alpha \to \alpha \to \alpha \leq^{\forall} [\,]\beta \to \gamma \to (\beta \sqcup \gamma)$$

To show this, we need to instantiate $\alpha$ so that $\alpha \to \alpha \to \alpha$ becomes a subtype of $\beta \to \gamma \to (\beta \sqcup \gamma)$, and taking $\alpha = \beta \sqcup \gamma$ suffices.

The typing scheme $[\,]\alpha \to \alpha \to \alpha$ describes a function type, accepting two inputs and producing one output. In ML, since all are labelled $\alpha$, each of the three values involved must be of the same type. On the contrary, the subtyping in MLsub means that the two inputs need not be of the same type: for any given choice of $\alpha$, subtyping allows the two inputs to be given at any two subtypes of $\alpha$. Because MLsub tracks data flow (as described in the introduction), it only requires a constraint between inputs and outputs: the negative uses of $\alpha$ (the two inputs) must both be subtypes of the positive uses of $\alpha$ (the output). More succinctly, in MLsub, *type variables label data flow*, a fact exploited in the automaton construction of Section 6.4.

### 3.3 Polar Types

Although our syntax includes all combinations of the lattice operators, the structure of programs does not allow the lattice operations $\sqcup$ and $\sqcap$ to appear arbitrarily. If a program chooses randomly to produce either an output of type $\tau_1$ or one of type $\tau_2$, the actual output type is $\tau_1 \sqcup \tau_2$. Similarly, if a program uses an input in a context where a $\tau_1$ is required and again in a context where a $\tau_2$ is, then the actual input type is $\tau_1 \sqcap \tau_2$. Generally, $\sqcup$ only arises when describing outputs, while $\sqcap$ only arises when describing inputs. In a similar vein, the least type $\bot$ appears only on outputs (of non-terminating programs), while the greatest type $\top$ appears only on inputs (an unused input). Thus, we carve out two subsets of $\mathcal{T}$: *positive* types $\tau^+$ (which describe outputs) and *negative* types $\tau^-$ (which describe inputs), defined by the following syntax (in which $\alpha$ is required to be covariant and guarded in $\mu\alpha.\tau^+$ and $\mu\alpha.\tau^-$):

$$\begin{aligned}\tau^+ &::= \text{bool} \mid \tau_1^- \to \tau_2^+ \mid \{\ell_1 : \tau_1^+, \ldots, \ell_n : \tau_n^+\} \mid \\ &\quad\ \alpha \mid \tau_1^+ \sqcup \tau_2^+ \mid \bot \mid \mu\alpha.\tau^+ \\ \tau^- &::= \text{bool} \mid \tau_1^+ \to \tau_2^- \mid \{\ell_1 : \tau_1^-, \ldots, \ell_n : \tau_n^-\} \mid \\ &\quad\ \alpha \mid \tau_1^- \sqcap \tau_2^- \mid \top \mid \mu\alpha.\tau^-\end{aligned}$$

A *polar typing scheme* is a typing scheme $[\Delta^-]\tau^+$ where $\Delta^-(x)$ is a negative type for $x \in \mathrm{dom}\,\Delta^-$ and $\tau^+$ is a positive type, while $\Pi$ is said to be polar if $\Pi(\hat{\mathbf{x}})$ is polar for all $\hat{\mathbf{x}} \in \mathrm{dom}\,\Pi$.

Polar typing schemes are a very restricted subset of the possible typing schemes. However, the principality result of the next section shows that this subset is large enough to perform type inference: every expression typeable with any typing scheme has a *principal* typing scheme which is polar. For this reason, the concrete algorithms of Section 5 manipulate only polar types and typing schemes.

## 4. Type Inference for MLsub

With our reformulated typing rules, the type inference problem is to find a *principal typing scheme* $[\Delta_\mathrm{p}]\tau_\mathrm{p}$ given inputs $\Pi, e$ such that

$$\Pi \Vdash e : [\Delta]\tau \text{ iff } [\Delta_\mathrm{p}]\tau_\mathrm{p} \leq^{\forall} [\Delta]\tau$$

An advantage of the reformulated rules is that the inputs and outputs of type inference are more clearly delineated, in contrast to

Milner's Algorithm $\mathcal{W}$ which takes $\Gamma, e$ and produces substitution with which to modify $\Gamma$ and a type $\tau$.

If $e$ is closed and has principal typing scheme $[]\tau_{\mathrm{p}}$, then this implies (using Theorem 6 and the definition of $\leq^{\forall}$) that:

$$\vdash e : \tau \text{ iff } \rho(\tau_{\mathrm{p}}) \leq \tau \text{ for some substitution } \rho$$

So, we must construct a function $P(\Pi; e)$, which for any polar typing environment $\Pi$ and expression $e$ produces a principal typing scheme (assuming $e$ is typeable).

This function is defined by induction on $e$, and is trivial when $e$ is a variable:

$$P(\Pi; \hat{\mathbf{x}}) = \Pi(\hat{\mathbf{x}})$$
$$P(\Pi; x) = [x : \alpha]\alpha$$

Let-bindings are only slightly more complicated:

$$\begin{aligned} P(\Pi; \text{let } \hat{\mathbf{x}} = e_1 \text{ in } e_2) &= [\Delta_1 \sqcap \Delta_2]\tau_2 \\ &\text{where } [\Delta_1]\tau_1 = P(\Pi; e_1) \\ &\text{and } [\Delta_2]\tau_2 = P(\Pi, \hat{\mathbf{x}} : [\Delta_1]\tau_1) \end{aligned}$$

Inferring types for introduction forms is also straightforward. To infer typing schemes for $\lambda$-bindings, we infer a typing scheme $[\Delta]\tau$ for the body, and then extract the argument's type from $\Delta$. To this end, we write $\Delta_x$ for $\Delta$ with $x$ removed from the domain, and extend the syntax $\Delta(x)$ by using the convention $\Delta(x) = \top$ for $x \notin \operatorname{dom}\Delta$, so that functions that do not use their argument are given types $\top \to \tau$.

$$\begin{aligned} P(\Pi; \text{true}) &= []\text{bool} \\ P(\Pi; \text{false}) &= []\text{bool} \\ P(\Pi; \lambda x.e) &= [\Delta_x](\Delta(x) \to \tau) \\ &\text{where } [\Delta]\tau = P(\Pi; e) \\ P(\Pi; \{\ell_1 = e_1, \ldots, \ell_n = e_n\}) &= \left[\prod_i \Delta_i\right] \{\ell_1 : \tau_1, \ldots, \ell_n : \tau_n\} \\ &\text{where } [\Delta_i]\tau_i = P(\Pi; e_i) \end{aligned}$$

The difficult case of type inference is inferring types for elimination forms: application, projection, and if. This mirrors Milner's Algorithm $\mathcal{W}$. Most cases of inference construct a principal typing scheme for an expression by gluing together the principal typing schemes of its subexpressions, but in the case of an application $e_1\, e_2$ it may be necessary to perform a substitution[3] on the principal types of $e_1$ and $e_2$. For instance, if $e_1 = \lambda x.x$ with principal typing scheme $[]\alpha \to \alpha$ and $e_2 = \text{true}$, it is necessary to substitute bool for $\alpha$ to type the application.

Specifically, we take $[\Delta_1]\tau_1$ and $[\Delta_2]\tau_2$ to be the principal typing schemes of $e_1$ and $e_2$ in the environment $\Gamma$. Not all instances of $[\Delta_1]\tau_1$ and $[\Delta_2]\tau_2$ result in $e_1\, e_2$ being typeable: in the above example, the instance $[]\bot \to \bot$ of $[\Delta_1]\tau_1$ leaves the expression untypeable. Only those instances allowing a subsequent use of the (APP) rule are permitted.

All type variables in a typing scheme are implicitly universally quantified, so without loss of generality we assume that $[\Delta_1]\tau_1$ and $[\Delta_2]\tau_2$ have no type variables in common. We can thus use a single substitution $\rho$ to assign types to the type variables in $[\Delta_1]\tau_1$ and $[\Delta_2]\tau_2$. In order for $e_1\, e_2$ to be typeable, $\rho$ must satisfy:

$$\rho(\tau_1) \leq \rho(\tau_2) \to \rho(\alpha)$$

for some fresh variable $\alpha$.

[3] This corresponds to Algorithm $\mathcal{W}$ finding a "generic instance", which involves removing $\forall$ quantifiers as well as substituting. With typing schemes, substitution suffices as we have no $\forall$ quantifiers to remove.

$$\begin{aligned} \mathsf{sub}_{\mathcal{U}}(\text{bool} = \text{bool}) &= \emptyset \\ \mathsf{sub}_{\mathcal{U}}(\tau_1 \to \tau_2 = \tau_3 \to \tau_4) &= \{\tau_1 = \tau_3, \tau_2 = \tau_4\} \\ \mathsf{sub}_{\mathcal{U}}(\{f\} = \{g\}) &= \{f(\ell) = g(\ell) \mid \ell \in D\} \\ &\text{where } D = \operatorname{dom} f = \operatorname{dom} g \end{aligned}$$

$$\begin{aligned} \mathcal{U}(\emptyset) &= [] \\ \mathcal{U}(\alpha = \alpha, C) &= \mathcal{U}(C) \\ \mathcal{U}(\alpha = \tau, C) &= \mathcal{U}(\theta_{\alpha=\tau} C) \circ \theta_{\alpha=\tau} \quad \alpha \notin \mathrm{ftv}(\tau) \\ \mathcal{U}(\tau = \alpha, C) &= \mathcal{U}(\theta_{\alpha=\tau} C) \circ \theta_{\alpha=\tau} \quad \alpha \notin \mathrm{ftv}(\tau) \\ \mathcal{U}(c, C) &= \mathcal{U}(\mathsf{sub}_{\mathcal{U}}(c), C) \end{aligned}$$

Figure 5: The classical unification algorithm (for finite terms)

By the definition of $\leq^{\forall}$, the instances $\mathsf{inst}([\Delta]\tau)$ of $[\Delta]\tau$ are:

$$\uparrow \{\rho([\Delta]\tau) \mid \rho \text{ a substitution}\}$$

where $\uparrow$ is upwards-closure under the subtyping order. We write $\mathsf{inst}([\Delta]\tau \mid \tau_1 \leq \tau_2)$ for the instances under a constraint, that is:

$$\uparrow \{\rho([\Delta]\tau) \mid \rho(\tau_1) \leq \rho(\tau_2)\}$$

The set of possible typing schemes for $e_1\, e_2$ is:

$$\uparrow \{\rho([\Delta_1 \sqcap \Delta_2]\alpha) \mid \rho(\tau_1) \leq \rho(\tau_2) \to \rho(\alpha)\}$$

To infer a type for $e_1\, e_2$ we must find a typing scheme $[\Delta]\tau$ whose instances are the above. If we can do this, we can define $P(\Pi; e_1\, e_2)$ as follows:

$$\begin{aligned} P(\Pi; e_1\, e_2) &= [\Delta]\tau \\ &\text{where } \mathsf{inst}([\Delta]\tau) = \mathsf{inst}([\Delta_1 \sqcap \Delta_2]\alpha \mid \tau_1 \leq \tau_2 \to \alpha) \\ &\text{and } [\Delta_1]\tau_1 = P(\Pi; e_1),\ [\Delta_2]\tau_2 = P(\Pi; e_2) \end{aligned}$$

The same reasoning can be used for projections and conditionals:

$$\begin{aligned} &P(\Pi;\, e.\ell) = [\Delta]\tau \\ &\text{where } \mathsf{inst}([\Delta]\tau) = \mathsf{inst}([\Delta_1]\alpha \mid \tau_1 \leq \{\ell : \alpha\}) \\ &\quad \text{and } [\Delta_1]\tau_1 = P(\Pi; e) \end{aligned}$$

$$\begin{aligned} &P(\Pi; \text{if } e_1 \text{ then } e_2 \text{ else } e_3) = [\Delta](\tau) \\ &\text{where } \mathsf{inst}([\Delta]\tau) = \mathsf{inst}([\Delta_1 \sqcap \Delta_2 \sqcap \Delta_3](\tau_2 \sqcup \tau_3) \mid \tau_1 \leq \text{bool}) \\ &\quad \text{and } [\Delta_1]\tau_1 = P(\Pi; e_1) \\ &\qquad\ [\Delta_2]\tau_2 = P(\Pi; e_2) \\ &\qquad\ [\Delta_3]\tau_3 = P(\Pi; e_3) \end{aligned}$$

So, to compute principal types, we must find a way of constructing $[\Delta]\tau$ such that

$$\mathsf{inst}([\Delta]\tau) = \mathsf{inst}([\Delta']\tau' \mid C)$$

where $C$ is some set of *constraints*. In general, the generated constraints are always of the form $\tau^+ \leq \tau^-$, which intuitively represents an attempt to use the output of one expression (of type $\tau^+$) in a particular context (requiring type $\tau^-$).

To construct such $[\Delta]\tau$, we introduce *biunification*, a variant of standard unification (as used to infer types in ML) that can handle subtyping constraints of the form $\tau^+ \leq \tau^-$.

### 4.1 Unification in ML

Type inference for ML uses unification to eliminate constraints. Figure 5 shows Martelli and Montanari's unification algorithm [15].

Unification decomposes constraints into a set of *atomic constraints*, which constrain a type variable to equal a term. An atomic

$$\mathsf{sub}_{\mathcal{U}^\mu}(\mu\alpha.\tau_1 = \tau_2) = \{\tau_1[\mu\alpha.\tau_1] = \tau_2\}$$
$$\mathsf{sub}_{\mathcal{U}^\mu}(\tau_1 = \mu\alpha.\tau_2) = \{\tau_1 = \tau_2[\mu\alpha.\tau_2]\}$$
$$\mathsf{sub}_{\mathcal{U}^\mu}(c) = \mathsf{sub}_{\mathcal{U}}(c)$$

$$\mathcal{U}^\mu(H;\ \emptyset) = [\,]$$
$$\mathcal{U}^\mu(H;\ \tau_1 = \tau_2, C) = \mathcal{U}^\mu(C) \quad \tau_1 = \tau_2 \in H$$
$$\mathcal{U}^\mu(H;\ \alpha = \alpha, C) = \mathcal{U}^\mu(H;\ C)$$
$$\mathcal{U}^\mu(H;\ \alpha = \tau, C) = \mathcal{U}^\mu(\theta_{\alpha=\tau}H;\ \theta_{\alpha=\tau}C) \circ \theta_{\alpha=\tau}$$
$$\mathcal{U}^\mu(H;\ \tau = \alpha, C) = \mathcal{U}^\mu(\theta_{\alpha=\tau}H;\ \theta_{\alpha=\tau}C) \circ \theta_{\alpha=\tau}$$
$$\mathcal{U}^\mu(H;\ c, C) = \mathcal{U}^\mu(H, c;\ \mathsf{sub}_{\mathcal{U}^\mu}(c), C)$$

Figure 6: The unification algorithm with recursive types

constraint $\alpha = \tau$ is eliminated using the substitution $\theta_{\alpha=\tau}$ which replaces all uses of $\alpha$ with $\tau$, an operation classically termed "variable elimination". When we extend this to subtyping constraints, elimination will not always remove a variable, so in this work we prefer the term *atomic constraint elimination*. Atomic constraint elimination is trivial in plain unification where we define, for all $\tau$ and $\alpha$ not free in $\tau$:

$$\theta_{\alpha=\tau} = [\tau/\alpha]$$

$\theta_{\alpha=\tau}$ fails if $\alpha$ is free in $\tau$ (the *occurs check*).

In MLsub, the presence of subtyping and recursive types make atomic constraint elimination more complex.

### 4.2 Extending Unification with Recursive Types

First, we extend the ML unification algorithm with support for recursive types $\mu\alpha.\tau$, as supported by ML variants such as OCaml (Figure 6). To ensure termination, the recursive unification algorithm $\mathcal{U}^\mu$ uses an extra argument $H$ to record previously-visited constraints. When processing a constraint $c$ of the form $\mu\alpha.\tau_1 = \tau_2$, the recursive type $\mu\alpha.\tau_1$ is unrolled and $c$ is added to $H$ so that it can be ignored if encountered again.

Unlike plain unification, this algorithm uses atomic constraint elimination $\theta_{\alpha=\tau}$ even when $\alpha$ is free in $\tau$ and the occurs check fails. We extend the definition of $\theta_{\alpha=\tau}$ to introduce a recursive type in this case:

$$\theta_{\alpha=\tau} = \begin{cases} [\tau/\alpha] & \text{if } \alpha \text{ not free in } \tau \\ [\mu\alpha.\tau/\alpha] & \text{otherwise} \end{cases}$$

Real implementations of this algorithm replace $H$ with a single mutable table $T$, which makes the algorithm more efficient. We use this trick in our automata-based implementation in Section 5.6.

### 4.3 Extending Unification with Subtyping

Unification cannot be used directly for MLsub, since unification uses equality rather than subtyping constraints. In particular, unification treats constraints symmetrically and thus loses information about the direction of data flow, which is vital for subtyping. Also, our careful distinction between input and output is not observed by unification, so the substitution it produces may not respect the syntactic restrictions we place on typing schemes about where positive and negative types may occur.

So, we define a variant of this algorithm which respects the direction of data flow (in particular, treats upper and lower bounds differently), and preserves our syntactic restrictions. To this end, we use *bisubstitutions* instead of substitutions, which we define as substitutions which may assign a different type to positive and negative uses of a type variable. We write $[\tau^+/\alpha^+, \tau^-/\alpha^-]$ for

$$\mathsf{sub}_{\mathcal{B}}(\tau_1^- \to \tau_1^+ \le \tau_2^+ \to \tau_2^-) = \{\tau_2^+ \le \tau_1^-, \tau_1^+ \le \tau_2^-\}$$
$$\mathsf{sub}_{\mathcal{B}}(\mathsf{bool} \le \mathsf{bool}) = \{\}$$
$$\mathsf{sub}_{\mathcal{B}}(\{f\} \le \{g\}) = \{f(\ell) \le g(\ell) \mid \ell \in \mathrm{dom}\, g\} \quad \text{where } \mathrm{dom}\, g \subseteq \mathrm{dom}\, f$$
$$\mathsf{sub}_{\mathcal{B}}(\mu\alpha.\tau^+ \le \tau^-) = \{\tau^+[\mu\alpha.\tau^+/\alpha] \le \tau^-\}$$
$$\mathsf{sub}_{\mathcal{B}}(\tau^+ \le \mu\alpha.\tau^-) = \{\tau^+ \le \tau^-[\mu\alpha.\tau^-/\alpha]\}$$
$$\mathsf{sub}_{\mathcal{B}}(\tau_1^+ \sqcup \tau_2^+ \le \tau^-) = \{\tau_1^+ \le \tau^-, \tau_2^+ \le \tau^-\}$$
$$\mathsf{sub}_{\mathcal{B}}(\tau^+ \le \tau_1^- \sqcap \tau_2^-) = \{\tau^+ \le \tau_1^-, \tau^+ \le \tau_2^-\}$$
$$\mathsf{sub}_{\mathcal{B}}(\bot \le \tau^-) = \{\}$$
$$\mathsf{sub}_{\mathcal{B}}(\tau^+ \le \top) = \{\}$$

$$\mathcal{B}(H;\ \emptyset) = []$$
$$\mathcal{B}(H;\ \tau_1 \le \tau_2, C) = \mathcal{B}(H;\ C) \quad \tau_1 \le \tau_2 \in H$$
$$\mathcal{B}(H;\ \alpha \le \alpha, C) = \mathcal{B}(H;\ C)$$
$$\mathcal{B}(H;\ \alpha \le \tau, C) = \mathcal{B}(\theta_{\alpha\le\tau}H;\ \theta_{\alpha\le\tau}\, C) \circ \theta_{\alpha\le\tau}$$
$$\mathcal{B}(H;\ \tau \le \alpha, C) = \mathcal{B}(\theta_{\tau\le\alpha}H;\ \theta_{\tau\le\alpha}\, C) \circ \theta_{\tau\le\alpha}$$
$$\mathcal{B}(H;\ c, C) = \mathcal{B}(H, c;\ \mathsf{sub}_{\mathcal{B}}(c), C)$$

Figure 7: The biunification algorithm

a bisubstitution which replaces positive occurrences of $\alpha$ with $\tau^+$ and negative occurrences of $\alpha$ with $\tau^-$. We also write $[\tau^-/\alpha^-]$ for a bisubstitution which leaves positive occurrences of $\alpha$ unchanged. That is, $[\tau^-/\alpha^-]$ has the same effect as $[\alpha/\alpha^+, \tau^-/\alpha^-]$, and similarly for $[\tau^+/\alpha^+]$. Bisubstitutions always make positive-for-positive and negative-for-negative replacements, so we may apply them without violating our syntactic restrictions.

The *biunification* algorithm takes as input a set of constraints of the form $\tau^+ \le \tau^-$ and produces a bisubstitution. The algorithm is shown in Figure 7, and it is instructive to compare it to unification with recursive types.

Note in particular the straightforward definition of $\mathsf{sub}_{\mathcal{B}}$. By ensuring that constraints are always of the form $\tau^+ \le \tau^-$, decomposing constraints is made easy. The syntactic restrictions on polar types ensure that hard cases like $\tau_1 \sqcap \tau_2 \le \tau_3$ do not arise.

Biunification treats recursive types in the same way as $\mathcal{U}^\mu$. Constraints involving $\sqcup$ on the left of $\le$ are decomposed into several constraints, since $\tau_1 \sqcup \tau_2 \le \tau_3$ iff $\tau_1 \le \tau_3$ and $\tau_2 \le \tau_3$ by the least-upper-bound property of $\sqcup$. The situation is similar for $\sqcap$-constraints on the right. The benefit of the syntactic restrictions on polar types is that we need not handle the awkward cases of $\sqcup$ on the right nor $\sqcap$ on the left.

Atomic constraint elimination $\theta_{\alpha=\tau}$ has been split in two: we use bisubstitutions $\theta_{\alpha\le\tau}$ to eliminate upper bounds and $\theta_{\tau\le\alpha}$ to eliminate lower bounds. Their definition is trickier than for plain unification, and is explained below.

### 4.4 Atomic Constraint Elimination in Biunification

Atomic constraints can be eliminated by using the $\sqcup$ and $\sqcap$ operators on types. Consider the following type with an atomic constraint, which expresses the type of a function which takes arguments of a type $\tau_0$ and returns them unchanged (assuming for now that $\alpha$ is not free in $\tau_0$):

$$\alpha \to \alpha \text{ where } \alpha \le \tau_0$$

That is, the function has type $\alpha \to \alpha$, whenever $\alpha$ is a subtype of $\tau_0$. Since $\alpha \sqcap \tau_0$ is always a subtype of $\tau_0$, we could equivalently describe the function as having the following type:

$$(\alpha \sqcap \tau_0) \to (\alpha \sqcap \tau_0)$$

However, this violates our syntactic restrictions, since $\sqcap$ appears in a positive position. The essential insight that lets us eliminate atomic constraints in polar types is that the upper bound $\tau_0$ on the type variable $\alpha$ only affects negative uses of $\alpha$: the function argument must be a $\tau_0$, but the result need not be used as a $\tau_0$. So, we encode the constraint on $\alpha$ by replacing only the negative occurrences of $\alpha$ with the type $\tau_0 \sqcap \alpha$, using the bisubstitution $[\tau_0 \sqcap \alpha/\alpha^-]$, giving the *unconstrained* type:

$$(\alpha \sqcap \tau_0) \to \alpha$$

Although it may not seem so at first, this is equivalent to the original constrained type. The original type can be instantiated with any type $\tau$ such that $\tau \leq \tau_0$, while the unconstrained type can be instantiated with any type $\tau$. For any instance of one, the other has an instance which gives a subtype.

For any instance $\alpha \mapsto \tau$ (where $\tau \leq \tau_0$) of the original type, we choose an instance $\alpha \mapsto \tau$ of the unconstrained type, giving:

$$(\tau \sqcap \tau_0) \to \tau \ \leq \ \tau \to \tau$$

which is true since $\tau \sqcap \tau_0 = \tau$. Similarly, for any instance $\alpha \mapsto \tau$ of the unconstrained type, we choose an instance $\alpha \mapsto \tau \sqcap \tau_0$, giving:

$$(\tau \sqcap \tau_0) \to (\tau \sqcap \tau_0) \leq (\tau \sqcap \tau_0) \to \tau$$

which holds since $\tau \sqcap \tau_0 \leq \tau$. This construction works in general: when $\alpha$ is not free in $\tau_0$, we eliminate the atomic constraint $\alpha \leq \tau_0$ by applying the bisubstitution $[\tau_0 \sqcap \alpha/\alpha^-]$.

As before, we handle the cases where $\alpha$ is free in $\tau$ by introducing a recursive type. To ensure that the substitution only affects uses of $\alpha$ of the correct polarity, we perform the recursion on a fresh type variable $\beta$, giving:

$$\theta_{\alpha \leq \tau} = \begin{cases} [\alpha \sqcap \tau/\alpha^-] & \text{if } \alpha \text{ not free in } \tau \\ [(\mu\beta.\alpha \sqcap [\beta/\alpha^-]\tau)/\alpha^-] & \text{otherwise } (\beta \text{ fresh}) \end{cases}$$

$\theta_{\tau \leq \alpha}$ is defined dually, as:

$$\theta_{\tau \leq \alpha} = \begin{cases} [\alpha \sqcup \tau/\alpha^+] & \text{if } \alpha \text{ not free in } \tau \\ [(\mu\beta.\alpha \sqcup [\beta/\alpha^+]\tau)/\alpha^+] & \text{otherwise } (\beta \text{ fresh}) \end{cases}$$

### 4.5 Correctness of Biunification

To prove the correctness of biunification, we must show that its output bisubstitutions are *stable* and *solve* the input constraints.

In standard unification, applying any substitution to a type yields an instance of the type. Indeed, this is how "instance" is usually defined. However, applying an arbitrary bisubstitution to a typing scheme does not always yield an instance of the typing scheme, since bisubstitutions may replace positive and negative occurrences of a variable with incompatible types. For instance, clearly

$$[\,]\alpha \to \alpha \not\leq^\forall [\,]\top \to \text{bool}$$

and yet the latter can be formed from the former by applying the bisubstitution $[\top/\alpha^-, \text{bool}/\alpha^+]$. In fact, $\xi([\Delta^-]\tau^+)$ is in general an instance of $[\Delta^-]\tau^+$ only when $\xi(\alpha^-) \leq \xi(\alpha^+)$ for all type variables $\alpha$.

So, we define a *stable bisubstitution* as any $\xi$ such that $\xi(\alpha^-) \leq \xi(\alpha^+)$ for all type variables $\alpha$, and $\xi^2 = \xi$. The idempotence condition is imposed for technical convenience, as it allows us to simplify the definition of solving. This is not peculiar to subtyping: careful accounts of standard unification prove the same about the output of unification algorithms.

We say that a bisubstitution $\xi$ *solves* a set of constraints $C$ when, for any polar typing scheme $[\Delta^-]\tau^+$,

$$\mathsf{inst}([\Delta^-]\tau^+ \mid C) = \mathsf{inst}(\xi([\Delta^-]\tau^+))$$

As we saw above, $[\tau_0 \sqcap \alpha/\alpha^-]$ solves $\alpha \leq \tau_0$ when $\alpha$ is not free in $\tau_0$. In general, we have:

**Lemma 7.** *For any atomic constraint $c$, $\theta_c$ is stable and solves $c$.*

**Theorem 8.** *For any set of constraints $C$ of the form $\tau^+ \leq \tau^-$, if the algorithm $\mathcal{B}(\emptyset; C)$ returns $\xi$ then $\xi$ is stable and solves $C$.*

Producing bisubstitutions that solve a given set of constraints suffices to discharge the side-conditions in Section 4, allowing type inference to produce principal types [6].

However, $\mathcal{B}(H; C)$ does not always produce an output bisubstitution. It can fail, upon encountering a constraint such as $\text{bool} \leq \top \to \bot$ for which $\mathsf{sub}_\mathcal{B}$ is not defined. In such cases, we report a type error, which is justified by the following theorem:

**Theorem 9.** *If the algorithm $\mathcal{B}(\emptyset; C)$ fails, then $C$ is unsatisfiable.*

Of course, these results leave open a third possibility, that $\mathcal{B}(H; C)$ does not terminate. We leave discussion of termination and complexity to Section 5.7, since the arguments are much simpler when expressed in terms of the next section's automata.

## 5. Simplifying Typing Schemes with Automata

This section introduces a compact representation of typing schemes as finite automata, as well as an efficient biunification algorithm and techniques for simplifying typing schemes to simpler, equivalent ones. This material is not strictly necessary to infer principal types, but without it, the inferred types quickly grow large.

Indeed, a common criticism of type systems with subtyping and inference is that they tend to produce large and unwieldy types. Systems based on constrained types are particularly prone to creating large types: without simplification, many such systems produce overly large type schemes [11].

Various algorithms for simplifying types appear in the literature, since many type systems with subtyping are impractical without them. Informal connections to standard constructions in automata have been made before: the removal of unreachable constraints by Eifrig et al. [8] corresponds to removing dead states from a finite automaton, while Pottier noted that the *canonisation* and *minimisation* algorithms he describes [18] are analogous to determinisation of an NFA and minimisation of the resulting DFA.

Our *representation theorem* (Section 5.3) generalises these techniques, by putting the connection between automata and subtyping on a sound formal footing: two automata represent the same type iff they accept equal languages. This allows us to use standard algorithms like removal of dead states, determinisation and minimisation, but also admits more advanced simplification algorithms such as those given by Ilie, Navarro and Yu [12] which operate directly on NFAs. Due to the representation theorem, any algorithm which simplifies standard finite automata also simplifies types.

### 5.1 Type Automata

Types are represented by *type automata*, a slight variant on standard non-deterministic finite automata. A type automaton consists of:

- a finite set $Q$ of *states*
- a designated *start state* $q_0 \in Q$
- a *transition table* $\delta \subseteq Q \times \Sigma_F \times Q$.
- for each state $q$, a *polarity* ($+$ or $-$) and a set of *head constructors* $H(q)$.

The alphabet $\Sigma_F$ contains one symbol for each field of each type constructor. $\Sigma_F$ contains the symbols $d, r$ (representing the domain and range of function types), and a symbol $\ell_i$ for each record label.

The transition table $\delta$ of a type automaton has the condition that $d$-transitions only connect states of differing polarity, and all other transitions connect states of the same polarity. This means that the polarity of every state reachable from $q$ is determined by that of $q$, and is in some sense redundant. However, we find it more clear to be explicit about state polarities.

We use $f$ to quantify over transition labels $d, r$, and write a transition from $q$ to $q'$ labelled by $f$ as $q \xrightarrow{f} q'$ (this notation should not be confused with $\to$ as used for function types).

Standard NFAs label each state as either "accepting" or "non-accepting". Instead, a type automaton labels each state $q$ with a set of *head constructors* $H(q)$, drawn from the set consisting of type variables, the symbol $\langle\to\rangle$ (representing function types), $\langle b\rangle$ (representing the boolean type) and $\langle L\rangle$ for any set $L$ of record labels $\ell$ (representing record types). We maintain the invariant that $H(q)$ does not contain both $\langle L\rangle$ and $\langle L'\rangle$ for distinct sets of labels $L, L'$.

### 5.2 Constructing Type Automata

The simplest way to construct a type automaton from a positive or negative type is to first construct a type automaton containing extra transitions labelled by $\epsilon$, and then to remove these transitions in a second pass. This mirrors the standard algorithm for constructing a nondeterministic finite automaton from a regular expression.

The grammar of Section 2.1 gives the syntax of positive and negative types. For the purposes of this section, we assume that all $\mu$ operators in a type bind distinct type variables. The set of *subterms* of a type is the set of types used in its construction, including the type itself, but omitting type variables bound by $\mu$. For instance, the subterms of $\mu\alpha.\beta \sqcup (\text{bool} \to \alpha)$ are:

$$\{\, \mu\alpha.\beta \sqcup (\text{bool} \to \alpha),\ \beta \sqcup (\text{bool} \to \alpha),\ \beta,\ \text{bool} \to \alpha,\ \text{bool} \,\}$$

The type automaton for a positive type $\tau_0$ has one state for every subterm $\tau$ of $\tau_0$, written $q(\tau)$ by slight abuse of notation. The start state $q_0$ is $q(\tau_0)$. The polarity of $q(\tau)$ matches that of $\tau$, $H(q(\alpha)) = \{\alpha\}$, $H(q(\text{bool})) = \{\langle b\rangle\}$, $H(q(\tau_1 \to \tau_2)) = \{\langle\to\rangle\}$, and $H(q(\{f\})) = \{\langle \text{dom } f\rangle\}$. In all other cases, $H(q(\tau)) = \emptyset$.

We further abuse notation by defining $q(\alpha) = q(\mu\alpha.\tau_1)$ for every type variable $\alpha$ bound by a recursive type $\mu\alpha.\tau_1$.

The type automaton has the following transition table:

$$\begin{aligned}
q(\tau_1 \sqcup \tau_2) &\xrightarrow{\epsilon} q(\tau_1) & q(\tau_1 \sqcup \tau_2) &\xrightarrow{\epsilon} q(\tau_2) \\
q(\tau_1 \sqcap \tau_2) &\xrightarrow{\epsilon} q(\tau_1) & q(\tau_1 \sqcap \tau_2) &\xrightarrow{\epsilon} q(\tau_2) \\
q(\tau_1 \to \tau_2) &\xrightarrow{d} q(\tau_1) & q(\tau_1 \to \tau_2) &\xrightarrow{r} q(\tau_2) \\
q(\mu\alpha.\tau_1) &\xrightarrow{\epsilon} q(\tau_1)
\end{aligned}$$

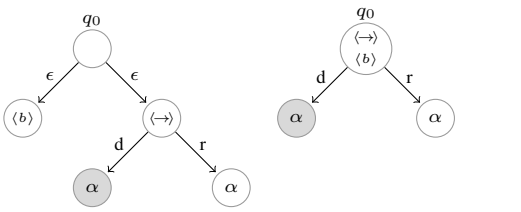
Finally, we remove $\epsilon$-transitions using the standard algorithm. For a state $q$, we define $E(q)$ as the set of states reachable from $q$ by following zero or more $\epsilon$-transitions, and then set:

$$H(q) = \bigcup_{q' \in E(q)} H(q')$$

$$q \xrightarrow{f} q' \text{ iff } \exists q'' \in E(q).\ q'' \xrightarrow{f} q'$$

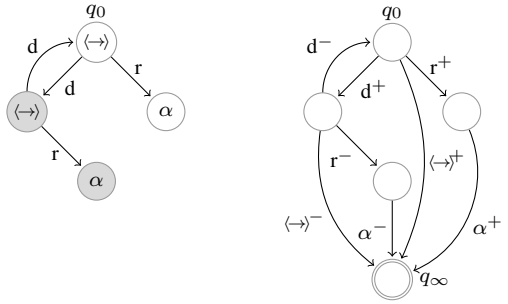
To maintain the invariant that $H(q)$ does not contain distinct $\langle L\rangle$, $\langle L'\rangle$, we replace multiple record symbols with their union or intersection in negative and positive states respectively.

The syntactic restrictions on positive and negative types are important in ensuring that this process generates a valid type automaton. The positivity and negativity constraints on function types and the covariance condition on $\mu$ types ensure that $d$ transitions connect states of unlike polarity and $r$ transitions connect ones of like polarity, while the guardedness condition on $\mu$ types avoids cycles of $\epsilon$-transitions.




(a) With $\epsilon$-transitions (b) $\epsilon$-transitions removed

Figure 8: Automata construction with $\epsilon$-transitions




(a) Type automaton (b) NFA

Figure 9: Type automaton and NFA for $\mu\beta.\,(\beta \to \alpha) \to \alpha$.

As an example, the type automaton for the positive type $(\alpha \to \alpha) \sqcup \text{bool}$ is shown in Figure 8, before and after removal of $\epsilon$-transitions, with negative states drawn shaded and $H(q)$ drawn inside the state $q$. An example with recursive types appears in Figure 9a.

The construction above does not preserve every detail of the syntax of types. For instance, $\alpha \sqcup (\bot \sqcup \text{bool})$ and $\text{bool} \sqcup \alpha$ are converted to identical automata. As we see in the next section, this causes no ambiguity: equivalent automata arise only from equivalent types.

### 5.3 Simplifying Type Automata

Type automata can be converted to standard NFAs, by labelling each transition with the polarity of the state from which it originates, and converting $H(q)$ to a set of transitions (also labelled with polarity) to a new accepting state $q_\infty$. For example, Figure 9a shows a type automaton, while Figure 9b shows the same automaton after conversion to an NFA. (The full details of this conversion can be found in the first author's thesis [6, Ch. 7]).

With this conversion of type automata to standard NFAs, we now talk about the language accepted by a type automaton. Thus, we say that the automaton in Figure 9a accepts the following language:

$$(d^+d^-)^*(d^+(r^-\alpha^- + \langle\to\rangle^-) + r^+\alpha^+ + \langle\to\rangle^+)$$

Now, we state our representation theorem:

**Theorem 10.** *For types $\tau_1, \tau_2$ (both positive or both negative) represented by automata accepting languages $L_1, L_2$, we have $\tau_1 \equiv \tau_2$ if and only if $L_1 = L_2$.*

Any algorithm for simplifying finite automata may therefore be used to simplify type automata. The canonical choice is to convert to a deterministic finite automaton using the subset construction

and then simplify using Hopcroft's algorithm. In fact, this describes the *canonisation* and *minimisation* algorithms of Pottier [18], although in his presentation they were individually proved correct for types rather than relying on a general representation theorem.

### 5.4 Simplifying Typing Schemes

The simplification techniques of the previous section preserve equivalence as types, and can therefore be used to simplify typing schemes by simplifying their component types. However, typing schemes admit more simplifications than those which preserve equivalence of their component types.

Recall that two typing schemes may be equivalent even though they have different numbers of type variables, as long as those type variables describe the same data flow (Section 3.2). For instance, the following two typing schemes for $choose$ are equivalent:

$$[\,]\alpha \to \alpha \to \alpha$$
$$[\,]\beta \to \gamma \to (\beta \sqcup \gamma)$$

More generally, consider a typing scheme

$$\phi(\tau_1^-, \ldots \tau_n^-; \tau_1^+, \ldots \tau_m^+)$$

parameterised by $n$ negative types and $m$ positive ones, where $\phi$ contains no type variables except via $\tau_i^-, \tau_j^+$. For instance,

$$\phi_{choose}(\tau_1^-, \tau_2^-;\ \tau_1^+) = [\,]\,\tau_1^- \to \tau_2^- \to \tau_1^+$$

An *instantiation* of $\phi$ is a typing scheme formed from $\phi$ where $\tau_i^-$ $(1 \le i \le n)$ is given as $\bigsqcap S$ for some finite set $S$ of type variables (taking $\bigsqcap \emptyset$ to be $\top$), while $\tau_j^+$ $(1 \le j \le m)$ is similarly given as $\bigsqcup S$ (taking $\bigsqcup \emptyset$ to be $\bot$). In this manner, the two typing schemes for $choose$ are $\phi_{choose}(\alpha, \alpha; \alpha)$ and $\phi_{choose}(\beta, \gamma; \beta \sqcup \gamma)$.

For a given instantiation of $\phi$, we say that there is a *flow edge* $i \rightsquigarrow j$ if $\tau_i^-$ and $\tau_j^+$ have a type variable in common. The two typing schemes for $choose$ both have the same two flow edges: $1 \rightsquigarrow 1$ and $2 \rightsquigarrow 1$. This is an instance of a general principle:

**Lemma 11.** *Any two instantiations of a typing scheme*

$$\phi(\tau_1^-, \ldots \tau_n^-; \tau_1^+, \ldots \tau_m^+)$$

*with the same flow edges are equivalent.*

Flow edges capture the notion of data flow from the introduction, and provide a means of simplifying typing schemes: relabelling type variables while preserving flow edges preserves typing scheme equivalence, so we can represent typing schemes with fewer type variables. Next, we show how this simplification can be implemented with *scheme automata*, which represent flow edges directly.

### 5.5 Scheme Automata

Typing schemes are represented by *scheme automata*. Scheme automata have a *domain* $D$, which is a set of $\lambda$-bound variables corresponding to $\mathrm{dom}(\Delta)$ in a typing scheme $[\Delta]\tau$. A scheme automaton with domain $D$ consists of:

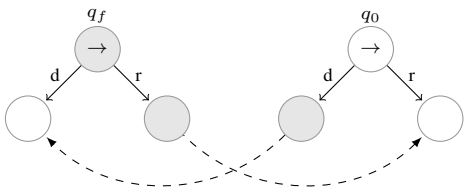
- a finite set $Q$ of *states*
- (multiple) designated *start states* $q_0$ and $q_x$ for $x \in D$
- a transition table $\delta \subseteq Q \times \{d, r\} \times Q$
- for each state $q$, a *polarity* ($+$ or $-$) and a set of *head constructors* $H(q)$
- a set of *flow edges* $q^- \rightsquigarrow q^+$

Instead of a single start state, a scheme automaton has distinguished negative states $q_x$ for each $x \in D$ (corresponding to $\Delta(x)$ in a typing scheme $[\Delta]\tau$), and a positive state $q_0$ (corresponding to $\tau$ in a typing scheme $[\Delta]\tau$). The transition table $\delta$ retains the polarity condition of type automata.

In a scheme automaton, the head constructors $H(q)$ of a set $q$ are similar to those of a type automaton, but do not contain type variables. Instead, type variables are represented implicitly as flow edges.

To construct a scheme automaton from a typing scheme $[\Delta]\tau$, we start by constructing type automata for $\tau$ and $\Delta(x)$ (for each $x$). The states and transitions of the scheme automata are given by the disjoint union of those of the type automata, and the start states $q_0$ and $q_x$ are the start states of each type automaton.

For each type variable $\alpha$, we add flow edges $q^- \rightsquigarrow q^+$ for every $q^-, q^+$ such that $\alpha \in H(q^-), \alpha \in H(q^+)$. Having represented type variables as flow edges in this way, we remove all type variables from the head constructors $H(q)$. Below, we see a scheme automaton for the typing scheme $[f : \alpha \to \beta]\alpha \to \beta$, with flow edges $q^- \rightsquigarrow q^+$ drawn as dashed lines.




Two typing schemes may be represented by the same scheme automaton, despite syntactic differences. For instance, the two typing schemes for $choose$ above are represented identically.

We simplify typing schemes by representing them as scheme automata, and then rendering these as a tuple of type automata using as few type variables as possible. To do this, we must find an assignment of type variables to states that exactly generates the flow edges of the scheme automaton. Doing this optimally is the NP-complete problem of *biclique decomposition*, but we have found that simple heuristics give good results.

The two typing schemes for $choose$ above are represented by the same scheme automaton, since they have the same flow edges. To display this to the user, the scheme automaton is converted back to type automata and then to a syntactic typing scheme. Our conversion heuristic chooses the $[\,]\,\alpha \to \alpha \to \alpha$ representation, as it involves introducing fewest type variables.

### 5.6 Biunification of Automata

As well as enabling simplification, type and scheme automata are useful to efficiently implement type inference. In fact, our implementation uses them throughout: types and typing schemes are always represented as automata, and syntactic types are only used for input and output. To this end, we implemented the biunification algorithm of Section 4.3 in terms of scheme automata, which turns out to be simpler than the description in terms of syntactic types. Our implementation is imperative, and destructively updates automata.

The algorithm $\mathcal{B}(H; C)$ takes as input a set of hypotheses $H$, a set of constraints $C$, and produces as output a bisubstitution. In the imperative implementation, the hypotheses and constraints are both represented by pairs of states of a scheme automaton. The output bisubstitution is not represented explicitly: instead, the scheme automaton is mutated. Since the scheme automaton represents $H$ and $C$, mutating it simultaneously applies a bisubstitution to $H$, $C$ and the output.

For states $q_1, q_2$ of a given scheme automaton (both positive or both negative) we define the operation $\mathrm{merge}(q_1, q_2)$ which adds the transitions and flow edges of $q_2$ to $q_1$. More specifically, $\mathrm{merge}(q_1, q_2)$ modifies the automaton by adding, for all $q_1', q_2', f$,

**function** biunify($q^+, q^-$)
  **if** $(q^+, q^-) \notin T$ **then**
    $T \leftarrow T \cup \{(q^+, q^-)\}$
    **if** $\exists x \in H(q^+), y \in H(q^-).\ x \not\leq y$ **then**
      `fail`
    **for** $q'^+$ where $q^- \rightsquigarrow q'^+$ **do**
      merge($q'^+, q^+$)
    **for** $q'^-$ where $q'^- \rightsquigarrow q^+$ **do**
      merge($q'^-, q^-$)
    **for** $q'^-, q'^+$ where $q^+ \xrightarrow{d} q'^-, q^- \xrightarrow{d} q'^+$ **do**
      biunify($q'^+, q'^-$)
    **for** $q'^-, q'^+, f \neq d$ where $q^+ \xrightarrow{r} q'^+, q^- \xrightarrow{f} q'^-$ **do**
      biunify($q'^+, q'^-$)

Figure 10: Biunification algorithm for scheme automata

- transitions $q_1 \xrightarrow{f} q_1'$ when $q_2 \xrightarrow{f} q_2'$
- flow edges $q_1' \rightsquigarrow q_2$ when $q_1' \rightsquigarrow q_1$ (if $q_1, q_2$ positive)
- flow edges $q_1 \rightsquigarrow q_2'$ when $q_2 \rightsquigarrow q_2'$ (if $q_1, q_2$ negative)

and by adding the head constructors $H(q_2)$ to $H(q_1)$.

If the state $q_1$ represents an occurrence of a type variable $\alpha$ (i.e. $\alpha \in H(q_1)$ in the type automaton representation, or $q_1$ has a flow edge in the scheme automaton representation), and $q_2$ represents some type $\tau$, then the effect of $\mathrm{merge}(q_1, q_2)$ is to perform the atomic constraint elimination $\theta_{\alpha \leq \tau}$ or $\theta_{\tau \leq \alpha}$ as per Section 4.4. The addition of new transitions by merge may introduce cycles, which corresponds to introducing $\mu$-types during atomic constraint elimination.

As earlier, the biunification algorithm operates on constraints $\tau^+ \leq \tau^-$, here represented as pairs of states $(q_1^+, q_2^-)$. Instead of threading the argument $H$ through all recursive calls, we use a single mutable table $T$ of previously-seen inputs. The biunification algorithm for automata is shown in Figure 10.

### 5.7 Termination and Complexity

Each recursive call to biunify either terminates immediately or adds a pair of states not previously present to $T$. Since there are finitely many such states, this must terminate.

Suppose the input automaton has $n$ states and $m$ transitions. There are $O(n^2)$ possible pairs of states, and therefore $O(n^2)$ possible recursive calls to biunify. Since biunify iterates over pairs of transitions, the total amount of work is bounded by their number, so the worst-case complexity is $O((n+m)^2)$.

However, this complexity is difficult to attain. In particular, in the common case where the automaton is a tree (that is, there are no states reachable by two routes nor cycles), each state can be visited only once, giving $O(n+m)$ complexity.

In practice, the algorithm is sufficiently performant that our online demo retypes the input program on each keystroke, without noticeable delay.

## 6. Related Work and Discussion

This work extends a long thread of research into *non-structural subtyping*, some of which we summarize below. This form of subtyping is called *non-structural* since it admits subtyping relationships between type constructors of different arities (e.g. $\bot \leq \bot \to \bot$). The alternative, *structural subtyping*, has been well-studied and is easier to deal with (see e.g. Henglein and Rehof [9]), but cannot express top or bottom types, or subtyping between records with different sets of fields as used in object-oriented programming.

### 6.1 Definition of the Subtyping Relation

The prototypical example of non-structural subtyping with recursive types is the system of Amadio and Cardelli. Their seminal paper [1] describes *ground types* $t$ as regular trees over the following signature:

$$t \ ::= \ \bot \mid \top \mid t \to t$$

Regular trees are trees with at most finitely many distinct subtrees, which are exactly the trees representable as finite closed terms using a fixpoint operator $\mu$.

These are equipped with a partial order $\leq$ which is bounded by $\bot$ and $\top$, where $\to$ is covariant in the range and contravariant in the domain. This partial order forms a lattice, which satisfies the equations of Fig. 3.

Amadio and Cardelli show that the subtyping order for ground types is decidable (while their original algorithm is inefficient, Kozen, Palsberg and Schwartzbach [14] later showed how to decide subtyping efficiently).

### 6.2 Type Variables and Polymorphism

However, combining Amadio and Cardelli's subtyping with polymorphism proved problematic. The most straightforward way of adding type variables to their system is to quantify over ground types $t$. We take *type terms* $\tau$ as given by the following syntax:

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

A *ground substitution* $\rho$ maps type variables to ground types $t$, and can be used to interpret a type term $\tau$ as a ground type $\rho(\tau)$. The subtyping relation on type terms is then defined by quantification over ground substitutions:

$$\tau_1 \leq \tau_2 \quad \text{iff} \quad \forall \rho.\ \rho(\tau_1) \leq \rho(\tau_2)$$

Handling type variables by quantifying over ground types is a common approach [10, 16, 21, 22], but turns out to have issues. As discussed in Section 1.2, defining type variables by quantifying over ground types allows reasoning by case analysis over type variables, which makes the polymorphic subtyping relation complex and difficult to decide, as well as non-extensible. These issues are resolved by adding type variables directly to the definition of types rather than quantifying over ground types.

Castagna and Xu [4] also introduce type variables by quantifying over ground types. However, they avoid the above issues by applying a *convexity* condition to the definition of ground types, which recovers in their system the intuitive idea that type variables are related only to themselves.

The complexities of this approach to type variables come up in the well-studied problem of *entailment*. An entailment problem asks whether some constraints imply others, where the constraints are often restricted to not include $\sqcup$, $\sqcap$ or $\mu$. For instance, the example (E) of Section 1.2 can be written as an entailment problem:

$$\alpha \leq \alpha \to \bot \ \models \ (\bot \to \top) \to \bot \leq \alpha$$

In the standard formulation where type variables quantify over ground types, entailment is not known to be decidable despite much research. The full first-order theory was shown undecidable by Su et al. [21], and sound but incomplete algorithms for entailment and subtyping have been published [18, 22]. We conjecture that some of these algorithms may be sound and complete for MLsub, since their incompleteness seems to arise from avoiding the sort of case-analysis necessary to prove (E).

### 6.3 Ad-hoc Polymorphism and Overloading

This work integrates ML-style parametric polymorphism with subtyping but does not consider any form *ad-hoc polymorphism*, in which functions are allowed to operate with different behaviours at

different types. For example, ad-hoc polymorphism is necessary to allow an *overloaded* operator $+$, which is simultaneously addition of integers (of type int $\to$ int $\to$ int) and concatenation of lists (of type $\forall\alpha.\text{list}[\alpha] \to \text{list}[\alpha] \to \text{list}[\alpha]$).

Kaes [13] presented a system extending ML, combining parametric polymorphic, subtyping and overloading, and a similar system lifting some of its restrictions was later given by Smith [20]. However, both systems infer types with an explicit collection of constraints, rather than just a type term. With their language $\text{ML}_{\leq}$, Bourdoncle and Merz [3] present a language combining ML with object-oriented programming, supporting CLOS-style multimethod dispatch, which also infers constrained types.

### 6.4 Automata and Simplification

Representing types as automata is a standard technique, which has been used to tackle subtyping, entailment and simplification. Kozen, Palsberg and Schwartzbach [14] gave an algorithm based on automata for efficiently deciding subtyping between ground types of Amadio and Cardelli's system. Henglein and Rehof [10] reduced the problem of NFA universality to entailment in the Amadio-Cardelli system, thus proving it PSPACE-hard. In fact, our representation theorem began life as an attempt to find a sort of converse to Henglein and Rehof's result. Niehren and Priesnitz [17] studied also entailment using a form of finite automata (their *cap-set automata*, explained in detail in Priesnitz's thesis [19]). Eifrig et al. [8], Trifonov and Smith [22], Pottier [18] all use some form of automata for simplifying types.

## 7. Conclusion

We have presented MLsub, a minimal language subsuming the Hindley-Milner calculus with let-polymorphism integrated seamlessly with subtyping. An implementation is available online.

We improve on previous work in this area by offering an algorithm to infer compact types (syntactic terms, rather than a term and a possibly-large set of constraints), and proving that these are principal.

We provide a general means of simplifying types and typing schemes: instead of specifying a particular simplification algorithm, our representation theorem shows that anything from the literature of finite automata and regular languages can be used.

Languages incorporating ML-style type inference often have interesting type system features not present in the core calculus, such as advanced module systems, GADTs, typeclasses, etc. We are interested in seeing which of these constructs can be implemented (or even continue to make sense) in the context of MLsub.

***Acknowledgements*** Thanks to Leo White for discussions about MLsub, particularly on the relationship between the ML-style and reformulated rules, to Derek Dreyer for helpful comments about the presentation of this paper, and to the anonymous referees. This work was funded by Trinity College, Cambridge, and OCaml Labs.

## References

[1] R. M. Amadio and L. Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15 (4), 1993.

[2] H. Bekič. Definable operations in general algebras, and the theory of automata and flowcharts. In C. Jones, editor, *Programming Languages and Their Definition*, volume 177 of *Lecture Notes in Computer Science*, pages 30–55. Springer Berlin Heidelberg, 1984.

[3] F. Bourdoncle and S. Merz. Type checking higher-order polymorphic multi-methods. In *POPL '97: 24th ACM Symposium on Principles of Programming Languages*, pages 302–315, Paris, Jan. 1997. ACM.

[4] G. Castagna and Z. Xu. Set-theoretic foundation of parametric polymorphism and subtyping. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 94–106. ACM, 2011.

[5] B. A. Davey and H. A. Priestley. *Introduction to lattices and order*. Cambridge University Press, 2002.

[6] S. Dolan. *Algebraic Subtyping*. PhD thesis, University of Cambridge, 2016. To appear.

[7] J. Eifrig, S. Smith, and V. Trifonov. Type inference for recursively constrained types and its application to OOP. *Electronic Notes in Theoretical Computer Science*, 1:132–153, 1995. MFPS XI, Mathematical Foundations of Programming Semantics, Eleventh Annual Conference.

[8] 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*, OOPSLA '95, pages 169–184. ACM, 1995.

[9] F. Henglein and J. Rehof. The complexity of subtype entailment for simple types. In *Logic in Computer Science, 1997. LICS'97. Proceedings., 12th Annual IEEE Symposium on*, pages 352–361. IEEE, 1997.

[10] F. Henglein and J. Rehof. Constraint automata and the complexity of recursive subtype entailment. In *25th International Colloquium on Automata, Languages and Programming*, ICALP '98, pages 616–627. Springer Berlin Heidelberg, 1998.

[11] 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*, POPL '95, pages 176–185. ACM, 1995.

[12] L. Ilie, G. Navarro, and S. Yu. On NFA reductions. In J. Karhumäki, H. Maurer, G. Păun, and G. Rozenberg, editors, *Theory Is Forever*, volume 3113 of *Lecture Notes in Computer Science*, pages 112–124. Springer Berlin Heidelberg, 2004.

[13] S. Kaes. Type inference in the presence of overloading, subtyping and recursive types. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming*, LFP '92, pages 193–204. ACM, 1992.

[14] D. Kozen, J. Palsberg, and M. I. Schwartzbach. Efficient recursive subtyping. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '93, pages 419–428. ACM, 1993.

[15] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4 (2):258–282, 1982.

[16] J. Niehren and T. Priesnitz. Entailment of non-structural subtype constraints. In P. Thiagarajan and R. Yap, editors, *Advances in Computing Science — ASIAN'99*, volume 1742 of *Lecture Notes in Computer Science*, pages 251–265. Springer Berlin Heidelberg, 1999.

[17] J. Niehren and T. Priesnitz. Non-structural subtype entailment in automata theory. In N. Kobayashi and B. C. Pierce, editors, *Theoretical Aspects of Computer Software*, volume 2215 of *Lecture Notes in Computer Science*, pages 360–384. Springer Berlin Heidelberg, 2001.

[18] F. Pottier. *Type inference in the presence of subtyping: from theory to practice*. PhD thesis, Université Paris 7, 1998.

[19] T. Priesnitz. *Subtype Satisfiability and Entailment*. PhD thesis, Saarland University, 2004.

[20] G. S. Smith. Principal type schemes for functional programs with overloading and subtyping. *Science of Computer Programming*, 23 (2):197–226, 1994.

[21] Z. Su, A. Aiken, J. Niehren, T. Priesnitz, and R. Treinen. The first-order theory of subtyping constraints. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 203–216. ACM, 2002.

[22] V. Trifonov and S. Smith. Subtyping constrained types. In R. Cousot and D. A. Schmidt, editors, *Static Analysis*, volume 1145 of *Lecture Notes in Computer Science*, pages 349–365. Springer Berlin Heidelberg, 1996.

[23] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and computation*, 115(1):38–94, 1994.