# Algebraic Subtyping

Stephen Dolan

University of Cambridge
Computer Laboratory

Trinity College

September 2016

This dissertation is submitted for
the degree of Doctor of Philosophy

# Declaration

This dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration except where specifically indicated in the text.

Neither it nor any substantial part has already been submitted nor is being concurrently submitted for a degree or diploma or other qualification at the University of Cambridge or any other institution.

This dissertation does not exceed the regulation length of 60,000 words, including tables and footnotes.

# Algebraic Subtyping

Stephen Dolan

## Summary

Type inference gives programmers the benefit of static, compile-time type checking without the cost of manually specifying types, and has long been a standard feature of functional programming languages. However, it has proven difficult to integrate type inference with subtyping, since the unification engine at the core of classical type inference accepts only equations, not subtyping constraints.

This thesis presents a type system combining ML-style parametric polymorphism and subtyping, with type inference, principal types, and decidable type subsumption. Type inference is based on *biunification*, an analogue of unification that works with subtyping constraints.

Making this possible are several contributions, beginning with the notion of an "extensible" type system, in which an open world of types is assumed, so that no typeable program becomes untypeable by the addition of new types to the language. While previous formulations of subtyping fail to be extensible, this thesis shows that adopting a more algebraic approach can remedy this. Using such an approach, this thesis develops the theory of biunification, shows how it is used to infer types, and shows how it can be efficiently implemented, exploiting deep connections between the algebra of regular languages and polymorphic subtyping.

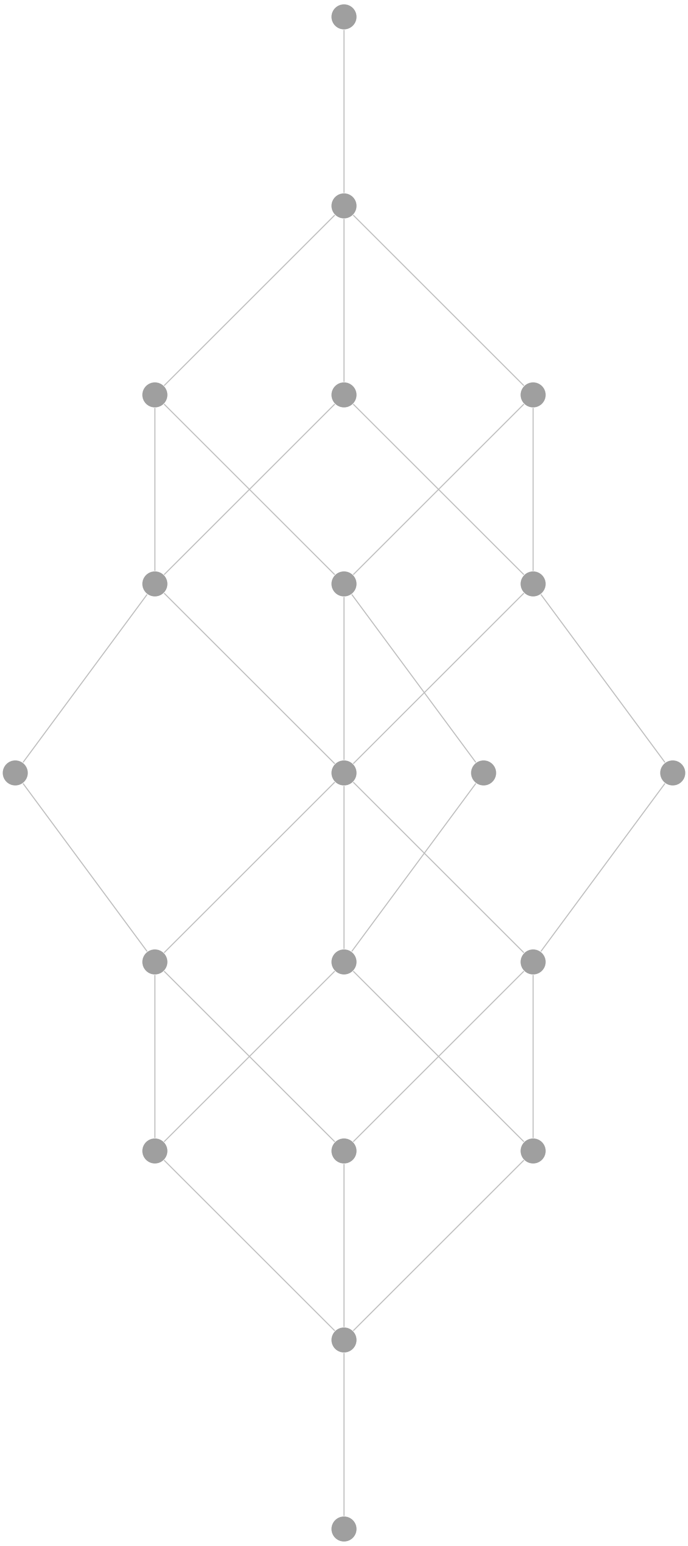
# Acknowledgements

First, I thank my supervisor Alan Mycroft, for his valuable advice, gentle guidance, and general willingness to engage in deep technical conversation about every bizarre tangent I wandered off along.

I am grateful for many conversations with Leo White, which improved both the ideas in this thesis and their presentation. I also thank Daan Leijen for helpful comments and vital encouragement, and all of the denizens of the Computer Lab in Cambridge who provided such a stimulating atmosphere, particularly my office-mate Raphaël Proust. I thank Trinity College and OCaml Labs for funding this work.

Especially, I thank Louise, for her unflagging love, support and companionship, which got me through darker times and made brighter ones that much brighter.

# Contents



# 1 Introduction

*The most important thing in the programming language is the name. A language will not succeed without a good name. I have recently invented a very good name and now I am looking for a suitable language.*

—Donald Knuth (attributed)

Programmers working in functional languages such as ML and its progeny have long enjoyed powerful *type inference*, being able to write

$$\mathsf{f}\ x\ y = \{\texttt{foo} : x,\ \texttt{bar} : y\}$$

and have their compiler statically determine that the function $\mathsf{f}$ returns a record, whose `foo` and `bar` fields have whatever types $\mathsf{f}$'s first and second arguments have. Symbolically,

$$\mathsf{f}\ :\ \forall \alpha\, \beta.\ \alpha \to \beta \to \{\texttt{foo} : \alpha,\ \texttt{bar} : \beta\}$$

For even longer, programmers have reasoned about programs (with perhaps less enjoyment) using *preconditions* and *postconditions*. In particular, the output of one function can be passed as the input to a second provided that the postcondition provided by the first satisfies the precondition required by the second. For instance, the output of $\mathsf{f}$ can be passed to the function $g$, written

$$g\ x = x.\texttt{foo}$$

since $g$ requires that only that its input have a field called `foo`, while $\mathsf{f}$ provides a output having such a field.

This reasoning is inherently asymmetrical. The function $\mathsf{f}$ may produce an output with a stronger postcondition than $g$ requires: in the example, $\mathsf{f}$'s output also has a field called `bar`. This is at odds with conventional approaches to type inference, which boil the program down to a collection of type equations, equating the types provided as outputs with those required as inputs. Such an approach leaves no room for the condition provided by $\mathsf{f}$ to be stronger than that required by $g$.

## 1.1 Types and data flow

By requiring that the type provided when a value is produced (e.g. the result of $\mathsf{f}$) be exactly equal to the type required when the value is consumed (e.g. the argument to g), conventional approaches to type inference ignore the direction of data flow. This makes heavy use of language features such as records quite cumbersome, and an object-oriented style of programming almost impossible. However, the downsides of ignoring data flow are not limited to features such as records and objects, and even cause some simple functions to be less useful than one might expect.

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:

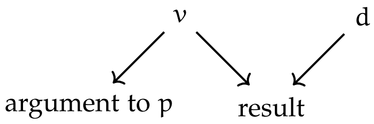
$$\texttt{select}\ p\ v\ d = \texttt{if}\ (p\ v)\ \texttt{then}\ v\ \texttt{else}\ d \tag{1.1}$$

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

$$\forall\alpha.\ (\alpha \to \texttt{bool}) \to \alpha \to \alpha \to \alpha \tag{1.2}$$

This type 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.

Type systems which support *subtyping* 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 type system described in this thesis supports subtyping. It ascribes to the `select` function the following type:

$$\forall\alpha\,\beta.\ (\alpha \to \texttt{bool}) \to \alpha \to \beta \to (\alpha \sqcup \beta) \tag{1.3}$$

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 both $\alpha$ and $\beta$.

## 1.2 Contributions

I contribute a type system, in the vein of ML, with:

- Decidable type inference and let-polymorphism
- Principal types
- Decidable polymorphic subsumption
- Compact inferred types

- An efficient[1] inference algorithm
- A rich type system, including records and variant types.

The system accepts all core ML programs. They are even typeable with the same type schemes, although my system may also give them more general types (such as the `select` function above).

Type inference with subtyping has traditionally been considered difficult, not least because the usual approach based on unification does not work: Hindley-Milner type inference relies on extracting equations from the programming and substituting one side for the other, but a subtyping constraint does not justify substitution.

There have been many previous systems that achieved various of the above points, but none that managed them all simultaneously. An account of the relationship to previous work is given in Chapter 10.

## 1.3 Design principles for subtyping

The major difference between the type system of this thesis and previous work lies not in the typing rules, syntax or semantics of programs, but in the collection of types themselves. Rather than just write down a definition of types and subtyping, we first consider some guiding principles for how those types should behave.

### 1.3.1 Extensibility

The first guiding principle is *extensibility*. I say that a type system is *extensible* when no typeable program becomes untypeable merely by the addition of more types to the language. In other words, extensible type systems are those which never rely on the non-existence of certain types: they assume an open world of types, rather than a closed one.

Extensibility is desirable for various reasons. Firstly, programming languages evolve and new features and types are added to subsequent versions of a language. In order to ensure that old programs written in earlier versions of the language continue to function, it is necessary that those old programs did not rely on the lack of the new types.

Even if we decide that our language is perfect and unchanging, it is still impractical for the compiler to know about all types. If the language supports user-defined types and separate compilation, then the compiler will have to compile programs knowing only about an incomplete collection of types.

The concept of extensibility has received little attention. I suspect that this is the case because, most of the time, it appears automatically. Absent subtyping, no special effort is usually needed to guarantee extensibility.

The situation is different when subtyping is introduced. Later in this chapter (Section 1.4), we see how previous approaches have broken extensibility with seemingly-natural definitions of subtyping. In Chapter 3, I demonstrate a different approach to subtyping which preserves extensibility.

[1] Efficient in the same sense as ML: tends to run fast, but with some effort pathological cases can be constructed

### 1.3.2 Algebra before syntax

The second guiding principle is to place more emphasis on the algebra than on the syntax of types. The algebra of types consists of the operations defined on types and relations between them: which types are subtypes of or equal to other types, whether types form a lattice, and so on. Knowing the algebra of types is quite useful when writing programs, as it allows the programmer to reason about the types in use being equivalent or at least compatible. Accordingly, this thesis will focus first on deciding what the right algebra of types is, and only secondarily on finding syntax to represent it.

More pithily, the idea is to

*Find the* **simplest** *algebra of types, and* **some** *syntax for them*

This is to be contrasted with the standard approach, which starts with the *simplest* syntax for types, and proceeds to build *some* algebra out of them. If minimality of syntax is the primary goal, it is all too easy to make the algebraic structure unusably complicated. We see several examples of this accidental complexity below.

One benefit of an algebraic approach is that it makes extensibility easier to attain. The basic tools of syntactic reasoning (definitions by abstract syntax, induction over terms) assume a closed world of terms, and so if extensibility holds it does so by coincidence or hard work[2]. On the other hand, the basic tools of algebra (axiomatic definitions, equational reasoning) are extensible by default.

## 1.4 Failures of extensibility

Next, we walk through a standard approach to subtyping, showing how the principle of extensibility can be quite easily broken by innocent-looking definitions, and how it can be repaired by a careful treatment of the algebra of types. The following is quite informal, and the full exposition is deferred to Chapter 3.

As a running example, we consider a simple functional language with records and a primitive boolean type. Informally, the types of this language are defined by an abstract syntax like:

$$\text{types} = \text{boolean type} \mid \text{function types} \mid \text{record types} \tag{1.4}$$

To support subtyping, the above types are equipped with some partial ordering. Most of the details of this ordering are not relevant at the moment (although we discuss them thoroughly later, in Chapter 3). The only salient point is that function types, record types, and the boolean type are pairwise incomparable: while there are some subtyping rules giving subtyping relationships between, say, two different record types, there are none between record and function types.

Not only are function and record types incomparable, but they also have no common upper or lower bounds. This fact prevents the partial order of types from forming a lattice. In order to perform type inference, it is very useful for types to form a lattice, since then the result type of a conditional expression can be typed as the least upper bound of the result types of its cases.

[2] Both of which the author takes pains to avoid.

So, subtyping systems often use a different definition of types which does indeed form a lattice:

$$\text{types} = \text{boolean type} \mid \text{function types} \mid \text{record types} \mid \bot \mid \top \tag{1.5}$$

The new type $\bot$ is a subtype of all types, and thus uninhabited, containing no values, while $\top$ is a supertype of all types, and thus uninformative, containing all values. The addition of these two types causes the partial order of types to form a lattice (for the usual definitions of function and record subtyping).

However, this lattice turns out to be quite ill-behaved in practice. In the presence of type variables, it is difficult to decide even quite simple subtyping properties [SAN$^+$02, Reh98, Pri04].

Below, we demonstrate that this bad behaviour is a result of its failure to be extensible, and by sticking to the design principles described in Section 1.3 we can avoid it.

### 1.4.1 Vacuous reasoning

Using the lattice defined by (1.5), it is the case for every function type $\tau_f$ and record type $\tau_s$ that

$$\tau_f \sqcap \tau_s \leqslant \text{bool} \tag{1.6}$$

where $\sqcap$ is the lattice meet operator. Thus, the lattice of (1.5) assumes that there are not, and will never be, values which are simultaneously records (have attributes) and functions (may be applied). I note that such values were not originally present, but were later added, to the programming languages C++, Python, and others.

Thus, extensibility demands we count statements like (1.6) as false. In particular, we should have separate, incomparable types $\tau_s \sqcap \tau_f$, $\tau_s \sqcap \text{bool}$, $\tau_f \sqcap \text{bool}$. That they are all currently uninhabited is not reason enough to identify them.

Let's re-examine the original definition (1.4), which defined the set of types to be the disjoint union of the sets of function types, record types, and the boolean type. Writing $+$ for disjoint union, we express this as:

$$\text{types} = \text{boolean type} + \text{function types} + \text{record types} \tag{1.7}$$

Disjoint union is the coproduct in the category of sets, so we have a good excuse to use the symbol $+$.

We introduce subtyping by equipping the set of types with a partial order. Since the coproduct of partial orders is given by disjoint union, the above definition is a good description of the poset of types.

Next, we desire that types form a lattice. The algebraically simplest thing to do is to continue in our current vein: leave (1.7) unchanged, but interpret it as a coproduct of lattices rather than a coproduct of posets or plain sets.

Unfortunately, (1.5) did not do this. Instead of the algebraically simplest definition, (1.5) uses the syntactically simplest definition: taking the disjoint union (coproduct of sets or posets), and adding just enough to make it a lattice (top and bottom elements). This does result in a lattice, but the resulting type system is not extensible and the lattice does not satisfy any useful universal properties (see Section 3.2). In order to extensibly define a lattice, or any other algebraic structure, we should use a lattice coproduct rather than attempting to impose structure on the disjoint union. Furthermore, we should exclude vacuous reasoning by demanding that the lattice be *distributive* (see Section 2.1.5).

So, we conclude that the extensible way to define a subtyping system is to define types as a *coproduct of distributive lattices*.

### 1.4.2 Closed-world polymorphism and free algebras

The above sets out our approach to defining monomorphic types, but the treatment of type variables has been conspicuously absent. In particular, we need to define the subtyping relationship between terms that contain type variables. Using $\to$ to construct function types, and $\alpha, \beta, \ldots$ for type variables, we ask whether

$$\alpha \to \alpha \leqslant \bot \to \top \tag{1.8}$$

In much work, the meaning of such propositions is defined by quantification over the ground (type-variable-free) types. That is, we interpret the above statement as

$$\forall \tau. \quad \tau \to \tau \leqslant \bot \to \top \tag{1.9}$$

This particular statement is easily shown true using the subtyping rule for functions, noting that $\tau \leqslant \top$ and $\bot \leqslant \tau$ for all $\tau$. However, similar questions lead to surprising difficulties. The following example is due to Pottier, and takes place in a lattice of types defined in the style of (1.5) without records or booleans. That is, the only types are function types, $\top$ and $\bot$.

$$(\bot \to \top) \to \bot \ \leqslant\ (\alpha \to \bot) \sqcup \alpha \tag{1.10}$$

Statement (1.10) holds, as can be shown by case analysis on $\alpha$: if $\alpha = \top$, then it holds trivially. Otherwise, $\alpha \leqslant \bot \to \top$ (the largest function type), and so $(\bot \to \top) \to \bot \leqslant \alpha \to \bot$ by contravariance.

However, this example is 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.

This is a separate issue from the one considered above. Even if we construct the lattice of ground types in an extensible fashion (as a coproduct of distributive lattices), we can still observe such behaviour. Instead, the problem is with the treatment of type variables by quantifying over ground types. By quantifying over ground types to interpret statements about type variables, we assume a closed world of types.

This situation has parallels in mainstream algebra. Suppose we work with the ring of two elements $\mathbb{B}$: this is an algebraic structure consisting of two elements $\{0, 1\}$, with addition defined as exclusive-or (an abelian group with identity 0), and multiplication defined as conjunction (a monoid with identity 1, distributing over addition).

It is the case in $\mathbb{B}$ that

$$x^2 = x$$

as is easily checked by case analysis. Indeed, $x^n = x$, from which one deduces that an arbitrary polynomial $\sum_i a_i x^i$ must be equal to $0$, $1$ or $x$.

However, this is not what is usually meant by "polynomial over $\mathbb{B}$". Instead, we form the ring of polynomials in $x$ over $\mathbb{B}$ (denoted $\mathbb{B}[x]$) by adding $x$ as an indeterminate, a fresh element distinct from any element in $\mathbb{B}$. Addition and multiplication are defined in $\mathbb{B}[x]$ by treating $x$ as subject only to the defining axioms of rings. So, in $\mathbb{B}[x]$,

$$x^3 + x^3 + x^2 = x^2$$
$$x^3 + x^3 + x^2 \neq x$$

because $x^3 + x^3 = (1+1)x^3 = 0x^3 = 0$, but $x^2 \neq x$. This interpretation of the variable $x$ is more algebraically well-behaved, as $\mathbb{B}[x]$ satisfies a useful universal property: If we have an arbitrary ring $\mathcal{R}$, a ring homomorphism $\phi : \mathbb{B} \to \mathcal{R}$, and an element $\hat{x} \in \mathcal{R}$, then there exists a unique homomorphism $\theta : \mathbb{B}[x] \to \mathcal{R}$ extending $\phi$ such that $\theta(x) = \hat{x}$.

In other words, $\mathbb{B}[x]$ is extensible, in the sense of Section 1.3.1: if $\mathcal{R}$ is some ring containing all the elements of $\mathbb{B}$ and possibly others (that is, equipped with a homomorphism $\mathbb{B} \to \mathcal{R}$), then any two terms interpreted equally in $\mathbb{B}[x]$ will be interpreted equally in $\mathcal{R}$, for all values of $x \in \mathcal{R}$.

Applying this idea to types is straightforward. We must avoid handling type variables by quantifying over ground types. Rather, define type variables as opaque indeterminates, without assuming they range over concrete ground types. Thus, we modify (1.7) to the following:

$$\text{types} = \text{boolean type} + \text{function types} + \text{record types} + \text{type variables} \quad (1.11)$$

Using this as the definition of types rather than something like (1.5) gives an extensible and algebraically natural lattice of types.

This definition of the subtyping lattice in an algebraically simple rather than syntactically simple manner is the main novelty of this thesis. Of course, there is a lot of work between this definition and a working programming language (giving this document its heft), but much of it proceeds along the same lines as previous work. It is only the extensible and algebraically well-behaved subtyping lattice that allow us to move past the usual sticking points of subtyping such as eliminating constraints, polymorphic subsumption, type scheme simplification, and the like.

## 1.5 Structure of the thesis

**Chapter 2** provides some mathematical background in order theory, category theory, and Kleene algebra. The order theory goes a little further than most treatments of the subtyping order, including describing why we might want the subtyping order to be a *distributive lattice* (Section 2.1.5), and analysing the interactions between subtyping and recursive types (Section 2.1.6), which are more subtle than usually given credit for. Category theory is used to formalise the idea of the "simplest" algebra of types that we sought in Section 1.3.2, while Kleene algebra becomes useful in the representation of types as automata in Chapters 7 and 8.

**Chapters 3 and 4** define the MLsub type system. The major novelty is the design and construction of an extensible lattice of types, which is built up step-by-step in Chapter 3. The concrete syntax, semantics and even typing rules are entirely standard, and appear in Chapter 4 along with a proof of soundness and related properties, as well as an alternative presentation of the type system (Section 4.3) upon which the inference algorithm is based.

**Chapters 5 and 6** together present the algorithm to infer principal types for MLsub. Chapter 5 introduces *polar types*, a syntactic restriction of types, and *biunification*, an analogue of unification for solving subtyping constraints over polar types. Chapter 6 shows that polar types suffice for type inference (that is, while not all types are polar, every typeable expression has a principal type which is), and specifies the inference algorithm based on biunification.

**Chapters 7 and 8** show how types may be represented compactly as automata. Section 7.2 proves the *representation theorem*, that types are equivalent iff their automata accept the same language, which allows standard algorithms of automata theory to operate on types, and Section 7.4 shows how biunification can be efficiently implemented directly on automata. Chapter 8 devises an algorithm to decide *subsumption*, which compares polymorphic type schemes and can be used for e.g. signature matching.

**Chapter 9** discusses some extensions of the minimal MLsub language, by showing how other type system features (such as user-defined types, variance annotations, sums, etc.) can be integrated easily into the framework developed here.

**Chapter 10** reviews related work, and **Chapter 11** presents some future work and concludes.

# 2 Background

*Order is a necessity for everyone, but not everyone understands it in the same way.*

—Fausto Cercignani

The central problem studied in this thesis is *type inference*, the act of examining a program and inferring its type.

When examining a program, we find that its subexpressions place *constraints* on what types we may assign to it. For instance, suppose we are given an application of a function of type $\tau_1 \to \tau_2$ to an argument of type $\tau_3$. The application can be typed only if values of type $\tau_3$ can be passed where a value of type $\tau_1$ is expected. Before the type inference algorithm can give a type to the application, this constraint must be dealt with.

In the classical Hindley-Milner type inference algorithm, the constraint is an equality constraint, with the application being typeable only when $\tau_1 = \tau_3$. Types are defined as first-order terms, and equality constraints are dealt with using unification.

*Subtyping* allows for more interesting constraints, in particular by not requiring symmetry: just because we can pass a $\tau_3$ whenever a $\tau_1$ is expected does not imply the reverse. This models many language features: naively, we might expect a record with more fields to work in any context where one with fewer suffices. However, the classical theory of unification does not apply to subtyping constraints, and so the standard approach to type inference fails.

Lacking unification (or an equivalent), previous work on inference with subtyping has used *constrained types*, in which constraints like $\tau_3 \leqslant \tau_1$ are simply attached to the type. In general, this can lead to unwieldy types, since the set of attached constraints grows with the size of the program [HM95]. Heuristic methods of reducing the size of the constraint set have been devised [Pot01, EST95a], but a general means is elusive not least because of the difficulty of determining whether two constraint sets are equivalent [SAN$^+$02, NP99, Reh98, Pri04]. This and other related work is discussed in Chapter 10.

In this thesis, we take an alternative approach. By using an algebraic approach to define the lattice of types, we construct a type system supporting an analogue of unification (the *biunification* of Chapter 5), allowing subtyping constraints to be eliminated just as equality constraints are in the Hindley-Milner system, and removing the need for constrained types altogether. The result is a system that infers types that are usually as compact as ML's, and where comparison between types is decidable.

This depends crucially on several subtleties of the construction of types, which are explored in Chapter 3. Before then, some mathematical background is needed, on order theory (for the subtyping relation itself), category theory (to formalise the "simplest algebra" of types sought in the introduction), and Kleene algebra (useful for the efficient automaton representation of types of Chapter 7).

What follows is a terse, incomplete and highly selective introduction to those topics, biased heavily towards what is needed to develop this thesis. For a better introduction to order theory, I recommend Davey and Priestley's wonderful *Introduction to Lattices and Order* [DP02], while for category theory I recommend Mac Lane's classic *Categories for the Working Mathematician* [ML78]. For a heady mix of both, as well as the sort of topology we touch on in Section Section 3.4, read Johnstone's *Stone Spaces* [Joh86]. For a thorough introduction to reasoning about fixed points and Kleene algebra, I recommend Backhouse's *Galois connections and fixed point calculus* [Bac02].

## 2.1 Order theory

A *preorder* $\leqslant$ on a set $A$ is a reflexive transitive binary relation on $A$. Any preorder induces an equivalence relation $\equiv$ on $A$ called its *kernel*, defined as $a \equiv b$ iff $a \leqslant b$ and $b \leqslant a$.

A *partial order* is a preorder additionally satisfying *antisymmetry*: if $a \leqslant b$ and $b \leqslant a$, then $a = b$. Equivalently, a partial order is a preorder whose kernel is equality. If $\leqslant$ is any preorder on $A$, then it generates a partial order on the quotient $A/{\equiv}$, that is, the set of equivalence classes of the kernel of $\leqslant$. A *poset* is a set equipped with a partial order.

If $S$ is a subset of some poset $A$, then an *upper bound* of $S$ is an element $a \in A$ such that $s \leqslant a$ for all $s \in S$. The *least upper bound* or *join* of $S$ is that upper bound $b$ of $S$ such that $b \leqslant a$ for every upper bound $a$ of $S$. The join of $S$ need not exist in general, but is unique if it does. We write $\bigsqcup S$ for the join of $S$, abbreviating $\bigsqcup\{a, b\}$ to $a \sqcup b$ and $\bigsqcup \emptyset$ to $\bot$. We also abbreviate $\bigsqcup\{a_i \mid i \in I\}$ as $\bigsqcup_{i \in I} a_i$. The join of the empty set $\bot$ is the least element of the partial order $A$.

The dual of the join is the *greatest lower bound* or *meet*. The meet of $S$ is written as $\bigsqcap S$, binary meets as $\sqcap$ and the meet of the empty set (greatest element) as $\top$.

### 2.1.1 Lattices

A *lattice* is a poset all of whose finite subsets have a meet and a join (we say it "has finite joins" and "has finite meets"). A *join-semilattice* is a poset which has finite joins, while a *meet-semilattice* has finite meets.

Some authors define lattices in terms of the binary join and meet operators $\sqcup$ and $\sqcap$, admitting as lattices orders lacking greatest or least elements, and using the term *bounded lattice* to imply their presence. By defining lattices in terms of finite, possibly-empty meets and joins, we employ the opposite convention: all lattices are bounded.

For example, the collection $\mathcal{P}_f(\mathbb{N})$ of finite subsets of $\mathbb{N}$ ordered by $\subseteq$ is a join-semilattice, with joins given by $\cup$ and least element $\emptyset$. It is not a meet-semilattice despite having all non-empty meets (given by intersections), since it has no greatest element.

A *complete lattice* is a poset which has arbitrary joins and arbitrary meets, while *complete join-semilattices* and *complete meet-semilattices* have one or the other. $\mathcal{P}_f(\mathbb{N})$ is not a complete join-semilattice, as it fails to have arbitrary joins: the union of $\{1\}, \{1,2\}, \{1,2,3\}, \ldots$ is not a finite set.

A subset $D$ of a partial order $A$ is said to be *directed* when any pair of elements of $D$ have an upper bound in $D$. In a join-semilattice, any subset closed under join is directed, but directedness is more general: directed sets are not required to contain *least* upper bounds. For instance, the subset of $\mathcal{P}_f(\mathbb{N})$ consisting of sets of even-numbered cardinality is directed: while the union of two finite sets of numbers both having even cardinality may not have an even number of elements, the union is certainly contained in some set of even cardinality. A *directed-complete partial order* is a poset of which any directed subset has a least upper bound. We say that such a poset *has directed joins*.

**Lemma 1** (Complete = Finite + Directed)**.** *A poset has arbitrary joins iff it has both finite and directed joins.*

*Proof.* Clearly a poset with arbitrary joins has finite and directed joins. For the converse, consider an arbitrary subset $S$. Let $T$ be the set:

$$\left\{ \bigsqcup S' \mid S' \text{ a finite subset of } S \right\}$$

All joins above are finite, so $T$ exists. $T$ is directed, so $\bigsqcup T$ exists and is equal to $\bigsqcup S$. □

This pattern reoccurs frequently: the behaviour of arbitrary subsets can be determined from the behaviour of finite subsets and directed subsets, since every set is the union of the directed family of its finite subsets.

The dual of a directed subset is a *codirected* subset (hence *codirected meet*, *codirected complete partial order*, etc.), and so we note that a poset has arbitrary meets iff it has both finite and codirected meets.

### 2.1.2 Lattices and subtyping

We define subtyping as a partial order on types, where if $s \leqslant t$ for types $s, t$ then all values of type $s$ are also values of type $t$. In order to support type inference, it is very useful to have the types under the subtyping order form a lattice.

Suppose we have a conditional expression of the form `if c then a else b`, where $a$ is an expression of type $s$ and $b$ is an expression of type $t$. Any type $u$ which is an upper bound of $\{s, t\}$ is an acceptable type of the entire expression, since by subtyping both $a$ and $b$ are of type $u$. Principal type inference requires that there be a single best type for the whole expression, so there should be a least such $u$. Thus, inferring types for conditional expressions requires that the subtype order is a join-semilattice.

Conversely, suppose we have a $\lambda$-abstraction which binds the variable $x$ and contains two subexpressions $a$ and $b$. If $a$ requires $x$ to be of type $s$ and $b$ requires $x$ to be of type $t$, then the best type for $x$ is $s \sqcap t$, which is the greatest type satisfying the requirements of both $a$ and $b$.

### 2.1.3 Suborders versus sublattices

A *suborder* of a poset $A$ is a subset of $A$ ordered by the restriction of $A$'s order. A *sublattice* of a lattice is a subset of a lattice closed under finite meets and finite joins (and similarly *sub-(complete lattice)*, *sub-(join semilattice)*, and so on.). The concept of a sublattice must be carefully separated from that of a suborder which happens to form a lattice. To clarify, consider the following three partially ordered sets:

- $\mathcal{P}(\mathbb{N})$, the subsets of $\mathbb{N}$
- $\mathcal{P}_+(\mathbb{N})$, the subsets of $\mathbb{N}$ closed under addition
- $\mathcal{P}_{f\top}(\mathbb{N})$, the subsets of $\mathbb{N}$ which are either finite or the entirety of $\mathbb{N}$

All sets are ordered using $\subseteq$. $\mathcal{P}_+(\mathbb{N})$ and $\mathcal{P}_{f\top}(\mathbb{N})$ are sub-orders of $\mathcal{P}(\mathbb{N})$, since they are partially ordered sets inheriting their order from $\mathcal{P}(\mathbb{N})$.

All three of these are complete lattices. In $\mathcal{P}(\mathbb{N})$, joins are given by union and meet by intersection. $\mathcal{P}_+(\mathbb{N})$ and $\mathcal{P}_{f\top}(\mathbb{N})$ have the same meets, but joins in $\mathcal{P}_+(\mathbb{N})$ are given by the closure of the union under addition, while joins in $\mathcal{P}_{f\top}(\mathbb{N})$ are given by the union only if finite, and $\mathbb{N}$ otherwise.

However, while $\mathcal{P}_+(\mathbb{N})$ is both a lattice and a sub-order of $\mathcal{P}(\mathbb{N})$, it is not a sublattice of $\mathcal{P}(\mathbb{N})$. In $\mathcal{P}_+(\mathbb{N})$, the join of $\{0\}$ and $\{2\}$ is the set of even natural numbers, that is, the closure of $\{0, 2\}$ under addition. Since the join of $\{0\}$ and $\{2\}$ in $\mathcal{P}(\mathbb{N})$ is $\{0, 2\}$, the two orders disagree on lattice structure, so $\mathcal{P}_+(\mathbb{N})$ is not a sublattice of $\mathcal{P}(\mathbb{N})$.

In $\mathcal{P}_{f\top}(\mathbb{N})$, finite joins and finite meets and are given by union and intersection, so $\mathcal{P}_{f\top}(\mathbb{N})$ is in fact a sublattice of $\mathcal{P}(\mathbb{N})$. While $\mathcal{P}_{f\top}(\mathbb{N})$ agrees with $\mathcal{P}(\mathbb{N})$ on arbitrary meets, they disagree on some infinite joins. So, we say that $\mathcal{P}_{f\top}(\mathbb{N})$ is a sub-(complete meet semilattice) but not a sub-(complete join semilattice) nor a sub-(complete lattice) of $\mathcal{P}(\mathbb{N})$.

### 2.1.4 Distributive lattices

A lattice is said to be *distributive* when meets distribute over joins and joins distribute over meets, that is, when the following identities hold:

$$a \sqcap (b \sqcup c) = (a \sqcap b) \sqcup (a \sqcap c)$$
$$a \sqcup (b \sqcap c) = (a \sqcup b) \sqcap (a \sqcup c)$$

Either of these identities implies the other (for specific $a, b, c$ these statements are independent, but if one holds for all $a, b, c$ then so does the other), so we say that the property of being distributive is *self-dual*.

Generalising distributivity to the infinite case is more complex. The distributive identities can be written as follows:

$$a \sqcap \bigsqcup_{b \in S} b = \bigsqcup_{b \in S} (a \sqcap b)$$
$$a \sqcup \bigsqcap_{b \in S} b = \bigsqcap_{b \in S} (a \sqcup b)$$

The distributivity condition above is equivalent to these identities for finite $S$: we say that meet distributes over finite joins, and join distributes over finite meets. This condition generalises in various ways: we might have a lattice in which:

- meets distribute over directed joins, or
- meets distribute over arbitrary joins, or

- joins distribute over codirected meets, or
- joins distribute over arbitrary meets

Unlike the finite case, these conditions are not self-dual. For instance, in the lattice of open sets of a topological space, meets always distribute over arbitrary joins, while joins do not in general distribute over arbitrary meets. An analogue of Lemma 1 holds: meet distributes over arbitrary joins iff meet distributes over finite joins and meet distributes over directed joins.

Still, these distributivity conditions are not fully infinitary, since they involve finite joins or meets distributing over possibly-infinite ones. A lattice is said to be *completely distributive* if it satisfies the following more complex condition, for an arbitrary family $x_{i,j}$ of elements indexed by $i \in I, j \in J(i)$:

$$\bigsqcap_{i \in I} \bigsqcup_{j \in J(i)} x_{i,j} = \bigsqcup_{f \in F} \bigsqcap_{i \in I} x_{i,f(i)}$$

where $f$ ranges over the set $F$ of functions of domain $I$ such that $f(i) \in J(i)$.

Complete distributivity is a self-dual property. However, there exist lattices in which finite meets distribute over arbitrary joins, and finite joins over arbitrary meets, and yet the lattice fails to be completely distributive[1].

### 2.1.5 Distributivity and subtyping

While there is clear intuition behind requiring the subtyping order to form a lattice (Section 2.1.2), it is less obvious why we might want the subtyping lattice to be distributive.

Distributive lattices have very well-behaved coproducts, which simplifies the subtyping relation (see Section 3.2.3). More generally, distributivity precludes the sort of vacuous reasoning that we decried in Section 1.4.1, in which we strove to avoid statements like (1.6), where

$$a \sqcap b \leqslant c$$

despite there being no relationship between $a$, $b$ and $c$. Distributivity turns out to be exactly the algebraic condition we can impose on the lattice which effectively forbids this sort of vacuous reasoning. While this is hard to see directly from the definition, there are several alternative presentations of distributivity which make the connection clearer. Below, we present four conditions, all equivalent to distributivity:

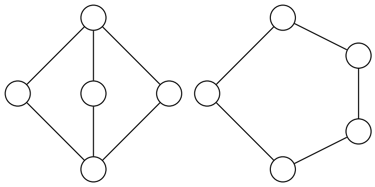
**Meet-distributivity** We can avoid the problem in (1.6) by demanding that all subtyping relationships between the meet of two types and another type follow from some property of the first two. Formally,

$$\text{If } a \sqcap b \leqslant c \text{, then } a' \sqcap b' = c \text{ for some } a' \leqslant a, b' \leqslant b$$

**Forbidden sublattices** Alternatively, we can simply demand that our lattice does not contain the two prototypical non-extensible lattices of Figure 2.1 ($M_3$ vacuously admits $a \sqcap b \leqslant c$, while $N_5$ suffers a similar problem by admitting $x \sqcap z \leqslant y$ vacuously):

$$\text{The lattice does not contain } M_3 \text{ or } N_5 \text{ as a sublattice}$$

[1] Coming up with examples is quite tricky, though. Some examples are complete atomless Boolean algebras (completely distributive complete Boolean algebras are atomic), such as the lattice of open-regular sets of a perfect topological space



(a) $M_3$ (b) $N_5$

Figure 2.1: The non-distributive lattices $M_3$ and $N_5$

**Representation as sets** Another condition is to require that our lattice of types be representable as a powerset lattice, by assuming a homomorphism $\phi$ mapping types to some set $A$, such that $\phi(\tau_1 \sqcap \tau_2) = \phi(\tau_1) \cup \phi(\tau_2)$, and similarly for joins. In other words,

There exists a lattice homomorphism to some powerset lattice

**Cut-elimination** We frequently need to decide statements like (1.6), of the form $a_0 \sqcap a_1 \sqcap \ldots \leqslant b_0 \sqcup b_1 \sqcup \ldots$. We call such statements *sequents*, by analogy with Gentzen's sequent calculus which also has a finite set of terms on the left, interpreted conjunctively, and a finite set of terms on the right, interpreted disjunctively. An important property of the sequent calculus is the *cut-elimination theorem*, which can be written in lattice notation as follows:

$$\frac{\bigsqcap_i A_i \leqslant c \sqcup \bigsqcup_j B_j \qquad \bigsqcap_i A_i \sqcap c \leqslant \bigsqcap_j B_j}{\bigsqcap_i A_i \leqslant \bigsqcup_j B_j}$$

The four conditions above are all equivalent [DP02, CC00] to distributivity. Distributivity has other side benefits: the coproduct of distributive lattices is rather more well-behaved than the coproduct of general lattices, which will ease construction of the subtyping lattice (see Section Section 3.2).

### 2.1.6 Recursive types and subtyping

In systems based on Hindley-Milner type inference, whether or not to include equirecursive types is a design decision, with no especially strong technical reasons to go either way. Including recursive types allows more programs, but also allows some questionable programs that would perhaps be better excluded.

In particular, the principality properties of type inference are unaffected by the presence of recursive types. The concrete inference algorithms are essentially the same, although care must be taken to implement unification correctly (either supporting unification of infinite terms, or having an occurs check).

However, with subtyping and in particular a least type $\bot$, recursive types cannot be omitted without losing principality. Consider the following term:[2]

$$f = \lambda g.f(\texttt{g true})$$

[2] This term can also be defined non-recursively, using a fixpoint combinator

The function $\mathsf{f}$ can be given any of the following types:

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

These are all non-recursive types. However, the *principal* type of the function $\mathsf{f}$ is the following recursive type:

$$(\mu\alpha.\text{bool} \to \alpha) \to \bot$$

Subtyping causes there to be terms typeable with finite types, yet whose principal types are recursive.

However, the usual characterisation of recursive types as being unique fixed points of their defining equations is not quite enough to handle subtyping correctly.

### 2.1.7 Fixed, pre-fixed and post-fixed points

Suppose $f : A \to A$ represents such a recursive definition, where $A$ is a poset. To interpret the definition, we frequently choose not just an arbitrary fixed point but the *least fixed point* $a$, giving us the reasoning principle:

$$\text{If } f(x) = x\text{, then } a \leqslant x$$

Or, even better, we require the *unique fixed point* $a$, giving the principle:

$$\text{If } f(x) = x\text{, then } a = x$$

Unfortunately, for our applications to subtyping, neither of these reasoning principles is strong enough. The issue is that the hypothesis $f(x) = x$ is an equation, whereas our knowledge about types consists of inequations $f(x) \leqslant x$ or $x \leqslant f(x)$. We need reasoning principles that have such conditions as their hypotheses.

A *pre-fixed point* of a function $f$ is some $a$ such that $f(a) \leqslant a$. Dually, a *post-fixed point* of $f$ is some $a$ such that $f(a) \geqslant a$. We denote the set of pre-fixed points of $f$ as $\mathsf{Pre}(f)$, the set of post-fixed points as $\mathsf{Post}(f)$, and the set of fixed points as $\mathsf{Fix}(f)$. Note that $\mathsf{Fix}(f) = \mathsf{Pre}(f) \cap \mathsf{Post}(f)$.

The useful properties of fixed points correspond to minimal, maximal or unique elements of $\mathsf{Pre}(f)$, $\mathsf{Post}(f)$ and $\mathsf{Fix}(f)$. The least or unique post-fixed point is uninteresting, since this is the least element of $A$ (if it exists). Dually, the greatest or unique pre-fixed point is equally uninformative. This leaves five useful properties that a point $a$ might have, listed below with their reasoning principles.

$$\begin{aligned}
\text{unique fixed point:}\quad & f(a) = a\text{, and if } f(x) = x\text{, then } a = x \\
\text{least fixed point:}\quad & f(a) = a\text{, and if } f(x) = x\text{, then } a \leqslant x \\
\text{greatest fixed point:}\quad & f(a) = a\text{, and if } f(x) = x\text{, then } a \geqslant x \\
\text{least pre-fixed point:}\quad & f(a) \leqslant a\text{, and if } f(x) \leqslant x\text{, then } a \leqslant x \\
\text{greatest post-fixed point:}\quad & f(a) \geqslant a\text{, and if } f(x) \geqslant x\text{, then } a \geqslant x
\end{aligned}$$

These properties are not all independent. Clearly, any point which is both the least and greatest fixed point is the unique fixed point. Less obviously, when we take $f$ to be monotone (that is, $x \leqslant y$ implies $f(x) \leqslant f(y)$), least pre-fixed points are least fixed points[3]:

[3] The categorically inclined will recognise this as Lambek's lemma.

**Lemma 2.** *If $a$ is the least pre-fixed point of monotone $f$, then $a$ is the least fixed point of $f$.*

*Proof.* $f(a) \leqslant a$ since $a$ is a pre-fixed point, so $f(f(a)) \leqslant f(a)$ by monotonicity of $f$. But this means $f(a)$ is a pre-fixed point, so $a \leqslant f(a)$. □

However, the converse of this does not hold! A least fixed point of $f$ is not necessarily a least pre-fixed point. Consider the function $f : \mathbb{R} \to \mathbb{R}$ defined by $f(x) = 2x$, which has a unique fixed point at $0$, yet every $x \leqslant 0$ is a pre-fixed point with no least such point. In other words, if a function has a least pre-fixed point, then it must also have a least fixed point, and they must coincide. But if a function has a least fixed point, it need not have a least pre-fixed point.

Naively, it might seem that this counterexample relies crucially on the lack of a least real number. This is not the case, which as demonstrated by the next a counterexample on a bounded poset. Consider the poset $P$, whose elements are the integers and a greatest element $\top$, ordered in the following unconventional way:

$$0 \leqslant 1 \leqslant 2 \leqslant 3 \leqslant \ldots \leqslant -3 \leqslant -2 \leqslant -1 \leqslant \top$$

Define the function $f : P \to P$ as follows:

$$f(x) = \begin{cases} n+1 & \text{if } x = n, n \in \mathbb{N} \\ -(n+1) & \text{if } x = -n, n \in \mathbb{N}, n \neq 0 \\ \top & \text{if } x = \top \end{cases}$$

In other words, $f$ moves non-negative integers one step right, negative integers one step left, and leaves $\top$ where it is. This is a monotone function, with a unique fixed point $\top$ which is also the greatest post-fixed point. However, this function has no least pre-fixed point, as $-n$ for every nonzero $n \in \mathbb{N}$ is a pre-fixed point.

A correct treatment of recursive types and subtyping requires least pre-fixed and greatest post-fixed points, rather than mere fixed points. Happily, most of the standard fixed-point theorems do in fact prove the least pre-fixed point property, although they are rarely stated in generality. Here is Tarski's fixed-point theorem, stated in least pre-fixed point form. The proof is essentially copied from that of Theorem 1 in Tarski's paper [Tar55] (with notation changed), although Tarski's paper and most subsequent literature state the result only in the weaker fixed-point form. First, we split off an independently useful lemma:

**Lemma 3.** *The meet of a set of pre-fixed points of a monotone function is a pre-fixed point.*

*Proof.* Let $P \subseteq \mathsf{Pre}(f)$, $u = \bigsqcap P$. For any $x \in P$, $u \leqslant x$, so $f(u) \leqslant f(x) \leqslant x$, making $f(u)$ a lower bound of $P$. Since $u$ is the greatest such lower bound, $f(u) \leqslant u$. □

**Theorem 4** (Tarski's pre-fixed point theorem). *If $f$ is a monotone function on a complete lattice $A$, then $f$ has a least pre-fixed point.*

*Proof.* $\bigsqcap \mathsf{Pre}(f)$ is a pre-fixed point by Lemma 3, and the least such by construction. □

Occasionally, we are interested not in the least pre-fixed point of a function $f$ but in the least pre-fixed point above some starting point $x$. If $x \in \mathsf{Post}(f)$, then this follows as a corollary:

**Lemma 5.** *For a complete lattice* $A$*, monotone* $f : A \to A$*, and* $x \in \mathsf{Post}(f)$*, then* $U_x$*, defined as*

$$U_x = \{y \mid y \in A, x \leqslant y\}$$

*is a complete lattice, and* $f$ *restricts to a monotone function* $U_x \to U_x$*.*

*Proof.* The meet or join of a set of elements above $x$ is above $x$, so $U_x$ is clearly complete. If $y \in U_x$ (that is, $x \leqslant y$), then $x \leqslant f(x) \leqslant f(y)$, so $f(y) \in U_x$ as well. □

**Corollary 6.** *For a complete lattice* $A$*, monotone* $f : A \to A$*, and* $x \in \mathsf{Post}(f)$*,* $f$ *has a least pre-fixed point above* $x$*.*

*Proof.* Apply Tarski's pre-fixed point theorem to $U_x$ and $f$. □

As a further corollary, we get Tarski's result that $\mathsf{Fix}(f)$ is a complete lattice:

**Corollary 7.** *For a complete lattice* $A$*, monotone* $f : A \to A$*,* $\mathsf{Fix}(f)$ *is a complete lattice (as a suborder, not sublattice, of* $A$*)*

*Proof.* For any $F \subseteq \mathsf{Fix}(f)$, $\bigsqcup F$ is a post-fixed point by the dual of Lemma 3, which has a least pre-fixed (therefore least fixed) point above it by the previous corollary, so $\mathsf{Fix}(f)$ has arbitrary least upper bounds. The dual argument gives arbitrary lower bounds. □

### 2.1.8 Other fixed point results and Bekič's construction

In order to obtain Tarski's result that every monotone function have a least pre-fixed point, it is not necessary to assume $A$ be a complete lattice. In fact, a directed-complete poset suffices, and this precisely characterises such posets. Specifically, due to results of Abian and Brown [AB61] and Markowsky [Mar76], we know the following are equivalent:

- $A$ is a directed-complete partial order.
- $A$ is *chain-complete*, having joins of totally ordered subsets.
- Every monotone $f : A \to A$ has a least fixed point.
- Every monotone $f : A \to A$ has a least pre-fixed point.

Conspicuously missing from this list is the property of having a fixed point (not necessarily a least one) for every monotone function, a condition known as the *fixed point property* and not known to be equivalent to anything in particular. Davis [Dav55] proved as a sort of converse to Tarksi's theorem that a lattice has a fixed point property iff it is complete. For posets not known to be lattices, the property is quite difficult to deal with, and it has been an open problem for several decades whether the product of two posets with the fixed point property retains the property. For a detailed discussion, read Schröder [Sch12].

These results are thoroughly classical. The axiom of choice is needed to equate directed-complete and chain-complete posets, and classical transfinite induction is the proof method of choice for many of the fixed point theorems.

I assume a classical setting throughout this thesis, but to the constructively-minded I point out Pataraia's beautiful proof that directed-complete posets have the fixed point property [Esc03], and Bauer's negative results [Bau12] showing that the fixed-point theorems fail constructively on chain-complete posets.

We sometimes need to find *simultaneous* fixed points, that is, given functions $f : A \times B \to A$ and $g : A \times B \to B$, find $a \in A, b \in B$ such that

$$f(a, b) = a \qquad\qquad g(a, b) = b$$

Such situations arise when resolving a set of mutually recursive type constraints. As mentioned above, we cannot conclude that $A \times B$ has the fixed point property just because $A$ and $B$ do. However, suppose that the fixed points (or least pre-fixed points) are assigned in a monotone fashion. That is, we have a fixed point or least pre-fixed point $\mu f$ for every monotone $f$, and moreover, if $f \leqslant f'$ pointwise then $\mu f \leqslant \mu f'$. In this case, a construction of Bekič[4] gives simultaneous fixed points of $f$ and $g$:

$$\mu(f \times g) = \begin{pmatrix} \mu a.f(a, \mu b.g(a, b)) \\ \mu b.g(\mu a.f(a, b), b) \end{pmatrix}$$

The stronger condition is needed to ensure that

$$a \mapsto f(a, \mu b.g(a, b))$$

is itself a monotone function, allowing the outer use of $\mu$.

This construction will be particularly useful in Chapter 5, when the finding of least pre-fixed points is restricted to functions of a particular syntactic form $\mu\alpha.t$. Bekič's construction allows us to use this single-variable construction to find simultaneous fixed points in multiple variables.

## 2.2 Semirings and Kleene algebra

A *semiring* consists of a set equipped with constants $0, 1$ and binary operators $+, \cdot$, such that $+$ is associative and commutative with identity $0$, $\cdot$ is associative with identity $1$, $+$ distributes over $\cdot$, and $0 \cdot a = a \cdot 0 = 0$.

The semirings of interest in this thesis are *idempotent semirings*, satisfying the extra condition $a + a = a$. This condition gives the semiring a semilattice structure, with $+$ as join and $0$ as the least element. In other words, an idempotent semiring is a semilattice with a binary operation $\cdot$ which preserves finite joins and has identity $1$.

While the connection between partial orders and subtyping is direct, the usefulness of semirings for subtyping is less obvious. The link arises from two important examples of semirings.

First, given a semilattice $A$, the semilattice morphisms $A \to A$ form an idempotent semiring, with composition as $\cdot$ and the identity function as $1$. Since *type substitutions* (Section 3.3.3) preserve $\sqcup$ and $\sqcap$, they are semilattice morphisms and form a semiring.

Second, the regular languages over some finite alphabet form an idempotent semiring, with $+$ as union, $0$ as the empty language, $\cdot$ as concatenation,

[4] The correct spelling of Hans Bekič's surname is unclear. In *Programming Languages and Their Definition* [BJ84], a collection of his work, his name is variously given as Bekic, Bekić and Bekič, although perhaps this has more to do with the limitations of typewriters than personal preference.

and 1 as the language consisting of the empty string. The regular languages have an efficient representation as finite automata, and so can be efficiently manipulated.

Section 7.2 exploits the semiring algebra of these two constructions, presenting algorithms which represent and manipulate types and type substitutions as though they were regular languages. To prove this valid, we must understand the equational theory of regular languages.

### 2.2.1 Axiomatising the regular languages

As well as the functions $+, \cdot$ and constants $0, 1$ of semiring theory, regular languages are described using the Kleene star, a unary operator $^*$. To justify manipulating types as regular languages, we must show that all equations true of regular languages hold of types.

Axiomatising the equations true of regular languages is quite tricky. The obvious approach is to augment the definition of an idempotent semiring with some identities concerning $^*$, for instance:

$$\begin{aligned} a^* &= 1 + a \cdot a^* & \qquad (a^*)^* &= a^* \\ a^* &= 1 + a^* \cdot a & \qquad (a+b)^* &= a^*(ba^*)^* \end{aligned}$$

While these identities do hold of regular languages, they are not *complete*: there are identities of regular languages not derivable from the rules of semirings augmented by the above. In fact, no finite list of identities completely axiomatises the equations of regular languages, a result first proved by Redko [Red64][5].

Instead, the behaviour of $^*$ can be specified using an identity and an implication. A *left-handed Kleene algebra* is an idempotent semiring with a unary operator $^*$ satisfying:

$$a^* = a \cdot a^* + 1$$
$$a \cdot x \leqslant x \Longrightarrow a^* \cdot x \leqslant x$$

**Theorem 8** (Completeness for left-handed Kleene algebras). *All equations of regular languages hold in every left-handed Kleene algebra. That is, two finite terms built of variables, $0$, $1$, $+$, $\cdot$ and $*$ denote equal regular languages iff they have the same interpretation in every left-handed Kleene algebra.*

This result was stated without proof by Conway [Con71], and finally proved by Boffa [Bof95], relying on work by Krob [Kro91]. Recently, Kozen and Silva [KS12] published a much shorter proof.

A *right-handed Kleene algebra* is an idempotent semiring satisfying flipped versions of the above rules:

$$a^* = a^* \cdot a + 1$$
$$x \cdot a \leqslant x \Longrightarrow x \cdot a^* \leqslant x$$

The completeness result holds for these as well, since every right-handed Kleene algebra is a left-handed one with the arguments to $\cdot$ swapped. However, right- and left-handed Kleene algebras are not equivalent: Kozen [Koz90] constructs a right-handed algebra which is not left-handed. The completeness

[5] This paper is out of print, in Russian, and quite difficult to get a hold of, so the interested reader is directed to a shorter proof by Conway [Con71, p. 105], which is merely out of print.

result means that their equational theories agree (that is, they have the same valid identities), but they disagree on implications or general first-order sentences.

An algebra which is both a left-handed and right-handed Kleene algebra is called a *Kleene algebra*, a concept which (with its completeness result) predates the one-handed variety [Koz90].

### 2.2.2 Kleene algebra via pre-fixed points

The behaviour of * in a (perhaps one-handed) Kleene algebra can be better understood as constructing pre-fixed points. In fact, left-handed Kleene algebras can be characterised as semirings with least pre-fixed points of *affine maps* $f(x) = a \cdot x + b$.

**Theorem 9.** *For an idempotent semiring* $A$*, the following are equivalent:*

- $A$ *is a left-handed Kleene algebra.*
- $A$ *contains a least pre-fixed point* $\mu x.\ a \cdot x + b$ *of every affine map* $f(x) = a \cdot x + b$*, such that*

$$(\mu x.\ a \cdot x + b) \cdot c = (\mu x.\ a \cdot x + b \cdot c)$$

*Proof.* ($\Rightarrow$) The affine map $f(x) = a \cdot x + b$ has least pre-fixed point $a^* \cdot b$ since it is a fixed point:

$$a \cdot (a^* \cdot b) + b = (a \cdot a^* + 1) \cdot b = a^* \cdot b$$

and given any pre-fixed point $y$ (with $a \cdot y + b \leqslant y$), $a^* \cdot y \leqslant y$ (since $a \cdot y \leqslant y$), and so $a^* \cdot b \leqslant y$ (since $b \leqslant y$). It satisfies the second condition since $(a^* \cdot b) \cdot c = a^* \cdot (b \cdot c)$.

($\Leftarrow$) Define $a^* = \mu x.a \cdot x + 1$, implying $a^* = a \cdot a^* + 1$. Given $a \cdot x \leqslant x$, we have that $x$ is a pre-fixed point of $y \mapsto a \cdot y + x$, so

$$a^* \cdot x = (\mu y.\ a \cdot y + 1) \cdot x = \mu y.\ a \cdot y + x \leqslant x$$

□

This characterisation tells us that the * operator on a given semiring making it into a left-handed Kleene algebra is *unique*. Either least pre-fixed points of affine maps exist or they don't, but there is never a choice of definitions for *.

Furthermore, suppose a given semiring has a (necessarily unique) operator * making it a left-handed Kleene algebra, and a (also necessarily unique) operator making it a right-handed Kleene algebra (which we now label † to disambiguate). By the completeness theorems, left- and right- handed Kleene algebras all identities of regular languages, in particular:

$$a^* = a^* \cdot a + 1$$
$$a^\dagger = a \cdot a^\dagger + 1$$

However, $a^\dagger$ is the least $x$ satisfying $x \cdot a + 1 \leqslant x$, while $a^*$ is the least $x$ satisfying $a \cdot x + 1 \leqslant x$, giving that $a^\dagger \leqslant a^*$ and $a^* \leqslant a^\dagger$, making * and † coincide.

So, the structure of a (one- or both-handed) Kleene algebra is completely determined by its underlying semiring! Adding the operator * is not a new operation, so much as an assertion that certain pre-fixed points already exist in the semiring.

### 2.2.3 Complete and *-continuous Kleene algebras

One- or both-handed Kleene algebras manage to capture all of the identities of regular languages, but are in some respects still quite ill-behaved. In particular, we might think that $a^*$ is given as the least upper bound of its powers:

$$1 + a + a \cdot a + a \cdot a \cdot a + \dots$$

However, this least upper bound is not guaranteed to exist in an arbitrary Kleene algebra, and even when it does it is not guaranteed to be equal to $a^*$! Note that this does not contradict the completeness theorem: the statement $a^* = 1 + a + a \cdot a + \dots$ is not an "identity" in the sense of an equation between finite terms.

This odd behaviour can be precluded by requiring stronger properties. A *complete idempotent semiring*[6] is an idempotent semiring where arbitrary, not just finite, joins exist and are preserved by multiplication. All such semirings are Kleene algebras, where $a^* = \bigsqcup_k a^k$.

However, complete idempotent semirings require a little too much. If we work with formal languages, demanding that all joins exist takes us out of the domain of regular languages and includes arbitrary formal languages. This defeats the point somewhat: the usefulness of Kleene algebras lies in their including sufficiently many joins and fixed points to describe recursive structures, but sufficiently few that finite automata can be used as representations. Specifically, in Section 7.2, we use Kleene algebras to describe those types that can be written using a finite, well-behaved syntax (the *polar types* of Chapter 5). This syntax is closed under finite join and forming recursive types, but definitely not under arbitrary joins.

A happy balance is found in the definition of *$^*$-continuous Kleene algebras*[7], which are subsets of a complete idempotent semiring containing $0$, $1$, and closed under $+$, $\cdot$ and $^*$. Thus, a $^*$-continuous semiring must have a well-behaved $^*$, but may be small enough to be described by finite syntax.

It is a theorem of Conway and Kozen that the $^*$-continuous Kleene algebras are exactly those semirings equipped with an operation $^*$ satisfying:

$$a \cdot b^* \cdot c = \bigsqcup_k a \cdot b^k \cdot c$$

## 2.3 Category theory

I assume some knowledge of category theory in order to construct the lattice of types in Chapter 3. However, a purely syntactic version of the construction appears in that chapter, and category theory is not needed to understand the rest of the thesis (although the syntactic construction will seem somewhat unmotivated and arbitrary without the categorical explanation).

Only basic category theory is assumed: categories, sums, products, and functors. Chapter 3 uses initial algebras to interpret recursive definitions, while free objects have a brief cameo, but these concepts are explained as needed.

[6] Kozen's terminology [Koz90]. Conway calls these **S**-algebras.

[7] Conway's **N**-algebras

### 2.3.1 Categories of orders

The development of the type system in this thesis is explained using a few order-themed categories, including:

**Pos** The category whose objects are partially ordered sets and whose morphisms are monotone functions.

**Lat** The category whose objects are lattices and whose morphisms are functions that preserve finite meets and finite joins.

**DLat** The category whose objects are distributive lattices and whose morphisms are functions that preserve finite meets and finite joins.

Note that by defining the morphisms of **Lat** and **DLat** in terms of finite rather than binary meets and joins, we ensure that morphisms preserve the least and greatest elements (as these are empty joins and meets respectively).

There is an important functor $(-)^{\mathrm{op}} : \mathbf{Pos} \to \mathbf{Pos}$, the *dual* functor, which acts on a poset by reversing the ordering. This is a functor: for any monotone $f : A \to B$, we have:

$$a \leqslant_A a' \implies f(a) \leqslant_B f(a')$$

Equivalently,

$$a \geqslant_A a' \implies f(a) \geqslant_B f(a')$$

so we can take $f^{\mathrm{op}} = f$ as a morphism from $A^{\mathrm{op}}$ to $B^{\mathrm{op}}$.

Since the dual of a lattice is a lattice, and likewise for a distributive lattice, the functor $(-)^{\mathrm{op}}$ is also well-defined for **Lat** and **DLat**:

$$\begin{aligned}(-)^{\mathrm{op}} &: \mathbf{Pos} \to \mathbf{Pos}\\ (-)^{\mathrm{op}} &: \mathbf{Lat} \to \mathbf{Lat}\\ (-)^{\mathrm{op}} &: \mathbf{DLat} \to \mathbf{DLat}\end{aligned}$$

Two other useful functors on the category of posets are:

$$(-)^{\top} : \mathbf{Pos} \to \mathbf{Pos} \qquad\qquad (-)^{\perp} : \mathbf{Pos} \to \mathbf{Pos}$$

The functor $(-)^{\top}$ acts on a poset by adjoining a new greatest element, and acts on morphisms by preserving the new element:

$$f^{\top}(x) = \begin{cases} \top & \text{if } x = \top \\ f(x) & \text{otherwise} \end{cases}$$

The functor $(-)^{\perp}$ is its dual, and we have $(L^{\top})^{\mathrm{op}} = (L^{\mathrm{op}})^{\perp}$. As with $(-)^{\mathrm{op}}$, the operations of adding new greatest and least elements are also well-defined for lattices, so we also use these as functors on **Lat** and **DLat**.

### 2.3.2 Concrete categories and free objects

These categories all have a similar structure: their objects are sets with some extra structure, and the morphisms are those functions between the sets which preserve this structure. This idea can be made formal as a *concrete category*, which is a category equipped with a functor $|-| : \mathcal{C} \to \mathbf{Set}$ known as a *forgetful functor*, whose effect is to forget any structure of $\mathcal{C}$ beyond plain sets and return the underlying sets and functions. For instance, the forgetful functor $|-| : \mathbf{Pos} \to \mathbf{Set}$ maps each poset $A$ to its set of elements $|A|$, ignoring the order with which $A$ is equipped. A **Pos**-morphism $f : A \to B$ (that is, a monotone function $f$ from $A$ to $B$) is mapped to the **Set**-morphism $|f| : |A| \to |B|$ (that is, a function $|f|$ from $A$ to $B$) by simply forgetting that $f$ preserves ordering.

For each of the categories **Pos**, **Lat** and **DLat**, products are taken the same way as in **Set**, by Cartesian products:

$$|A \times B| = |A| \times |B|$$

However, this is not the case for sums in **Lat** and **DLat**, since the sum of two lattices is given by a construction quite unlike disjoint union:

$$|A + B| \neq |A| + |B|$$

This distinction is of central importance in Section 3.2.

A poset, lattice or distributive lattice can be made out of any set using a construction called the *free object*, hence the terms *free lattice* and *free distributive lattice* (free posets being rather boring). Given a set $V$ whose elements are called *generators*, the free lattice $\mathsf{Free}(V)$ is a lattice consisting of all terms built from elements of $V$ and the lattice operations, quotiented by equivalence under the lattice laws. For $V = a, b$, $\mathsf{Free}(V)$ contains the following elements:

$$\bot, a \sqcap b, a, b, a \sqcup b, \top$$

Since the laws for lattices and distributive lattices differ, the free lattice and the free distributive lattice on the same set $V$ are quite different, when $V$ has more than two elements. For instance, the free distributive lattice on three generators contains 20 elements (arranged as on page 8), while the free lattice on three generators is infinite.

The defining property of a free lattice (or free object in general) is that a function from the generators to some lattice $A$ can be extended uniquely to a morphism from $\mathsf{Free}(V)$ to $A$. More formally, there is a natural bijection between the functions (**Set**-morphisms) $V \to |A|$ and the morphisms $\mathsf{Free}(V) \to A$.

Free (distributive) lattices are used in Section 3.3.2 to formalise type variables, where their defining property is used to understand substitution: an assignment of types to type variables (a function) can be extended to a lattice morphism. In other words, type terms with variables can be uniquely interpreted as types once you know what types are assigned to each variable.

### 2.3.3 Aside: orders versus categories for subtyping

A partial order (in fact, any preorder) can be interpreted as a category, by taking the objects of the category to be the elements of the partial order, and supposing a single morphism $a \to b$ whenever $a \leqslant b$. The reflexivity of the order provides identity morphisms, and transitivity provides composition.

This is a deep connection, with many of the concepts of category theory being mirrored precisely in order theory. Most definitions and theorems carry over directly, with the assumption of uniqueness of morphisms simplifying the order-theoretic versions.

The question arises, therefore, of why to bother with order theory at all? Subtyping could be expressed by placing types in a category instead of a partial order, removing the need for orders. Apart from missing out on the usefulness of various order-theoretic results that do not generalise well to categories (in particular, the fixed-point theorems), there is a deeper reason not to do this.

Subtyping between types is done implicitly by a type system, without any program-level marker to signify the point at which subtyping happens. As a programmer, I expect to be able to control any operation that the program does which affects its result. In other words, when the compiler implicitly applies subtyping, I require that it makes no choices in doing so: between two types there should not be a choice of possible subtyping routes. Categorically, morphisms should be unique, which selects from possible categories exactly those that correspond to orders.

# 3 Constructing types

*What are numbers and what should they be?*

—Richard Dedekind

*Numbers are usually integers.*

—GNU assembler reference manual

For the bulk of this thesis, we study a minimal calculus with subtyping, containing booleans, records, and functions. These three features are just enough to make the subtyping relation realistically awkward, while remaining minimal:

- Booleans give a base type, incomparable to types made with any other type constructor.

- Functions give type constructor with type parameters of different variance: function types are contravariant in their domain and covariant in their range.

- Records give subtyping relations between type constructors of different arity, since a record type with more fields is a subtype of one with fewer.

Including all of these types keeps us out of several well-known easier special cases: function types mean we cannot assume purely *covariant subtyping*, while record types mean we cannot use *structural subtyping*[1], and so we are forced to tackle the problem in generality. In Chapter 9, we see how the system developed to handle boolean, record and function types smoothly extends to other features, like variant types (sums), mutable references, and effects.

The typing rules for this minimal calculus are the standard typing rules for ML, augmented with the standard subtyping rule, so I unimaginatively call it MLsub. Two presentations of the typing rules (the familiar ML-style ones, and an equivalent reformulation that makes the inference algorithm clearer) are presented and discussed in detail in the next chapter.

[1] The terminology here is unfortunate. In the subtyping literature, *structural subtyping* (as opposed to *non-structural subtyping*) refers to a subtyping relation that relates only type constructors of the same arity, precluding width subtyping on records. In object-orientation literature, *structural typing* (as opposed to *nominal typing*) refers to subtyping relations that are implicitly formed from similar structures, rather than having to be explicitly declared via subclassing annotations. Confusingly, by these definitions, structural typing requires non-structural subtyping!

The syntax, semantics and even typing rules of MLsub are entirely standard. The novelty that makes principal type inference possible lies instead in a part of the language often treated too briefly: the definition of the types themselves.

Conventionally, types get a one-line definition laying out their syntax:

$$\tau ::= \mathrm{bool} \mid \tau_1 \to \tau_2 \mid \{\ell_1 : \tau_1, \dots, \ell_n : \tau_n\}$$

Such a definition can be read in two ways. First, we have the syntactic viewpoint: types are trees of symbols according to the above grammar. Alternatively, we have the algebraic viewpoint: types are members of the initial algebra of a functor on **Set**.

The syntactic viewpoint enables easy implementation: representing syntactic terms as data structures in a compiler is straightforward. The algebraic viewpoint enables reasoning: that this is an *initial* algebra allows results about it to be transported to any other algebra, giving extensibility. Happily, both viewpoints coincide.

When we introduce subtyping, the story is much the same. The syntactic presentation is modified by adding an inductively defined subtyping relation, while the algebraic presentation is modified by taking an initial algebra in **Pos** instead of **Set**. Again, either viewpoint gives the same answer.

However, once we require types to form a lattice rather than a mere poset (as required for inference), the two viewpoints diverge sharply. Previous work has (perhaps unthinkingly) consistently chosen the syntactic route, leaving a system bereft of universal properties or useful algebraic structure. Our goal here is to construct a collection of types and a subtyping relation, with better algebraic structure than previous approaches.

The algebraic construction of the collection of types is carried out in stages, defining the following systems of types, each with more structure than the previous:

| | |
|---|---|
| $\mathcal{T}_s$ | simple types |
| $\mathcal{T}_o$ | simple types, subtyping order |
| $\mathcal{T}_b$ | simple types, subtyping order (bounded) |
| $\mathcal{T}_l$ | simple types, subtyping lattice |
| $\mathcal{T}_V$ | simple types, subtyping lattice, type variables |
| $\widetilde{\mathcal{T}_V}$ | simple types, subtyping lattice, type variables, recursive types |

While systems $\mathcal{T}_s$, $\mathcal{T}_o$ and $\mathcal{T}_b$ are entirely standard, $\mathcal{T}_l$ diverges from previous work. The conventional approach to constructing a subtyping lattice builds instead the lattice $\mathcal{T}_l{}'$, which while having a pleasantly minimal syntactic definition fails to be extensible or algebraically well-behaved.

A similar situation occurs when moving from $\mathcal{T}_l$ to $\mathcal{T}_V$. The standard approach is to first define ground types, and introduce type variables as meta-variables quantifying over these ground types. Section 1.4 shows how this approach breaks extensibility, so instead we define $\mathcal{T}_V$ as a *free algebra*, which amounts to treating type variables as opaque indeterminates.

Once we have added recursive types to give $\widetilde{\mathcal{T}_V}$, we have a lattice of types that is particularly well-behaved algebraically, enabling complete and principal type inference. In subsequent chapters, we take $\mathcal{T} = \widetilde{\mathcal{T}_V}$, and do not refer to the others.

## 3.1 Simple types

Our first attempt at defining types defines the set $\mathcal{T}_s$ of *simple types*, having functions, records and booleans. Syntactically, this is:

$$\tau \in \mathcal{T}_s ::= \mathrm{bool} \mid \tau_1 \to \tau_2 \mid \{\ell_1 : \tau_1, \dots, \ell_n : \tau_n\}$$

We assume some set $\mathcal{L}$ of record labels, over which $\ell_i$ range. 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 $\mathrm{dom}\, f = \{\ell_1, \ell_2\}$, $f(\ell_1) = \tau_1$, $f(\ell_2) = \tau_2$. We assume there to be only finitely many record labels: a finite program can only refer to finitely many anyway, and having only finitely many type constructors makes our lives easier when reasoning about completeness and limits in Section 3.4.

An inductive definition like the above can be read as the initial algebra of a functor. We construct three functors, $\mathsf{Bool}_s$, $\mathsf{Func}_s$ and $\mathsf{Rec}_s$ for boolean, function and record types, and combine these into a functor $\mathcal{F}_s$ of which $\mathcal{T}_s$ is the initial algebra.

There is only one boolean type, so the functor $\mathsf{Bool}_s : \mathbf{Set} \to \mathbf{Set}$ is the constant functor returning the set 1, which is a singleton set (the terminal object of **Set**):

$$\mathsf{Bool}_s(A) = 1$$

Function types are given by a pair of types (the domain and the range), so the functor $\mathsf{Func}_s$ returns a product:

$$\mathsf{Func}_s(A) = A \times A$$

Record types are given by a partial function from labels (elements of $\mathcal{L}$) to types. Since the class of partial functions $A \rightharpoonup B$ is isomorphic to the class of total functions $A \to (B + 1)$, we choose to write the functor $\mathsf{Rec}_s$ as follows:

$$\mathsf{Rec}_s(A) = \mathcal{L}^{A+1}$$

Above, we choose to interpret $\mathcal{L}^{(-)}$ as an iterated product, rather than anything more exotic:

$$\mathcal{L}^X = X \times X \times \dots \times X$$

where the number of terms in the product is the (assumed finite) cardinality of $\mathcal{L}$.

Thus, the functor $\mathcal{F}_s : \mathbf{Set} \to \mathbf{Set}$ is defined as:

$$\mathcal{F}_s(A) = \mathsf{Bool}_s(A) + \mathsf{Func}_s(A) + \mathsf{Rec}_s(A)$$

and the simple types $\mathcal{T}_s$ form the initial algebra of $\mathcal{F}_s$.

### 3.1.1 Algebras, initial and otherwise

In general, a $\mathcal{F}_s$-*algebra* $(A, f)$ consists of a set $A$ and a function (**Set**-morphism) $f : \mathcal{F}_s(A) \to A$. In other words, a $\mathcal{F}_s$-algebra is a set equipped with some interpretation of boolean, function types and record types, that is, a set with a distinguished element bool, a function $- \to - : A \times A \to A$, and functions $\{\ell_1 : -, \dots \ell_n : -\} : A^n \to A$.

$$\begin{array}{rl}
(\textsc{Bool}) & \dfrac{}{\mathrm{bool} \leqslant \mathrm{bool}} \\[2ex]
(\textsc{Func}) & \dfrac{\tau_1' \leqslant \tau_1 \quad \tau_2 \leqslant \tau_2'}{\tau_1 \to \tau_2 \leqslant \tau_1' \to \tau_2'} \\[2ex]
(\textsc{RecWidth}) & \dfrac{}{\{\ell_1 : \tau_1, \ldots, \ell_n : \tau_n, \ldots \ell_{n+m} : \tau_{n+m}\} \leqslant \{\ell_1 : \tau_1, \ldots \ell_n : \tau_n\}} \\[2ex]
(\textsc{RecDepth}) & \dfrac{\tau_1 \leqslant \tau_1' \quad \ldots \quad \tau_n \leqslant \tau_n'}{\{\ell_1 : \tau_1, \ldots, \ell_n : \tau_n\} \leqslant \{\ell_1 : \tau_1', \ldots, \ell_n : \tau_n'\}}
\end{array}$$

Figure 3.1: Subtyping rules

A morphism between $\mathcal{F}_s$-algebras $(A, f)$ and $(B, g)$ is a function $\alpha : A \to B$ which respects the $\mathcal{F}_s$-algebra structure, i.e. makes the following diagram commute:

$$\begin{tikzcd}
\mathcal{F}_s(A) \arrow[r, "f"] \arrow[d, "\mathcal{F}_s(\alpha)"] & A \arrow[d, "\alpha"] \\
\mathcal{F}_s(B) \arrow[r, "g"] & B
\end{tikzcd}$$

In this way, $\mathcal{F}_s$-algebras form a category.

$\mathcal{T}_s$ is an $\mathcal{F}_s$-algebra, because it is equipped with an interpretation of boolean, function and record types. Moreover, it is the *initial* $\mathcal{F}_s$-algebra, meaning that for any other $\mathcal{F}_s$-algebra $(A, f)$, there is an $\mathcal{F}_s$-algebra morphism $\alpha : (\mathcal{T}_s, \mathrm{id}) \to (A, f)$ which preserves the interpretation of booleans, functions and record types. This translates into the following universal property:

**Proposition 10.** *For any $\mathcal{F}_s$-algebra $(A, f)$, there is a unique function $\alpha : \mathcal{T}_s \to A$ such that:*

$$\begin{aligned}
\alpha(\mathit{bool}) &= \mathit{bool} \\
\alpha(\tau_1 \to \tau_2) &= \alpha(\tau_1) \to \alpha(\tau_2) \\
\alpha(\{\ell_i : \tau_i\}) &= \{\ell_i : \alpha(\tau_i)\}
\end{aligned}$$

In other words, working with the initial algebra provides extensibility: anything we can construct in $\mathcal{T}_s$ can be mapped in a structure-preserving way to any other $\mathcal{F}_s$-algebra, even if the other algebra defines other types as well.

### 3.1.2 Subtyping

Next, we equip our types with a subtyping order. Syntactically, this is done by defining the relation $\leqslant$ inductively according to the rules of Figure 3.1. Since record types are defined by a partial function rather than a list, we need no permutation rules. It is a routine proof that the rules of Figure 3.1 do in fact define a partial order, and we write $\mathcal{T}_o$ for the poset ordered by $\leqslant$ whose carrier set is $\mathcal{T}_s$.

Algebraically, we again take $\mathcal{T}_o$ to be an initial algebra, but this time we take the initial algebra in **Pos** rather than **Set**. The functor $\mathsf{Bool}_o : \mathbf{Pos} \to \mathbf{Pos}$ looks the same as before, although this time 1 denotes the singleton poset rather than the singleton set:

$$\mathsf{Bool}_o(A) = 1$$

Function types are ordered contravariantly in their domain and covariantly in their range, so we define:

$$\mathsf{Func}_o(A) = A^{op} \times A$$

where $(-)^{op} : \mathbf{Pos} \to \mathbf{Pos}$ is the functor that maps every poset $(X, \leqslant)$ to its dual $(X, \geqslant)$.

Records are partial functions from $\mathcal{L}$ to types, which as before we represent as total functions from $\mathcal{L}$ to types with one extra element. Since (RecWidth) tells us that records lacking fields are ordered above records containing them, the extra element is added as a new greatest element to the poset, giving

$$\mathsf{Rec}_o(A) = \mathcal{L}^{(A^\top)}$$

where $(-)^\top : \mathbf{Pos} \to \mathbf{Pos}$ is the functor that adjoins a new top element.

These functors directly encode the subtyping rules of Fig. 3.1, so we define $\mathcal{F}_o : \mathbf{Pos} \to \mathbf{Pos}$ as:

$$\mathcal{F}_o(A) = \mathsf{Bool}_o(A) + \mathsf{Func}_o(A) + \mathsf{Rec}_o(A)$$

and $\mathcal{T}_o$ is the initial algebra of $\mathcal{F}_o$. By being initial, this algebra has the same useful extensibility properties as $\mathcal{T}_s$.

This construction extends the construction of $\mathcal{T}_s$ in a strong sense. We have the forgetful functor $|-|^{\mathbf{Pos}} : \mathbf{Pos} \to \mathbf{Set}$ which turns a poset into a plain set by dropping its ordering. Since $|A^{op}|^{\mathbf{Pos}} = |A|^{\mathbf{Pos}}$ and $|A^\top|^{\mathbf{Pos}} = |A|^{\mathbf{Pos}} + 1$, the poset functors $\mathsf{Bool}_o$, $\mathsf{Func}_o$, and $\mathsf{Rec}_o$ agree with their counterparts in **Set**:

$$\begin{aligned}
|\mathsf{Bool}_o(A)|^{\mathbf{Pos}} &= \mathsf{Bool}_s(|A|^{\mathbf{Pos}}) \\
|\mathsf{Func}_o(A)|^{\mathbf{Pos}} &= \mathsf{Func}_s(|A|^{\mathbf{Pos}}) \\
|\mathsf{Rec}_o(A)|^{\mathbf{Pos}} &= \mathsf{Rec}_s(|A|^{\mathbf{Pos}})
\end{aligned}$$

### 3.1.3 Least and greatest types

Syntactically, it is straightforward to add least and greatest types. We define the *bounded types* $\mathcal{T}_b$ inductively by extending the syntax:

$$\tau \in \mathcal{T}_b ::= \mathsf{bool} \mid \tau \to \tau \mid \{\ell_1 : \tau_1, \ell_2 : \tau_2, \ldots\} \mid \bot \mid \top$$

and order it by the rules of Figure 3.1, augmented with two new rules:

$$\frac{}{\bot \leqslant \tau} \qquad \frac{}{\tau \leqslant \top}$$

Algebraically, the construction is similar. We work now in the category $\mathbf{Pos}_\bot^\top$ of *bounded posets*: those with a least and greatest element, and monotone maps which preserve the least and greatest elements. We add the new elements to the definition of $\mathsf{Bool}_b$, $\mathsf{Func}_b$ and $\mathsf{Rec}_b$:

$$\begin{aligned}
\mathsf{Bool}_b(A) &= 1_\bot^\top \\
\mathsf{Func}_b(A) &= (A^{op} \times A)_\bot^\top \\
\mathsf{Rec}_b(A) &= (\mathcal{L}^{(A^\top)})_\bot^\top
\end{aligned}$$

where the functor $(-)_\bot^\top$ augments its input with new least and greatest elements. Since the functors $(-)^{op}$ and $(-)^\top$ preserve boundedness, we are

$$\begin{aligned}\text{bool} \sqcup \text{bool} &= \text{bool}\\ \text{bool} \sqcap \text{bool} &= \text{bool}\end{aligned}$$

$$\begin{aligned}(\tau_1 \to \tau_2) \sqcup (\tau_1' \to \tau_2') &= (\tau_1 \sqcap \tau_1') \to (\tau_2 \sqcup \tau_2')\\ (\tau_1 \to \tau_2) \sqcap (\tau_1' \to \tau_2') &= (\tau_1 \sqcup \tau_1') \to (\tau_2 \sqcap \tau_2')\end{aligned}$$

$$\begin{aligned}\{f\} \sqcup \{g\} = {} & \{h\}\\ & \text{where } \operatorname{dom} h = \operatorname{dom} f \cap \operatorname{dom} g\\ & \text{and } h(\ell) = f(\ell) \sqcup g(\ell)\\ \{f\} \sqcap \{g\} = {} & \{h\}\\ & \text{where } \operatorname{dom} h = \operatorname{dom} f \cup \operatorname{dom} g\\ & \text{and } h(\ell) = \begin{cases} f(\ell) \sqcap g(\ell) & \text{if } \ell \in \operatorname{dom} f,\ \ell \in \operatorname{dom} g\\ f(\ell) & \text{if } \ell \in \operatorname{dom} f,\ \ell \notin \operatorname{dom} g\\ g(\ell) & \text{if } \ell \notin \operatorname{dom} f,\ \ell \in \operatorname{dom} g \end{cases}\end{aligned}$$

Figure 3.2: Meet and join

justified in treating them as functors $\mathbf{Pos}_\perp^\top \to \mathbf{Pos}_\perp^\top$. The definition of $\mathcal{F}_b$ is as before:

$$\mathcal{F}_b(A) = \mathsf{Bool}_b(A) + \mathsf{Func}_b(A) + \mathsf{Rec}_b(A)$$

In this category, coproducts are constructed by taking the disjoint union and identifying the least elements of each summand, as well as the greatest elements, so we end up adding only a single new top and bottom element, despite $(-)_\perp^\top$ appearing in each summand. Again, the algebra $\mathcal{T}_b$ is the initial algebra of this functor $\mathcal{F}_b$.

## 3.2 A (distributive) lattice of types

So far, the syntactic and algebraic approaches have given the same results. However, they suggest different moves as soon as we require a lattice rather than a plain poset of types.

The syntactically simplest lattice to build is to keep the syntax of $\mathcal{T}_b$ and notice that it is, in fact, already a lattice!

$$\mathcal{T}_l' = \mathcal{T}_b$$

I give it a new name $\mathcal{T}_l'$ to emphasize that we now consider it equipped with meets and joins, rather than just a bounded ordering. The computation of meet and join in $\mathcal{T}_l'$ is as shown in Figure 3.2. Cases not matched by Figure 3.2 (e.g. bool $\sqcup$ (bool $\to$ bool)) are $\top$ (for unmatched joins) or $\perp$ (for unmatched meets).

The algebraically simplest lattice to build is to keep the definition of $\mathcal{F}_b$, but use a category of lattices instead of posets. In fact, I work with the category **DLat** of *distributive* lattices, since these have better extensibility properties (see

Section 2.1.5). So, we define $\mathcal{F}_l : \mathbf{DLat} \to \mathbf{DLat}$ as follows:

$$\begin{aligned}
\mathsf{Bool}_l(A) &= 1^\top_\perp \\
\mathsf{Func}_l(A) &= (A^{\mathrm{op}} \times A)^\top_\perp \\
\mathsf{Rec}_l(A) &= (\mathcal{L}^{(A^\top)})^\top_\perp \\
\mathcal{F}_l(A) &= \mathsf{Bool}_l(A) + \mathsf{Func}_l(A) + \mathsf{Rec}_l(A)
\end{aligned}$$

The functors $(-)^{\mathrm{op}}$, $(-)^\top$, and $(-)^\top_\perp$ preserve the property of being a distributive lattice, so we are justified in taking them to be functors $\mathbf{DLat} \to \mathbf{DLat}$. Since the definition of $\mathcal{F}_4$ lies in $\mathbf{DLat}$ rather than $\mathbf{Pos}^\top_\perp$, the coproduct is a coproduct of distributive lattices rather than of bounded posets. Taking the initial algebra of this functor gives a distributive lattice $\mathcal{T}_l$, which is quite different from $\mathcal{T}_l'$.

### 3.2.1 Syntactic construction

While it is not vital that $\mathcal{T}_l$ be built using the most minimal syntax, we do desire *some* syntax describing it, so an explicit construction is useful. We first show a construction for $\mathcal{F}_l(A)$, where $A$ is an arbitrary distributive lattice, and then iterate this construction to build the initial algebra. We begin by defining syntactic *terms over* $A$ representing the three component functors and $\mathcal{F}_l$ using $x_i$ to range over $A$:

$$\begin{aligned}
t_B &::= \perp \mid \mathrm{bool} \mid \top \\
t_F &::= \perp \mid x_1 \to x_2 \mid \top \\
t_R &::= \perp \mid \{\ell_1 : x_1, \dots, \ell_n : x_n\} \mid \top \\
t &::= t \sqcup t \mid t \sqcap t \mid t_B \mid t_F \mid t_R
\end{aligned}$$

Each term $t_B, t_F, t_R$ can be interpreted as an element $[\![t_B]\!]_B$, $[\![t_F]\!]_F$, $[\![t_R]\!]_R$ of $\mathsf{Bool}_l(A)$, $\mathsf{Func}_l(A)$, $\mathsf{Rec}_l(A)$ as follows, writing $*$ for the unique member of the terminal object 1, and $i$ for the inclusion map from $L$ to $L^\top_\perp$:

$$\begin{array}{lll}
[\![\perp]\!]_B = \perp & [\![\mathrm{bool}]\!]_B = i(*) & [\![\top]\!]_B = \top \\
[\![\perp]\!]_F = \perp & [\![x_1 \to x_2]\!]_F = i((x_1, x_2)) & [\![\top]\!]_F = \top \\
[\![\perp]\!]_R = \perp & [\![\{f\}]\!]_R = i(f) & [\![\top]\!]_R = \top
\end{array}$$

We write $\mathsf{inj}_{\{B,F,R\}}$ for the injection maps into the coproduct, and interpret terms $t$ straightforwardly:

$$\begin{array}{ccc}
[\![t_1 \sqcup t_2]\!] = [\![t_1]\!] \sqcup [\![t_2]\!] & & [\![t_1 \sqcap t_2]\!] = [\![t_1]\!] \sqcap [\![t_2]\!] \\
[\![t_B]\!] = \mathsf{inj}_B([\![t_B]\!]_B) & [\![t_F]\!] = \mathsf{inj}_F([\![t_F]\!]_F) & [\![t_R]\!] = \mathsf{inj}_R([\![t_R]\!]_R)
\end{array}$$

However, many terms have the same interpretation, for instance:

$$[\![\mathrm{bool} \sqcup \mathrm{bool}]\!] = [\![\mathrm{bool}]\!]$$

The inclusion map $i$ preserves non-empty meets and joins (it is not a homomorphism because it does not preserve empty meets and joins, due to the addition of new $\perp$ and $\top$ elements). Therefore,

$$[\![(x_1 \to x_2) \sqcup (x_1' \to x_2')]\!] = [\![(x_1 \sqcap x_1') \to (x_2 \sqcup x_2')]\!]$$

In fact, when two terms are equal by any of the rules of Fig. 3.2 then they have equal interpretations, so the elements of $\mathcal{F}_l(A)$ are equivalence classes of terms t quotiented by the smallest equivalence relation including the equations of Fig. 3.2 and those of distributive lattices. We write $\tau$ for these equivalence classes, reserving t for syntactic terms.

Next, we build $\mathcal{T}_l$ by iterating this construction. The elements of $\mathcal{F}_l(0)$ are the equivalence classes of terms containing no nesting of the boolean, function or record type constructors, such as:

$$\bot \to \top$$
$$\text{bool} \sqcap (\top \to \top)$$
$$(\bot \to \top) \sqcup (\text{bool} \sqcap \{\ell : \bot\})$$

Similarly, $\mathcal{F}_l(\mathcal{F}_l(0))$ contains types with booleans, functions or record types nested to a depth of at most 2, such as bool → bool, but not bool → (bool → bool). In general, $\mathcal{F}_l^k(0)$ contains those types representable by terms with at most k levels of nesting, and the initial algebra is given by taking the disjoint union of $\mathcal{F}_l^k(0)$ for all k and removing duplicates.

More formally, we start with the initial object of **DLat**, which is the distributive lattice $0 = \{\top, \bot\}$. For any distributive lattice L, there is a unique morphism $! : 0 \to L$, so we can build the following *direct system*:

$$0 \xrightarrow{!} \mathcal{F}_l(0) \xrightarrow{\mathcal{F}_l(!)} \mathcal{F}_l(\mathcal{F}_l(0)) \xrightarrow{\mathcal{F}_l(\mathcal{F}_l(!))} \dots$$

The *direct limit* of this system is the universal object $\mathcal{T}_l$ having injection morphisms $\mathsf{inj}_k : \mathcal{F}_l^k(0) \to \mathcal{T}_l$ such that:

$$\mathsf{inj}_{k+1} \circ \mathcal{F}_l^k(!) = \mathsf{inj}_k$$

Explicitly, we build $\mathcal{T}_l$ as the disjoint union of $\mathcal{F}_l^k(0)$ for all k:

$$\mathcal{T}_l = \biguplus_k \mathcal{F}_l^k(0) / \sim$$

quotiented by the equivalence relation $\sim$, which is the smallest equivalence relation satisfying:

$$x \sim \mathcal{F}_l^k(!)(x)$$

for all $x \in \mathcal{F}_l^k(0)$.

To reduce clutter, I generally avoid writing the interpretation maps $[\![-]\!]$ from syntactic terms over A to $\mathcal{F}_l(A)$ and the injections $\mathsf{inj}_k$ from $\mathcal{F}^k(0)$ to $\mathcal{T}_l$. Thus, I abuse notation by writing bool → bool for the type $\tau \in \mathcal{T}_l$ more properly denoted:

$$\mathsf{inj}_2\, [\![ [\![\text{bool}]\!] \to [\![\text{bool}]\!] ]\!]$$

The upshot of this is that while we may interpret bits of syntax t as types $\tau \in \mathcal{T}_l$, we cannot assume that each type $\tau$ is the interpretation of a unique bit of syntax. So, we must avoid reasoning by pattern-matching on the syntax of types, unless we are very careful to respect the equivalences between different syntactic representations of the same type.

### 3.2.2 Comparing the lattices

Essentially, $\mathcal{T}_l$ differs from $\mathcal{T}_l'$ by making fewer terms equal to $\bot$ and $\top$. Meets and joins such as $\text{bool} \sqcap (\top \to \bot)$ are freely added to $\mathcal{T}_l$, whereas $\mathcal{T}_l'$ equates them with $\bot$.

Due to its definition as an initial algebra, $\mathcal{T}_l$ has a universal property:

**Proposition 11.** *For any $\mathcal{F}_l$-algebra $(A, f)$, there is a unique monotone function $\alpha : \mathcal{T}_l \to A$ such that:*

$$\begin{aligned} \alpha(\tau_1 \sqcup \tau_2) &= \alpha(\tau_1) \sqcup \alpha(\tau_2) & \alpha(\tau_1 \sqcap \tau_2) &= \alpha(\tau_1) \sqcap \alpha(\tau_2) \\ \alpha(\bot) &= \bot & \alpha(\top) &= \top \\ \alpha(\mathit{bool}) &= \mathit{bool} & \alpha(\tau_1 \to \tau_2) &= \alpha(\tau_1) \to \alpha(\tau_2) \\ \alpha(\{\ell_i : \tau_i\}) &= \{\ell_i : \alpha(\tau_i)\} \end{aligned}$$

The first conditions can be given more concisely by stating that $\alpha$ is a lattice homomorphism (a morphism in **DLat**), rather than just a monotone function.

The lattice $\mathcal{T}_l'$ does not satisfy this universal property, and there may be no lattice homomorphism $\mathcal{T}_l' \to (A, f)$ for a $\mathcal{F}_l$-algebra $(A, f)$. This is the underlying reason for the failure of extensibility we saw in Section 1.4.1: since $\mathcal{T}_l'$ defines the meet of any function type and any record type to be $\bot$, it cannot have a meet-preserving map to a lattice that does not.

While the construction of $\mathcal{T}_l'$ is natural from a syntactic point of view (simply noticing an already-present lattice structure), it is deeply odd from an algebraic one. We re-explain the construction of $\mathcal{T}_l'$ algebraically by starting with a lattice $A$ (initially, the one-element lattice), and iteratively extending it. So, we build the lattices $\mathsf{Bool}_l(A)$, $\mathsf{Func}_l(A)$ and $\mathsf{Rec}_l(A)$. Next, instead of taking their coproduct (as in the construction of $\mathcal{T}_l$), we perform the following steps to make $\mathcal{T}_l'$:

- Forget the lattice structure, taking $\mathsf{Bool}_l(A)$, $\mathsf{Func}_l(A)$ and $\mathsf{Rec}_l(A)$ as mere bounded posets.
- Take a coproduct in $\mathbf{Pos}_\bot^\top$, still ignoring lattice structure.
- Notice the happy accident that the result is a lattice.

Syntactically concise, but not an algebraically satisfying process. In fact, the algebra of $\mathcal{T}_l'$ is remarkably complicated: while calculating meets and joins of elements of $\mathcal{T}_l'$ is fairly straightforward (and can be done efficiently [KPS93]), once type variables and lattice operations are introduced the subtyping problem is not even known to be decidable, despite a vast amount of research on the problem [Pot98b, Reh98, Pri04, SAN$^+$02].

### 3.2.3 Components and coproducts

The lattice $\mathcal{T}_l$ is the initial algebra of $\mathcal{F}_l$, and $\mathcal{T}_l = \mathcal{F}_l(\mathcal{T}_l)$. Expanding definitions, that is:

$$\mathcal{T}_l = 1_\bot^\top + (\mathcal{T}_l^{\mathrm{op}} \times \mathcal{T}_l)_\bot^\top + (\mathcal{L}^{(\mathcal{T}_l^\top)})_\bot^\top$$

This equation is of the form

$$\mathcal{T}_l = \sum_i (C_i)_\bot^\top$$

where $C_i$ ranges over the three *components*: boolean, function and record types. The functor $(-)_\bot^\top$ adds an extra top and bottom element to each component, which are identified by the sum. This ensures that the components are disjoint: the greatest function type is distinguished from the greatest record type. In Section 3.3, we consider adding type variables as separate components, while in Chapter 9 we consider various other type constructors (e.g. sum types) as new components, but the general pattern of defining the lattice of types as a sum of components persists.

Given two types from the same component, it is straightforward to determine their subtyping relationship based on the definition of that component. For instance, given two function types $\tau_1 \to \tau_2$ and $\tau_1' \to \tau_2'$, we know that $\tau_1 \to \tau_2 \leqslant \tau_1' \to \tau_2'$ iff $\tau_1' \leqslant \tau_1$ and $\tau_2 \leqslant \tau_2'$, since the function type component is $\mathcal{T}_l^{\mathrm{op}} \times \mathcal{T}_l$ (that is, two type parameters, with the first contravariant and the second covariant).

However, it is less obvious when the types are from different components. Consider the following general problem, for types $\tau_i, \tau_i'$ drawn from components $C_i$:

$$\bigsqcap_i \tau_i \leqslant \bigsqcup_i \tau_i'$$

Trivially, if $\tau_k \leqslant \tau_k'$ for some $k$ then the above holds, since:

$$\bigsqcap_i \tau_i \leqslant \tau_k \leqslant \tau_k' \leqslant \bigsqcup_i \tau_k'$$

Happily, a standard theorem about the coproduct of distributive lattices is that this holds only in the above trivial case:

**Proposition 12.** *$\bigsqcap_i \tau_i \leqslant \bigsqcup_i \tau_i'$ iff $\tau_k \leqslant \tau_k'$ for some $k$.*

*Proof.* See e.g. Grätzer [Grä09, p.131]. Somewhat confusingly, Grätzer and others refer to the coproduct as the "free product". □

This simple characterisation of subtyping greatly simplifies the problem of deciding subsumption (see Chapter 8), and is a distinct technical advantage of making the subtyping order form a distributive lattice. Other systems, taking a less algebraic and more syntactic approach, were much less simple in this regard. For instance, in the lattice $\mathcal{T}_l'$ described previously, Proposition 12 is violated since the following holds:

$$\mathrm{bool} \leqslant (\top \to \bot) \sqcup \{\ell : \bot\}$$

because the right-hand side is equal to $\top$. This made the subtyping problem inordinately difficult in previous work that took this approach, since subtyping relationships may obtain even though no type constructor appears on both sides of $\leqslant$.

## 3.3 Type variables

The lattice $\mathcal{T}_l$ is distributive, and contains booleans, record and function types with a useful universal property. However, it is not quite enough to treat type variables in a satisfying way.

Suppose we have an $\mathcal{F}_l$-algebra $(A, f)$, and a set of type variables $V$. Type variables are represented abstractly in $A$ by means of some function $i : V \to |A|$, where $|A|$ is the underlying set of the lattice $A$ (that is, the functor $|-|$ is the forgetful functor from **DLat** to **Set**). An *assignment* of variables is another function $\rho : V \to |A|$. The essential property that we need is that any assignment $\rho$, mapping type variables to types, can be uniquely extended to a *substitution* $\hat{\rho}$, mapping types to types. There are two subtly different ways to formalise this, depending on exactly what we allow $\rho(v)$ to range over:

**Definition 13** (Closed-world variable assignment)**.** *For any assignment* $\rho : V \to |A|$*, there is a unique morphism* $\hat{\rho} : (A, f) \to (A, f)$*, such that* $\hat{\rho}(i(v)) = \rho(v)$ *for all variables* $v \in V$ *(or equivalently,* $|\hat{\rho} \cdot i| = \rho$*).*

This definition is "closed-world", because it only allows for quantification over A. If we have a larger structure of types B, this definition gives us no way to assign elements of B to variables. To be extensible, we need to be able to substitute elements of an arbitrary algebra of types for type variables, giving the following open-world version:

**Definition 14** (Open-world variable assignment)**.** *For any* $\mathcal{F}_l$*-algebra* $(B, g)$ *and assignment* $\rho : V \to |B|$*, there is a unique morphism* $\hat{\rho} : (A, f) \to (B, g)$*, such that* $\hat{\rho}(i(v)) = \rho(v)$ *for all variables* $v \in V$ *(or equivalently,* $|\hat{\rho} \cdot i| = \rho$*).*

This is the categorical definition of a *free object*[2]. If A satisfies the open-world variable assignment condition, then A is a *free object over* V.

### 3.3.1 Open versus closed-world type variables

One might wonder why we need to distinguish the open-world and closed-world descriptions of type variables, when no special attention is paid to the fact in most treatments of ML-like languages.

In fact, without subtyping, Herbrand's theorem saves us from needing to draw the distinction. Herbrand's theorem tells us that an existential formula falsifiable at all is falsifiable in a syntactic Herbrand structure, or in other words, that anything that can go wrong under open-world variable assignment can already go wrong in a closed world.

Section 1.4.2 showed that this is not the case once a subtyping lattice is introduced, since Eq. (1.10) is falsified only in an open-world extension of the type system. Herbrand's theorem does not apply, since we are not looking for a counterexample among all logical structures but only among the lattice-ordered ones. Thus, we have to be quite careful about open- versus closed-world variable assignment, since we can no longer rely on their coincidence.

### 3.3.2 Constructing free algebras

As explained in Section 1.4.2, the idea is to extend the syntax of types with explicit type variables. This is an intuitive, but still essentially syntactic approach: we treat variables opaquely by adding them directly to the syntax. Algebraically, adding a set V of variables to the definition of $\mathcal{F}_l$ amounts to working with the initial algebra of the functor $\mathcal{F}_V : \mathbf{DLat} \to \mathbf{DLat}$, defined as:

$$\begin{aligned}\mathcal{F}_V(A) &= \mathcal{F}_l(A) + \mathsf{Free}(V)\\ &= \mathsf{Bool}_l(A) + \mathsf{Func}_l(A) + \mathsf{Rec}_l(A) + \mathsf{Free}(V)\end{aligned}$$

In fact, the initial algebra of $\mathcal{F}_V$ (which we call $\mathcal{T}_V$) is the free object over V in the category of $\mathcal{F}_l$-algebras, called the *free* $\mathcal{F}_l$*-algebra over* V:

**Theorem 15.** *The initial algebra of* $\mathcal{F}_V$ *is the free* $\mathcal{F}_l$*-algebra over* V*.*

*Proof.* Let $(A, f_V)$ be the initial $\mathcal{F}_V$-algebra, and suppose we are given some $\mathcal{F}_l$-algebra $(B, g)$ and a variable assignment $\rho : V \to |B|$.

[2] See Section 2.3.2 for a more introductory description of free objects

By the universal property of $\mathsf{Free}$, there is a unique morphism $\alpha : \mathsf{Free}(V) \to B$ such that $|\alpha| = \rho$, making $(B, g + \alpha)$ a $\mathcal{F}_V$-algebra. By initiality, there is a unique morphism $\hat{\rho} : A \to B$ making this diagram commute:

$$\begin{tikzcd}
\mathcal{F}_V(A) \arrow[r, "f_V"] \arrow[d, "\mathcal{F}_V(\hat{\rho})"] & A \arrow[d, "\hat{\rho}"] \\
\mathcal{F}_V(B) \arrow[r, "g+\alpha"] & B
\end{tikzcd}$$

The morphism $f_V$ is $f + i$, for some $f : \mathcal{F}_l(A) \to A$ and $i : \mathsf{Free}(V) \to A$. The morphism $\mathcal{F}_V(\hat{\rho})$ is $\mathcal{F}_l(\hat{\rho}) + 1$, allowing us to rewrite the above diagram as:

$$\begin{tikzcd}
\mathcal{F}_l(A) \arrow[r, "f"] \arrow[d, "\mathcal{F}_l(\hat{\rho})"] & A \arrow[d, "\hat{\rho}"] & \mathsf{Free}(V) \arrow[l, "i"'] \arrow[d, "1"] \\
\mathcal{F}_l(B) \arrow[r, "g"] & B & \mathsf{Free}(V) \arrow[l, "\alpha"']
\end{tikzcd}$$

This gives $\hat{\rho}$ as the unique $\mathcal{F}$-algebra morphism $(A, f) \to (B, g)$ (the left square) making $\hat{\rho} \cdot i = \alpha$ (the right square) and therefore $|\hat{\rho} \cdot i| = |\alpha| = \rho$. □

The concrete construction of $\mathcal{T}_V$ is the same as that of $\mathcal{T}_l$: the direct limit (that is, disjoint union with duplicates identified) of the following:

$$0 \xrightarrow{!} \mathcal{F}_V(0) \xrightarrow{\mathcal{F}_V(!)} \mathcal{F}_V(\mathcal{F}_V(0)) \xrightarrow{\mathcal{F}_V(\mathcal{F}_V(!))} \dots$$

Just like $\mathcal{T}_l$, the lattice $\mathcal{T}_V$ can be represented as a sum $\sum_i (C_i)^\top_\bot$ of components $C_i$. Since $\mathsf{Free}$ preserves sums (being a left adjoint), and the free (distributive or otherwise) lattice on one generator is $1^\top_\bot$, we have:

$$\mathsf{Free}(V) = \sum_{\nu \in V} 1^\top_\bot$$

So, the components $C_i$ consist of booleans, functions, records, plus one component for each type variable.

### 3.3.3 Properties of substitutions

For any assignment $\rho : V \to \mathcal{T}_V$, the construction above gives a substitution $\hat{\rho} : \mathcal{T}_V \to \mathcal{T}_V$ which allows us to substitute types for type variables. The general categorical formulation of this ensures $\hat{\rho}$ is a morphism of **DLat**, meaning that for any assignment $\rho$,

$$\hat{\rho}(\bot) = \bot \qquad \hat{\rho}(\top) = \top$$
$$\hat{\rho}(\tau_1 \sqcup \tau_2) = \hat{\rho}(\tau_1) \sqcup \hat{\rho}(\tau_2) \qquad \hat{\rho}(\tau_1 \sqcap \tau_2) = \hat{\rho}(\tau_1) \sqcap \hat{\rho}(\tau_2)$$

It would not have been much more effort to prove this by hand, by induction on a syntactic representation of $\mathcal{T}_V$. However, once we extend $\mathcal{T}_V$ with recursive types in the next section, this categorical formulation gives us the above properties without the hassle of syntactic reasoning under $\mu$-binders.

According to Definition 14, the substitution $\hat{\rho}$ corresponding to an assignment $\rho$ is a morphism of $\mathcal{F}_V$-algebras, not just a morphism of **DLat**. In other words, we know that $\hat{\rho}$ respects the algebraic structure, giving the expected properties of substitutions:

$$\begin{aligned}\hat{\rho}(\mathrm{bool}) &= \mathrm{bool} \\ \hat{\rho}(\tau_1 \to \tau_2) &= \hat{\rho}(\tau_1) \to \hat{\rho}(\tau_2) \\ \hat{\rho}(\{\ell_1 : \tau_1, \dots, \ell_n : \tau_n\}) &= \{\ell_1 : \hat{\rho}(\tau_1), \dots, \ell_n : \hat{\rho}(\tau_n)\}\end{aligned}$$

Although we generally take the set $V$ of variables to be finite, there is an easy mechanism to construct fresh variables. The inclusion map from the set $V$ to $V + 1$ gives a function $i : V \to \mathcal{T}_{V+1}$. Since $\mathcal{T}_V$ is a free algebra, we can read $i$ as an assignment and get the substitution $\hat{i} : \mathcal{T}_V \to \mathcal{T}_{V+1}$. In other words, if we need a fresh variable, we can map types from $\mathcal{T}_V$ (the algebra of types with variables drawn from $V$) to $\mathcal{T}_{V+1}$ (the algebra of types with variables drawn from $V$, and one extra).

Having established that these substitutions work the same way as their familiar syntactic counterparts, in the rest of this thesis we will generally gloss over the distinction between an assignment $\rho$ (mapping type variables to types) and its associated substitution $\hat{\rho}$ (mapping types to types). In later chapters, we abuse notation slightly: having defined $\rho$'s action on type variables, we call $\rho$ a "substitution" and write $\rho(\tau)$ for what should technically be $\hat{\rho}(\tau)$.

## 3.4 Recursive types

To interpret the recursive type $\mu\alpha.\mathrm{bool} \to \alpha$, we need to find a fixed point, a solution $\tau$ of $\tau = \mathrm{bool} \to \tau$. It is easy to see that $\mathcal{T}_V$ contains no such solution, so solve such equations we need some *completeness* property.

Throughout this chapter we have avoided making choices as much as possible, deriving the definition of types from the desired components (functions, records and booleans), and the desired properties (first a set, then a poset, then a lattice, and so on.). This leads us to an algebraically simple definition, which as we'll see later on makes type inference much easier.

However, now that we need recursive types, we find that there are multiple ways to proceed and it seems we are forced to choose. We must construct a *completion* of $\mathcal{T}_V$, and there are several ways we might do so:

- By using a *terminal coalgebra* instead of an initial algebra, we admit more elements. Syntactically, this amounts to including elements represented by infinite terms, such as:

$$\mathrm{bool} \to (\mathrm{bool} \to (\mathrm{bool} \to \dots))$$

- By giving $\mathcal{T}_V$ a metric, we may find its *metric completion*, giving a complete metric space where Banach's theorem applies.

- By using the *order completion* of $\mathcal{T}_V$, we can construct a complete lattice, so that the Knaster-Tarski theorem applies.

All of these techniques are useful: infinite syntactic terms give a useful intuition, Banach's theorem proves uniqueness of solutions to equations like $\tau = \mathrm{bool} \to \tau$, and the Knaster-Tarski theorem proves that solutions are least pre-fixed and greatest post-fixed points (as explained in Section 2.1.7, we need both properties to correctly treat the interaction of recursive types and subtyping).

Happily, we need make no choice: all three coincide.

### 3.4.1 Completion via coalgebra

We constructed $\mathcal{T}_V$ as an initial algebra, taking the direct limit of:

$$0 \xrightarrow{!} \mathcal{F}_V(0) \xrightarrow{\mathcal{F}_V(!)} \mathcal{F}_V^2(0) \xrightarrow{\mathcal{F}_V^2(!)} \ldots$$

Thus, $\mathcal{T}_V$ is the universal object with injections $\mathsf{inj}_k : \mathcal{F}_V^k(0) \to \mathcal{T}_l$ such that:

$$\mathsf{inj}_{k+1} \circ \mathcal{F}_V^k(!) = \mathsf{inj}_k$$

Since distributive lattices also have a terminal object $1$ (the one-element lattice $\{* = \top = \bot\}$), with a unique morphism $¡ : 1 \to L$, we can construct the dual of the above, the inverse system:

$$1 \xleftarrow{¡} \mathcal{F}_V(1) \xleftarrow{\mathcal{F}_V(¡)} \mathcal{F}_V^2(1) \xleftarrow{\mathcal{F}_V^2(¡)} \ldots$$

The *terminal coalgebra* $\widetilde{\mathcal{T}_V}$ is the inverse limit of the above system, that is, the universal object having *projection* morphisms $\pi_k : \widetilde{\mathcal{T}_V} \to \mathcal{F}_V^k(1)$ such that:

$$\mathcal{F}_V^k(¡) \circ \pi_{k+1} = \pi_k$$

Explicitly, we construct $\widetilde{\mathcal{T}_V}$ as the subset of the product of $\mathcal{F}^i(1)$ where the components agree:

$$\widetilde{\mathcal{T}_V} = \{\tilde{\tau} \mid \forall k.\ \tilde{\tau}_k \in \mathcal{F}_V^k(1),\ \tilde{\tau}_k = \mathcal{F}_V(¡)(\tilde{\tau}_{k+1})\}$$

Like the elements of $\mathcal{F}_V^k(0)$, the elements of $\mathcal{F}_V^k(1)$ can be represented by finite terms, using the same representation of $\mathcal{F}_V$ described in Section 3.2.1. I continue to use the convention introduced at the end of that section to both omit explicit interpretation maps (writing $\text{bool} \to \text{bool} \in \mathcal{F}_V^2(0)$, even though $\mathcal{F}_V^2(0)$ consists of equivalence classes of syntactic terms) and to omit explicit injections into the direct limit (writing $\text{bool} \to \text{bool} \in \mathcal{T}_V$).

The terms of $\mathcal{F}_V^k(1)$ contain type constructors nested to a depth of at most $k$, so that $* \to * \in \mathcal{F}_V(1)$, and $\top \to (\top \to *) \in \mathcal{F}_V^2(1)$. Intuitively, these are types whose syntax has been truncated at a depth $k$, with any deeper structure replaced with $*$, so we call them *truncations*. The elements $\tilde{\tau}$ of $\widetilde{\mathcal{T}_V}$ are infinite sequences of truncations, where each is a truncation of the next.

There is a canonical morphism $\phi$ from $\mathcal{T}_V$ (the initial algebra of $\mathcal{F}$) to $\widetilde{\mathcal{T}_V}$ (the terminal coalgebra of $\mathcal{F}$), which maps every finite type (element of $\mathcal{T}_V$) to its sequence of truncations (element of $\widetilde{\mathcal{T}_V}$). For example, for a type $\tau = \{\ell_1 : \text{bool} \to \text{bool}\} \sqcup \text{bool}$, $\phi(\tau)$ is:

$$\begin{aligned}
\phi(\tau)_0 &= * \\
\phi(\tau)_1 &= \{\ell_1 : *\} \sqcup \text{bool} \\
\phi(\tau)_2 &= \{\ell_1 : * \to *\} \sqcup \text{bool} \\
\phi(\tau)_3 &= \{\ell_1 : \text{bool} \to \text{bool}\} \sqcup \text{bool} \\
\phi(\tau)_{3+i} &= \phi(\tau)_3
\end{aligned}$$

Since every element $\tau$ of $\mathcal{T}_V$ has a finite syntactic representation, $\phi(\tau)$ is *eventually constant*: $\phi(\tau)_i = \phi(\tau)_j$ for all $i, j$ greater than some $i_0$. However, not all elements of $\widetilde{\mathcal{T}_V}$ are of this form. For instance, $\widetilde{\mathcal{T}_V}$ contains an element $\tilde{\tau}$

given by:

$$\begin{aligned}\tilde{\tau}_0 &= *\\ \tilde{\tau}_1 &= \mathsf{bool} \to *\\ \tilde{\tau}_2 &= \mathsf{bool} \to (\mathsf{bool} \to *)\\ \tilde{\tau}_3 &= \mathsf{bool} \to (\mathsf{bool} \to (\mathsf{bool} \to *)\\ \tilde{\tau}_4 &= \dots\end{aligned}$$

This is a solution to $\tilde{\tau} = \mathsf{bool} \to \tilde{\tau}$, and therefore is not $\phi(\tau)$ for any $\tau$.

**Algebra structure of $\widetilde{\mathcal{T}_V}$** Since $\widetilde{\mathcal{T}_V}$ is the terminal coalgebra of $\mathcal{F}$, Lambek's lemma tells us that $\widetilde{\mathcal{T}_V}$ and $\mathcal{F}_V(\widetilde{\mathcal{T}_V})$ are isomorphic. In particular, there is a morphism $f : \mathcal{F}_V(\widetilde{\mathcal{T}_V}) \to \widetilde{\mathcal{T}_V}$, giving $\widetilde{\mathcal{T}_V}$ the structure of an $\mathcal{F}_V$-*algebra*. This allows us to interpret the type constructors bool, → and $\{\dots\}$ as operations on $\widetilde{\mathcal{T}_V}$, giving e.g. an element $\tilde{\tau_1} \to \tilde{\tau_2} \in \widetilde{\mathcal{T}_V}$ for $\tilde{\tau_1}, \tilde{\tau_2} \in \widetilde{\mathcal{T}_V}$.

Explicitly characterising this morphism is useful. There are maps $\lambda_i : \mathcal{F}_V(\widetilde{\mathcal{T}_V}) \to \mathcal{F}_V^i(1)$ given by:

$$\lambda_i = \begin{cases} ! & \text{if } i = 0\\ \mathcal{F}_V(\pi_k) & \text{if } i = k+1 \end{cases}$$

Since $\widetilde{\mathcal{T}_V}$ is the terminal coalgebra of $\mathcal{F}_V$, this implies there is a morphism $f : \mathcal{F}_V(\widetilde{\mathcal{T}_V}) \to \widetilde{\mathcal{T}_V}$ such that:

$$\mathcal{F}_V(\pi_k) = \pi_{k+1} \circ f$$

Since $f$ is used to interpret the type constructors, this implies that e.g.

$$\pi_{k+1}(\tilde{\tau_1} \to \tilde{\tau_2}) = \pi_k(\tilde{\tau_1}) \to \pi_k(\tilde{\tau_2})$$

Intuitively, we truncate a function type to $k+1$ levels by truncating its domain and result types to $k$ levels.

**Representation as infinite terms** The most direct way to represent the elements of $\widetilde{\mathcal{T}_V}$ syntactically is as infinite sequences of truncations, as above. However, it is also possible to represent the final coalgebra $\widetilde{\mathcal{T}_V}$ by defining syntax *coinductively*. Since the definition of $\mathcal{F}_V$ has not changed, we still use the inductive definition of terms over $A$ from Section 3.2.1 (augmented with type variables as per Section 3.3). Thus, we need a mixture of inductive and coinductive definitions: the structure of $\mathcal{F}_V$ is inductive, and the final coalgebra is coinductive:

$$\left.\begin{aligned} t_B &::= \bot \mid \mathsf{bool} \mid \top\\ t_F &::= \bot \mid t_1 \to t_2 \mid \top\\ t_R &::= \bot \mid \{\ell_1 : t_1, \dots, \ell_n : t_n\} \mid \top \end{aligned}\right\}\text{(coinductively)}$$
$$t ::= t_1 \sqcup t_2 \mid t_1 \sqcap t_2 \mid t_B \mid t_F \mid t_R \mid \alpha \qquad \text{(inductively)}$$

Syntactically, this amounts to representing elements of $\widetilde{\mathcal{T}_V}$ with infinite terms, where only finitely many occurrences of ⊔ and ⊓ are allowed before a type constructor (bool, →, or $\{\dots\}$). Alternatively, infinite paths through the syntax of a term must have infinitely many type constructors.

We only use this representation for illustrative purposes: reasoning formally using it requires an awkward mix of induction and coinduction, and care to respect the equivalence of different syntax for the same type. The algebraic reasoning permitted by the other characterisations of $\widetilde{\mathcal{T}_V}$ is much more direct, despite requiring more work to set up.

### 3.4.2 Completion via metrics

We define a family of equivalence relations $\approx_k$ on $\widetilde{\mathcal{T}_V}$ as follows:

$$\tilde{\tau_1} \approx_k \tilde{\tau_2} \text{ iff } \pi_k\tilde{\tau_1} = \pi_k\tilde{\tau_2}$$

and extend it to $\mathcal{T}_V$:

$$\tau_1 \approx_k \tau_2 \text{ iff } \phi(\tau_1) \approx_k \phi(\tau_2)$$

Intuitively, $\tau_1 \approx_k \tau_2$ if $\tau_1$ and $\tau_2$ are equal to a depth $k$, that is, if it is necessary to look under $k$ layers of type constructors to spot the difference. Note that $\tau_1 \approx_{k+1} \tau_2$ implies $\tau_1 \approx_k \tau_2$ (since equality at $\pi_{k+1}$ implies equality at $\pi_k$ by definition of inverse limits), and if $\tau_1 \approx_k \tau_2$ for all $k$, then $\tau_1 = \tau_2$.

We can use $\approx_k$ to define *ultrametrics* on $\mathcal{T}_V$ and $\widetilde{\mathcal{T}_V}$. An ultrametric $d$ on a set $A$ is a function $d : A \times A \to \mathbb{R}$ such that:

$$\begin{aligned} &d(x,y) \geqslant 0 \\ &d(x,y) = 0 \text{ iff } x = y \\ &d(x,z) \leqslant \mathsf{max}(d(x,y), d(y,z)) \end{aligned}$$

The difference between an ultrametric and an ordinary metric is the use of $\mathsf{max}$ instead of $+$ in the definition above. The ultrametric $d$ on $\mathcal{T}_V$ is defined as:

$$d(\tau_1, \tau_2) = \begin{cases} 0 & \text{if } \tau_1 = \tau_2 \\ 2^{-k} & \text{where } k \text{ is the least natural number such that } \tau_1 \not\approx_k \tau_2 \end{cases}$$

Intuitively, the closer this metric measures $\tau_1$ and $\tau_2$, the greater the depth to which they are equal. The ultrametric on $\widetilde{\mathcal{T}_V}$ is defined identically.

**Completeness and completion** A *Cauchy sequence* in $\widetilde{\mathcal{T}_V}$ is an infinite sequence $\tilde{\tau_i}$ whose elements become arbitrarily close to each other. Formally, the sequence is Cauchy if:

$$\forall \epsilon\ \exists n_0\ \forall i, j > n_0.\ d(\tilde{\tau_i}, \tilde{\tau_j}) < \epsilon$$

or equivalently,

$$\forall k\ \exists n_0\ \forall i, j > n_0.\ \pi_k\tilde{\tau_i} = \pi_k\tilde{\tau_j}$$

Again equivalently, this means that there exists a sequence of truncations $x_i$ such that:

$$\forall k\ \exists n_0\ \forall i > n_0.\ \pi_k\tilde{\tau_i} = x_i$$

However, the sequence $x_i$ determines a member $\tilde{x} \in \widetilde{\mathcal{T}_V}$ (given by $\pi_k\tilde{x} = x_k$), and the above shows that $\lim_i \tilde{\tau_i} = \tilde{x}$. Therefore, every Cauchy sequence in $\widetilde{\mathcal{T}_V}$ converges: $\widetilde{\mathcal{T}_V}$ is a *complete metric space*.

$\mathcal{T}_V$ is not complete, since $\tilde{x}$ may not be of the form $\phi(\tau)$ for $\tau \in \mathcal{T}_V$ (we saw an example of this in the previous section). However, for every $\tilde{\tau} \in \widetilde{\mathcal{T}_V}$, we can find some $\tau \in \mathcal{T}_V$ such that $\phi(\tau) \approx_k \tilde{\tau}$, since the truncation $\pi_k \tilde{\tau}$ is the truncation of at least one $\tau \in \widetilde{\mathcal{T}_V}$ (intuitively, we can take a syntactic representation of any truncation, replace all $*$ with $\top$, and have a syntactic representation of some type with that truncation). Thus, while not all elements of $\widetilde{\mathcal{T}_V}$ are of the form $\phi(\tau)$, they are all within distance $2^{-k}$ of some $\phi(\tau)$, for all $k$.

The map $\phi : \mathcal{T}_V \to \widetilde{\mathcal{T}_V}$ preserves $\approx_k$, and so preserves distances (it is an *isometry*). Since every element of $\widetilde{\mathcal{T}_V}$ is arbitrarily close to the image of $\phi$, we say that the image of $\phi$ is *dense* in $\widetilde{\mathcal{T}_V}$. Since $\widetilde{\mathcal{T}_V}$ is complete, $\phi$ is therefore a *completion* of $\mathcal{T}_V$, so we may equivalently define $\widetilde{\mathcal{T}_V}$ as the metric completion of $\mathcal{T}_V$.

Completions of metric spaces can also be characterised by a universal property. If $A$ is any complete metric space, and $f : \mathcal{T}_V \to A$ is continuous/non-distance-increasing/contractive, then there is a unique $\tilde{f} : \widetilde{\mathcal{T}_V} \to A$ which is also continuous/non-distance-increasing/contractive, such that $\tilde{f} \circ \phi = f$. In other words, any function that specifies values for the dense subset $\phi(\mathcal{T}_V)$ can be extended to specify values for the whole of $\widetilde{\mathcal{T}_V}$.

**Unique fixed points** Since $\widetilde{\mathcal{T}_V}$ is complete, Banach's fixed point theorem proves that every *contractive function* has a unique fixed points. A function is contractive when it brings points closer together by some factor, that is, when there exists some $\lambda < 1$ such that $d(f(x), f(y)) < \lambda d(x, y)$. In terms of the relations $\approx_k$, a function $f : \widetilde{\mathcal{T}_V} \to \widetilde{\mathcal{T}_V}$ is contractive iff:

$$\tilde{\tau}_1 \approx_k \tilde{\tau}_2 \implies f(\tilde{\tau}_1) \approx_{k+1} f(\tilde{\tau}_2)$$

The $\mathcal{F}_V$-algebra structure of $\widetilde{\mathcal{T}_V}$ gives an interpretation of the type constructors bool, → and $\{\dots\}$, mapping $\mathcal{F}_V(\widetilde{\mathcal{T}_V})$ to $\widetilde{\mathcal{T}_V}$. Furthermore, → is a contractive function from $\widetilde{\mathcal{T}_V} \times \widetilde{\mathcal{T}_V}$ to $\mathcal{T}_V$:

**Proposition 16.** *If* $\tau_1 \approx_k \tau_1'$ *and* $\tau_2 \approx_k \tau_2$ *then* $\tau_1 \to \tau_2 \approx_{k+1} \tau_1' \to \tau_2'$

*Proof.* $\pi_{k+1}(\tilde{\tau}_1 \to \tilde{\tau}_2) = \pi_k(\tilde{\tau}_1) \to \pi_k(\tilde{\tau}_2) = \pi_k(\tilde{\tau}_1') \to \pi_k(\tilde{\tau}_2') = \pi_{k+1}(\tilde{\tau}_1' \to \tilde{\tau}_2')$ □

Similarly, record type constructors are contractive. Since $\pi_k$ are lattice homomorphisms, the lattice operations $\sqcup$ and $\sqcap$ preserve $\approx_k$: they are therefore non-distance-increasing functions, which may be composed with a contractive function to yield a contractive function.

So, any function produced by composing $\sqcup$, $\sqcap$ and type constructors is contractive, as long as any uses of the argument to the function are guarded by an occurrence of → or $\{\dots\}$. For instance, consider:

$$f(\tau) = \text{bool} \to \tau$$

Above, $f$ is a contractive function, so it has a unique solution which we call $\mu\alpha.\text{bool} \to \alpha$.

### 3.4.3 Completion via orders

Characterising $\widetilde{\mathcal{T}_V}$ as a metric completion gives us Banach's theorem and unique fixed points of contractive maps. However, it does not quite prove the least pre-fixed point or greatest post-fixed point properties that we require.

$\widetilde{\mathcal{T}_V}$ is the inverse limit of the distributive lattices $\mathcal{F}_V^k(1)$. Since the set of variables $V$ and the set of record labels $\mathcal{L}$ were both assumed to be finite, and the operations used to define $\mathcal{F}_V$ preserve finiteness, each lattice $\mathcal{F}_V^k(1)$ (and $\mathcal{F}_V^k(0)$) is finite. $\widetilde{\mathcal{T}_V}$ is therefore a *profinite distributive lattice*, meaning a distributive lattice arising as the inverse limit of a system of finite distributive lattices.

Profinite distributive lattices are complete, since finite distributive lattices are trivially complete and completeness is preserved by inverse limits. Therefore, the Knaster-Tarski theorem applies to $\widetilde{\mathcal{T}_V}$, giving us least pre-fixed and greatest post-fixed points of monotone functions. For instance, given the monotone function $\alpha \mapsto \mathrm{bool} \to \alpha$, we have least and greatest fixed points which we label $\mu^+\alpha.\mathrm{bool} \to \alpha$ and $\mu^-\alpha.\mathrm{bool} \to \alpha$, and the following reasoning principles:

$$\begin{aligned}\mathrm{bool} \to \tau \leqslant \tau &\implies \mu^+\alpha.\mathrm{bool} \to \alpha \leqslant \tau \\ \tau \leqslant \mathrm{bool} \to \tau &\implies \tau \leqslant \mu^-\alpha.\mathrm{bool} \to \alpha\end{aligned}$$

Since $\alpha \mapsto \mathrm{bool} \to \alpha$ is contractive, we use the results of the previous section to conclude that

$$\mu^+\alpha.\mathrm{bool} \to \alpha = \mu^-\alpha.\mathrm{bool} \to \alpha$$

We write this as simply $\mu\alpha.\mathrm{bool} \to \alpha$. Thus, contractive monotone functions have a fixed point, which is the greatest pre-fixed point, the least post-fixed point, and the unique fixed point, which are all the properties we need to reason about recursive types.

The construction of $\widetilde{\mathcal{T}_V}$ above used ultrametric techniques, and turned out to be a profinite distributive lattice. This was not coincidence: profinite distributive lattices are intimately connected with ultrametrics. We spend the rest of this section exploring this connection, leading to a third characterisation of $\widetilde{\mathcal{T}_V}$: the *initial* algebra of a functor in the category **ProfDLat** of profinite distributive lattices.

**Profiniteness** $\widetilde{\mathcal{T}_V}$ is not just complete, but profinite. The profinite distributive lattices is an unusually well-behaved class having several equivalent characterisations, and is a natural generalisation of finite distributive lattices.

Finite distributive lattices have a very simple representation theory. Given any poset $P$, its *lower sets* are the collection $\mathsf{Down}(P)$ of downwards-closed subsets of $P$. $\mathsf{Down}(P)$ is always a distributive lattice (with meet and join given by intersection and union), and Birkhoff's celebrated representation theorem for finite posets says that all finite distributive lattices arise as $\mathsf{Down}(P)$ for some finite $P$.

Naively generalised to the infinite case, this theorem fails as there are infinite distributive lattices not of the form $\mathsf{Down}(P)$ for any $P$. Priestley generalised in one direction, determining representations for all distributive lattices (which specialise to $\mathsf{Down}(P)$ in the finite case). Generalising in the other direction, determining which distributive lattices are of the form $\mathsf{Down}(P)$, we get exactly the profinite distributive lattices:

**Theorem 17.** *For a distributive lattice $L$, the following conditions are equivalent:*

*(i)* $L$ *is isomorphic to* $\mathsf{Down}(P)$ *for some poset* $P$.

*(ii)* $L$ *is the inverse limit of a system of finite distributive lattices.*

*(iii)* L *can be topologised with a Stone topology.*

*(iv)* L *is completely distributive and algebraic.*

*Proof.* The equivalence of (i), (ii) and (iii) is shown by Johnstone [Joh86, p. 249]. Winskel [Win83] shows the equivalence of (i) and (iv). □

We note that case (iii), topologising L with a Stone topology, is essentially what we were doing by defining an ultrametric on $\widetilde{\mathcal{T}_V}$. A Stone topology is a topology which is *compact*, *Hausdorff* and 0-*dimensional*. The topology induced by an ultrametric is always Hausdorff and 0-dimensional [DG56]. Each of the relations $\approx_k$ has only finitely many equivalence classes, since the equivalence classes are members of $\mathcal{F}_V^k(1)$ which we already noted is finite. In metric terms, this means that for any $k$, there are finitely many elements of $\widetilde{\mathcal{T}_V}$ (representatives of the equivalence classes of $\approx_k$) such that every other $\tilde{\tau} \in \widetilde{\mathcal{T}_V}$ is within distance $2^{-k}$ of one of them, and likewise for $\mathcal{T}_V$. This property is called being *totally bounded*, and it is a standard theorem that a metric space is compact iff it is both complete and totally bounded.

Thus, the ultrametric induces a Stone topology on $\widetilde{\mathcal{T}_V}$ (indeed, the *unique* Stone topology on $\widetilde{\mathcal{T}_V}$, since a lattice can be topologised with a compact Hausdorff topology in at most one way [Joh86, p. 277]).

**Canonical extension** As well as building $\widetilde{\mathcal{T}_V}$ as a terminal coalgebra, or equivalently as the metric completion of $\mathcal{T}_V$, we now show that the same structure arises as the *canonical extension* of $\mathcal{T}_V$, a purely order-theoretic notion.

There are several inequivalent completion procedures in order theory, the best-known of which is the *Dedekind-MacNeille completion* DM (also known as *completion by cuts*), which builds the reals from the rationals. However, DM won't suit our purposes, because DM(L) is not in general a distributive lattice even when L is. In fact, the lattice structure of DM(L) generally has nothing to do with the lattice structure of L: DM is best thought of as a completion from posets to complete lattices (and indeed, is adjoint to the forgetful functor from complete lattices to posets).

There is a more suitable completion procedure: the *canonical extension* CanonExt(L) of a distributive lattice L is complete and preserves all joins and meets from L. More precisely, CanonExt(L) is complete, completely distributive and algebraic (i.e. profinite), and is universal with this property (that is, CanonExt is left adjoint to the forgetful functor from complete, completely distributive algebraic lattices to distributive lattices [GJ94, DHP07]).

**Theorem 18.** $\widetilde{\mathcal{T}_V} = \mathsf{CanonExt}(\mathcal{T}_V)$

*Proof.* We must show that $\widetilde{\mathcal{T}_V}$ satisfies the universal property of $\mathsf{CanonExt}(\mathcal{T}_V)$. That is, we are given a homomorphism $\psi : \mathcal{T}_V \to L$ for some profinite distributive lattice L, and must find a unique complete homomorphism $\alpha : \widetilde{\mathcal{T}_V} \to L$ such that $\alpha \circ \phi = \psi$.

We take L to be topologised as a Stone space, and $\psi$ to be continuous (which we may do by Theorem 17). Since $\widetilde{\mathcal{T}_V}$ is the metric completion of $\mathcal{T}_V$, this gives us a unique continuous $\alpha : \widetilde{\mathcal{T}_V} \to L$ such that $\alpha \circ \phi = \psi$, and it remains only to show that $\alpha$ is a complete homomorphism.

To show $\alpha$ is a complete homomorphism, we must show it preserves all meets and joins. We prove this for joins, as the case for meets is identical. Since $\alpha$ preserves all joins iff it preserves finite and directed ones, and since continuous functions between compact Hausdorff topological lattices always preserve directed joins [Joh86, p. 273], we need only prove $\alpha$ preserves empty and binary joins.

That $\alpha$ preserves empty joins is trivial, since $\phi$ and $\psi$ both preserve $\bot$. For binary joins, consider $\alpha(\tilde{\tau} \sqcup \tilde{\tau}')$. Since $\phi(\mathcal{T}_V)$ is dense in $\widetilde{\mathcal{T}_V}$, $\tilde{\tau}$ and $\tilde{\tau}'$ are given as limits of elements of $\mathcal{T}_V$:

$$\begin{aligned}\tilde{\tau} &= \lim_i \phi(\tau_i)\\ \tilde{\tau}' &= \lim_i \phi(\tau_i')\end{aligned}$$

for some sequences $\tau_i, \tau_i' \in \mathcal{T}_V$. By continuity of $\sqcup$, $\alpha$ and $\psi$, and the fact that $\psi$ is a complete homomorphism:

$$\begin{aligned}\alpha(\tilde{\tau} \sqcup \tilde{\tau}') &= \alpha((\lim_i \phi(\tau_i)) \sqcup (\lim_i \phi(\tau_i')))\\ &= \lim_i \alpha(\phi(\tau_i)) \sqcup \lim_i \alpha(\phi(\tau_i'))\\ &= \lim_i \psi(\tau_i) \sqcup \lim_i \psi(\tau_i')\\ &= \psi(\tilde{\tau}) \sqcup \psi(\tilde{\tau}')\\ &= \psi(\tilde{\tau} \sqcup \tilde{\tau}')\end{aligned}$$

□

**$\widetilde{\mathcal{T}_V}$ as an initial algebra** $\mathsf{CanonExt}$ is a functor, left adjoint to the forgetful functor from profinite distributive lattices to distributive lattices (in fact, it is the same as the *profinite completion* [GJ94, DHP07]). As such, it preserves colimits. Since $\mathcal{T}_V$ is defined as a direct limit (a type of colimit), this allows us to simplify the definition, noting that $\mathsf{CanonExt}(L) = L$ when $L$ is finite:

$$\begin{aligned}\widetilde{\mathcal{T}_V} &= \mathsf{CanonExt}\left(\varinjlim\ 0 \xrightarrow{!} \mathcal{F}_V(0) \xrightarrow{\mathcal{F}_V(!)} \mathcal{F}_V^2(0) \xrightarrow{\mathcal{F}_V^2(!)} \dots\right)\\ &= \varinjlim\ \mathsf{CanonExt}(0) \xrightarrow{!} \mathsf{CanonExt}(\mathcal{F}_V(0)) \xrightarrow{\mathcal{F}_V(!)} \mathsf{CanonExt}(\mathcal{F}_V^2(0)) \xrightarrow{\mathcal{F}_V^2(!)} \dots\\ &= \varinjlim\ 0 \xrightarrow{!} \mathcal{F}_V(0) \xrightarrow{\mathcal{F}_V(!)} \mathcal{F}_V^2(0) \xrightarrow{\mathcal{F}_V^2(!)} \dots\end{aligned}$$

where the latter two limits are taken in the category of $\mathsf{ProfDLat}$ of profinite distributive lattices. For any finite set of variables $V$, this category has free objects $\widetilde{\mathsf{Free}}(V)$ (which are the same as $\mathsf{Free}(V)$ in **DLat**). So, we define the functor $\widetilde{\mathcal{F}_V} : \mathsf{ProfDLat} \to \mathsf{ProfDLat}$ in exactly the same way as $\mathcal{F}_V$:

$$\begin{aligned}\widetilde{\mathsf{Bool}_V}(A) &= 1_\bot^\top\\ \widetilde{\mathsf{Func}_V}(A) &= (A^{\mathrm{op}} \times A)_\bot^\top\\ \widetilde{\mathsf{Rec}_V}(A) &= (\mathcal{L}^{(A^\top)})_\bot^\top\\ \widetilde{\mathcal{F}_V}(A) &= \widetilde{\mathsf{Bool}_V}(A) + \widetilde{\mathsf{Func}_V}(A) + \widetilde{\mathsf{Rec}_V}(A) + \widetilde{\mathsf{Free}}(V)\end{aligned}$$

$\widetilde{\mathcal{T}_V}$ is both the initial algebra and the terminal coalgebra of $\widetilde{\mathcal{F}_V}$.

## 3.5 Summary

Each of the stages in the construction of $\widetilde{\mathcal{T}_V}$ is characterised as the initial algebra of a functor:

| | is the initial algebra of... | adding... |
|---|---|---|
| $\mathcal{T}_\mathrm{s}$ | $\mathcal{F}_\mathrm{s} : \mathbf{Set} \to \mathbf{Set}$ | simple types |
| $\mathcal{T}_\mathrm{o}$ | $\mathcal{F}_\mathrm{o} : \mathbf{Pos} \to \mathbf{Pos}$ | subtyping order |
| $\mathcal{T}_\mathrm{b}$ | $\mathcal{F}_\mathrm{b} : \mathbf{Pos}_\bot^\top \to \mathbf{Pos}_\bot^\top$ | least and greatest types |
| $\mathcal{T}_\mathrm{l}$ | $\mathcal{F}_\mathrm{l} : \mathbf{DLat} \to \mathbf{DLat}$ | distributive subtyping lattice |
| $\mathcal{T}_\mathrm{V}$ | $\mathcal{F}_\mathrm{V} : \mathbf{DLat} \to \mathbf{DLat}$ | type variables |
| $\widetilde{\mathcal{F}_\mathrm{V}}$ | $\widetilde{\mathcal{F}_\mathcal{V}} : \mathbf{ProfDLat} \to \mathbf{ProfDLat}$ | recursive types |

Many of these functors have similar definitions: $\mathcal{F}_\mathrm{s}$ and $\mathcal{F}_\mathrm{o}$ have the same definition (albeit in different categories), as do $\mathcal{F}_\mathrm{b}$ and $\mathcal{F}_\mathrm{l}$ and also $\mathcal{F}_\mathrm{V}$ and $\widetilde{\mathcal{F}_\mathrm{V}}$. The general idea is that we change the definition of $\mathcal{F}$ when we want to add new *elements* to the collection of types: for instance, when adding least and greatest types (moving from $\mathcal{F}_\mathrm{o}$ to $\mathcal{F}_\mathrm{b}$) or when adding type variables (moving from $\mathcal{F}_\mathrm{l}$ to $\mathcal{F}_\mathrm{V}$).

If, on the other hand, we want to demand more *structure* from the collection of types, for instance, that it be a partial order (moving from $\mathcal{F}_\mathrm{s}$ to $\mathcal{F}_\mathrm{o}$), a lattice (moving from $\mathcal{F}_\mathrm{b}$ to $\mathcal{F}_\mathrm{l}$) or closed under recursive equations (moving from $\mathcal{F}_\mathrm{V}$ to $\widetilde{\mathcal{F}_\mathrm{V}}$), then we should keep the same definition of $\mathcal{F}$ but interpret it in a more suitable category. In this fashion, we end up with a collection of types with the right structure, retaining the extensibility properties of being an initial algebra.

# 4 The type system

> *Come to think of it, though, that may just be one of the dangers of the formal approach to thought: it is too easy to be tidy and elegant without having sufficient justification for the particular systematization.*
>
> —Dana Scott

The Hindley-Milner type system occupies a coveted sweet spot in language design: enough polymorphism is available that interesting programs can be written and typechecked, and yet no type annotations are necessary. More formally, the type inference algorithm of the Hindley-Milner system is:

- **Sound**: Type inference generates only type schemes for which a valid derivation exists.

- **Complete**: Every typeable program is ascribed a type scheme by the inference algorithm.

- **Principal**: The type scheme given to a program by type inference is at least as general as any other scheme for the program.

Polymorphism in the Hindley-Milner system distinguishes *types* from *type schemes*. Types are monomorphic, built out of type constructors, variables and base types, but no $\forall$ quantifiers. Type schemes are polymorphic, consisting of a possibly-empty sequence of $\forall$ quantifiers before a type. This form of polymorphism, which allows $\forall$ only at the top level of a type scheme, is called *prenex polymorphism*.

Typing rules for the Hindley-Milner system appear in Figure 4.1. Of several equivalent presentations, we choose one without explicit rules for generalisation and specialisation. Instead, type schemes are introduced by the rule (LET) (hence the alternative name *let-polymorphism* for prenex polymorphism) and eliminated by the rule (VAR-$\forall$). In this way, an expression bound to an identifier by `let` may be used at any monomorphic instantiation of its polymorphic type.

All versions of ML treat variables bound by $\lambda$ and variables bound by let quite differently. The presentation below makes this explicit by using separate syntactic categories for both kinds of variable: variables $x$ are bound by $\lambda$ and given monomorphic types $\tau$, while variables $\hat{x}$ are bound by let and given polymorphic type schemes $\forall\vec{\alpha}.\tau$ where $\vec{\alpha}$ denotes a finite sequence of distinct

Type environments:

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

Typing rules:

$$\begin{array}{lll}
(\textsc{Var-}\lambda) & \dfrac{}{\Gamma \vdash x : \tau} & \Gamma(x) = \tau \\
(\textsc{Var-}\forall) & \dfrac{}{\Gamma \vdash \hat{\mathbf{x}} : \tau[\vec{\tau}/\vec{\alpha}]} & \Gamma(\hat{\mathbf{x}}) = \forall\vec{\alpha}.\tau \\
(\textsc{Abs}) & \dfrac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2} & \\
(\textsc{App}) & \dfrac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 e_2 : \tau_2} & \\
(\textsc{Let}) & \dfrac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, \hat{\mathbf{x}} : \forall\vec{\alpha}.\tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \text{let } \hat{\mathbf{x}} = e_1 \text{ in } e_2 : \tau_2} & \vec{\alpha} \text{ not free in } \Gamma
\end{array}$$

Figure 4.1: Hindley-Milner type system with function types. See text for additional rules concerning record and boolean types and subtyping

type variables. Typing environments $\Gamma$ mix both kinds of variable, mapping $\lambda$-bound variables $x$ to types and let-bound variables $\hat{\mathbf{x}}$ to type schemes.

The language we study is a straightforward extension of ML with subtyping, called MLsub. Types $\tau$ range over the distributive lattice $\widetilde{\mathcal{T}_V}$, defined in the previous chapter, which we refer to from now on simply as $\mathcal{T}$ (leaving the set of type variables $V$ implicit). Expressions $e$ have the following syntax:

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

Above, $x$ and $\hat{\mathbf{x}}$ range over program variables, and $\ell$ ranges over some set $\mathcal{L}$ of record labels. In the record construction syntax $\{\ldots\}$, the $\ell_i$ are assumed distinct.

We begin with the typing rules of the Hindley-Milner type system, as presented in Fig. 4.1. The typing rules of MLsub are these, augmented with a subtyping rule and rules for typing booleans and records, given below. The dynamic semantics are given by a small-step operational semantics, shown in Section 4.1.4. Both the typing rules and the operational semantics are entirely standard, and appear more-or-less verbatim in textbooks [Pie02].

The subtyping rule of MLsub is straightforward, allowing replacement of a type with any supertype, according to the subtyping relation:

$$(\textsc{Sub}) \quad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash e : \tau'} \; \tau \leqslant \tau'$$

Simple as it is, this rule causes difficulties. Unlike the others, this rule is not syntax-directed, so determining exactly when it should be applied is hard. More importantly, the asymmetry of the subtyping relation between types means that unification is not a suitable means of solving constraints between types, breaking Milner's Algorithm W and its descendants.

The major contribution of this thesis is a type system in the Hindley-Milner vein, which includes the above (Sub) rule with a nontrivial subtyping relation, while preserving soundness, completeness and principality of type inference. Despite repeated attempts, this has not been done before.

Presentations of a minimal Hindley-Milner calculus often limit themselves to function types, as the extension to larger type languages (e.g. including sum and product types) presents no new difficulties. However, a subtyping relation including only function types is qualitatively easier to deal with than a more realistic one. So, to make the presentation more interesting, I introduce booleans:

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

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

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

as well as records:

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

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

The typing rules for booleans and records are also entirely standard.

## 4.1 Properties of the type system

Since the typing rules of MLsub are just those of ML augmented with a standard subtyping rule, most of the familiar properties of the ML type system hold for MLsub.

### 4.1.1 Instantiation

The first property we prove is *instantiation*, which allows us to replace type variables with types throughout a type derivation. First, we say that two type schemes are alpha-equivalent if they disagree only on the names of bound variables. That is, $\forall\vec{\alpha}.\tau_1$ and $\forall\vec{\beta}.\tau_2$ are alpha-equivalent when $\tau_2 = \tau_1[\vec{\beta}/\vec{\alpha}]$.

We write $\operatorname{dom}\Gamma$ for the set of $\lambda$- and let-bound variables mentioned in $\Gamma$, and write $\Gamma_1 \equiv \Gamma_2$ if $\Gamma_1$ and $\Gamma_2$ have the same domain, assign equal types to each $\lambda$-bound variable, and assign alpha-equivalent type schemes to each let-bound variable. Typing respects this equivalence relation:

**Proposition 19.** *If* $\Gamma_1 \vdash e : \tau$ *and* $\Gamma_2 \equiv \Gamma_1$*, then* $\Gamma_2 \vdash e : \tau$

*Proof.* Straightforward induction on derivations, noting in case (Var-$\forall$) that $\tau_1[\vec{\tau}/\vec{\alpha}]$ and $\tau_2[\vec{\tau}/\vec{\beta}]$ are equal when $\forall\vec{\alpha}.\tau_1$ and $\forall\vec{\beta}.\tau_2$ are alpha-equivalent. □

We formalise instantiation by applying a type variable substitution to both the environment and the resulting type of a type derivation. We write $[\tau_1/\alpha, \tau_2/\beta]$ for a substitution which maps $\alpha \mapsto \tau_1$, $\beta \mapsto \tau_2$ and $\gamma \mapsto \gamma$ for all other type variables $\gamma$. Per Section 3.3.3, we leave implicit the step of turning an assignment (operating on type variables) into a substitution (operating on types), giving e.g. $[\tau_1/\alpha](\alpha \to \alpha) = \tau_1 \to \tau_1$.

Substitutions are lattice homomorphisms (Section 3.3.3), so they preserve the subtyping relation:

$$\tau_1 \leqslant \tau_2 \quad \Longrightarrow \quad \rho(\tau_1) \leqslant \rho(\tau_2)$$

We extend type variable substitutions $\rho$ to operate on type schemes by defining

$$\rho(\forall\vec{\alpha}.\tau) = \forall\vec{\alpha}.\rho'(\tau)$$

where $\rho'$ is the substitution that is the identity on the variables $\vec{\alpha}$, and agrees with $\rho$ otherwise. We then extend substitutions to operate on whole type environments, writing $\rho(\Gamma)$ for the application of $\rho$ pointwise to the types and type schemes in $\Gamma$.

The instantiation lemma allows us to apply substitutions to both the environment and type of a derivation, preserving validity:

**Lemma 20** (Instantiation). *If* $\Gamma \vdash e : \tau$ *then* $\rho(\Gamma) \vdash e : \rho(\tau)$.

*Proof.* Induction on the derivation, which is straightforward except for the cases involving type variables ((Var-$\forall$) and (Let)).

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

We have $\rho(\Gamma)(\hat{\mathbf{x}}) = \forall\vec{\alpha}.\rho'(\tau)$ where $\rho'$ is as above. We use (Var-$\forall$) to instantiate this type scheme with $\rho(\vec{\tau})/\vec{\alpha}$, giving:

$$\rho(\Gamma) \vdash \hat{\mathbf{x}} : \rho'(\tau)[\rho(\vec{\tau})/\vec{\alpha}]$$

However, since $\rho'$ agrees with $\rho$ on variables not in $\vec{\alpha}$, $\rho'(\tau)[\rho(\vec{\tau})/\vec{\alpha}] = \rho(\tau[\vec{\tau}/\vec{\alpha}])$, so $\rho(\Gamma) \vdash \hat{\mathbf{x}} : \rho(\tau[\vec{\tau}/\vec{\alpha}])$.

Case (Let):

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

Were we to directly apply the inductive hypothesis to give $\rho(\Gamma) \vdash e_1 : \rho(\tau_1)$, we would get stuck trying to apply (Let) afterwards, since even if $\rho(\vec{\alpha})$ are variables, they need not be free in $\rho(\Gamma)$.

Instead, we devise an alternative substitution $\rho_2$ which maps $\vec{\alpha}$ to distinct variables $\vec{\beta}$ not free in $\rho(\Gamma)$, and agrees with $\rho$ on all other variables. Since $\vec{\alpha}$ are not free in $\Gamma$, $\rho_2(\Gamma) = \rho(\Gamma)$. Applying the inductive hypothesis (with $\rho_2$) gives:

$$\rho(\Gamma) \vdash e_1 : \rho_2(\tau_1)$$

Applying the inductive hypothesis (with $\rho$) to the second hypothesis gives:

$$\rho(\Gamma, \hat{\mathbf{x}} : \forall\vec{\alpha}.\tau_1) \vdash e_2 : \rho(\tau_2)$$

Let $\rho'$ be as above so that $\rho(\hat{\mathbf{x}}) = \forall\vec{\alpha}.\rho'(\tau_1)$. Since $\rho'$ and $\rho_2$ both agree with $\rho$ except on $\vec{\alpha}$ (with $\rho'$ being the identity on $\vec{\alpha}$ and $\rho_2$ mapping them to $\vec{\beta}$), $\forall\vec{\alpha}.\rho'(\tau_1)$ and $\forall\vec{\beta}.\rho_2(\tau_1)$ are alpha-equivalent. So, by Proposition 19,

$$\rho(\Gamma), \forall\vec{\beta}.\rho_2(\tau_1) \vdash e_2 : \rho(\tau_2)$$

The rule (Let) now applies, giving the result. □

### 4.1.2 Weakening

The next property is *weakening*, which allows us to weaken any type derivation by replacing $\Gamma$ with some $\Gamma'$ making stronger assumptions. We formalise the strength relationship between two type environments by writing $\Gamma_2 \leqslant \Gamma_1$ whenever:

- $\operatorname{dom}\Gamma_2 \supseteq \operatorname{dom}\Gamma_1$
- $\Gamma_2(\hat{\mathbf{x}})$ and $\Gamma_1(\hat{\mathbf{x}})$ are alpha-equivalent for $\hat{\mathbf{x}} \in \operatorname{dom}\Gamma_1$

- $\Gamma_2(x) \leqslant \Gamma_1(x)$ for $x \in \operatorname{dom} \Gamma_1$

That is, if $\Gamma_2 \leqslant \Gamma_1$ then $\Gamma_2$ has more bindings, and assigns a subtype of $\Gamma_1$'s type to every λ-bound variable in $\Gamma_1$. For let-bound variables, we require $\Gamma_1$ and $\Gamma_2$ to be alpha-equivalent, and we leave subtyping between polymorphic type schemes (*subsumption*) until Section 4.2.

This relation is a preorder:

**Proposition 21.** *$\leqslant$ is a preorder on environments.*

*Proof.* Reflexivity is trivial, and transitivity follows from transitivity of both $\supseteq$ and subtyping. □

By antisymmetry of subtyping, the kernel of this relation is $\equiv$ (that is, if $\Gamma_1 \leqslant \Gamma_2$ and $\Gamma_2 \leqslant \Gamma_1$, then $\Gamma_1 \equiv \Gamma_2$).

The weakening lemma is as follows:

**Lemma 22** (Weakening)**.** *If $\Gamma_1 \vdash e : \tau$ and $\Gamma_2 \leqslant \Gamma_1$, then $\Gamma_2 \vdash e : \tau$*

*Proof.* Induction on the height of the derivation, which is straightforward except for the cases involving variables: (Var-λ), (Var-∀), (Abs) and (Let).

Case (Var-λ): We have $\Gamma_1(x) = \tau$, and $\Gamma_2(x) \leqslant \tau$ since $\Gamma_2 \leqslant \Gamma_1$. So, $\Gamma_2(x) \leqslant \tau$, and the result follows by applying (Var-λ) followed by (Sub).

Case (Var-∀): The type schemes are alpha-equivalent, so the result follows from Proposition 19.

Case (Abs): $\Gamma_2 \leqslant \Gamma_1$ implies $\Gamma_2, x : \tau \leqslant \Gamma_1, x : \tau$, so the inductive hypothesis applies.

Case (Let): Given

$$(\textsc{Let})\quad \frac{\Gamma_1 \vdash e : \tau \quad \Gamma_1, \hat{x} : \forall\vec{\alpha}.\tau \vdash e' : \tau'}{\Gamma_1 \vdash \text{let } \hat{x} = e \text{ in } e' : \tau'}\ \vec{\alpha} \text{ not free in } \Gamma_1$$

We could use the inductive hypothesis to show $\Gamma_2 \vdash e : \tau$, but unfortunately $\vec{\alpha}$ need not be free in $\Gamma_2$ so the side-condition of (Let) would not be satisfied.

Instead, we first apply Lemma 20 with a substitution $\rho$ mapping $\vec{\alpha}$ to distinct variables $\vec{\beta}$ not free in either $\Gamma_1$ or $\Gamma_2$ and other variables to themselves, giving:

$$\Gamma_1 \vdash e : \tau[\vec{\beta}/\vec{\alpha}]$$

Since the construction in Lemma 20 does not change the height of the derivation, we may then apply the inductive hypothesis to get $\Gamma_2 \vdash e : \tau[\vec{\beta}/\vec{\alpha}]$. Since $\forall\vec{\beta}.\tau[\vec{\beta}/\vec{\alpha}]$ is alpha-equivalent to $\forall\vec{\alpha}.\tau$, we have:

$$\Gamma_2, \hat{x} : \forall\vec{\beta}.\tau[\vec{\beta}/\vec{\alpha}] \leqslant \Gamma_1, \hat{x} : \forall\vec{\alpha}.\tau$$

The inductive hypothesis then tells us

$$\Gamma_2, \hat{x} : \forall\vec{\beta}.\tau[\vec{\beta}/\vec{\alpha}] : e' : \tau'$$

from which (Let) applies, giving the result. □

### 4.1.3 Substitution

Since MLsub distinguishes two kinds of variable, there are two substitution lemmas, for λ- and let-bound variables.

Given an environment $\Gamma$, we write $\Gamma_x$ for $\Gamma$ with any bindings for $x$ removed, and similarly $\Gamma_{\hat{x}}$. The first substitution lemma handles λ-bound variables:

**Lemma 23** (Substitution (λ-bound))**.** *If* $\Gamma \vdash e : \tau$, $\Gamma(x) = \tau_x$ *and* $\Gamma_x \vdash e' : \tau_x$ *then* $\Gamma_x \vdash e[e'/x] : \tau$.

*Proof.* Induction on the typing derivation of $e$. Straightforward application of the induction hypothesis in all cases except (Var-λ) and (Abs).

Case (Var-λ): if the variable being typed is $x$, the variable being substituted, then $e[e'/x]$ is $e'$, $\tau = \tau_x$, $\Gamma_x \vdash e' : \tau_x$ by hypothesis, so $\Gamma \vdash e[e'/x] : \tau$ by weakening. If the variable being typed is $y$, some variable other than $x$, then $e[e'/x] = e$ and $\Gamma \vdash e : \tau$ by hypothesis.

Case (Abs): if the variable being abstracted is $x$, the variable being substituted, then $x$ is not free in $e$ so $\Gamma_x \vdash e[e'/x] : \tau$ since $e[e'/x] = e$ and $\Gamma \vdash e : \tau$ by hypothesis. Otherwise, the variable being abstracted is some other variable $y$, and $e = \lambda y.e_1$ and the induction hypothesis applies. □

The second lemma handles let-bound variables, and has a slightly more complex statement because of polymorphism:

**Lemma 24** (Substitution (let-bound))**.** *If* $\Gamma \vdash e : \tau$, $\Gamma(\hat{x}) = \forall\vec{\alpha}.\tau_x$ *and for all* $\vec{\tau}$, $\Gamma_{\hat{x}} \vdash e' : \tau_x[\vec{\tau}/\vec{\alpha}]$, *then* $\Gamma_{\hat{x}} \vdash e[e'/\hat{x}] : \tau$.

*Proof.* Induction on typing derivation of $e$, straightforward except in case (Var-∀) and (Let).

Case (Var-∀): As before, the variable being typed is either the same variable as $\hat{\mathbf{x}}$, or some other variable $\hat{\mathbf{y}}$. If $\hat{\mathbf{x}}$, the hypothesis gives $\Gamma \vdash e : \tau_x[\vec{\tau}/\vec{\alpha}]$, and otherwise $e[e'/\hat{x}] = e$ and the result is trivial.

Case (Let): If the variable being bound is $\hat{\mathbf{x}}$, then $e = \text{let } \hat{\mathbf{x}} = e_1 \text{ in } e_2$ for some $e_1, e_2$. The induction hypothesis applies to the typing of $e_1$, and then (Let) applies giving the result. Otherwise $e = \text{let } \hat{\mathbf{y}} = e_1 \text{ in } e_2$ for some other variable $y$, and the induction hypothesis applies to both $e_1$ and $e_2$, which again gives the result after applying (Let). □

### 4.1.4 Soundness

Next, we show that typeable MLsub programs do not go wrong, with respect to a standard small-step call-by-value operational semantics. The soundness proof is by proving progress and preservation theorems [WF94].

We first identify the *values*, ranged over by $v$, which are those expressions of the following forms:

$$v ::= \mathsf{true} \mid \mathsf{false} \mid \lambda x.e \mid \{\ell_1 = v_1, \ldots, \ell_n = v_n\}$$

Next, we define *reduction contexts* $R$ as follows:

$$\begin{aligned} R ::= \ & \bullet\, e \mid v\, \bullet \mid \\ & \{\ell_1 = v, \ldots, \ell_i = \bullet, \ell_{i+1} = e, \ldots\} \mid \bullet.\ell \mid \\ & \text{if } \bullet \text{ then } e \text{ else } e \\ & \text{let } \hat{x} = \bullet \text{ in } e \end{aligned}$$

We write $R[e]$ for the expression formed by substituting $e$ into the hole in $R$ marked by $\bullet$. Finally, the small-step transition relation is the least relation $\longrightarrow$

including:

$$\begin{aligned}
(\lambda x.e)v &\longrightarrow e[v/x] \\
\{\ell_1 = v_1, \ldots \ell_i = v_i, \ldots \ell_n = v_n\}.\ell_i &\longrightarrow v_i \\
\text{if true then } e_1 \text{ else } e_2 &\longrightarrow e_1 \\
\text{if false then } e_1 \text{ else } e_2 &\longrightarrow e_2 \\
\text{let } \hat{\mathbf{x}} = v \text{ in } e &\to e[v/\hat{\mathbf{x}}] \\
R[e] &\longrightarrow R[e'] \text{ if } e \longrightarrow e'
\end{aligned}$$

Despite the fact that the typing rules and operational semantics of MLsub are both entirely standard, the soundness result does not follow directly, since the definition of types $\mathcal{T} = \widetilde{\mathcal{T}_V}$ is larger than the collection of types usually used. Happily, the standard proof of soundness for ML via progress and preservation theorems turns out not to depend very much on the exact collection of types in use!

The tricky part of a progress and preservation proof is the *inversion lemma*, which proves that, say, a value of type $\tau_1 \to \tau_2$ must be a $\lambda$-abstraction, and generally relies on some inversion principle for types. In the present work, we do not have a strong inversion principle for the subtyping relation: just because $\tau \leqslant \tau_1 \to \tau_2$ does not imply that $\tau$ is a function type (it may be $\bot$, or even something like $(\tau_1 \to \tau_2) \sqcap \beta \sqcap \text{bool}$). However, the following weaker inversion principles turn out to suffice:

**Lemma 25** (Subtype inversion)**.** *If* $\tau_1 \to \tau_2 \leqslant \tau_1' \to \tau_2'$*, then* $\tau_1' \leqslant \tau_1$, $\tau_2 \leqslant \tau_2'$. *If* $\{f\} \leqslant \{g\}$*, then* $f(\ell) \leqslant g(\ell)$ *for* $\ell \in \operatorname{dom} g$.

*Proof.* Direct from the construction of $\widetilde{\mathcal{F}_V}(\widetilde{\mathcal{T}_V})$, which is isomorphic to $\widetilde{\mathcal{T}_V}$. □

**Lemma 26** (Value inversion)**.** *If* $\Gamma \vdash v : s \to t$*, then* $v = \lambda x.e$.
*If* $\Gamma \vdash v :$ *bool, then* $v \in$ *{true, false}.*
*If* $\Gamma \vdash v : \{\ell : t\}$*, then* $v = \{\ldots, \ell = v, \ldots\}$.

*Proof.* Values can only be typed with the rules (Abs), (Bool) or (Cons). Violations of the above condition would require an application of (Sub), which would require a subtyping relationship between a function and a boolean, a function and a record, or a boolean and a record type, all of which are prohibited by Proposition 12 of Section 3.2.3. □

In other words, we have the standard subtyping rules of co/contra-variance for function types and width and depth subtyping for record types, and a limited inversion principle: if we have a subtyping relation between two function types, we may derive subtyping between their domains and ranges, and if we have a subtyping relation between two record types we may derive subtyping between their fields. We also know that there are no subtyping relations between the boolean type and any function type, between any function type and any record type, or between the boolean type and any record type.

Note that we do *not* assume that this necessarily describes all subtyping relations: we may have additional types with unspecified subtyping relations. In particular, given that $\tau' \leqslant \tau_1 \to \tau_2$ we cannot conclude that $\tau'$ is of the form $\tau_1' \to \tau_2'$.

With these lemmas in hand, the actual soundness proof consists of proving progress and preservation theorems.

**Theorem 27** (Progress)**.** *If* $\vdash e : \tau$ *then either* $e$ *is a value, or* $e \longrightarrow e'$ *for some* $e'$.

*Proof.* Induction on the typing derivation of $e$. Cases (Var-$\lambda$) and (Var-$\forall$) are impossible. In cases (Abs), (True) and (False), $e$ is a value. In the other cases:

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

By induction hypotheses, $e_1$ progresses or is a value. If it progresses, then so does $e_1\ e_2$, so assume it is the value $v_1$. Similarly, if $e_2$ progresses then so does $v_1\ e_2$, so assume it is a value. Now we have $v_1\ v_2$, but $v_1 = \lambda x.e_1'$ by value inversion, so the term progresses.

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

By IH, each $e_i$ progresses or is a value. If any progress, then so does $e$, so assume all are values. Therefore $e$ is a value as well.

$$(\textsc{Proj}) \quad \frac{\Gamma \vdash e' : \{\ell : \tau\}}{\Gamma \vdash e'.\ell : \tau}$$

By IH, $e'$ progresses or is a value. If it progresses, then so does $e'$, so assume it is a value. By value inversion, it has a field labelled $\ell$, so $e = e'.\ell$ progresses.

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

By IH, $e_1$ progresses or is a value. If it progresses, so does $e$, so assume it is a value. By value inversion, it is true or false, so $e$ progresses.

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

By IH, $e_1$ progresses or is a value. Either way, $e$ progresses. □

**Theorem 28** (Preservation)**.** *If* $\vdash e : \tau$ *and* $e \longrightarrow e'$, *then* $\vdash e' : \tau$.

*Proof.* Induction on the reduction $e \to e'$, and examination of the typing derivation of $\vdash e : \tau$.

$(\lambda x.e)v \longrightarrow e[v/x]$: The general case of a typing derivation of $(\lambda x.e)v : \tau$ looks like:

$$\cfrac{\cfrac{\cfrac{\cfrac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2}\ (\textsc{Abs})}{\Gamma \vdash \lambda x.e : \tau_3 \to \tau_4}\ (\textsc{Sub}) \qquad \Gamma \vdash v : \tau_3}{\Gamma \vdash (\lambda x.e)v : \tau_4}\ (\textsc{App})}{\Gamma \vdash (\lambda x.e)v : \tau}\ (\textsc{Sub})$$

Since applications of rule (Sub) are not syntax-directed, it may be applied more or fewer times than above. However, any derivation can be brought into the above form where (Sub) is used exactly once in each allowable location, since reflexivity of subtyping allows a use of (Sub) to be inserted and transitivity allows multiple uses to be combined.

The side-conditions of (Sub) specify:

$$\tau_1 \to \tau_2 \leqslant \tau_3 \to \tau_4 \qquad\qquad \tau_4 \leqslant \tau$$

By Lemma 25, $\tau_2 \leqslant \tau_4$ and $\tau_3 \leqslant \tau_1$, so by subtyping (i.e. applying (Sub))) we have

$$\Gamma, x : \tau_1 \vdash e : \tau \qquad\qquad \Gamma \vdash v : \tau_1$$

from which Lemma 23 gives $\Gamma \vdash e[v/x] : \tau$.

$\{\ell_1 = v_1, \ldots \ell_n = v_n\}.\ell_i \longrightarrow v_i$: Typing derivations look like this (with (Sub) again applied exactly once in each possible location):

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash v_1 : \tau_1 \quad \ldots \quad \Gamma \vdash v_i : \tau_i \quad \ldots \quad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash \{\ell_1 = v_1, \ldots \ell_i = v_i, \ldots \ell_n = v_n\} : \{f\}}\;(\textsc{Cons})}{\Gamma \vdash \{\ell_1 = v_1, \ldots \ell_i = v_i, \ldots \ell_n = v_n\} : \{g\}}\;(\textsc{Sub})}{\Gamma \vdash \{\ell_1 = v_1, \ldots \ell_i = v_i, \ldots \ell_n = v_n\}.\ell_i : \tau'}\;(\textsc{Proj})}{\Gamma \vdash \{\ell_1 = v_1, \ldots \ell_i = v_i, \ldots \ell_n = v_n\}.\ell_i : \tau}\;(\textsc{Sub})$$

The side-conditions of (Sub) are that $\tau' \leqslant \tau$ and $\{f\} \leqslant \{g\}$. Since $\ell_i$ is in the domain of both,

$$\tau_i = f(\ell_i) \leqslant g(\ell_i) = \tau' \leqslant \tau$$

so by subtyping, $\Gamma \vdash v_i : \tau$.

if b then $e_1$ else $e_2 \longrightarrow e_i$: Typing derivations are (again with (Sub) inserted exactly once where possible):

$$\dfrac{\dfrac{\Gamma \vdash b : \mathrm{bool} \quad \Gamma \vdash e_1 : \tau' \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash \text{if } b \text{ then } e_1 \text{ else } e_2 : \tau'}\;(\textsc{If})}{\Gamma \vdash \text{if } b \text{ then } e_1 \text{ else } e_2 : \tau}\;(\textsc{Sub})$$

By side-condition of (Sub), $\tau \leqslant \tau'$, so $\Gamma \vdash e_i : \tau$.

let $\hat{\mathbf{x}} = v$ in $e \to e[v/\hat{\mathbf{x}}]$: Typing derivations are (again with (Sub) inserted exactly once where possible):

$$\dfrac{\dfrac{\Gamma \vdash v : \tau_1 \quad \Gamma, \hat{\mathbf{x}} : \forall\vec{\alpha}.\tau_1 \vdash e : \tau_2}{\Gamma \vdash \text{let } \hat{\mathbf{x}} = v \text{ in } e : \tau_2}\;(\textsc{Let})}{\Gamma \vdash \text{let } \hat{\mathbf{x}} = v \text{ in } e : \tau}\;(\textsc{Sub})$$

Instantiation (Lemma 20) proves that $\Gamma \vdash v : \tau_1[\vec{\tau}/\vec{\alpha}]$ by choosing $\rho$ such that $\rho(\vec{\alpha}) = \vec{\tau}$ (note $\rho(\Gamma) = \Gamma$ since $\vec{\alpha}$ are not free in $\Gamma$), and the result follows by applying Lemma 24 and (Sub) (since $\tau_2 \leqslant \tau$).

$R[e] \longrightarrow R[e']$: The typing derivation of $\vdash R[e] : \tau$ has $\vdash e : \tau'$ as a hypothesis for some $\tau'$, by case analysis of possible $R$. Replacing this hypothesis with $\vdash e' : \tau'$ (from the induction hypothesis) gives $\vdash R[e'] : \tau$.

□

## 4.2 Typing schemes and subsumption

Until this point, we have ignored the important question of *subsumption*, the analogue of subtyping between type schemes, since the relation $\leqslant$ relates two environments only when they assign alpha-equivalent type schemes to let-bound variables. This is a subtle affair, since subsumption must combine both ways in which types are manipulated by the type system: *instantiation*, as done by rule (Var-∀), and *subtyping*, as done by rule (Sub).

We tackle this in two parts. In this part, we introduce *typing schemes* which describe the interaction of instantiation and subtyping in the absence of free let-bound variables. In Section 4.3, we then show how these typing schemes are used to build the *reformulated typing rules* for MLsub, a type system equivalent to the original but based on typing schemes instead of type schemes. The inference algorithm of Chapter 6 and the subsumption algorithm of Chapter 8 are both based on typing schemes.

First, we introduce *monotype environments* $\Delta$ which are type environments binding no let-bound variables (and so lacking $\forall$-quantifiers):

$$\Delta ::= \epsilon \mid \Delta, x : \tau$$

Monotype environments inherit the ordering $\leqslant$ of standard environments. Note that two monotype environments $\Delta_1$, $\Delta_2$ have a greatest lower bound $\Delta_1 \sqcap \Delta_2$ in this ordering which is also a monotype environment, where $\operatorname{dom}(\Delta_1 \sqcap \Delta_2) = \operatorname{dom}\Delta_1 \cup \operatorname{dom}\Delta_2$, and $(\Delta_1 \sqcap \Delta_2)(x) = \Delta_1(x) \sqcap \Delta_2(x)$, interpreting $\Delta_i(x) = \top$ if $x \notin \operatorname{dom}\Delta_i$ (for $i \in \{1, 2\}$).

A *typing scheme* $[\Delta]\tau$ is a pair of a monotype environment $\Delta$ and a type $\tau$, representing both the type of an expression and the types of its free $\lambda$-bound variables. Substitutions can be applied to typing schemes by applying them to both components, so $\rho([\Delta]\tau)$ is $[\rho(\Delta)]\rho(\tau)$. We extend the subtyping relation $\leqslant$ to typing schemes, covariantly in $\tau$ and contravariantly in $\Delta$:

$$[\Delta_2]\tau_2 \leqslant [\Delta_1]\tau_1 \text{ iff } \tau_2 \leqslant \tau_1,\ \Delta_1 \leqslant \Delta_2$$

However, there is a more subtle ordering we place on typing schemes called *subsumption*, which allows for the instantiation of type variables as well as subtyping. The subsumption relation $\leqslant^\forall$ is defined as follows:

$$[\Delta_2]\tau_2 \leqslant^\forall [\Delta_1]\tau_1 \text{ iff } \rho([\Delta_2]\tau_2) \leqslant [\Delta_1]\tau_1 \text{ for some substitution } \rho$$

In other words, $[\Delta_2]\tau_2$ subsumes $[\Delta_1]\tau_1$ if some substitution instance of $[\Delta_2]\tau_2$ is a subtype of $[\Delta_1]\tau_1$.

**Proposition 29.** *$\leqslant^\forall$ is a preorder.*

*Proof.* $\leqslant^\forall$ is trivially reflexive. For transitivity, assume $[\Delta_3]\tau_3 \leqslant^\forall [\Delta_2]\tau_2$ and $[\Delta_2]\tau_2 \leqslant^\forall [\Delta_1]\tau_1$. We have, by monotonicity of substitutions:

$$\rho([\Delta_3]\tau_3) \leqslant [\Delta_2]\tau_2,\ \rho'([\Delta_2]\tau_2) \leqslant [\Delta_1]\tau_1 \implies \rho'(\rho([\Delta_3]\tau_3)) \leqslant [\Delta_1]\tau_1$$

and so $[\Delta_3]\tau_3 \leqslant^\forall [\Delta_1]\tau_1$ using the substitution $\rho' \circ \rho$. □

The subsumption relation $\leqslant^\forall$ can alternatively be characterised as the smallest order including both subtyping and instantiation.

Lemmas 20 and 22 prove that weakening and instantiation preserve typeability, implying the following proposition:

**Proposition 30.** *For let-free $e$, if $\Delta_1 \vdash e : \tau_1$ and $[\Delta_1]\tau_1 \leqslant^\forall [\Delta_2]\tau_2$, then $\Delta_2 \vdash e : \tau_2$.*

This definition of subsumption is simpler than the subsumption relation used by Trifonov and Smith [TS96] and Pottier [Pot98b], which has more the flavour of a simulation relation, requiring that for every instance of $[\Delta_1]\tau_1$ there is an instance of $[\Delta_2]\tau_2$ which is a subtype:

$$[\Delta_2]\tau_2 \leqslant^\forall_{\text{sim}} [\Delta_1]\tau_1 \text{ iff } \forall\rho'\exists\rho.\ \rho([\Delta_2]\tau_2) \leqslant \rho'([\Delta_1]\tau_1)$$

Thanks to the algebraic construction of $\mathcal{T}$, the two definitions are equivalent:

**Proposition 31.** *$\leqslant^{\forall}$ and $\leqslant^{\forall}_{sim}$ are the same relation.*

*Proof.* Suppose $[\Delta_2]\tau_2 \leqslant^{\forall} [\Delta_1]\tau_1$, so there is some $\rho_0$ such that $\rho_0([\Delta_2]\tau_2) \leqslant [\Delta_1]\tau_1$. For any $\rho'$, take $\rho = \rho' \cdot \rho_0$, and $\rho([\Delta_2]\tau_2) \leqslant \rho'([\Delta_1]\tau_1)$ follows by monotonicity of substitution. Alternatively, supposing $[\Delta_2]\tau_2 \leqslant^{\forall}_{\mathrm{sim}} [\Delta_1]\tau_1$, take $\rho'$ to be the identity substitution, giving a $\rho$ such that $\rho([\Delta_2]\tau_2) \leqslant [\Delta_1]\tau_1$. □

This result did not hold for previous formulations of the lattice of types, as the proof uses the identity substitution which maps each type variable to itself. This requires that the type variables have a presence of their own in the subtyping lattice, rather than quantifying over ground types (see Section 3.3 for a thorough discussion). With the extensible formulation as a free algebra, subsumption as simulation and subsumption as generalisation plus subtyping coincide.

### 4.2.1 Equivalence of typing schemes

The subsumption relation $\leqslant^{\forall}$ is a preorder, not a partial order, and it induces a nontrivial equivalence relation $\equiv^{\forall}$. We leave the algorithm for deciding $\leqslant^{\forall}$ and $\equiv^{\forall}$ until Chapter 8. For now, we merely give some examples of the non-triviality.

Any two typing schemes which are alpha-equivalent subsume each other, since each is a substitution instance of the other. In effect, typing schemes are always *closed*, never having free type variables. Relations between a typing scheme and its environment are instead expressed using the $\Delta$ part of a typing scheme $[\Delta]\tau$.

However, typing schemes can be equivalent by $\equiv^{\forall}$ without being alpha-equivalent. Suppose we have a function first which takes two (curried) arguments and returns the first. The second argument, therefore, is unused and does not affect the result type. ML expresses this using polymorphism, giving a typing scheme like:

$$\mathsf{first} : [\,]\alpha \to \beta \to \alpha$$

The same effect can be expressed with subtyping, giving instead:

$$\mathsf{first} : [\,]\alpha \to \top \to \alpha$$

We need not choose: these are equivalent by $\equiv^{\forall}$. The second is an instantiation of the first, while the first is a subtype of the second.

Suppose instead we have the function choose which again takes two arguments, but returns one or the other randomly. The ML-style typing scheme is now:

$$\mathsf{choose} : [\,]\alpha \to \alpha \to \alpha$$

while the subtyping-style typing scheme is:

$$\mathsf{choose} : [\,]\beta \to \gamma \to (\beta \sqcup \gamma)$$

The latter directly expresses the subtyping constraints: choose may take arguments at two different types, but will return their upper bound. So, the second, more complicated type may appear to be more general, and indeed it subsumes the first by instantiation.

Perhaps surprisingly, the first also subsumes the second, making both equivalent! 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 the instantiation $\alpha = \beta \sqcup \gamma$ suffices. Intuitively, these typing schemes are equivalent because they describe the same data flow: both allow input to flow from either argument to the result.

So, the type inference algorithm for MLsub sees both of these as equivalent. Indeed, the efficient automaton-based representation of typing schemes introduced in Chapter 7 represents both typing schemes for choose identically, while Section 7.3.3 shows how the conversion from automata back to syntactic types picks the simpler, former typing scheme for choose for display to the user.

## 4.3 Reformulated typing rules

Next, we reformulate the typing rules of MLsub into an equivalent form based on typing schemes $[\Delta]\tau$ rather than on type schemes $\forall\vec{\alpha}.\tau$.

The standard typing rules record constraints between a type and the environment by reusing the same type variable in both, which necessitates a careful treatment of free and bound type variables (notably in the side-condition of (Let)). Instead, we consider typing schemes to always be *closed*, that is, never having any free variables. Constraints between a type and the environment are recorded using the $\Delta$ part of a typing scheme $[\Delta]\tau$, relying on coincidence of program variable names $x$ rather than type variable names $\alpha$.

Describing a type system in this style is not novel, but what is new is that, thanks to the lack of explicit constraints, we can prove this equivalent to the standard ML-style formulation (in Section 4.3.1). Therefore, in this work the reformulated rules are not a type system defined oddly, but simply an alternative representation that clarifies the process of type inference (Chapter 6).

Instead of type environments $\Gamma$, which map variables to types or type schemes, the reformulated rules use *typing environments* $\Pi$ to assign typing schemes (not type schemes) to let-bound variables:

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

The reformulated rules produce judgements of the form $\Pi \Vdash e : [\Delta]\tau$, and are shown in their entirety in Fig. 4.2. For these typing rules, I assume a prior stage of alpha-renaming that ensures all $\lambda$- and let-binders bind distinct names. Alternatively, de Bruijn indices can be used, or some tedious relabelling can be done during typing to keep track of the domains of various $\Pi$ and $\Delta$.

The role of instantiation is played by the (Sub) rule since the relation $\leqslant^{\forall}$ includes substitution of types for type variables. Note the lack of side-conditions about free type variables when generalising in the (Let)-rule: mentions of a type variable $\alpha$ in one typing scheme $[\Delta]\tau$ have no relation to mentions in any other, and the connection between constraints on the same program variable $x$ arising from different parts of the program is maintained by the presence of $x$ in multiple $\Delta$, not by coincidence of type variable names. Accordingly, freshening of type variables which is usually done in rule (Var) by systems based on ML can in fact be done at any time using (Sub) and an substitution that permutes variables.

As with the original typing rules, these reformulated rules allow weakening. We write $\Pi_2 \leqslant^{\forall} \Pi_1$ when $\operatorname{dom}\Pi_2 \supseteq \operatorname{dom}\Pi_1$ and $\Pi_2(\hat{x}) \leqslant^{\forall} \Pi_1(\hat{x})$ for $\hat{x} \in \operatorname{dom}\Pi_1$.

$$(\textsc{Var-}\Pi)\quad \frac{}{\Pi \Vdash \hat{x} : [\Delta]\tau}\ \Pi(\hat{x}) = [\Delta]\tau$$

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

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

$$(\textsc{App})\quad \frac{\Pi \Vdash e_1 : [\Delta]\tau \to \tau' \quad \Pi \Vdash e_2 : [\Delta]\tau}{\Pi \Vdash e_1\ e_2 : [\Delta]\tau'}$$

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

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

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

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

$$(\textsc{Cons})\quad \frac{\Pi \Vdash e_1 : [\Delta]\tau_1 \quad \dots \quad \Pi \Vdash e_n : [\Delta]\tau_n}{\Pi \Vdash \{\ell_1 = e_1, \dots, \ell_n = e_n\} : [\Delta]\{\ell_1 : \tau_1, \dots, \ell_n : \tau_n\}}$$

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

$$(\textsc{Sub})\quad \frac{\Pi \Vdash e : [\Delta]\tau}{\Pi \Vdash e : [\Delta']\tau'}\ [\Delta]\tau \leqslant^{\forall} [\Delta']\tau'$$

Figure 4.2: Reformulated MLsub typing rules

**Proposition 32** (Weakening for $\Vdash$)**.** *If* $\Pi_1 \Vdash e : [\Delta]\tau$ *and* $\Pi_2 \leqslant^{\forall} \Pi_1$*, then* $\Pi_2 \Vdash e : [\Delta]\tau$.

*Proof.* By straightforward induction on derivations, using rule ($\textsc{Sub}$) in case ($\textsc{Var-}\Pi$). □

### 4.3.1 Equivalence of original and reformulated rules

These rules are equivalent to the original rules, in that

$$\vdash e : \tau \text{ iff } \Vdash e : [\,]\tau$$

To show this we need a means of converting between $\Gamma$-style environments and $\Pi$-style environments. First, we show how a type environment $\Gamma$ may be split into a monomorphic part $m(\Gamma)$ and a polymorphic part $p(\Gamma)$:

$$\begin{aligned} m(\epsilon) &= \epsilon & p(\epsilon) &= \epsilon \\ m(\Gamma, x : \tau) &= m(\Gamma), x : \tau & p(\Gamma, x : \tau) &= p(\Gamma) \\ m(\Gamma, \hat{x} : \forall\vec{\alpha}.\tau) &= m(\Gamma) & p(\Gamma, \hat{x} : \forall\vec{\alpha}.\tau) &= p(\Gamma), x : [m(\Gamma)]\tau \\ & & &\text{(provided } \vec{\alpha} \text{ not free in } \Gamma) \end{aligned}$$

The last case's requirement that $\vec{\alpha}$ not be free in $\Gamma$ can always be satisfied by renaming the bound variables $\vec{\alpha}$ in $\forall\vec{\alpha}.\tau$, if necessary.

**Lemma 33.** *If* $\Gamma \vdash e : \tau$*, then* $p(\Gamma) \Vdash e : [m(\Gamma)]\tau$

*Proof.* Induction on the derivation of $\Gamma \vdash e : \tau$. Most cases are straightforward, since they do not manipulate the environment and a direct analogue exists in $\Vdash$. The exceptions are ($\textsc{Var-}\lambda$), ($\textsc{Var-}\forall$), ($\textsc{Abs}$), ($\textsc{Let}$) and ($\textsc{Sub}$).

Case (Var-$\lambda$): if $\Gamma(x) = \tau$, then $m(\Gamma)(x) = \tau$ so $[x : \tau]\tau \leqslant^{\forall} [m(\Gamma)]\tau$ and the result follows by applying (Var-$\Delta$) and (Sub).

Case (Var-$\forall$): We have $\Gamma(x) = \forall\vec{\alpha}.\tau$, and we assume $\vec{\alpha}$ are not free in $\Gamma$, possibly by renaming the bound variables. Then, $[m(\Gamma)]\tau \leqslant^{\forall} [m(\Gamma)]\tau[\vec{\tau}/\vec{\alpha}]$, and since $p(\Gamma)(\hat{\mathbf{x}}) \leqslant^{\forall} [m(\Gamma)]\tau$ the result follows by (Var-$\Gamma$) and (Sub).

Case (Abs): Since $p(\Gamma, x : \tau) = \Gamma$ and $m(\Gamma, x : \tau) = m(\Gamma), x : \tau$, the result follows directly by applying (Abs).

Case (Let): Given an application of (Let):

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

we have by the induction hypothesis (noting how the side-condition means $p(\Gamma, x : \forall\vec{\alpha}.\tau_1)$ need perform no renaming):

$$p(\Gamma) \Vdash e_1 : [m(\Gamma)]\tau_1 \qquad p(\Gamma), x : [m(\Gamma)]\tau_1 \Vdash e_2 : [m(\Gamma)]\tau_2$$

and applying (Let) gives the result.

Case (Sub): If $\tau \leqslant \tau'$, then also $[m(\Gamma)]\tau \leqslant^{\forall} [m(\Gamma)]\tau'$, so the (Sub) rule can be directly applied. □

For the converse, an environment $\Gamma$ and a monomorphic environment $\Delta$ may be combined into a single environment $\Gamma \sqcap \Delta$, whose domain is their domains' union, and where $(\Gamma \sqcap \Delta)(x) = \Gamma(x) \sqcap \Delta(x)$ for $\lambda$-bound variables common to both domains. This allows us to combine separate $\Pi$ and $\Delta$ into a single environment as follows:

$$\begin{aligned} r(\epsilon) &= \epsilon \\ r(\Pi, \hat{\mathbf{x}} : [\Delta]\tau) &= (r(\Pi) \sqcap \Delta), x : \forall\vec{\alpha}.\tau \\ &\qquad (\vec{\alpha} \text{ are those variables free in } \tau \text{ but not } \Delta) \end{aligned}$$

**Lemma 34.** *If* $\Pi \Vdash e : [\Delta]\tau$, *then* $r(\Pi) \sqcap \Delta \vdash e : \tau$.

*Proof.* Induction on the derivation of $\Pi \Vdash e : [\Delta]\tau$. Again, the result is trivial for those rules that do not touch the environment and have direct analogues in $\vdash$, leaving (Var-$\Delta$), (Var-$\Pi$), (Abs), (Let), (Sub).

Case (Var-$\Delta$): $(r(\Pi) \sqcap (x : \tau))(x) \leqslant \tau$, so the result follows by (Var-$\lambda$) and (Sub).

Case (Var-$\Pi$): If $\Pi(\hat{\mathbf{x}}) = [\Delta]\tau$, then $(r(\Pi) \sqcap \Delta)(\hat{\mathbf{x}}) = \forall\vec{\alpha}.\tau$, where $\vec{\alpha}$ are the variables free in $\tau$ but not $\Delta$, so the result follows by (Var-$\forall$), where each type variable is replaced with itself.

Case (Abs): By the induction hypothesis, $r(\Pi) \sqcap (\Delta, x : \tau) \vdash e : \tau'$. By the assumption of distinct variable names, $x$ must not be in the domain of $r(\Pi)$ or $\Delta$, so $(r(\Pi) \sqcap (\Delta, x : \tau)) = (r(\Pi) \sqcap \Delta), x : \tau$ and (Abs) applies.

Case (Let): By the induction hypothesis,

$$r(\Pi) \sqcap \Delta_1 \vdash e_1 : \tau_1 \qquad r(\Pi) \sqcap \Delta_1 \sqcap \Delta_2, x : \forall\vec{\alpha}.\tau_1 \vdash e_2 : \tau_2$$

where $\vec{\alpha}$ are those variables free in $\tau'$ but not $\Delta$. By weakening, $r(\Pi) \sqcap \Delta_1 \sqcap \Delta_2 \vdash e_1 : \tau_1$, and so (Let) applies, giving the result.

Case (Sub): By the induction hypothesis, $r(\Pi) \sqcap \Delta \vdash e : \tau$ and $[\Delta]\tau \leqslant^{\forall} [\Delta']\tau'$, so $\rho([\Delta]\tau) \leqslant [\Delta']\tau'$ for some $\rho$. By Lemma 20, we therefore have $\rho(r(\Pi) \sqcap \Delta) \vdash e : \rho(\tau)$. Since typing schemes are closed (that is, do not have free variables), we may assume (by renaming if necessary) that the variables mentioned in $[\Delta]\tau$ are disjoint from those mentioned in $\Pi$, and thus that $\rho(r(\Pi) \sqcap \Delta) = r(\Pi) \sqcap \rho(\Delta) \leqslant r(\Pi) \sqcap \Delta'$. So, by weakening and subtyping, $r(\Pi) \sqcap \Delta' \vdash e : \tau'$. □

**Theorem 35.** $\vdash e : \tau$ *iff* $\Vdash e : []\tau$

*Proof.* Using the above two lemmas for the two directions, noting that $p(\epsilon) = \epsilon$, $m(\epsilon) = \epsilon$ and $r(\epsilon) = \epsilon$. □

These reformulated rules are at the core of the type inference algorithm in Chapter 6.

# 5 Polarity and biunification

*How can intuition deceive us on this point?*

—Henri Poincaré

MLsub is defined in terms of the lattice $\mathcal{T} = \widetilde{\mathcal{T}_V}$, which is algebraically well-behaved: it is distributive, an initial algebra, and so on. However, $\mathcal{T}$ is uncountable (due to the presence of infinite types), and contains many odd-looking types such as:

$$\mathsf{bool} \sqcup ((\bot \to \bot) \sqcap \{\mathtt{a} : \mathsf{bool}\})$$

The purpose of this chapter is to carve out a subset of $\mathcal{T}$ called the *polar types*, and show that they support an analogue of unification we call *biunification*. The polar types are defined syntactically, and there are only countably many of them.

Not all types (i.e. elements of $\mathcal{T}$) can be written as polar types. However, restricting to polar types loses no expressiveness, because of the following central result:

*The principal type of any typeable expression is a polar type.*

This result is stated and proved in detail in Chapter 6, and the proof is by construction: the type inference algorithm produces polar principal types.

The relationship between types (elements of $\mathcal{T}$) and polar types is similar to that between the real numbers and pairs of integers representing rationals. The real numbers have useful completeness properties, allowing the taking of limits (while $\mathcal{T}$ is a complete lattice, allowing the taking of arbitrary meets and joins), but reals are uncountably many and difficult to compute with directly (again, just like $\mathcal{T}$). On the other hand, pairs of integers $\frac{\mathrm{m}}{\mathrm{n}}$ are easy to compute with (as are polar types), although some operations like the cube root are not always defined (like polar types, which have certain restrictions on taking meets and joins). Finally, while every fraction $\frac{\mathrm{m}}{\mathrm{n}}$ can be interpreted as a real (and every polar type as an element of $\mathcal{T}$), there are generally multiple representations for a given number: $\frac{1}{2}$ and $\frac{2}{4}$ have the same interpretation as reals (similarly, there are syntactically distinct polar types whose interpretations in $\mathcal{T}$ agree).

The central idea of polar types is to distinguish between the types used to describe inputs (called *negative types*) and those used to describe outputs (*positive types*). Due to subtyping, output types represent lower bounds (a program produces a $\tau$, which may be used at any supertype of $\tau$), while input types represent upper bounds (a program requires a $\tau$, and any subtype of $\tau$ may be provided).

For safety of execution, each subexpression of a program constrains its inputs and outputs in some way, and to check types is to check the conjunction of those constraints. In a lattice, the conjunction of lower bound constraints is described with $\sqcup$, while the conjunction of upper bounds is described with $\sqcap$:

$$a \leqslant x \text{ and } b \leqslant x \text{ iff } a \sqcup b \leqslant x$$
$$x \leqslant a \text{ and } x \leqslant b \text{ iff } x \leqslant a \sqcap b$$

Equivalently, 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 once a context where a $\tau_1$ is required and again in a context where a $\tau_2$ is, then the actual input type is $\tau_1 \sqcap \tau_2$.

So, we follow Pottier [Pot98b] and restrict polar types so as to only allow $\sqcup$ in output (positive) types, while $\sqcap$ is allowed only in input (negative types). A good portion of the apparent difficulty of solving subtyping constraints comes from allowing $\sqcup$ in negative positions or $\sqcap$ in positive, which roughly amounts to admitting disjunctions of typing constraints. Enforcing polarity removes disjunctions, and makes possible the *biunification* algorithm for eliminating constraints.

## 5.1 Polar types

Polar types are defined as syntactic terms, split into *positive type terms* $t^+$ and *negative type terms* $t^-$, defined simultaneously:

$$\begin{aligned} t^+ &::= \alpha \mid t^+ \sqcup t^+ \mid \bot \mid \text{bool} \mid t^- \to t^+ \mid \{\ell_1 : t_1^+, \dots, \ell_n : t_n^+\} \mid \mu\alpha.t^+ \\ t^- &::= \alpha \mid t^- \sqcap t^- \mid \top \mid \text{bool} \mid t^+ \to t^- \mid \{\ell_1 : t_1^-, \dots, \ell_n : t_n^-\} \mid \mu\alpha.t^- \end{aligned}$$

Recursive types (those beginning $\mu$) have two additional syntax restrictions. First, recursive types must be *guarded*, in that occurrences of $\alpha$ inside $\mu\alpha.t$ must be underneath at least one $\to$ (allowing $\mu\alpha.\text{bool} \to \alpha$ but not $\mu\alpha.\text{bool} \sqcup \alpha$). Secondly, recursive types must be *covariant*, in that $\alpha$ may only occur in $\mu\alpha.t$ to the left of an even number of $\to$ signs (allowing $\mu\alpha.\,(\alpha \to \text{bool}) \to \text{bool}$ but ruling out $\mu\alpha.\,\alpha \to \text{bool}$).

We use the Roman letter t for these to indicate that they are purely syntactic: $\bot \sqcup \bot$ and $\bot$ are different positive type terms, leaving the Greek letter $\tau$ to refer to elements of $\mathcal{T}$. The syntactic nature of $t^+, t^-$ means we can pattern-match on syntax, which is how we define the biunification algorithm of Section 5.3.3.

Syntactic type terms $t^+, t^-$ denote elements $\tau$ of $\mathcal{T}$. Due to the guardedness and covariance restrictions, recursive terms have unique interpretations which are both least pre-fixed and greatest post-fixed points of their defining equations (see Section 3.4). We write $t_1^+ \equiv t_2^+$ when $t_1^+$ and $t_2^+$ denote the same element $\tau$ of $\mathcal{T}$, even though $t_1^+$ and $t_2^+$ may be syntactically different.

We often leave the step moving from the syntactic object $t^+$ to its algebraic denotation $\tau$ implicit. In particular, given a type substitution $\rho$, we write $\rho(t^+)$ for $\rho$ applied to the denotation of $t^+$, and we write $t_1^+ \leqslant t_2^+$ when their denotations are subtypes.

In this way, polar types give a syntax for a particular subset of $\mathcal{T}$. This subset is expressive enough to write the type of the introduction's `select` function (1.1) as a positive type term:

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

However, the syntax cannot express types such as $(\alpha \sqcup \beta) \to (\alpha \sqcup \beta)$ as either a positive or a negative type term. Despite these restrictions, polar types suffice to describe the principal type of any MLsub expression, a fact which underlies the approach to type inference presented in this thesis.

Note that neither the denotations of positive nor negative type terms form a sublattice of $\mathcal{T}$: positive types are not closed under meet, and negative types are not closed under join. However, positive types are a sub-(join semilattice) of $\mathcal{T}$, while negative types are a sub-(meet semilattice).

### 5.1.1 Recursive types

Our first syntactic restriction, *guardedness*, is necessary to ensure uniqueness of fixpoints (since $\phi(\alpha) = \alpha$ has many fixed points) and appears in most presentations of recursive types. Our second syntactic restriction, *covariance*, is necessary to ensure existence of greatest pre-fixed points and least post-fixed points: without the covariance condition, $\phi$ could fail to be monotonic and have many pre-fixed points but no least one.

Our syntax does not actually require that $\alpha$ appear at all in $\mu\alpha.t^+$, but this causes no issues: if $\alpha$ is not free in $t^+$, then $\mu\alpha.t^+ = t^+$ as constant functions have unique fixed points.

While the syntax for polar type terms $t^+, t^-$ requires guardedness and covariance, we can weaken both of these requirements in derived operations. First, we note that every polar type term can be separated into a guarded and unguarded part:

**Proposition 36.** *For all positive type terms $t^+$ and variables $\alpha$, there exist positive type terms $t^+_\alpha$ and $t^+_g$ such that $t^+_\alpha \in \{\bot, \alpha\}$, $\alpha$ is guarded in $t^+_g$, and $t^+$ is equivalent to $t^+_\alpha \sqcup t^+_g$.*

*Proof.* By induction on the syntax of $t^+$.

Case $\alpha$: $t^+_\alpha = \alpha, t^+_g = \bot$.

Case $\beta$, $\beta \neq \alpha$: $t^+_\alpha = \bot, t^+_g = \beta$

Case $\bot$: $t^+_\alpha = \bot, t^+_g = \bot$.

Case $t \sqcup t'$: by induction, $t = t_\alpha \sqcup t_g$, $t' = t'_\alpha \sqcup t'_g$, so $t = (t_\alpha \sqcup t'_\alpha) \sqcup (t_g \sqcup t'_g)$, where $t_\alpha \sqcup t'_\alpha$ can be written as $\bot$ or $\alpha$ and $\alpha$ is guarded in $t_g \sqcup t'_g$.

Case bool, $t \to t, \{\dots\}$: $t^+_\alpha = \bot, t^+_g = t^+$ is guarded.

Case $\mu\beta.t'$: by induction, $t' = t'_\alpha \sqcup t'_g$, and by the fixed-point property $t^+ = t'_\alpha \sqcup t'_g[\mu\beta.t'/\alpha]$, where $t'_g[\mu\beta.t'/\alpha]$ denotes the replacement of occurrences of $\alpha$ in $t'_g$ (which must be positive) by $\mu\beta.t'$. □

Using this fact, we can weaken the guardedness condition and define a general least pre-fixed point operator $\mu^+$ as follows:

$$\mu^+\alpha.t^+ = \mu\alpha.t^+_g$$

where $t^+$ separates into $t^+_\alpha$ and $t^+_g$ as above. We still assume $\alpha$ occurs covariantly in $t^+$, but it may be unguarded.

**Proposition 37.** $\mu^+\alpha.t^+$ *is the least pre-fixed point of* $\alpha \mapsto t^+$.

*Proof.* If $\alpha$ is in fact guarded in $t^+$, then clearly $\mu^+\alpha.\, t^+ = \mu\alpha.\, t^+_g$, which is the least pre-fixed point. Otherwise, $t^+ = \alpha \sqcup t^+_g$ and $\mu^+\alpha.\, t^+$ is the least pre-fixed point of $\alpha \mapsto \alpha \sqcup t^+_g$. Since, for all $x$, $x \sqcup t^+_g \leqslant x$ iff $t^+_g \leqslant x$, $\alpha \mapsto t^+$ and $\alpha \mapsto \alpha \sqcup t^+_g$ have the same pre-fixed points, and thus the same least pre-fixed point. □

We thus have a least pre-fixed point operator $\mu^+$ on positive types which does not require guardedness, and dually a greatest post-fixed point operator $\mu^-$. Guardedness is still required to show that they coincide, since for instance:

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

So, the guardedness restriction can be relaxed, but it still seems that the covariance restriction reduces expressiveness because the syntax excludes e.g. $\mu\alpha.\alpha \to \alpha$, even though by contractivity (Section 3.4.2) a unique type $\tau = \tau \to \tau$ exists in $\mathcal{T}$. However, with some effort, polar types can indeed express this type.

The trick is to separate the positive and negative occurrences of $\alpha$: we look for *two* types $\tau^+_1, \tau^-_2$ such that:

$$\tau^+_1 = \tau^-_2 \to \tau^+_1 \qquad\qquad \tau^-_2 = \tau^+_1 \to \tau^-_2$$

Here, $\tau_1$ depends on itself only covariantly, as does $\tau_2$. We can thus introduce well-formed $\mu$-types:

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

Substitution gives:

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

These types are the same, since the syntaxes are $\alpha$-equivalent. Alternatively, we can see $\tau_1 = \tau_2$ since they both satisfy $\tau = \tau \to \tau$, which we know has a unique solution by contractivity.

This trick is an instance of a general construction due to Bekič [Bek84], who showed how to use a finite collection of least pre-fixed point operators on ordered sets (here $\mu^+$ on positive types and $\mu^-$ on negative ones) to construct a least pre-fixed point operator on their product. This allows the construction of recursive types like $\tau = \tau \to \tau$ using polar types as above, but more generally allows finding the least solution of finite systems of simultaneous equations over polar types, where "least" means having the smallest positive types and largest negative ones.

### 5.1.2 Polar typing schemes

A polar typing scheme $[D^-]t^+$ is a typing scheme (Section 4.2) where the types $D^-(x)$ of $\lambda$-bound variables are given by negative type terms, and the type of the result $t^+$ is given by a positive type term.

A polar typing scheme denotes pointwise a typing scheme $[\Delta]\tau$, and we write $\rho([D^-]t^+)$ to mean the application of $\rho$ to this denotation.

Our inference algorithm works solely with polar typing schemes, and the principality result (Section 6.1) shows that these suffice: every typeable program has a principal typing scheme which is polar.

## 5.2 Unification and subtyping

Unification is at the heart of the Hindley-Milner type inference algorithm for ML, and in this chapter we define and explore a subtyping-aware analogue called *biunification* which plays the same role for MLsub.

Constraints arise when inferring types for programs written in ML. Suppose an application $f\,x$, where the environment $\Gamma$ informs us that

$$\begin{aligned} f &: (\alpha \to \beta) \to (\alpha \to \beta) \\ x &: \mathrm{int} \to \mathrm{int} \end{aligned}$$

Type checking can proceed only under the constraint $\alpha \to \beta = \mathrm{int} \to \mathrm{int}$. Unification allows us to *eliminate* this constraint, turning it into the substitution $[\mathrm{int}/\alpha, \mathrm{int}/\beta]$. After applying this substitution, we have

$$\begin{aligned} f &: (\mathrm{int} \to \mathrm{int}) \to (\mathrm{int} \to \mathrm{int}) \\ x &: \mathrm{int} \to \mathrm{int} \end{aligned}$$

and type inference may proceed by applying $(\textsc{App})$.

Previous work on extending ML-style type inference with subtyping generally does not manage to eliminate constraints. Instead, previous work modifies the typing judgement to include an explicit set of constraints, giving a judgement form $\Gamma \vdash e : \tau | C$. Unfortunately, this set of constraints quickly grows unwieldy, growing linearly in the size of the program [HM95]. In a naive implementation, the set of constraints $C$ is simply the program text written clumsily.

Much research concentrated on methods for optimising the constraint set $C$, by converting it to smaller but equivalent forms. However, these methods are heuristic: no construction of a minimal constraint set was found, nor even a means to determine whether two constrained types are equivalent or in the subsumption relation (see Section 10.3).

The biunification technique described below allows subtyping constraints to be eliminated, and so does not require adding a constraint set $C$ to the typing judgement. It is the main ingredient in the algorithm for computing principal types in the next chapter.

Several situations where unification is used in Hindley-Milner type inference can be distinguished based on the direction of data flow, which is relevant when considering subtyping. As explained previously, types for outputs are given by positive types $t^+$, while types for inputs are given by negative types $t^-$. The cases are as follows:

**An input flows to an output** In ML, using a $\lambda$-bound input directly as an output does not require any actual unification. Instead, when typing $\lambda x.e$ a fresh type variable $\alpha$ is generated in the $(\textsc{Abs})$ rule, and entered into the environment as the type of $x$. In the reformulated rules for MLsub, the same effect occurs in the $(\textsc{Var}\text{-}\Delta)$ rule, where the type for $x$ is given principally by $[x : \alpha]\alpha$ (since typing schemes are closed, freshness is not a concern here: see Section 4.2.1).

**Two inputs or two outputs flow together** Two outputs flow together in the result type of an if-expression, and an input flows to two places when a $\lambda$-bound variable is used twice. In ML, both of these situations cause the input or output types to be unified, which as we saw in Section 1.1

results in a loss of precision. In MLsub, two output types $\tau_1^+$ and $\tau_2^+$ flowing together give a result of type $\tau_1^+ \sqcup \tau_2^+$, while an input used at types $\tau_1^-$ and $\tau_2^-$ is given type $\tau_1^- \sqcap \tau_2^-$.

It may seem odd that the equivalent of an operation which may fail in ML (unification) is an operation which always succeeds in MLsub (introduction of $\sqcup$ or $\sqcap$). However, note that different types emerging from an if-expression or being used in a λ-abstraction cannot cause a program to go wrong, but merely for an output or input to be under- or over-constrained.

**An output flows to an input** When the output of one expression flows to the input of another, they must be compatible. Elimination forms, such as application and projection, require their input to be the output of a suitable introduction form. These are the situations which cause programs to go wrong if not typed correctly, and are the constraints that biunification must handle. Since the constraints involve an output flowing to an input, they are of the form $t^+ \leqslant t^-$.

Given a set of constraints (for ML, of the form $t_1 = t_2$, and for us of the form $t_1^+ \leqslant t_2^-$), the purpose of the unification algorithm is to compute a substitution that solves those constraints.

There are two terms that must be carefully defined: "substitution" and "solves". Substitutions as normally conceived are awkward here, because substituting a polar type term for a type variable does not in general preserve the polarity restrictions, since type variables may be used both positively and negatively. This issue is resolved in the next section, while the issue of knowing what it means to "solve" constraints hinges on subtleties of ML's type schemes described in Section 5.2.2.

### 5.2.1 Bisubstitutions

To operate on polar type terms, we generalise from substitutions to *bisubstitutions*, which map type variables to a pair of a positive and a negative type term. We write:

$$\xi = [t^+/\alpha^+, t^-/\alpha^-]$$

for the bisubstitution $\xi$ which maps positive occurrences of $\alpha$ to $t^+$ and negative ones to $t^-$, written $\xi(\alpha^+) = t^+$ and $\xi(\alpha^-) = t^-$. The types $t^+$ and $t^-$ are not in general assumed to be related, although later we prove that biunification produces only *stable* bisubstitutions, which do relate the two.

A bisubstitution $\xi$ may be applied to a positive or negative term by replacing positive occurrences of $\alpha$ with $\xi(\alpha^+)$ and negative ones with $\xi(\alpha^-)$. This maintains the syntactic structure of polar types, mapping positive types $t^+$ to positive types $\xi(t^+)$ and negative types $t^-$ to negative types $\xi(t^-)$ as follows:

$$\begin{aligned}
\xi(t_1^+ \sqcup t_2^+) &= \xi(t_1^+) \sqcup \xi(t_2^+) & \xi(t_1^- \sqcap t_2^-) &= \xi(t_1^-) \sqcap \xi(t_2^-) \\
\xi(\bot) &= \bot & \xi(\top) &= \top \\
\xi(\mathrm{bool}) &= \mathrm{bool} & \xi(\mathrm{bool}) &= \mathrm{bool} \\
\xi(t^- \to t^+) &= \xi(t^-) \to \xi(t^+) & \xi(t^+ \to t^-) &= \xi(t^+) \to \xi(t^-) \\
\xi(\{f\}) &= \{f \circ \xi\} & \xi(\{f\}) &= \{f \circ \xi\} \\
\xi(\mu\alpha.t^+) &= \mu\alpha.\xi'(t^+) & \xi(\mu\alpha.t^-) &= \mu\alpha.\xi'(t^-)
\end{aligned}$$

where $\xi'(\alpha^+) = \alpha$, $\xi'(\alpha^-) = \alpha$, and $\xi'$ otherwise agrees with $\xi$.

Bisubstitutions can be composed, defining $\xi\zeta$ as follows:

$$\xi\zeta(\alpha^+) = \xi(\zeta(\alpha^+)) \qquad \xi\zeta(\alpha^-) = \xi(\zeta(\alpha^-))$$

Composition is associative with identity 1, defined as the identity bisubstitution mapping every variable to itself.

We also write $\xi([D^-]t^+)$ to describe a bisubstitution acting pointwise on a polar typing scheme $[D^-]t^+$, giving another polar typing scheme.

We write $\xi \equiv \zeta$ when two bisubstitutions $\xi$ and $\zeta$ map the same variable to equivalent types, that is, when $\xi(\alpha^+) \equiv \zeta(\alpha^+)$ and $\xi(\alpha^-) \equiv \zeta(\alpha^-)$ for all type variables $\alpha$. Up to $\equiv$, bisubstitutions preserve $\sqcap$ and $\sqcup$:

Bisubstitutions preserve $\sqcup$ and $\sqcap$:

**Proposition 38.** $\xi(t_1^+ \sqcup t_2^+) = \xi(t_1^+) \sqcup \xi(t_2^+), \xi(t_1^- \sqcap t_2^-) = \xi(t_1^-) \sqcap \xi(t_2^-)$

*Proof.* Induction on the syntax of $t_1^\pm, t_2^\pm$. Alternatively, this can be shown by taking a bisubstitution to be an ordinary substitution $\rho$ by splitting each variable into separate positive and negative copies, since we know from Section 3.3.3 that substitutions preserve meets and joins. □

The latter argument proves a slightly stronger result: bisubstitutions preserve *all* meets and joins, including infinite ones.

We may add two bisubstitutions as follows:

$$(\xi + \zeta)(\alpha^+) = \xi(\alpha^+) \sqcup \zeta(\alpha^+) \qquad (\xi + \zeta)(\alpha^-) = \xi(\alpha^-) \sqcap \zeta(\alpha^-)$$

These equations extend to all polar types:

**Proposition 39.** $(\xi + \zeta)(t^+) = \xi(t^+) \sqcup \zeta(t^+)$, $(\xi + \zeta)(t^-) = \xi(t^-) \sqcap \zeta(t^-)$

*Proof.* Induction on the syntax of $t^+, t^-$. □

The operation $+$ is idempotent, commutative and associative up to $\equiv$, (inheriting these properties from $\sqcup$ and $\sqcap$), and has identity 0, defined as:

$$0(\alpha^+) = \bot \qquad 0(\alpha^-) = \top$$

0 acts on types by replacing all of their free variables with $\top$ and $\bot$. Since $\xi(t^+) = t^+$ when $t^+$ has no free variables, $\xi 0 = 0$. However, it is not the case that $0\xi = 0$, since e.g.

$$(0[(\top \to \alpha)/\alpha^+])(\alpha^+) = 0(\top \to \alpha) = \top \to \bot \not\equiv \bot = 0(\alpha^+)$$

The operation $+$ defines a semilattice, which induces an ordering on bisubstitutions:

$$\xi \leqslant \zeta \text{ iff } \xi + \zeta \equiv \zeta$$

This is a preorder, or a partial order up to $\leqslant$. Expanding the definition of $+$, we see that $\xi \leqslant \zeta$ iff, for all type variables $\alpha$, $\xi(\alpha^+) \leqslant \zeta(\alpha^+)$ and $\zeta(\alpha^-) \leqslant \xi(\alpha^-)$: thus, bisubstitutions are ordered covariantly in the types they assign to positive variable occurrences, and contravariantly in the types they assign to negative variable occurrences.

Bisubstitution addition distributes over composition on both sides:

**Proposition 40.** $\xi \cdot (\zeta_1 + \zeta_2) \equiv \xi \cdot \zeta_1 + \xi \cdot \zeta_2$

*Proof.* $(\xi \cdot (\zeta_1 + \zeta_2))(\alpha^+) = \xi(\zeta_1(\alpha^+) \sqcup \zeta_2(\alpha^+))$, which is $\xi(\zeta_1(\alpha^+)) \sqcup \xi(\zeta_2(\alpha^+))$ by Proposition 38, and dually for $\alpha^-$. □

**Proposition 41.** $(\zeta_1 + \zeta_2) \cdot \xi \equiv \xi \cdot \zeta_1 + \xi \cdot \zeta_2$

*Proof.* $((\zeta_1 + \zeta_2) \cdot \xi)(\alpha^+) = (\zeta_1 + \zeta_2)(\xi(\alpha^+))$, which is $\zeta_1(\xi(\alpha^+)) \sqcup \zeta_2(\xi(\alpha^+))$ by Proposition 39, and dually for $\alpha^-$. □

In particular, these results imply that both application and composition of bisubstitution are monotonic.

### 5.2.2 Parameterisation and typing

ML is often described by placing it in the framework of some more complex type theory, like System F or $F_\omega$ or the Calculus of Constructions. This viewpoint describes ML as the rank-1 fragment of such a system, where $\forall$ quantifiers are restricted to only occur at top-level, and analyses the behaviour of ML as a special case of the larger theory.

This viewpoint misses a crucial fact about ML type schemes, which is important for MLsub. In systems based on ML, the following type schemes are equivalent:

$$\forall\alpha\,\forall\beta\,.\,\alpha \to \beta \to \alpha$$
$$\forall\beta\,\forall\alpha\,.\,\alpha \to \beta \to \alpha$$

They are equivalent in the sense that, for instance, a module in SML or OCaml containing a value of the first type can be ascribed to contain a value of the second. In systems that describe polymorphism as $\lambda$-abstraction over types (including F, $F_\omega$ and CoC), the two type schemes above are written something like:

$$\Lambda\alpha\,\Lambda\beta\,.\,\alpha \to \beta \to \alpha$$
$$\Lambda\beta\,\Lambda\alpha\,.\,\alpha \to \beta \to \alpha$$

These two are not equivalent, even up to renaming. For instance, applying both to the type $\mathbb{N}$ gives the following clearly distinct types:

$$\Lambda\beta\,.\,\mathbb{N} \to \beta \to \mathbb{N}$$
$$\Lambda\alpha\,.\,\alpha \to \mathbb{N} \to \alpha$$

The presence of explicit type application in F, $F_\omega$ and CoC makes the exact parameterisation of a polymorphic type relevant. Conversely, in ML, the parameterisation is irrelevant and all that matters is the set of possible instances. Indeed, the above ML type schemes could be written as set comprehensions, making their equivalence clearer:

$$\{\alpha \to \beta \to \alpha \mid \alpha, \beta \text{ types}\}$$
$$\{\alpha \to \beta \to \alpha \mid \beta, \alpha \text{ types}\}$$

Thus, when manipulating constraints, an ML type checker need only preserve equivalence of the set of instances, and not equivalence of the parameterisation. This freedom is not much used in plain ML, since unification happens to preserve equivalence of the parameterisation. However, this freedom is what allows MLsub to eliminate subtyping constraints. Consider this example of a constrained set of natural numbers:

$$\{n^2 + 17 \mid n \in \mathbb{N},\ n \geqslant 5\}$$

We can remove the constraint $n \geqslant 5$ by substituting $\max(n, 5)$ for $n$:

$$\{\max(n, 5)^2 + 17 \mid n \in \mathbb{N}\}$$

This second set does not have the same parameterisation, quantifying over all $n$ rather than over just those $n \geqslant 5$, and yet it defines the same set.

### 5.2.3 The instances of a typing scheme

The goal of the inference algorithm (Chapter 6) is to produce, given an input expression $e$, a polar typing scheme $[D^-]t^+$ that subsumes all other possible typing schemes for $e$. It is generally the case that $e$ is typeable with many different non-polar typing schemes, but the most general one will be polar.

Accordingly, we define the set of (not necessarily polar) *instances* of a polar typing scheme $[D^-]t^+$ to be the set of typing schemes it subsumes, that is:

$$\{[\Delta]\tau \mid [D^-]t^+ \leqslant^\forall [\Delta]\tau\}$$

Expanding the definition of subsumption, this is equivalently,

$$\{[\Delta]\tau \mid \exists \rho. \rho([D^-]t^+) \leqslant [\Delta]\tau\}$$

We write $C$ for a finite sequence of constraints $t_1^+ \leqslant t_1^-, \dots t_n^+ \leqslant t_n^-$, where each constraint has the form $t^+ \leqslant t^-$. (The order of constraints isn't relevant, but we use sequences rather than sets to better match the concrete algorithm described below, which processes constraints in a particular order). We write $\xi C$ for the application of the bisubstitution $\xi$ to both sides of the constraints in $C$ (yielding another sequence of constraints), and we say $\rho$ *satisfies* $C$ (written $\rho \models C$) whenever $\rho(t_i^+) \leqslant \rho(t_i^-)$ for all constraints $t_i^+ \leqslant t_i^-$ in $C$.

Given a polar typing scheme $[D^-]t^+$, its *instances under C* are:

$$\{[\Delta]\tau \mid \exists \rho \models C.\ \rho([D^-]t^+) \leqslant [\Delta]\tau\}$$

The instances of $[D^-]t^+$ under $C$ represent the ways in which $[D^-]t^+$ can be used, subject to $C$. As such, it is larger for more general $[D^-]t^+$ and smaller $C$:

**Proposition 42.** *If* $[D_1^-]t_1^+ \leqslant^\forall [D_2^-]t^+$ *and* $C_1 \subseteq C_2$*, then all instances of* $[D_1^-]t_1^+$ *under* $C_1$ *are instances of* $[D_2^-]t_2^-$ *under* $C_2$.

We say that $\xi$ *solves* $C$ when, for any polar typing scheme $[D^-]t^+$, the set of instances of $\xi([D^-]t^+)$ is equal to the set of instances of $[D^-]t^+$ under $C$. The object of the rest of this chapter is to construct and prove the correctness of the *biunification* algorithm, which for any satisfiable $C$ constructs a bisubstitution $\xi$ solving it.

As an example, consider the typing scheme for the identity function $[\,]\alpha \to \alpha$ and the constraint $\alpha \leqslant \{\texttt{awake} : \text{bool}\}$. Simply substituting $\alpha$'s bound into the type (as per unification) does not solve the constraint, as it leaves us with the typing scheme

$$[\,]\{\texttt{awake} : \text{bool}\} \to \{\texttt{awake} : \text{bool}\}$$

which has lost information: we should be able to pass $\{\texttt{awake} = \text{true}, \texttt{number} = 5\}$ to this function and know that the result contains a field `number`. Specifically, the typing scheme

$$[\,]\{\texttt{awake} : \text{bool}, \texttt{number} : \text{int}\} \to \{\texttt{awake} : \text{bool}, \texttt{number} : \text{int}\}$$

is one of the instances of $[]\alpha \to \alpha$ under $\alpha \leqslant \{\texttt{awake} : \text{bool}\}$, and yet is not an instance of $[]\{\texttt{awake} : \text{bool}\} \to \{\texttt{awake} : \text{bool}\}$.

Following the example on natural numbers above, we notice that the constraint $\alpha \leqslant \{\texttt{awake} : \text{bool}\}$ is equivalent to $\alpha = \alpha \sqcap \{\texttt{awake} : \text{bool}\}$, so we make a replacement, giving:

$$[]\{\texttt{awake} : \text{bool}\} \sqcap \alpha \to \{\texttt{awake} : \text{bool}\} \sqcap \alpha$$

This typing scheme has the correct set of instances. For example, we can pass the previous sample argument by instantiating $\alpha = \{\texttt{awake} : \text{bool}, \texttt{number} : \text{int}\}$ and see that the return type still mentions a `number` field. Sadly, this typing scheme is no longer polar, since $\sqcap$ appears positively.

However, this typing scheme has the same set of instances as the following, which is indeed polar:

$$[](\{\texttt{awake} : \text{bool}\} \sqcap \alpha) \to \alpha$$

Intuitively, a constraint $\alpha \leqslant t^-$ gives a new upper bound to $\alpha$, and thus should only affect occurrences of $\alpha$ used as upper bounds (the negative ones). This is not a new insight: it was used by Eifrig, Smith and Trifonov [EST95a] and Pottier [Pot01] to simplify systems of constraints. Here, we use this insight to not just simplify but entirely eliminate constraints, producing a polar typing scheme. In general, this requires some special handling of recursive types, described below in Section 5.3.

### 5.2.4 Comparison with unification

The concept of a bisubstitution $\xi$ solving subtyping constraints $C$ is the analogue of the concept of *most general unifier* in standard unification.

A substitution $\rho$ is said to *unify* (or to be a *unifier* of) some finite set of equations $E = \{t_i = t'_i \mid 1 \leqslant i \leqslant n\}$ when $\rho(E)$ is trivial, that is, $\rho(t_i) = \rho(t'_i)$ for $1 \leqslant i \leqslant n$.

This does not seem to fit with the definition of "solves" above, since if a bisubstitution $\xi$ solves constraints $C$, it is not necessarily the case that $\xi(C)$ is trivial. For instance, if $c = \alpha \leqslant \text{bool}$, then the bisubstitution $\theta_c = [\alpha \sqcap \text{bool}/\alpha^-]$ solves $\{c\}$ (see below), and yet $\theta_c(c) = c$ which is a nontrivial constraint.

However, while there is no direct analogue of "unifier" with bisubstitutions, the concept of "most general unifier" fits nicely. A substitution $\rho$ is a *most general unifier* of $E$ when it unifies $E$, and for any $\rho_2$ unifying $E$ there exists some $\rho'_2$ such that $\rho_2 = \rho'_2 \circ \rho$. That is, all unifiers factor through a most general unifier. When we apply the concepts of instances and instances under $E$ (from Section 5.2.3) to equations and unifiers, we find that most general unifiers agree with our definition of "solves" above:

**Proposition 43.** *$\rho$ is a most general unifier of $E$ iff, for any type $t$, the instances of $\rho(t)$ agree with the instances of $t$ under $E$.*

*Proof.* The instances of $t$ under $E$ is the set:

$$\{\rho_2(t) \mid \rho_2 \text{ unifies } E\}$$

By the definition of most general unifiers, this is equivalently:

$$\{\rho'_2(\rho(t)) \mid \rho'_2 \text{ a substitution}\}$$

which is the definition of the instances of $\rho(t)$. □

So, bisubstitutions solving constraints is the analogue of most general unifiers, although in a slightly non-obvious way.

## 5.3 Solving constraints with bisubstitutions

As in standard unification, biunification works by breaking complex constraints down into a collection of simpler ones. We first describe how these simple constraints are solved, and then move on to the biunification algorithm.

Below, we solve *atomic* constraints, and in the next section we present the *biunification algorithm* which solves arbitrary constraints.

### 5.3.1 Atomic constraints

The simplest constraints are those between type variables and constructed types. We say that a positive or negative type term is *constructed* when it is of the form $t \to t$, $\{\ell_1 : t_1, \dots, \ell_n : t_n\}$ or bool (that is, it is an application of a type constructor rather than a type variable or a lattice operator).

A constraint is *atomic* if it is of the form $\alpha \leqslant t^-$ (with $t^-$ constructed), $t^+ \leqslant \alpha$ (with $t^+$ constructed) or $\alpha \leqslant \beta$.

If $\alpha$ appears in $t^-$, then the bisubstitution $[\alpha \sqcap t^-/\alpha^-]$ does not in general solve $\alpha \leqslant t^-$. With standard unification, the *occurs check* sidesteps this difficulty, by failing whenever $\alpha$ appears in $t^-$. Since we support recursive types (indeed, we are forced to if we are to have principality: see Section 2.1.6) we have no such easy escape. In general, solving a constraint $\alpha \leqslant t^-$ may require us to introduce a recursive type.

So, we solve $\alpha \leqslant t^-$ with the following bisubstitution $\theta$ (see Section 5.1.1 for the definition of $\mu^-$):

$$\theta = [\mu^-\beta.\, \alpha \sqcap [\beta/\alpha^-](t^-)/\alpha^-]$$

Note that by unrolling,

$$\theta(\alpha^-) = \alpha \sqcap [\theta(\alpha^-)/\alpha^-](t^-) = \alpha \sqcap \theta(t^-)$$

Due to its definition via the greatest post-fixed point operator $\mu^-$, $\theta(\alpha^-)$ is the greatest lower bound of its unrollings:

**Proposition 44.** $\theta(\alpha^-) = \bigsqcup_n t_n$, *where* $t_0 = \top$, $t_{n+1} = \alpha \sqcap [t_n/\alpha^-]t^-$.

When $\alpha$ does not occur in $t^-$, $\theta$ is equivalent to $[\alpha \sqcap t^-/\alpha^-]$. In general, $\theta$ can be understood as an iterated version of the bisubstitution $[\alpha \sqcap t^-/\alpha^-]$. Intuitively, when the occurs check fails, we make the substitution infinitely many times.

**Lemma 45.** $\theta(\alpha^-) = \bigsqcup_n t'_n$, *where* $t'_n = [\alpha \sqcap t^-/\alpha^-]^n(\alpha^-)$

*Proof.* First, we note that $t'_n$ satisfies a recurrence similar to that of $t_n$ above:

$$\begin{aligned} t'_{n+1} &= [\alpha \sqcap t^-/\alpha^-]^{n+1}(\alpha^-) \\ &= [[\alpha \sqcap t^-/\alpha^-]^n/\alpha^-](\alpha \sqcap t^-) \\ &= [t'_n/\alpha^-](\alpha \sqcap t^-) \\ &= t'_n \sqcap [t'_n/\alpha^-]t^- \end{aligned}$$

It is clear from this recurrence that the sequence $t'_n$ decreases.

Next, we show that it satisfies the *same* recurrence as $t_n$, proving that $\alpha \sqcap [t'_n/\alpha^-]t^- = t'_n \sqcap [t'_n/\alpha^-]t^-$. Since $t'_n \leqslant t'_0 = \alpha$, the $\geqslant$ relation is trivial. The $\leqslant$ relation simplifies to:

$$\alpha \sqcap [t'_n/\alpha^-]t^- \leqslant t'_n$$

We prove this by induction on $n$. In case $n = 0$, $\alpha \sqcap t^{-} \leqslant \alpha$. In case $n + 1$, we note (using $t'_{n+1} \leqslant t'_{n}$ and the induction hypothesis):

$$\alpha \sqcap [t'_{n+1}/\alpha^{-}]t^{-} \leqslant \alpha \sqcap [t'_{n}/\alpha^{-}] \leqslant t'_{n}$$

so

$$\alpha \sqcap [t'_{n+1}/\alpha^{-}]t^{-} \leqslant t'_{n} \sqcap [t'_{n}/\alpha^{-}]t' = t'_{n+1}$$

Therefore, sequences $t_n$ and $t'_n$ satisfy the same recurrence, with $t_n$ starting from $\top$ and $t'_n$ starting from $\alpha$. Since the recurrence is monotone, that means that $t'_n \leqslant t_n$, while since $t_1 \leqslant \alpha$ we have $t_{n+1} \leqslant t'_n$. The sequences are sandwiched between each other, so have the same greatest lower bound. □

This construction solves atomic constraints $\alpha \leqslant t^{-}$, a fact proved in Section 5.4.2. Using this construction (and its dual), we can solve any atomic constraint:

- $[\mu^{-}\beta.\, \alpha \sqcap [\beta/\alpha^{-}](t^{-})/\alpha^{-}]$ solves $\alpha \leqslant t^{-}$
- $[\mu^{+}\beta.\, \alpha \sqcup [\beta/\alpha^{+}](t^{+})/\alpha^{+}]$ solves $t^{+} \leqslant \alpha$
- $[\mu^{-}\beta.\, \alpha \sqcap [\beta/\alpha^{-}](\gamma)/\alpha^{-}] \equiv [\alpha \sqcap \gamma/\alpha^{-}]$ solves $\alpha \leqslant \gamma$

For any atomic constraint $c$, we write $\theta_c$ for the bisubstitution solving it per the above cases.

In the last case, we have a choice between using $[\alpha \sqcap \beta/\alpha^{-}]$ or $[\alpha \sqcup \beta/\beta^{+}]$ to solve the constraint $\alpha \leqslant \beta$. Again, this parallels standard unification where we may solve a constraint $\alpha = \beta$ either by replacing occurrences of $\alpha$ with $\beta$ or vice versa.

### 5.3.2 Decomposing constraints

Given a non-atomic constraint $t^{+} \leqslant t^{-}$, the biunification algorithm breaks it down into subconstraints. Atomic subconstraints are solved using $\theta_c$ from the previous section, while complex subconstraints are handled recursively.

For a constraint $t^{+} \leqslant t^{-}$, its *immediate subconstraints* $\mathsf{subi}(t^{+} \leqslant t^{-})$ are defined as follows:

$$\begin{aligned}
\mathsf{subi}(t_1^{-} \to t_1^{+} \leqslant t_2^{+} \to t_2^{-}) &= \{t_2^{+} \leqslant t_1^{-}, t_1^{+} \leqslant t_2^{-}\} \\
\mathsf{subi}(\mathsf{bool} \leqslant \mathsf{bool}) &= \{\} \\
\mathsf{subi}(\{\ell_1 : t_1^{+}, \ldots, \ell_n : t_n^{+}, \ldots \ell_{n+k} : t_{n+k}^{+}\} \leqslant \{\ell_1 : t_1^{-}, \ldots, \ell_n : t_n^{-}\}) & \\
&= \{t_1^{+} \leqslant t_1^{-}, \ldots, t_n^{+} \leqslant t^{-}n\} \\
\mathsf{subi}(\mu\alpha.t^{+} \leqslant t^{-}) &= \{t^{+}[\mu\alpha.t^{+}/\alpha] \leqslant t^{-}\} \\
\mathsf{subi}(t^{+} \leqslant \mu\alpha.t^{-}) &= \{t^{+} \leqslant t^{-}[\mu\alpha.t^{-}/\alpha]\} \\
\mathsf{subi}(t_1^{+} \sqcup t_2^{+} \leqslant t^{-}) &= \{t_1^{+} \leqslant t^{-}, t_2^{+} \leqslant t^{-}\} \\
\mathsf{subi}(t^{+} \leqslant t_1^{-} \sqcap t_2^{-}) &= \{t^{+} \leqslant t_1^{-}, t^{+} \leqslant t_2^{-}\} \\
\mathsf{subi}(\bot \leqslant t^{-}) &= \{\} \\
\mathsf{subi}(t^{+} \leqslant \top) &= \{\}
\end{aligned}$$

The function $\mathsf{subi}$ is partial: for instance, $\mathsf{subi}(\mathsf{bool} \leqslant \bot \to \top)$ fails. The cases for which it succeeds are characterised as follows:

**Proposition 46.** *For all $t^{+}, t^{-}$, exactly one of the following hold:*

- $t^{+} \leqslant t^{-}$ *is an atomic constraint*

- $t^+ \leqslant t^-$ *is unsatisfiable*
- $\mathsf{subi}(t^+ \leqslant t^-)$ *succeeds*

*Proof.* Case analysis on $t^+, t^-$. □

The ease of defining subi is the primary technical purpose of the restricted syntax of polar type terms. There is no way, in general, to decompose a constraint $t_1 \sqcap t_2 \leqslant t_3$ into a conjunction of simpler constraints, but the syntax of polar type terms excludes such examples.

Inspection of the definition of subi shows that the immediate subconstraints are equivalent to the original, and applying bisubstitutions to immediate subconstraints is the same as applying them to the original constraint:

**Proposition 47.** *If* $\mathsf{subi}(t^+ \leqslant t^-)$ *succeeds, then* $\rho \models t^+ \leqslant t^-$ *iff* $\rho \models \mathsf{subi}(t^+ \leqslant t^-)$

**Proposition 48.** *If* $\mathsf{subi}(t^+ \leqslant t^-)$ *succeeds, then* $\mathsf{subi}(\xi t^+ \leqslant \xi t^-) = \xi(\mathsf{subi}(t^+ \leqslant t^-))$.

### 5.3.3 The biunification algorithm

The biunification algorithm operates on a sequence of constraints C, and either fails or returns a bisubstitution that solves C. To handle recursive constraints, the biunification algorithm takes a second argument H, which is a set of constraints that have already been seen and may be skipped if seen again. We write the two-argument version as $\mathsf{biunify}(H; C)$ and take $\mathsf{biunify}(C) = \mathsf{biunify}(\emptyset; C)$. $\mathsf{biunify}(H; C)$ is the following recursive function:

$$\begin{aligned}
\mathsf{biunify}(H; \epsilon) &= 1 && \text{(EMP)}\\
\mathsf{biunify}(H; c, C) &= \mathsf{biunify}(H; C) \text{ if } c \in H && \text{(HYP)}\\
\mathsf{biunify}(H; c, C) &= \mathsf{biunify}(\theta_c(H \cup \{c\}); \theta_c C) \cdot \theta_c \text{ if c atomic} && \text{(ATOM)}\\
\mathsf{biunify}(H; c, C) &= \mathsf{biunify}(H \cup \{c\}; \mathsf{subi}(c), C) \text{ if } \mathsf{subi}(c) \text{ succeeds} && \text{(SUB)}
\end{aligned}$$

Constraints are processed one at a time. Ones that have been seen before are skipped (case (HYP)), and complex constraints are broken down into simpler ones (case (SUB)). When an atomic constraint c is reached, it is eliminated using $\theta_c$, and its solution is applied both to the output of a recursive call to biunify, and to the remaining constraints.

The algorithm adds all previously seen constraints to H for recursive calls. For correctness, this need only be done when decomposing constraints with recursive types, but doing it for all constraints allows H to act as a memoisation table, bringing the complexity of biunification from exponential to polynomial. We leave further discussion of the algorithm's time complexity until Section 7.4.1, after an efficient representation of constraints has been introduced. As well as performing better than a naive representation, the efficient representation makes the termination argument much easier.

This is a natural development from the standard unification algorithm. Indeed, suppose that we replace $\theta_c$ in the definition of biunify with the substitutions that eliminate equality constraints (e.g. using substitution $[\mathrm{bool}/\alpha]$ instead of the bisubstitution $[\alpha \sqcap \mathrm{bool}/\alpha^-]$), and we delete from the definition of subi all lines having to do with records, subtyping or recursive types (leaving just the first two lines, concerning booleans and functions). Doing this, we find we have precisely recovered Martelli and Montanari's unification algorithm [MM82]!

## 5.4 Correctness of biunification

Leaving aside the issue of termination, we must still prove correctness of the biunification algorithm. In particular, we must show that if biunification fails, then the input constraints $C$ are unsatisfiable, while if it succeeds the resulting bisubstitution really does solve $C$.

This is a surprisingly tricky business, much more so than for standard unification. The extra complexity comes from our more subtle definition of what it means for a bisubstitution to "solve" a constraint (see Section 5.2.3). The remainder of this chapter will be spent on proving the correctness of biunification.

Despite the length, even this proof is not fully rigourous (in particular, the treatment of the table $H$ of already-seen constraints is somewhat informal). It is odd that such a short algorithm requires such a detailed proof; we expect that a more concise argument exists, but leave its development as future work.

### 5.4.1 Stability and idempotence

Proving that these solutions are correct is made difficult by the complex definition of what it means for a bisubstitution to solve some constraints, which hinges on the notion of *instances* (Section 5.2.3). In this section, we describe a class of bisubstitutions for which the definition of solving is simpler.

In standard unification, applying any substitution a type yields an instance of the type. Indeed, this is how "instance" is usually defined. In our setting, however, things are more subtle. 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

$$[\,]\alpha \to \alpha \not\leqslant^{\forall} [\,]\top \to \mathrm{bool}$$

and yet the latter can be formed from the former by applying the bisubstitution $[\top/\alpha^-, \mathrm{bool}/\alpha^+]$. In order to map typing schemes to instances, a bisubsitution must map positive and negative occurrences of the same variable to related types:

**Lemma 49.** *$\xi([D^-]t^+)$ is an instance of arbitrary $[D^-]t^+$ iff $\xi(\alpha^-) \leqslant \xi(\alpha^+)$ for all type variables $\alpha$.*

*Proof.* Construct the substitution $\rho(\alpha) = \xi(\alpha^-)$. $\rho([D^-]t^+)$ is an instance of $[D^-]t^+$, and $\rho([D^-]t^+) \leqslant \xi([D^-]t^+)$ so $\xi([D^-]t^+)$ is too.

Conversely, if $\xi(\alpha^-) \not\leqslant \xi(\alpha^+)$ for a particular type variable $\alpha$, then $[\,]\xi(\alpha^-) \to \xi(\alpha^+)$ is not an instance of $[\,]\alpha \to \alpha$. □

Our second condition is *idempotence*, the requirement that $\xi^2 = \xi$. We impose this condition for technical convenience, as it allows us to simplify the definition of solving (conditions I and II). This is not new: many accounts of standard unification prove that the unification algorithm produces not merely most general unifiers, but most general *idempotent* unifiers [LMM88].

So, we define a *stable bisubstitution* as any $\xi$ such that $\xi^2 \equiv \xi$, and $\xi(\alpha^-) \leqslant \xi(\alpha^+)$ for all type variables $\alpha$. It is not the case that if $\xi$ solves $C$, then $\xi$ must be stable, any more than it is the case that most general unifiers must be idempotent. However, as we see below, any $C$ which can be solved can be solved by some stable $\xi$, so it does no harm to restrict our attention to stable bisubstitutions. Again, the situation parallels standard unification, where any unifiable terms can be unified by a most general idempotent unifier.

Recall that $\xi$ solves C if, for any polar typing scheme $[D^-]t^+$, the set of instances of $\xi([D^-]t^+)$ equals the set of instances of $[D^-]t^+$ under C. Expanding definitions, this is equivalent to the following two conditions:

$$\forall\rho \models C\,\exists\rho'\,\forall t^+, t^-.\ \rho'(\xi t^+) \leqslant \rho(t^+), \rho(t^-) \leqslant \rho'(\xi t^-) \tag{I}$$

$$\forall\rho'\,\exists\rho \models C\,\forall t^+, t^-.\ \rho(t^+) \leqslant \rho'(\xi t^+), \rho'(\xi t^+) \leqslant \rho(t^-) \tag{II}$$

These are quite unwieldy, but can be simplified under the assumption that $\xi$ is stable. First, we prove a useful lemma:

**Lemma 50.** *If* $\rho_0 \models C$*, then* $\rho \circ \rho_0 \models C$ *for arbitrary* $\rho$

*Proof.* For any $t^+ \leqslant t^- \in C$, $\rho_0(t^+) \leqslant \rho_0(t^-)$ so $\rho(\rho_0(t^+)) \leqslant \rho(\rho_0(t^-))$. □

**Lemma 51.** *If* $\xi$ *is stable, then* $\xi$ *solves* C *iff the following two conditions hold:*

$$\forall\rho \models C\,\forall t^+, t^-.\ \rho(\xi t^+) = \rho(t^+), \rho(t^-) = \rho(\xi t^-) \tag{Ia}$$

$$\exists\rho_0 \models C\,\forall t^+, t^-.\ \rho_0(t^+) \leqslant \xi t^+, \xi t^- \leqslant \rho_0(t^-) \tag{IIa}$$

*Proof.* Trivially, Ia implies I, by choosing $\rho' = \rho$. For the converse, suppose $\rho \models C$ and so by I $\rho'(\xi t^+) \leqslant \rho(t^+), \rho(t^-) \leqslant \rho'(\xi t^-)$ (for some $\rho'$, and all $t^+, t^-$). For arbitrary $\alpha$, since $\xi(\alpha^-) \leqslant \xi(\alpha^+)$,

$$\rho'(\xi(\alpha^-)) \leqslant \rho'(\xi(\alpha^+)) \leqslant \rho(\alpha) \leqslant \rho'(\xi(\alpha^-)) \leqslant \rho'(\xi(\alpha^+))$$

making all terms in the above equal. Therefore, $\rho'(\xi t^+) = \rho(t^+)$, and $\rho'(\xi t^-) = \rho(t^-)$. Since $\xi^2 = \xi$,

$$\rho(\xi t^+) = \rho'(\xi^2 t^+) = \rho'(\xi t^+) = \rho(t^+)$$

giving condition Ia.

Condition II implies IIa by taking $\rho$ to be the identity substitution. For the converse, suppose $\rho_0 \models C$ and $\rho_0(t^+) \leqslant \xi t^+$, $\xi t^- \leqslant \rho_0(t^-)$. Given any $\rho'$, $\rho' \circ \rho_0 \models C$ by Lemma 50, and $(\rho' \circ \rho_0)(t^+) \leqslant \rho'(\xi t^+)$ since $\rho_0(t^+) \leqslant \xi t^+$ (and dually for negative type terms $t^-$), proving II. □

In fact, the conditions can be simplified further. Instead of proving conditions Ia, IIa for all types, it suffices just to analyse their behaviour on type variables:

**Lemma 52.** *Conditions Ia and IIa are equivalent to:*

$$\forall\rho \models C\,\forall\alpha.\ \rho(\xi(\alpha^+)) = \rho(\alpha) = \rho(\xi(\alpha^-)) \tag{Ib}$$

$$\exists\rho_0 \models C\,\forall\alpha.\ \xi(\alpha^-) \leqslant \rho_0(\alpha) \leqslant \xi(\alpha^+) \tag{IIb}$$

*Proof.* Clearly Ia, IIa imply Ib, IIb, and the converse follows by induction on $t^+, t^-$. □

It is not in general the case that the composition of two stable bisubstitutions is stable. A useful special case in which this holds is the following:

**Lemma 53** (Stability of compositions)**.** *If* $\zeta, \theta$ *are stable bisubstitutions where* $\zeta \cdot \theta \equiv \zeta + \theta$*, then* $\theta \cdot \zeta$ *is stable.*

*Proof.* The first condition follows from monotonicity:

$$\zeta\theta(\alpha^-) \leqslant \zeta(\alpha^-) \leqslant \alpha \leqslant \zeta(\alpha^+) \leqslant \zeta\theta(\alpha^+)$$

The second condition uses Propositions 40 and 41:

$$(\theta\zeta)^2 \equiv \theta\zeta\theta\zeta \equiv \theta(\zeta + \theta)\zeta \equiv \theta\zeta^2 + \theta^2\zeta \equiv \theta\zeta + \theta\zeta \equiv \theta\zeta$$ □

### 5.4.2 Solving atomic constraints

Our first use of stable bisubstitutions is to prove that the bisubstitutions $\theta_c$ of Section 5.3.1 do indeed solve atomic constraints c. We prove this using conditions (Ib), (IIb) above, which require that we first prove $\theta$ to be stable.

**Lemma 54.** *For any type variable $\alpha$ and negative type term $t^-$, the bisubstitution $\theta_c$ is stable.*

*Proof.* By unrolling, $\theta(\alpha^-) \equiv \alpha \sqcap [\theta(\alpha^-)/\alpha^-](t^-) = \alpha \sqcap \theta(t^-)$, so clearly $\theta(\alpha^-) \leqslant \alpha = \theta(\alpha^+)$, and for any type variable $\gamma \neq \alpha$, $\theta(\gamma^-) = \gamma = \theta(\gamma^-)$.

To show that $\theta^2 \equiv \theta$, it suffices to show that $\theta^2(\alpha^-) \equiv \theta(\alpha^-)$, since $\theta$ is the identity for $\alpha^+$ and all other type variables. Expanding, we must show:

$$\mu^-\beta.\theta(\alpha^-) \sqcap [\beta/\alpha^-]t^- = \mu^-\beta.\alpha \sqcap [\beta/\alpha^-]t^-$$

For brevity, let $\phi(\tau)$ be the monotone function mapping $\tau$ to $[\tau/\alpha^-]t^-$. We show that the equation above holds by showing that both sides have the same post-fixed points, and therefore the same greatest such point. To do so, we must show:

$$\tau \leqslant \theta(\alpha^-) \sqcap \phi(\tau) \text{ iff } \tau \leqslant \alpha \sqcap \phi(\tau)$$

or equivalently,

$$\tau \leqslant \theta(\alpha^-) \wedge \tau \leqslant \phi(\tau) \text{ iff } \tau \leqslant \alpha \wedge \tau \leqslant \phi(\tau)$$

The ($\Rightarrow$) case is trivial, since $\theta(\alpha^-) \leqslant \alpha$. For the converse, if $\tau \leqslant \alpha \sqcap \phi(\tau)$ then $\tau$ is a post-fixed point of $\alpha \sqcap \phi(\tau)$, and thus less than the greatest such point which is $\theta(\alpha^-)$. □

**Theorem 55.** *The bisubstitution $\theta$ (as defined above) solves the constraint $\alpha \leqslant t^-$*

*Proof.* $\theta$ is stable, so we prove the lemma by verifying conditions Ib, IIb.

(Ib) Suppose $\rho \models \alpha \leqslant t^-$. $\rho(\theta(\alpha^+)) = \rho(\alpha)$ trivially, since $\theta$ is the identity on all positive variables. The situation is the same for negative variables other than $\alpha$, so we need only prove $\rho(\alpha) = \rho(\theta(\alpha^-))$, or equivalently

$$\rho(\alpha) = \rho(\mu^-\beta.\, \alpha \sqcap ([\beta/\alpha^-](t^-)))$$

Using the sequence $t^-_n$ as defined in Proposition 44, we prove by induction on $n$ that $\rho(\alpha) = \rho(t^-_n)$. Case $n = 0$ is trivial, since $\alpha \sqcap \top \equiv \alpha$.

In case $n+1$, we note that since $\rho(\alpha) \leqslant \rho(t^-_n)$, $\rho(t^-) \leqslant \rho([t^-_n/\alpha^-]t^-)$ since $\alpha^-$ appears only covariantly in $t^-$. Since $\rho(\alpha) \leqslant \rho(t^-)$, this implies $\rho(\alpha) \leqslant \rho([t^-_n/\alpha^-]t^-)$ and therefore $\rho(\alpha) = \rho(t^-_{n+1})$.

Since $\theta(\alpha^-)$ is the meet of $t^-_n$, and $\rho$ preserves meets, $\rho(\theta(\alpha^-)) = \rho(\alpha)$.

(IIb) Choose $\rho_0(\beta) = \theta(\beta^-)$ for all type variables $\beta$. Since $\theta$ is stable,

$$\theta(\beta^-) = \rho_0(\beta) \leqslant \theta(\beta^+)$$

To prove (IIb), we must also show $\rho_0 \models \alpha \leqslant t^-$. Since $\rho_0 = [\theta(\alpha^-)/\alpha]$ and $\theta(\alpha^-) \leqslant \theta(\alpha^+)$, $\theta(t^-) \leqslant \rho_0(t^-)$ as positive occurrences of $\alpha$ are contravariant in $t^-$. So, it suffices to show that $\theta(\alpha^-) \leqslant \theta(t^-)$, which holds since $\theta(\alpha^-) = \alpha \sqcap \theta(t^-)$. □

### 5.4.3 Solving multiple constraints

Next, we see how to construct a bisubstitution that solves a system of multiple constraints, by breaking it into parts.

**Lemma 56.** *Suppose* $\xi_1$*,* $\xi_2$ *and* $\xi_2 \cdot \xi_1$ *are all stable, and* $\xi_1$ *solves* $C_1$ *and* $\xi_2$ *solves* $\xi_1(C_2)$*. Then* $\xi_2 \cdot \xi_1$ *solves* $C_1, C_2$*.*

*Proof.* We verify conditions (Ib), (IIb), since all bisubstitutions involved are assumed stable.

(Ib) Given $\rho \models C_1, C_2$, since $\xi_1$ solves $C_1$ we have $\rho(\xi_1(t^+)) = \rho(t^+)$ and $\rho(t^-) = \rho(\xi_1(t^-))$ by condition (Ia). Therefore, $\rho \models \xi_1(C_2)$ since, for $t^+ \leqslant t^- \in C_2$,

$$\rho(\xi_1(t^+)) = \rho(t^+) \leqslant \rho(t^-) = \rho(\xi_1(t^-))$$

Since $\xi_2$ solves $\xi_1(C_2)$, that proves $\rho(\xi_2(t^+)) = \rho(t^+)$ and $\rho(t^-) = \rho(\xi_2(t^-))$, so:

$$\rho(\xi_2(\xi_1(\alpha^+))) = \rho(\xi_1(\alpha^+)) = \rho(\alpha) = \rho(\xi_1(\alpha^-)) = \rho(\xi_2(\xi_1(\alpha^-)))$$

(IIb) By condition (IIa), we have:

$$\rho_1 \models C_1 \forall t^+, t^-.\ \rho_1(t^+) \leqslant \xi_1(t^+),\ \xi_1(t^-) \leqslant \rho_1(t^-)$$
$$\rho_2 \models \xi_1(C_2) \forall t^+, t^-.\ \rho_2(t^+) \leqslant \xi_2(t^+),\ \xi_2(t^-) \leqslant \rho_2(t^-)$$

Let $\rho_0 = \rho_2 \circ \rho_1$. We have:

$$\xi_2(\xi_1(\alpha^-)) \leqslant \rho_2(\xi_1(\alpha^-)) \leqslant \rho_2(\rho_1(\alpha)) \leqslant \rho_2(\xi_1(\alpha^+)) \leqslant \xi_2(\xi_1(\alpha^+))$$

By Lemma 50, $\rho_0 \models C_1$. For $t^+ \leqslant t^- \in C_2$, we note that $\rho_2 \models \xi_1(t^+) \leqslant \xi_1(t^-)$, giving:

$$\rho_2(\rho_1(t^+)) \leqslant \rho_2(\xi_1(t^+)) \leqslant \rho_2(\xi_1(t^-)) \leqslant \rho_2(\rho_1(t^-))$$

so $\rho_0 \models C_2$. □

This result allows us to solve a set of constraints by first solving one of them, applying the solution to the rest, and then solving those, just as biunify does in case (ATOM). However, this only handles the success case: we need the following lemma to show correctness in the failure case, so that we know biunify fails only on unsatisfiable constraints.

**Lemma 57.** *Suppose* $\xi_1$*,* $\xi_2$ *and* $\xi_2 \cdot \xi_1$ *are all stable, and* $\xi_1$ *solves* $C_1$*. If* $\xi_1(C_2)$ *is unsatisfiable, then so is* $C_1, C_2$*.*

*Proof.* Suppose $\rho \models C_1, C_2$. Consider an arbitrary $t^+ \leqslant t^- \in C_2$. Since $\xi_1$ solves $C_1$, by condition (Ia) we know:

$$\rho(\xi_1(t^+)) = \rho(t^+) \leqslant \rho(t^-) = \rho(\xi_1(t^-))$$

so $\rho$ also solves $\xi_1(C_2)$, a contradiction. □

### 5.4.4 Stability of biunification

The results so far refer mostly to stable bisubstitutions. To apply them to biunify, we must show that it produces stable results. We prove a slightly stronger result:

**Lemma 58** (Stability of $\mathsf{biunify}$)**.** *If $\zeta$ is stable, and* $\mathsf{biunify}(\zeta H; \zeta C) = \xi$*, then $\xi\zeta$ is stable.*

*Proof.* By induction on the structure of recursive calls to $\mathsf{biunify}$.

(EMP): $\xi = 1$, so $\xi\zeta = \zeta$ is stable.

(HYP): We have $\mathsf{biunify}(\zeta H, \zeta C) = \xi$, so $\xi\zeta$ is stable by the inductive hypothesis.

(ATOM): Let $\xi'$ be the output of the recursive call to:

$$\mathsf{biunify}(\theta_{\zeta(c)}(\zeta H \cup \{\zeta(c)\}); \theta_{\zeta(c)}(\zeta(C)))$$

We have $\xi' \cdot \theta_{\zeta(c)} = \xi$, where $\zeta(c)$ is atomic.

The inductive hypothesis shows that $\xi' \cdot (\theta_{\zeta(c)} \cdot \zeta)$ is stable, which by associativity proves $\xi\zeta$ stable, but to use it we must satisfy its precondition that $\theta_{\zeta(c)} \cdot \zeta$ is stable. We know that both $\theta_{\zeta(c)}$ and $\zeta$ are stable (by Lemma 54 for $\theta_{\zeta(c)}$ and by assumption for $\zeta$), so we use Lemma 53 and need only prove that $\zeta \cdot \theta_{\zeta(c)} \equiv \zeta + \theta$.

For all variables $\gamma$,

$$\zeta \cdot \theta_{\zeta(c)}(\gamma^+) = \zeta(\gamma^+) \qquad\qquad (\zeta + \theta)(\gamma^+) = \zeta(\gamma^+) \sqcup \gamma$$

Since $\gamma \leqslant \zeta(\gamma^+)$ by stability, $\zeta(\gamma^+) = \zeta(\gamma^+) \sqcup \gamma$. A similar argument holds in the negative case, when $\gamma \neq \alpha$. In case $\alpha^-$, we recall that $\theta_{\zeta(c)} = \bigsqcap_n t'_n$, where $t'_n = [\alpha \sqcap \zeta(t^-)/\alpha^-]^n(\alpha^-)$ as in Lemma 45. We prove that $\zeta(t'_n) = \zeta(\alpha^-) \sqcap t'_n$. Let $\xi$ be the bisubstitution $[\alpha \sqcap \zeta(t^-)/\alpha^-]$, so that $t'_{n+1} = \xi(t'_n)$. We show that $\zeta\xi = \zeta + \xi$ as follows (it is trivial for variables other than $\alpha^-$):

$$\zeta\xi(\alpha^-) = \zeta(\alpha^-) \sqcap \zeta^2(t^-) = \zeta(\alpha^-) \sqcap \zeta(t^-) = (\zeta + \xi)(\alpha^-)$$

Therefore, by induction on $n$, $\zeta(t'_n) = \zeta(\alpha^-) \sqcap t'_n$, since in case $n = 0$ this simplifies to $\zeta(\alpha^-) = \zeta(\alpha^-) \sqcap \alpha$ as above, while in case $n + 1$,

$$\zeta(t'_{n+1}) = \zeta(\xi(t'_n)) = \zeta(t'_n) + \xi(t'_n) = \zeta(\alpha^-) \sqcap t'_n \sqcap t'_{n+1} = \zeta(\alpha^-) \sqcap t'_{n+1}$$

Since $\theta_{\zeta(c)}(\alpha^-)$ is the meet of $t'_n$, this proves that $\zeta(\theta_{\zeta(c)}(\alpha^-)) = \zeta(\alpha^-) \sqcap \theta_{\zeta(c)}(\alpha^-)$ and therefore that $\zeta\theta = \zeta + \theta$.

(SUB): We have $\mathsf{biunify}(\zeta H \cup \{\zeta c\}; \mathsf{subi}(\zeta c), \zeta C) = \xi$. By Proposition 48, $\mathsf{subi}(\zeta c) = \zeta(\mathsf{subi}(c))$, so the inductive hypothesis applies showing $\xi\zeta$ to be stable. □

As a corollary, if $\mathsf{biunify}(H; C) = \xi$ then $\xi$ is stable, by the above lemma taking $\zeta = 1$.

### 5.4.5 Biunification of unsatisfiable constraints

Here, we prove the easier half of the correctness result for $\mathsf{biunify}$, showing that it fails only on unsatisfiable results.

**Theorem 59.** *If* $\mathsf{biunify}(H; C)$ *fails, then* $C$ *is unsatisfiable.*

*Proof.* Induction on the structure of recursive calls to $\mathsf{biunify}$. Case (EMP) cannot fail, so assume $C = c, C'$.

If $\mathsf{biunify}$ immediately fails because $c$ is not in $H$, not atomic, and $\mathsf{subi}(c)$ fails, then $c$ (and therefore $C$) is unsatisfiable by Proposition 46.

Otherwise, a failure must be due to a recursive call to $\mathsf{biunify}$. In cases (HYP) and (SUB), the argument to the recursive call (either $C'$ or $\mathsf{subi}(c), C'$)

implies C (by Proposition 47), so if the argument is unsatisfiable then so is C. Case (ATOM) is similarly handled, using Lemma 57 (with Lemmas 54 and 58 providing the required stability preconditions) to show that unsatisfiablity of $\theta_c C'$ implies unsatisfiablity of C. □

### 5.4.6 Atomic subconstraints suffice

Biunification works by breaking complex constraints down into atomic subconstraints, like standard unification. To justify this, we must show that the resulting collection of atomic constraints is equivalent to the original collection of complex constraints, but this is made tricky by the presence of recursive types.

We define the set of *subconstraints* of a constraint is the least set containing it and closed under $\mathsf{subi}$, and write $\mathsf{at}(t^+ \leqslant t^-)$ for those subconstraints of $t^+ \leqslant t^-$ which are atomic, and $\mathsf{at}(C)$ for the union of $\mathsf{at}(t^+ \leqslant t^-)$ for $t^+ \leqslant t^- \in C$.

**Proposition 60.** *Any constraint has only finitely many subconstraints.*

*Proof.* Define the set $\mathsf{subt}(t^+)$ of *immediate subterms* of a positive type term $t^+$ as follows:

$$\begin{aligned}
\mathsf{subt}(\alpha) &= \{\} \\
\mathsf{subt}(t_1^+ \sqcup t_2^+) &= \{t_1^+, t_2^+\} \\
\mathsf{subt}(\bot) &= \{\} \\
\mathsf{subt}(\mathsf{unit}) &= \{\} \\
\mathsf{subt}(t^- \to t^+) &= \{t^-, t^+\} \\
\mathsf{subt}(\{\ell_1 : t_1, \ldots, \ell_n : t_n\}) &= \{t_1^+, \ldots t_n^+\} \\
\mathsf{subt}(\mu\alpha.t^+) &= \{t^+[\mu\alpha.t^+/\alpha]\}
\end{aligned}$$

The immediate subterms of a negative type term are defined dually, and the *subterms* of a positive or negative type term $t^+$ or $t^-$ is the least set containing $t^+$ or $t^-$ and closed under $\mathsf{subt}$. Immediate subterms are either syntactic subterms, or the one-step unrolling in the case of recursive types. Although naively computing subterms will fail to terminate (as recursive types can be unrolled arbitrarily), each positive or negative type term nonetheless has a finite set of subterms, since the recursion is regular.

By inspection of $\mathsf{subi}$, subconstraints are composed of pairs of subterms of the original constraint, of which there are only finitely many. □

An assignment $\rho$ satisfies constraints C iff $\rho$ satisfies $\mathsf{at}(C)$, hence solving the atomic subconstraints of C suffices to solve C. We prove this result in two parts, first proving it in the absence of recursive types and then extending the result:

**Lemma 61.** *If* $C$ *is satisfiable and contains no occurrence of* $\mu$*, then* $\rho \models C$ *iff* $\rho \models \mathsf{at}(C)$.

*Proof.* By induction on the total size of C, considering an arbitrary $t^+ \leqslant t^- \in C$. By Proposition 46, either $t^+ \leqslant t^-$ is atomic or $\mathsf{subi}(t^+ \leqslant t^-)$ succeeds (we know by assumption that it is not unsatisfiable).

If it is atomic, then trivially $\rho \models t^+ \leqslant t^-$ iff $\rho \models at(t^+ \leqslant t^-)$. If $\mathsf{subi}(t^+ \leqslant t^-)$ succeeds, then the result follows by the induction hypothesis (noting that the total size of $\mathsf{subi}(t^+ \leqslant t^-)$ is strictly smaller than that of $t^+ \leqslant t^-$, when $t^+ \leqslant t^-$ contains no $\mu$) and using Proposition 47 to show equivalence. □

**Lemma 62.** *If* $C$ *is satisfiable, then* $\rho \models C$ *iff* $\rho \models \mathsf{at}(C)$*.*

*Proof.* Recall from Section 3.4.1 the *projection morphisms* $\pi_k$, which truncate a type to $k$ levels of type constructor nesting. By definition of $\mathcal{T}$, $\tau_1 \leqslant \tau_2$ iff $\pi_k\tau_1 \leqslant \pi_k\tau_2$ for all $k$.

We write $\rho \models_k C$ ("$\rho$ satisfies $C$ to depth $k$") when $\pi_k(\rho(t^+)) \leqslant \pi_k(\rho(t^-))$ for $t^+ \leqslant t^- \in C$. Informally, $\rho \models_k C$ when $\rho$ satisfies $C$ as long as we do not look under more than $k$ type constructors.

Recursive types $\mu\alpha.t$ are equal to their unrollings $[\mu\alpha.t/\alpha]t$, so we may unroll any recursive types appearing in $C$ without affecting whether $\rho \models C$. By unrolling all $\mu$-types, we increase the minimum depth (constructor nesting level) at which a $\mu$ may be found by at least 1, since recursive types are guarded (see Section 5.1.1).

Form $C_k$ from the constraints $C$, by unrolling all $\mu$-types $k$ times, and then truncating the result at depth $k$ (that is, replacing all subterms below $k$ or more type constructors with $\bot$ or $\top$). Since unrolling preserves equivalence, and the truncation affects the type only below $k$ constructors, we have:

$$\rho \models_k C_k \text{ iff } \rho \models_k C$$

The set $\mathsf{at}(C_k)$ consists of truncated versions of some elements of $\mathsf{at}(C)$. It is not the case that $\mathsf{at}(C_k) = \mathsf{at}(C)_k$, since not all constraints are truncated together, but since the constraints in $\mathsf{at}(C_k)$ are all truncated versions of constraints in $\mathsf{at}(C)$, we do have:

$$\rho \models \bigcup_k \mathsf{at}(C_k) \text{ iff } \rho \models \mathsf{at}(C)$$

Since $C_k$ contains no $\mu$ (all occurrences of $\mu$ having been removed by truncation), $\rho \models_k C_k$ iff $\rho \models \mathsf{at}(C_k)$ by Lemma 61, so:

$$\begin{aligned}
\rho \models C &\Leftrightarrow \forall k.\rho \models_k C \\
&\Leftrightarrow \forall k.\rho \models_k C_k \\
&\Leftrightarrow \forall k.\rho \models_k \mathsf{at}(C_k) \\
&\Leftrightarrow \rho \models \bigcup_k \mathsf{at}(C_k) \\
&\Leftrightarrow \rho \models \mathsf{at}(C)
\end{aligned}$$ □

**Lemma 63.** *For stable* $\xi$ *and satisfiable* $C$*,* $\rho \models \mathsf{at}(\xi(C))$ *iff* $\rho \models \xi(\mathsf{at}(C))$

*Proof.* For non-atomic constraints $c \in C$, $\mathsf{subi}(\xi(c)) = \xi(\mathsf{subi}(c))$ by Proposition 48, while for atomic constraints $c \in C$, $\rho \models \mathsf{at}(\xi(c))$ iff $\rho \models \xi(c)$ (by Lemma 62) iff $\rho \models \xi(\mathsf{at}(c))$ (since $\mathsf{at}(c) = \{c\}$). □

### 5.4.7 Biunification of satisfiable constraints

Finally, we have assembled enough machinery to argue that if $\mathsf{biunify}(\emptyset; C)$ succeeds, then it produces a bisubstitution solving $C$.

The first argument $H$ to biunify is essentially a memoisation table, to handle recursive types and improve performance. We first show the correctness of a version of biunification without it, dubbed $\mathsf{biunify}'$:

$$\begin{aligned}
\mathsf{biunify}'(\epsilon) &= 1 && \text{(EMP')}\\
\mathsf{biunify}'(c, C) &= \mathsf{biunify}'(\theta_c C) \cdot \theta_c \text{ if } c \text{ atomic} && \text{(ATOM')}\\
\mathsf{biunify}'(c, C) &= \mathsf{biunify}'(\mathsf{subi}(c), C) \text{ if } \mathsf{subi}(c) \text{ succeeds} && \text{(SUB')}
\end{aligned}$$

**Lemma 64.** *If* $\mathsf{biunify}'(C) = \xi$*, then* $\xi$ *solves* $\mathsf{at}(C)$.

*Proof.* Induction on the structure of recursive calls to $\mathsf{biunify}'$. In case (EMP'), $1$ trivially solves $\epsilon$, while in case (SUB') we note that $c, C$ and $\mathsf{subi}(c), C$ have the same atomic subconstraints, so the inductive hypothesis applies directly.

In case (ATOM'), since $c$ is atomic, we have:

$$\mathsf{at}(c, C) = \{c\} \cup \mathsf{at}(C)$$

Let $\xi$ be the output of the recursive call to $\mathsf{biunify}'(\theta_c C)$. By the inductive hypothesis, $\xi$ solves $\mathsf{at}(\theta_c C)$, and we must show that $\xi \cdot \theta_c$ solves $\{c\} \cup \mathsf{at}(C)$.

By Lemma 54, $\theta_c$ is stable, and therefore by Lemma 58 so are $\xi$ and $\xi \cdot \theta_c$. This lets us apply Lemma 56 to show that $\xi \cdot \theta_c$ solves $\{c\} \cup \mathsf{at}(C)$, since $\theta_c$ solves $c$ (Theorem 55) and $\xi$ solves $\theta_c(\mathsf{at}(C))$ (Lemma 63). □

So results that $\mathsf{biunify}'(C)$ produces solve $C$, although it may fail to terminate when it encounters recursive types. The addition of a memoisation table $H$ solves this:

**Theorem 65.** *If* $\mathsf{biunify}(\emptyset; C) = \xi$*, then* $\xi$ *solves* $C$.

*Proof.* $\mathsf{biunify}$ defined the same way as $\mathsf{biunify}'$, except for the extra argument $H$ and rule (HYP). $\mathsf{biunify}$ does a depth-first search of the atomic constraints that $\mathsf{biunify}'$ visits, using $H$ to mark already-visited nodes so that they can be skipped on a revisit. When an atomic constraint $c$ is solved, $\theta_c H$ is marked visited, since Lemma 63 combined with condition (Ia) shows that that for any $\rho \models c$, $\rho \models \mathsf{at}(H)$ iff $\rho \models \theta_c \mathsf{at}(H)$. $\mathsf{biunify}$ therefore produces a result $\xi$ solving $\mathsf{at}(C)$ (by the same argument as Lemma 64), which solves $C$ by Lemma 62. □

The argument that $\mathsf{biunify}$ does indeed terminate is left until Section 7.4.1, as it is simpler using the representation of types introduced in Chapter 7.

# 6 Principal type inference

*It is not really difficult to construct a series of inferences, each dependent upon its predecessor and each simple in itself. If, after doing so, one simply knocks out all the central inferences and presents one's audience with the starting-point and the conclusion, one may produce a startling, though perhaps a meretricious, effect.*

—Sir Arthur Conan Doyle

This chapter describes the type inference algorithm for MLsub. The algorithm is much the same as Milner's Algorithm $\mathcal{W}$, with two important differences: first, biunification (Section 5.2) replaces unification, and second, the use of the reformulated typing rules makes the treatment of let-polymorphism slightly different.

## 6.1 Principality

We say that an expression $e$ is *typeable under* $\Pi$ if $\Pi \Vdash e : [\Delta]\tau$ for some $[\Delta]\tau$. A particular typing scheme $[\Delta_{\mathsf{pr}}]\tau_{\mathsf{pr}}$ is said to be *principal* for $e$ under $\Pi$ if the following holds, for all $[\Delta]\tau$:

$$\Pi \Vdash e : [\Delta]\tau \text{ iff } [\Delta_{\mathsf{pr}}]\tau_{\mathsf{pr}} \leqslant^{\forall} [\Delta]\tau$$

In other words, a principal typing scheme subsumes all others. A principal typing scheme, if it exists, is unique up to $\equiv^{\forall}$, so we also speak of *the* principal typing scheme (of $e$ under $\Pi$). This is a direct analogue of Damas's [DM82] principality result for the Hindley-Milner type system, which in the reformulated typing rules states that if $\Pi \Vdash e : [\Delta]\tau$ then $[\Delta]\tau$ is a substitution instance of the principal typing scheme $[\Delta_{\mathsf{pr}}]\tau_{\mathsf{pr}}$. However, the principality property is rather awkward to state in the traditional rules, as substitutions must be applied both to the context $\Gamma$ and the type $\tau$.

The main result of this chapter, which motivates the efficient representation studied in the following chapters, is that not only does every expression have a principal typing scheme, but that principal typing scheme is polar.

**Proposition 66.** *If $e$ is typeable under polar typing environment $\Pi$, then there exists a typing scheme $[D^{-}_{\mathsf{pr}}]t^{+}_{\mathsf{pr}}$ which is both polar and principal for $e$ under $\Pi$.*

As well as a standard principality result for the polar version of the MLsub type system, this states a slightly stronger property: not only is there a best polar typing scheme, but that polar typing scheme subsumes any other typing schemes, even non-polar ones.

Due to the equivalence of the reformulated rules with the original rules (Section 4.3.1), a principality result for the original rules is a straightforward corollary:

**Corollary 67.** *If* $\vdash e : \tau$*, then* $e$ *has a positive principal type* $t^+_{\mathsf{pr}}$*, such that*

$$\vdash e : \tau' \textit{ implies } \exists \rho.\, \rho(t^+_{\mathsf{pr}}) \leqslant \tau'$$

*Proof.* If $\vdash e : \tau$, then $\Vdash e : [\,]\tau$ by Theorem 35, and there is some $[D^-_{\mathsf{pr}}]t^+_{\mathsf{pr}} \leqslant^{\forall} [\,]\tau$ by Proposition 66. By definition of $\leqslant^{\forall}$, $[D^-_{\mathsf{pr}}]$ must be empty, and $\rho(t^+_{\mathsf{pr}}) \leqslant \tau$ for some $\rho$. □

The remainder of this chapter consists of a proof of Proposition 66, which can be read as an algorithm for computing principal types, or determining that none exist. We develop this algorithm case-by-case, along with its proof. The complete algorithm is collected afterwards as a set of inference rules, in Fig. 6.1.

### 6.1.1 Example

As an example, consider the problem of finding a principal typing scheme for the following expression:

$$(\lambda x.\, x.\mathtt{foo}.\mathtt{bar})\{\mathtt{foo} = y\}$$

Let's label the function $(\lambda x.\, x.\mathtt{foo}.\mathtt{bar})$ as $e_1$ and its argument $\{\mathtt{foo} = y\}$ as $e_2$. Suppose that we have already computed the principal typing schemes for $e_1$ and $e_2$:

$$\begin{aligned}(\lambda x.\, x.\mathtt{foo}.\mathtt{bar}) &: [\,]\{\mathtt{foo} : \{\mathtt{bar} : \alpha\}\} \to \alpha\\ \{\mathtt{foo} = y\} &: [y : \beta]\{\mathtt{foo} : \beta\}\end{aligned}$$

It is convenient to assume that the type variables used in the principal type of $e_1$ are distinct from those used in the principal type of $e_2$, which we may always do since typing schemes are closed and permit $\alpha$-renaming (see Section 4.2.1).

The typing rule $(\textsc{App})$ cannot be directly applied to the principal types of $e_1$ and $e_2$, since they are not of the correct form. However, the application $e_1\, e_2$ may be typed by first applying $(\textsc{Sub})$ to $e_1$:

$$\frac{\Vdash e_1 : [\,]\{\mathtt{foo} : \{\mathtt{bar} : \alpha\}\} \to \alpha}{\Vdash e_1 : [y : \{\mathtt{bar} : \gamma\}]\{\mathtt{foo} : \{\mathtt{bar} : \gamma\}\} \to \gamma}\ (\textsc{Sub})$$

and then to $e_2$:

$$\frac{\Vdash e_2 : [y : \beta]\{\mathtt{foo} : \beta\}}{\Vdash e_2 : [y : \{\mathtt{bar} : \gamma\}]\{\mathtt{foo} : \{\mathtt{bar} : \gamma\}\}}\ (\textsc{Sub})$$

This brings the typing schemes into the correct form for the hypotheses of $(\textsc{App})$, giving

$$\Vdash e_1\, e_2 : [y : \{\mathtt{bar} : \gamma\}]\gamma$$

This is a typing scheme for $e_1\, e_2$, but how do we know that it is principal, and if so how may we construct it algorithmically? First, we note that any typing derivation for the principal typing scheme of $e_1\ e_2$ must have the same basic shape as the above:

$$\dfrac{\dfrac{\Vdash e_1 : [\,]\{\mathtt{foo} : \{\mathtt{bar} : \alpha\}\} \to \alpha}{\Vdash e_1 : [\Delta]\tau \to \tau'}\ (\textsc{Sub}) \quad \dfrac{\Vdash e_2 : [y : \beta]\{\mathtt{foo} : \beta\}}{\Vdash e_2 : [\Delta]\tau}\ (\textsc{Sub})}{\Vdash e_1\ e_2 : [\Delta]\tau'}\ (\textsc{App})$$

It is safe to assume that $(\textsc{Sub})$ is applied exactly once to the hypotheses, since reflexivity allows us to insert $(\textsc{Sub})$ if it is absent, and transitivity allows us to collapse multiple uses to one. It is also safe to assume that $(\textsc{Sub})$ is not applied to the conclusion, since to do so always gives a less general typing scheme, by definition.

For the above to be valid, we must have

$$\begin{aligned} [\,]\{\mathtt{foo} : \{\mathtt{bar} : \alpha\}\} \to \alpha &\leqslant^{\forall} [\Delta]\tau \to \tau' \\ [y : \beta]\{\mathtt{foo} : \beta\} &\leqslant^{\forall} [\Delta]\tau \end{aligned}$$

By the definition of $\leqslant^{\forall}$, this amounts to finding an substitution $\rho$ such that the following constraints hold. Note that since we ensured the type variables in both expressions are distinct, we may use the same $\rho$ for both subsumptions.

$$\begin{aligned} \rho(\{\mathtt{foo} : \{\mathtt{bar} : \alpha\}\} \to \alpha) &\leqslant \tau \to \tau' \\ \Delta(y) &\leqslant \rho(\beta) \\ \rho(\{\mathtt{foo} : \beta\}) &\leqslant \tau \end{aligned}$$

The constraint on $\Delta$ can easily be removed by choosing $\Delta = [y : \rho(\beta)]$, a choice which is principal since it is more general than any other $\Delta$ satisfying the above. Solutions to the other two constraints consist of a choice of $\rho$, $\tau$ and $\tau'$. To reduce the number of unknowns, we introduce two fresh type variables $\delta$ and $\epsilon$ and choose $\tau = \rho(\delta)$ and $\epsilon = \rho(\tau')$. For brevity, we introduce the notation $\rho \models \tau_1 \leqslant \tau_2$ as shorthand for $\rho(\tau_1) \leqslant \rho(\tau_2)$, so that the remaining two constraints can be equivalently rewritten as:

$$\rho \models \begin{aligned} \{\mathtt{foo} : \{\mathtt{bar} : \alpha\}\} \to \alpha &\leqslant \delta \to \epsilon \\ \{\mathtt{foo} : \beta\} &\leqslant \delta \end{aligned}$$

The challenge is to produce a typing scheme which subsumes every typing scheme $[y : \rho(\beta)]\rho(\epsilon)$ for $\rho$ satisfying the above constraints. Most previous work on inference with subtyping punts this question by introducing constraints directly to the syntax of typing schemes, producing as a principal typing scheme something like

$$[y : \beta]\epsilon \mid \{\mathtt{foo} : \{\mathtt{bar} : \alpha\}\} \to \alpha \leqslant \delta \to \epsilon,\ \{\mathtt{foo} : \beta\} \leqslant \delta$$

In the current work, we can use biunification to eliminate these subtyping constraints, just as the Hindley-Milner type inference algorithm uses unification to eliminate equality constraints.

## 6.2 Principal type inference

For every typing environment $\Pi$ and expression $e$, define the *typing set* $(\Pi \Vdash e)$ to be the following set of typing schemes:

$$(\Pi \Vdash e) = \{[\Delta]\tau \mid \Pi \Vdash e : [\Delta]\tau\}$$

Due to the (SUB) rule, typing sets are always closed under subsumption: if $[\Delta_2]\tau_2 \leqslant^{\forall} [\Delta_1]\tau_1$ and $[\Delta_2]\tau_2 \in (\Pi \Vdash e)$, then $[\Delta_1]\tau_1 \in (\Pi \Vdash e)$.

If an expression $e$ has a principal typing scheme $[\Delta_{\mathsf{pr}}]\tau_{\mathsf{pr}}$ under $\Pi$, then the typing set $(\Pi \Vdash e)$ contains exactly those typing schemes subsumed by $[\Delta_{\mathsf{pr}}]\tau_{\mathsf{pr}}$. That is,

$$(\Pi \Vdash e) = \{[\Delta]\tau \mid [\Delta_{\mathsf{pr}}]\tau_{\mathsf{pr}} \leqslant^{\forall} [\Delta]\tau\}$$

Using the language of Section 5.2.3, $e$ has a principal typing scheme $[\Delta_{\mathsf{pr}}]\tau_{\mathsf{pr}}$ under $\Pi$ exactly when the typing set $(\Pi \Vdash e)$ is the set of instances of $[\Delta_{\mathsf{pr}}]\tau_{\mathsf{pr}}$.

So, the principality property (Section 6.1) can be restated as follows: For every polar typing environment $\Pi$ and expression $e$, the typing set $(\Pi \Vdash e)$ is either empty or is the set of instances of some polar typing scheme $[D^-]t^+$.

Returning to our example application $e_1\ e_2$, we see that the typing set $(\Vdash e_1\ e_2)$ is:

$$\left\{ [\Delta]\tau \;\middle|\; \exists\rho \models \begin{array}{r} \texttt{\{foo : \{bar : }\alpha\texttt{\}\}} \to \alpha \leqslant \delta \to \epsilon \\ \texttt{\{foo : }\beta\texttt{\}} \leqslant \delta \end{array},\ \rho([y : \beta]\epsilon) \leqslant [\Delta]\tau \right\}$$

Again using the language of Section 5.2.3, we see that this is exactly the set of instances of $[y : \beta]\epsilon$ under the constraints C defined as follows:

$$\begin{array}{r} \texttt{\{foo : \{bar : }\alpha\texttt{\}\}} \to \alpha \leqslant \delta \to \epsilon \\ \texttt{\{foo : }\beta\texttt{\}} \leqslant \delta \end{array}$$

Given a bisubstitution $\xi$ that solves C, we have:

$$(\Vdash e_1\ e_2) = \xi([y : \beta]\epsilon)$$

The biunification algorithm will either produce such a $\xi$, in which case $\xi([y : \beta]\epsilon)$ is a principal typing scheme for $e_1\ e_2$, or prove that C is unsatisfiable, in which case $e_1\ e_2$ is untypeable and $(\Vdash e_1\ e_2)$ is empty.

Our task is to develop an algorithm which constructs a principal polar typing scheme for any input program $e$, in any polar typing context $\Pi$, or shows that none exists. We do this by induction on $e$, using biunification to eliminate any constraints that arise.

### 6.2.1 Principality for functions

We begin with λ-bound variables, functions and λ-abstractions.

**λ-bound variables** Suppose $e = x$. λ-bound variables are typeable using (VAR-Δ) at any typing scheme of the form $[x : \tau]\tau$. The typing scheme $[x : \alpha]\alpha$ subsumes all of these (consider the substitution $\rho(\alpha) = \tau$), so the principal typing scheme is $[x : \alpha]\alpha$.

Note that we do not consult $\Pi$ here: unlike the Hindley-Milner type system, the reformulated typing rules keep the types of λ-bound variables on the right of the turnstile.

**λ-abstractions** Suppose $e = \lambda x.e$. λ-abstractions are typeable using (ABS), which requires that $e$ be typeable. Thus, if $(\Pi \Vdash e)$ is empty, then so is $(\Pi \Vdash \lambda x.e)$. Otherwise, suppose by the inductive hypothesis that $e$ has principal polar typing scheme $[D^-]t^+$ under $\Pi$.

As we noted before, any number of applications of (Sub) can be collapsed to just one, so the general form of a typing derivation for $\lambda x.e$ is:

$$\dfrac{\dfrac{\Pi \Vdash e : [D^-]t^+}{\Pi \Vdash e : [\Delta, x : \tau]\tau'}\,(\textsc{Sub})}{\Pi \Vdash \lambda x.e : [\Delta]\tau \to \tau'}\,(\textsc{Abs})$$

For (Sub) to be applicable, we must have:

$$[D^-]t^+ \leqslant^{\forall} [\Delta, x : \tau]\tau'$$

Write $D^-_x$ for $D^-$ with $x$ removed from its domain, and note that the above subsumption is equivalent to:

$$[D^-_x, x : D^-(x)]t^+ \leqslant^{\forall} [\Delta, x : \tau]\tau'$$

By function subtyping, this is equivalent to:

$$[D^-_x]D^-(x) \to t^+ \leqslant^{\forall} [\Delta]\tau \to \tau'$$

This means that the order of (Abs) and (Sub) can be swapped above, giving this derivation of the same typing scheme:

$$\dfrac{\dfrac{\dfrac{\Pi \Vdash \lambda x.e : [D^-]t^+}{\Pi \Vdash e : [D^-_x, x : D^-(x)]t^+}\,(\textsc{Sub})}{\Pi \Vdash \lambda x.e : [D^-_x]D^-(x) \to t^+}\,(\textsc{Abs})}{\Pi \Vdash \lambda x.e : [\Delta]\tau \to \tau'}\,(\textsc{Sub})$$

This shows that any typing scheme for $\lambda x.e$ under $\Pi$ is an instance of the polar typing scheme $[D^-_x]D^-(x) \to t^+$, so this is the principal typing scheme.

**Applications** Suppose $e = e_1\, e_2$. As before, if either $(\Pi \Vdash e_1)$ or $(\Pi \Vdash e_2)$ is empty, then $e_1\, e_2$ is untypeable and $(\Pi \Vdash e_1\, e_2)$ is also empty. Otherwise, suppose $[D^-_1]t^+_1$ and $[D^-_2]t^+_2$ are the principal polar typing schemes of $e_1$ and $e_2$.

The rule (Abs) requires that $e_1$ be of type $\tau \to \tau'$ and $e_2$ be of type $\tau$, for some $\tau, \tau'$, and we may need to use rule (Sub) on one or both of them to bring this about. By transitivity, we need apply (Sub) only once. So, typing derivations for $e_1\, e_2$ look like:

$$\dfrac{\dfrac{\Pi \Vdash e_1 : [D^-_1]t^+_1}{\Pi \Vdash e_1 : [\Delta]\tau \to \tau'}\,(\textsc{Sub}) \qquad \dfrac{\Pi \Vdash e_2 : [D^-_2]t^+_2}{\Pi \Vdash e_2 : [\Delta]\tau}\,(\textsc{Sub})}{\Pi \Vdash e_1\, e_2 : [\Delta]\tau'}\,(\textsc{App})$$

The applications of (Sub) require

$$[D^-_1]t^+_1 \leqslant^{\forall} [\Delta]\tau \to \tau' \qquad\qquad [D^-_2]t^+_2 \leqslant^{\forall} [\Delta]\tau$$

The subsumption relation $\leqslant^{\forall}$ is defined in terms of type variable substitutions (see Section 4.2). Since typing schemes are closed, we can assume that $[D^-_1]t^+_1$ and $[D^-_2]t^+_2$ have no type variables in common, allowing us to use a single substitution for both subsumptions. So, we require an substitution $\rho$ such that

$$\begin{aligned} \rho(t^+_1) &\leqslant \tau \to \tau' & \qquad \rho(t^+_2) &\leqslant \tau \\ \Delta &\leqslant \rho(D^-_1) & \qquad \Delta &\leqslant \rho(D^-_2) \end{aligned}$$

By introducing two fresh type variables $\alpha, \beta$, included in the domain of $\rho$, and taking $\tau = \rho(\alpha), \tau' = \rho(\beta)$, we can write the typing set $(\Pi \Vdash e)$ in the following form:

$$\left\{[\Delta]\tau \;\mid\; \exists\rho.\ \rho(t_1^+) \leqslant \rho(\alpha \to \beta),\ \rho(t_2^+) \leqslant \rho(\alpha),\ \rho([D_1^- \sqcap D_2^-]\beta) \leqslant [\Delta]\tau\right\}$$

This is equivalent to the instances of

$$[\xi(D_1^- \sqcap D_2^-)]\xi(\beta^+)$$

for some $\xi$ which solves the constraints $C$:

$$C = \{t_1^+ \leqslant \alpha \to \beta,\ t_2^+ \leqslant \alpha\}$$

Such a $\xi$ can be found as $\mathsf{biunify}(C)$, giving the principal polar typing scheme $\xi([D_1^- \sqcap D_2^-]\beta^+)$.

### 6.2.2 Principality for booleans and records

Inferring principal types for booleans and records is similar to inferring types for functions. They follow the same pattern as above: introduction rules, which do not require their hypotheses to be of a particular form, do not require use of biunification, while elimination rules do.

**Boolean literals** Suppose $e = \text{true}$ or $e = \text{false}$: These are typeable using (True) and (False), both giving principal type $(\Pi \Vdash e) = []\text{bool}$.

**Conditionals** Suppose $e = \text{if } e_1 \text{ then } e_2 \text{ else } e_3$: If any of $e_1$, $e_2$, $e_3$ are untypeable, then so is $e$. Otherwise, assume that $e_1, e_2, e_3$ have polar principal typing schemes $[D_1^-]t_1^+$, $[D_2^-]t_2^+$, $[D_3^-]t_3^+$ under $\Pi$.

By a similar line of reasoning to that for applications above, the typing set $(\Pi \Vdash e)$ is given by:

$$\left\{ [\Delta]\tau \;\middle|\; \exists\rho.\ \begin{array}{l} \rho(t_1^+) \leqslant \text{bool} \\ \rho(t_2^+) \leqslant \rho(\alpha) \\ \rho(t_3^+) \leqslant \rho(\alpha) \end{array},\ \rho([D_1^- \sqcap D_2^- \sqcap D_3^-]\alpha) \leqslant [\Delta]\tau \right\}$$

and the principal type is found by applying biunification to the constraints above.

**Record constructors** Suppose $e = \{\ell_1 = e_1, \ldots, \ell_n = e_n\}$: Like $\lambda$-abstraction, typing the record construction syntax is just a matter of shuffling types around. If any of the components $e_i$ are untypeable, then so is $e$. Otherwise, we have polar principal typing schemes $[D_i^-]t_i^+$ for $e_i$ under $\Pi$, and the principal typing scheme of $e$ is

$$[D_1^- \sqcap \ldots \sqcap D_n^-]\{\ell_1 : t_1^+, \ldots, \ell_n : t_n^+\}$$

**Record projections** Suppose $e = e_1.\ell$: Again, if $e_1$ is untypeable then so is $e$. Otherwise, $(\Pi \Vdash e_1) = [D_1^-]t_1^+$ and $(\Pi \Vdash e)$ is given by:

$$\left\{[\Delta]\tau \;\middle|\; \exists\rho.\ \rho(t_1^+) \leqslant \{\ell : \alpha\},\ \rho([D_1^-]\alpha) \leqslant [\Delta]\tau\right\}$$

for some fresh type variable $\alpha$, so the principal type is

$$\xi([D_1^-]\alpha)$$

where $\xi = \mathsf{biunify}(t_1^+ \leqslant \{\ell : \alpha\})$.

### 6.2.3 Principality for let-bindings

Due to the construction of the reformulated typing rules (Section 4.3), it is relatively straightforward to infer principal types for let-bindings and let-bound variables. In particular, since typing schemes are closed (Section 4.2.1), we need not worry about free variables in typing schemes.

**Let-bound variables** Suppose $e = \hat{x}$: If $\hat{x} \notin \operatorname{dom} \Pi$, then $\hat{x}$ is untypeable. Otherwise, the principal typing scheme is simply $\Pi(\hat{x})$.

**Let-bindings** Suppose $e = \text{let } \hat{x} = e_1 \text{ in } e_2$: If $e_1$ is untypeable, then so is $e$. Otherwise, $e_1$ has principal type $[D_1^-]t_1^+$ under $\Pi$, and a general typing derivation for $e$ looks like:

$$\dfrac{\dfrac{\Pi \Vdash e_1 : [D_1^-]t_1^+}{\Pi \Vdash e_1 : [\Delta_1]\tau_1}\ (\textsc{Sub}) \qquad \Pi, x : [\Delta_1]\tau_1 \Vdash e_2 : [\Delta_2]\tau_2}{\Pi \Vdash \text{let } x = e_1 \text{ in } e_2 : [\Delta_1 \sqcap \Delta_2]\tau_2}\ (\textsc{Let})$$

The tricky part here is that we have a choice of many different $[\Delta_1]\tau_1$ to insert into $\Pi$, not all of which are polar. So, we cannot directly apply the inductive hypothesis to the right-hand hypothesis above, because $\Pi, x : [\Delta_1]\tau_1$ need not be polar.

However, since $[D_1^-]t_1^+ \leqslant^\forall [\Delta_1]\tau_1$, weakening (Proposition 32) gives us the following:

$$\Pi, x : [D_1^-]t_1^+ \Vdash e_2 : [\Delta_2]\tau_2$$

That is, the most general choice is to insert $x$ into $\Pi$ with the principal type of $e_1$. $\Pi, x : [D_1^-]t_1^+$ is indeed polar, so the inductive hypothesis applies giving a principal typing scheme $[D_2^-]t_2^+$ for $e_2$. Then, we have the following polar principal typing scheme for $e$:

$$[D_1^- \sqcap D_2^-]t_2^+$$

## 6.3 Summary of the algorithm

The algorithm described case-by-case in the previous section can be written compactly as a set of inference rules. We introduce a judgement form $\Pi \lhd e : [D^-]t^+$, stating that $[D^-]t^+$ is the principal typing scheme of $e$ under the polar typing context $\Pi$. This judgement form is defined according to the syntax-directed rules of Fig. 6.1, which is a summary of the previous section.

The elimination rules ((App), (If) and (Proj)) refer to the biunification algorithm in a side-constraint. The type inference algorithm can be implemented by representing typing schemes as syntax and using the biunification algorithm of Section 5.3.3, but this tends to generate overly complex typing schemes slowly. Instead, typing schemes should be represented as automata, and use an optimised version of biunification as described in the next chapter.

$$(\textsc{Var-}\Pi) \quad \frac{}{\Pi \triangleright x : [D^-]t^+} \quad \Pi(x) = [D^-]t^+$$

$$(\textsc{Var-}\Delta) \quad \frac{}{\Pi \triangleright x : [x : \alpha]\alpha}$$

$$(\textsc{Abs}) \quad \frac{\Pi \triangleright e : [D^-]t^+}{\Pi \triangleright \lambda x.\, e : [D^-_x]D^-(x) \to t^+}$$

$$(\textsc{App}) \quad \frac{\Pi \triangleright e_1 : [D^-_1]t^+_1 \quad \Pi \triangleright e_2 : [D^-_2]t^+_2}{\Pi \triangleright e_1\ e_2 : \xi([D^-_1 \sqcap D^-_2]\alpha)} \quad \xi = \mathsf{biunify}(t^+_1 \leqslant t^+_2 \to \alpha)$$

$$(\textsc{Let}) \quad \frac{\Pi \triangleright e_1 : [D^-_1]t^+_1 \quad \Pi, x : [D^-_1]t^+_1 \triangleright e_2 : [D^-_2]t^+_2}{\Pi \triangleright \text{let } x = e_1 \text{ in } e_2 : [D^-_1 \sqcap D^-_2]t^+_2}$$

$$(\textsc{True}) \quad \frac{}{\Pi \triangleright \text{true} : [\,]\text{bool}}$$

$$(\textsc{False}) \quad \frac{}{\Pi \triangleright \text{false} : [\,]\text{bool}}$$

$$(\textsc{If}) \quad \frac{\Pi \triangleright e_1 : [D^-_1]t^+_1 \quad \Pi \triangleright e_2 : [D^-_2]t^+_2 \quad \Pi \triangleright e_3 : [D^-_3]t^+_3}{\Pi \triangleright \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \xi([D^-_1 \sqcap D^-_2 \sqcap D^-_3]\alpha)} \quad \xi = \mathsf{biunify}\begin{pmatrix} t^+_1 \leqslant \text{bool} \\ t^+_2 \leqslant \alpha \\ t^+_3 \leqslant \alpha \end{pmatrix}$$

$$(\textsc{Cons}) \quad \frac{\Pi \triangleright e_1 : [D^-_1]t^+_1 \quad \ldots \quad \Pi \triangleright e_n : [D^-_n]t^+_n}{\Pi \triangleright \{\ell_1 = e_1, \ldots, \ell_n = e_n\} : [D^-_1 \sqcap \ldots \sqcap D^-_n]\{\ell_1 : t^+_1, \ldots, \ell_n : t^+_n\}}$$

$$(\textsc{Proj}) \quad \frac{\Pi \triangleright e : [D^-]t^+}{\Pi \triangleright e.\ell : \xi([D^-]\alpha)} \quad \xi = \mathsf{biunify}(t^+ \leqslant \{\ell : \alpha\})$$

Figure 6.1: Type inference algorithm

# 7 Representation of types

*There can never be surprises in logic.*

—Ludwig Wittgenstein

*Machines take me by surprise with great frequency.*

—Alan Turing

A common criticism of type systems with subtyping and inference is that they tend to produce large and unwieldy types. These types then require *simplification* to convert them into smaller but equivalent types. Systems based on constrained types are particularly prone to creating large types: without simplification, many such systems produce type schemes with a constraint set of size linear in that of the program, even in the absence of polymorphism [HM95].

MLsub's typing schemes do not include a set of constraints, but a naive implementation of the algorithms of the previous chapter will still generate large types. For instance, given the function

$$\lambda x.\ \text{if } x.p \text{ then } x.q \text{ else } x.q$$

a naive implementation of the algorithms assigns the following type

$$(\{p : \text{bool}\} \sqcap \{q : \alpha\} \sqcap \{q : \beta\}) \to (\alpha \sqcup \beta)$$

instead of the simpler, but equivalent

$$\{p : \text{bool},\ q : \alpha\} \to \alpha$$

This chapter presents a *representation* of polar types and polar typing schemes as finite automata, which allows efficient implementation of type inference, as well as a method for simplification which converts complex types into simpler but equivalent forms.

We define *type automata* (Section 7.1), show how they represent positive and negative types (Section 7.1.2), and show how they may be simplified to smaller but equivalent automata (Section 7.2).

We also describe the simplification of typing schemes (Section 7.3), and how typing schemes may be represented by *scheme automata* (Section 7.3.1). Finally, we implement the biunification algorithm of the previous section in terms of automata (Section 7.4), giving an algorithm which runs faster than that of the previous chapter, as well as producing more compact types.

Representing types as some flavour of automata is a standard technique in dealing with recursive types and subtyping. The novelty here is to treat the automata as standard finite automata, accepting a regular language, from which we gain the *representation theorem*:

*Equivalent types have automata accepting equal languages*

Thus, the representation is not simply a mapping from syntactic type terms to automata, but an *embedding* of types into regular languages. In some ways, this is a converse to Henglein and Rehof's result [HR98] on the complexity of subtype comparisons by reduction from NFA problems: the present result is that the two are equivalent[1].

The connection has been informally noted before. It was not lost on Pottier [Pot98b] that the *canonisation* and *minimisation* algorithms strongly resembled the standard algorithms for turning a nondeterministic finite automaton into a minimal deterministic one. However, this is the first time the general relation has been demonstrated.

The practical upshot of this is to admit a large class of new simplification algorithms for types: namely, all of those appearing in the literature on regular languages. Thanks to the representation theorem, any algorithm for simplifying standard finite automata is automatically valid for simplifying types.

## 7.1 Type automata

Types are represented by *type automata*, a slight variant on standard nondeterministic 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:

$$\Sigma_F = \{d, r\} \cup \{\ell \mid \ell \in \mathcal{L}\}$$

The symbols $d$ and $r$ represent the domain and range of function types, and a symbol $\ell$ represents each record label.

The transition table $\delta$ of a type automaton has the condition that $d$-transitions only connect states of differing polarity (due to the contravariance of function domains), while transitions labelled by any other element of $\Sigma_F$ 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 $\Sigma_F$, 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).

[1] However, there are subtle differences between Henglein and Rehof's subtyping order and the present work, making the results incomparable, strictly speaking. See Chapter 10 for details.

### 7.1.1 Head constructors

The set of *head constructors* $H(q)$ are drawn from the set consisting of the symbols $\langle\rightarrow\rangle$ (representing function types), $\langle\mathtt{b}\rangle$ (representing the boolean type) and $\langle L\rangle$ for any set $L$ of record labels $\ell$ (representing record types) and type variables (representing themselves). The elements of $H(q)$ represent components of the lattice of types (see Section 3.2.3). They are partially ordered compatibly with the subtyping order, with the order relation inductively defined as follows:

$$\frac{}{\alpha \leqslant \alpha} \qquad \frac{}{\langle\mathtt{b}\rangle \leqslant \langle\mathtt{b}\rangle} \qquad \frac{}{\langle\rightarrow\rangle \leqslant \langle\rightarrow\rangle} \qquad \frac{L_1 \supseteq L_2}{\langle L_1\rangle \leqslant \langle L_2\rangle}$$

Polar types, in general, are a join (positive types) or meet (negative types) of types constructed from different components. Accordingly, the set $H(q)$ represents the components used in a particular polar type, and so contains at most one type constructor from each component. That is, $H(q)$ never contains two distinct record constructors $\langle L_1\rangle$ and $\langle L_2\rangle$.

For instance, if the head constructors $H(q^+)$ of the positive state $q^+$ are as follows:

$$\{\alpha, \langle\rightarrow\rangle, \langle\mathtt{b}\rangle\}$$

then $q^+$ represents some type $\alpha \sqcup (t_1^+ \rightarrow t_2^+) \sqcup \mathrm{bool}$. Dually, the same set of head constructors on a negative state $q^-$ represents some type $\alpha \sqcap (t_1^- \rightarrow t_2^-) \sqcap \mathrm{bool}$.

We define two operations $\widetilde{\sqcup}$ and $\widetilde{\sqcap}$ on sets of head constructors. Both of these are simply set union, with the invariant of at most one record type maintained by reducing according to the following rules:

$$\langle L_1\rangle \,\widetilde{\sqcup}\, \langle L_2\rangle = \langle L_1 \cap L_2\rangle \qquad\qquad \langle L_1\rangle \,\widetilde{\sqcap}\, \langle L_2\rangle = \langle L_1 \cup L_2\rangle$$

Generally, a type is represented by an automaton state $q$ by representing its type constructor (e.g. function) in $H(q)$, and representing its fields (e.g. domain and range) with transitions.

### 7.1.2 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 5.1 gives the syntax of positive and negative types. 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 $\mu\alpha.\beta \sqcup (\mathrm{bool} \rightarrow \alpha)$ are:

$$\{\, \mu\alpha.\beta \sqcup (\mathrm{bool} \rightarrow \alpha),\ \beta \sqcup (\mathrm{bool} \rightarrow \alpha),\ \beta,\ \mathrm{bool} \rightarrow \alpha,\ \mathrm{bool} \,\}$$

The type automaton for a positive type $t_0$ has one state for every subterm $t$ of $t_0$, written $q(t)$ by slight abuse of notation. The start state $q_0$ is $q(t_0)$. The polarity of $q(t)$ matches that of $t$, $H(q(\alpha)) = \{\alpha\}$, $H(q(\mathrm{bool})) = \{\langle\mathtt{b}\rangle\}$, $H(q(t_1 \rightarrow t_2)) = \{\langle\rightarrow\rangle\}$, and $H(q(\{f\}) = \{\langle\ell\rangle \mid \ell \notin \mathrm{dom}\, f\}$. In all other cases, $H(q(t)) = \emptyset$.

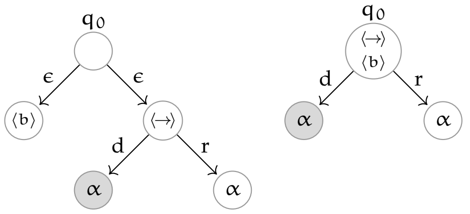


Figure 7.1: Automata construction with ϵ-transitions

We further abuse notation by defining $q(\alpha) = q(\mu\alpha.t_1)$ for every type variable $\alpha$ bound by a recursive type $\mu\alpha.t_1$.

The type automaton has the following transition table:

$$\begin{aligned} q(t_1 \sqcup t_2) &\xrightarrow{\epsilon} q(t_1) & q(t_1 \sqcup t_2) &\xrightarrow{\epsilon} q(t_2) \\ q(t_1 \sqcap t_2) &\xrightarrow{\epsilon} q(t_1) & q(t_1 \sqcap t_2) &\xrightarrow{\epsilon} q(t_2) \\ q(t_1 \to t_2) &\xrightarrow{d} q(t_1) & q(t_1 \to t_2) &\xrightarrow{r} q(t_2) \\ q(\mu\alpha.t_1) &\xrightarrow{\epsilon} q(t_1) & q(\{f\}) &\xrightarrow{\ell} q(f(\ell)) \text{ for } \ell \in \operatorname{dom} f \end{aligned}$$

Finally, we remove ϵ-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 ϵ-transitions, and then set:

$$H(q) = \begin{cases} \widetilde{\bigsqcup}_{q' \in E(q)} H(q') & \text{if } q \text{ positive} \\ \widetilde{\bigsqcap}_{q' \in E(q)} H(q') & \text{if } q \text{ negative} \end{cases}$$

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

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 ϵ-transitions.

As an example, the type automaton for the positive type $(\alpha \to \alpha) \sqcup \text{bool}$ is shown in Figure 7.1, before and after removal of ϵ-transitions. An example with recursive types appears in Figure 7.2a. In both figures, negative states are drawn shaded and $H(q)$ are drawn inside the state $q$.

### 7.1.3 Deconstructing automata

Given an automaton, converting it back to a concrete polar type term is straightforward. For each state $q^+$, $q^-$ of the automaton, we construct a term $t(q^+)$, $t(q^-)$ of matching polarity. We construct $t(q^+)$ as the join of the following elements:

- $\alpha$, if $\alpha \in H(q^+)$
- bool, if $\langle b \rangle \in H(q^+)$

- $t^-_d \to t^+_r$, if $\langle\to\rangle \in H(q^+)$, where:

$$t^-_d = \bigsqcap\{t(q') \mid q^+ \xrightarrow{d} q'\}$$
$$t^+_r = \bigsqcup\{t(q') \mid q^+ \xrightarrow{r} q'\}$$

- $\{f\}$, if $\langle L\rangle \in H(q^+)$, where $\operatorname{dom} f = L$ and:

$$f(\ell) = \bigsqcup\{t(q') \mid q^+ \xrightarrow{\ell} q'\}$$

Note that if none of the above cases apply, $t(q^+)$ is a join of zero elements, or $\bot$. The negative case $t(q^-)$ is constructed dually, as a meet.

Naively implemented, this algorithm does not terminate on automata containing cycles, but this can be avoided by introducing $\mu$-types to handle backward edges.

The polar type term recovered from an automaton is $t(q_0)$, where $q_0$ is the start state.

## 7.2 Simplifying type automata

The constructions described above operate at a very concrete level, turning pieces of syntax (polar type terms) into labelled graphs (automata). In fact, it does not even preserve every detail of the syntax of types, since for instance, $\alpha \sqcup (\bot \sqcup \text{bool})$ and $\text{bool} \sqcup \alpha$ are converted to identical automata, and this automaton is deconstructed back into $\alpha \sqcup \text{bool}$.

In this section, we introduce the *representation theorem*, showing that the construction above can be understood at a higher level, as an embedding of polar types into regular languages:

$$\text{Polar types} \underset{\mathcal{R}}{\overset{\mathcal{E}}{\rightleftarrows}} \text{Regular languages}$$

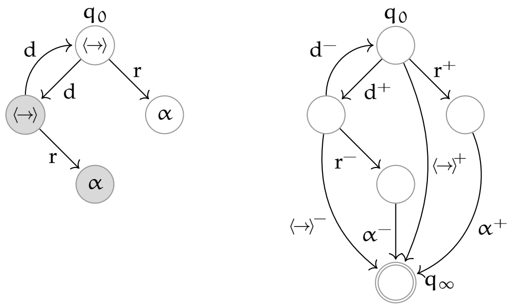
The mapping $\mathcal{E}$ is a formalisation of the last section's construction of automata, turning types into regular expressions, while $\mathcal{R}$ does the opposite job, turning regular expressions into types. We write $E_1 \equiv E_2$ when the regular expressions $E_1$,$E_2$ denote the same language. Below, we show that $\mathcal{E}$ maps equivalent types to equal languages (Section 7.2.1), $\mathcal{R}$ is a left inverse of $\mathcal{E}$ (Section 7.2.2), and that $\mathcal{R}$ maps equal languages to equivalent types (Section 7.2.3). Taken together, these results imply, for any polar types $t_1, t_2$ of the same polarity:

$$t_1 \equiv t_2 \qquad \text{iff} \qquad \mathcal{E}(t_1) \equiv \mathcal{E}(t_2)$$

We need not work hard to develop algorithms for simplifying types: the representation theorem reduces this problem to the well-studied problem of compactly representing regular languages.

### 7.2.1 Encoding types as regular languages

Standard non-deterministic finite automata (NFAs) do not label their states with head constructors, but instead label each state as either "accepting" or "non-accepting". However, using a careful encoding, we can convert type automata to standard NFAs in such a way that the subtyping order is reflected as the subset order between languages.



(a) Type automaton (b) NFA

Figure 7.2: Type automaton and NFA for $\mu\beta.\,(\beta \to \alpha) \to \alpha$.

We work over the alphabet $\Sigma$, which type variables, $\Sigma_F$, and a symbol for each of record, boolean and function types and each record field. Two copies of each symbol appear, labelled by $^+$ and $^-$:

$$\begin{aligned}\Sigma = \;& V^+ \cup V^- \cup \\ & \Sigma_F^+ \cup \Sigma_F^- \cup \\ & \{\langle\to\rangle^+, \langle b\rangle^+, \langle\{\}\rangle^+\} \cup \{\langle \ell\rangle^+ \mid \ell \text{ a label}\} \cup \\ & \{\langle\to\rangle^-, \langle b\rangle^-, \langle\{\}\rangle^-\} \cup \{\langle \ell\rangle^- \mid \ell \text{ a label}\}\end{aligned}$$

The encoding of $H(q^-)$ for a negative state $q^-$ as a language over $\Sigma$ is relatively straightforward. We write $\mathcal{E}^-(h)$ for the encoding of a particular head constructor $h \in H(q^-)$, and later extend $\mathcal{E}$ to work on positive constructors $h \in H(q^+)$ and eventually entire type terms. The result of $\mathcal{E}$ is a regular language, which we write using regular expressions $E$ (where $\lambda$ ranges over $\Sigma$):

$$E ::= \emptyset \mid \epsilon \mid \lambda \mid E + E \mid E \cdot E \mid E^*$$

The notation is standard: elements of $\Sigma$ are used as singleton languages, we write $\cdot$ for concatenation of languages, $+$ for union and $^*$ for Kleene star, while is the empty language (the identity of $+$) and $\epsilon$ is the language containing only the empty string (the identity of $\cdot$).

For a head constructor $h \in H(q^-)$, $\mathcal{E}^-(h)$ is defined as:

$$\begin{aligned}\mathcal{E}^-(\langle\to\rangle) &= \langle\to\rangle^- \\ \mathcal{E}^-(\langle b\rangle) &= \langle b\rangle^- \\ \mathcal{E}^-(\langle L\rangle) &= \langle\{\}\rangle^- + \{\langle \ell\rangle^- \mid \ell \in L\}\end{aligned}$$

Multiple head constructors are combined with $\cup$. That is,

$$\mathcal{E}^-(H(q^-)) = \bigcup_{h \in H(q^-)} \mathcal{E}^-(h)$$

Note that

$$\mathcal{E}^-(H(q_1^-) \,\widetilde{\sqcap}\, H(q_2^-)) = \mathcal{E}^-(H(q_1^-)) + \mathcal{E}^-(H(q_2^-))$$

since $\widetilde{\sqcap}$ on records is defined by taking the union of the fields.

We do something similar for positive states $q^+$. However, since the join of two record types takes the *intersection* of their fields, we must use a slightly trickier encoding. For positive types, we define $\mathcal{E}^+(h)$ in terms of the labels *absent*:

$$
\begin{aligned}
\mathcal{E}^+(\langle\to\rangle) &= \langle\to\rangle^+ \\
\mathcal{E}^+(\langle \mathtt{b}\rangle) &= \langle \mathtt{b}\rangle^+ \\
\mathcal{E}^+(\langle L\rangle) &= \langle\{\}\rangle^+ + \{\langle\ell\rangle^+ \mid \ell \notin L\}
\end{aligned}
$$

By using this encoding of record fields, we have:

$$\mathcal{E}^+(H(q_1^+) \mathbin{\widetilde{\sqcup}} H(q_2^+)) = \mathcal{E}^+(H(q_1^+)) + \mathcal{E}^+(H(q_2^+))$$

We extend this encoding to work on arbitrary polar type terms:

$$
\begin{aligned}
\mathcal{E}^+(\alpha) &= \alpha^+ & \mathcal{E}^-(\alpha) &= \alpha^- \\
\mathcal{E}^+(t_1 \sqcup t_2) &= \mathcal{E}^+(t_1) + \mathcal{E}^+(t_2) & \mathcal{E}^-(t_1 \sqcap t_2) &= \mathcal{E}^-(t_1) + \mathcal{E}^-(t_2) \\
\mathcal{E}^+(\bot) &= \emptyset & \mathcal{E}^-(\top) &= \emptyset \\
\mathcal{E}^+(t_1 \to t_2) &= \ \ d^+ \cdot \mathcal{E}^-(t_1) + r^+ \cdot \mathcal{E}^+(t_2) & \mathcal{E}^-(t_1 \to t_2) &= \ \ d^- \cdot \mathcal{E}^+(t_1) + r^- \cdot \mathcal{E}^-(t_2) \\
&\quad + \mathcal{E}^+(\langle\to\rangle) & &\quad + \mathcal{E}^-(\langle\to\rangle) \\
\mathcal{E}^+(\text{bool}) &= \mathcal{E}^+(\langle \mathtt{b}\rangle) & \mathcal{E}^-(\text{bool}) &= \mathcal{E}^-(\langle \mathtt{b}\rangle) \\
\mathcal{E}^+(\{f\}) &= \ \ \textstyle\bigcup_{\ell\in\text{dom}\, f} \ell^+ \cdot \mathcal{E}^+(f(\ell)) & \mathcal{E}^-(\{f\}) &= \ \ \textstyle\bigcup_{\ell\in\text{dom}\, f} \ell^- \cdot \mathcal{E}^-(f(\ell)) \\
&\quad + \textstyle\bigcup_{\ell\notin\text{dom}\, f} \ell^+ \cdot \Sigma^* & &\quad + \mathcal{E}^-(\langle\text{dom}\, f\rangle) \\
&\quad + \mathcal{E}^+(\langle\text{dom}\, f\rangle) & & \\
\mathcal{E}^+(\mu\alpha.t) &= \mathcal{O}^+_\alpha(t)^* \cdot \mathcal{E}^+([\bot/\alpha]t) & \mathcal{E}^-(\mu\alpha.t) &= \mathcal{O}^-_\alpha(t)^* \cdot \mathcal{E}^-([\top/\alpha]t)
\end{aligned}
$$

The auxiliary operations $\mathcal{O}^\pm_\alpha(t)$ describe the occurrences of a bound variable $\alpha$ within a $\mu$-binder, and are defined as follows:

$$
\begin{aligned}
\mathcal{O}^+_\alpha(\alpha) &= \epsilon & \mathcal{O}^-_\alpha(\alpha) &= \epsilon \\
\mathcal{O}^+_\alpha(\beta) &= \emptyset & \mathcal{O}^-_\alpha(\beta) &= \emptyset \\
\mathcal{O}^+_\alpha(t_1 \sqcup t_2) &= \mathcal{O}^+_\alpha(t_1) + \mathcal{O}^+_\alpha(t_2) & \mathcal{O}^-_\alpha(t_1 \sqcap t_2) &= \mathcal{O}^-_\alpha(t_1) + \mathcal{O}^-_\alpha(t_2) \\
\mathcal{O}^+_\alpha(\bot) &= \emptyset & \mathcal{O}^-_\alpha(\top) &= \emptyset \\
\mathcal{O}^+_\alpha(t_1 \to t_2) &= d^+ \cdot \mathcal{O}^-_\alpha(t_1) + r^+ \cdot \mathcal{O}^+_\alpha(t_2) & \mathcal{O}^-_\alpha(t_1 \to t_2) &= d^- \cdot \mathcal{O}^+_\alpha(t_1) + r^- \cdot \mathcal{O}^-_\alpha(t_2) \\
\mathcal{O}^+_\alpha(\text{bool}) &= \emptyset & \mathcal{O}^-_\alpha(\text{bool}) &= \emptyset \\
\mathcal{O}^+_\alpha(\{f\}) &= \bigcup_{\ell\in\text{dom}\, f} \ell^+ \cdot \mathcal{O}^+_\alpha(f(\ell)) & \mathcal{O}^-_\alpha(\{f\}) &= \bigcup_{\ell\in\text{dom}\, f} \ell^- \cdot \mathcal{O}^-_\alpha(f(\ell)) \\
\mathcal{O}^+_\alpha(\mu\alpha.t) &= \emptyset & \mathcal{O}^-_\alpha(\mu\alpha.t) &= \emptyset \\
\mathcal{O}^+_\alpha(\mu\beta.t) &= \mathcal{O}^+_\beta(t)^* \cdot \mathcal{O}^+_\alpha([\bot/\beta]t) & \mathcal{O}^-_\alpha(\mu\beta.t) &= \mathcal{O}^-_\beta(t)^* \cdot \mathcal{O}^-_\alpha([\top/\beta]t)
\end{aligned}
$$

The operation $\mathcal{O}^\pm_\alpha(t)$ produces exactly that part of $\mathcal{E}^\pm(t)$ referencing occurrences of the variable $\alpha$. More formally, we have:

**Proposition 68.** *If $\alpha$ occurs only positively in $t^+$, then*

$$\mathcal{E}^+(t^+) \equiv \mathcal{O}^+_\alpha(t^+) \cdot \alpha^+ + \mathcal{E}^+([\bot/\alpha^+]t^+)$$

A type automata for a polar type $t^+$ as constructed per Section 7.1.2, can be converted to an NFA accepting $\mathcal{E}^+(t^+)$. Figure 7.2a gives an example of a type automaton, while Figure 7.2b shows the same automaton, after conversion to a standard NFA. The conversion is done by forgetting polarities of states and adding a single accepting state $q_\infty$ with transitions from each state $q$ to $q_\infty$ labelled by each element of $\mathcal{E}(H(q))$.

Note in particular the extra term $\ell \cdot \Sigma^*$ in the encoding $\mathcal{E}^+(\mathsf{\{f\}})$, which helps ensure that the encoding preserves the subtyping order. For example, consider two type terms

$$\begin{aligned} t_1 &= \{\ell_1 : \alpha, \ell_2 : \text{bool}\} \\ t_2 &= \{\ell_2 : \text{bool}, \ell_3 : \beta\} \end{aligned}$$

which may be read as either positive or negative. Taking them to be negative types, we have:

$$\begin{aligned} \mathcal{E}^-(t_1) &= \ell_1^- \cdot \alpha^- + \ell_2^- \cdot \langle \mathsf{b} \rangle^- + \langle \mathsf{\{\}} \rangle^- + \langle \ell_1 \rangle^- + \langle \ell_2 \rangle^- \\ \mathcal{E}^-(t_2) &= \ell_2^- \cdot \langle \mathsf{b} \rangle^- + \ell_3^- \cdot \beta^- + \langle \mathsf{\{\}} \rangle^- + \langle \ell_2 \rangle^- + \langle \ell_3 \rangle^- \end{aligned}$$

The type $t_3 = \{\ell_1 : \alpha, \ell_2 : \text{bool}, \ell_3 : \beta\}$ is the meet of these two types (that is, $t_1 \sqcap t_2 \equiv t_3$), and we have:

$$\mathcal{E}^-(t_3) = \ell_1^- \cdot \alpha^- + \ell_2^- \cdot \langle \mathsf{b} \rangle^- + \ell_3^- \cdot \beta^- + \langle \mathsf{\{\}} \rangle^- + \langle \ell_1 \rangle^- + \langle \ell_2 \rangle^-$$

So, $\mathcal{E}^-$ respects the meet:

$$\mathcal{E}^-(t_3) = \mathcal{E}^-(t_1) + \mathcal{E}^-(t_2)$$

Conversely, by taking them to be positive types we have (writing $\ell$ for an arbitrary record label not among $\ell_1, \ell_2, \ell_3$):

$$\begin{aligned} \mathcal{E}^+(t_1) &= \ell_1^+ \cdot \alpha^+ + \ell_2^+ \cdot \langle \mathsf{b} \rangle^+ + \ell_3 \cdot \Sigma^* + \ell \cdot \Sigma^* + \langle \mathsf{\{\}} \rangle^+ + \langle \ell \rangle^+ + \langle \ell_3 \rangle^+ \\ \mathcal{E}^+(t_2) &= \ell_1 \cdot \Sigma^* + \ell_2^+ \cdot \langle \mathsf{b} \rangle^+ + \ell_3^+ \cdot \beta^+ + \ell \cdot \Sigma^* + \langle \mathsf{\{\}} \rangle^+ + \langle \ell \rangle^+ + \langle \ell_1 \rangle^+ \end{aligned}$$

The type $t_4 = \{\ell_2 : \text{bool}\}$ is their meet, and we have:

$$\mathcal{E}^+(t_4) = \ell_1^+ \cdot \Sigma^* + \ell_2^+ \cdot \langle \mathsf{b} \rangle^+ + \ell_3 \cdot \Sigma^* + \ell \cdot \Sigma^* + \langle \mathsf{\{\}} \rangle^+ + \langle \ell \rangle^+ + \langle \ell_1 \rangle^+ + \langle \ell_3 \rangle^+$$

and therefore (since $\Sigma^* + L = \Sigma^*$ for all $L$):

$$\mathcal{E}^+(t_4) = \mathcal{E}^+(t_1) + \mathcal{E}^+(t_2)$$

These relationships hold in general, for arbitrary $t_1, t_2$. In fact, a stronger result holds: $\mathcal{E}$ maps equivalent type terms to equal languages. To show this, we first require a lemma to allow unrolling recursive types:

**Lemma 69.** $\mathcal{E}^+(\mu\alpha.t^+) = \mathcal{E}^+([\mu\alpha.t^+/\alpha]t^+)$*, and dually for* $\mathcal{E}^-$

*Proof.* Note that $\mathcal{O}^+_\alpha$ separates the occurrences of $\alpha$ in $t^+$ from the rest of $\mathcal{E}^+(t^+)$, and so:

$$\mathcal{E}^+([t_2^+/\alpha]t^+) = \mathcal{O}^+_\alpha(t^+) \cdot \mathcal{E}^+(t_2^+) + \mathcal{E}^+([\bot/\alpha]t^+)$$

Since $\mathcal{O}^+_\alpha(t^+)^* = \mathcal{O}^+_\alpha(t^+) \cdot \mathcal{O}^+_\alpha(t^+)^* + \epsilon$,

$$\begin{aligned} \mathcal{E}^+(\mu\alpha.t^+) &= \mathcal{O}^+_\alpha(t^+)^* \cdot \mathcal{E}^+([\bot/\alpha]t^+) \\ &= (\mathcal{O}^+_\alpha(t^+) \cdot \mathcal{O}^+_\alpha(t^+)^* + \epsilon) \cdot \mathcal{E}^+([\bot/\alpha]t^+) \\ &= \mathcal{O}^+_\alpha(t^+) \cdot \mathcal{E}^+(\mu\alpha.t^+) + \mathcal{E}^+([\bot/\alpha]t^+) \\ &= \mathcal{E}^+([\mu\alpha.t^+/\alpha]t^+) \end{aligned}$$

The dual case is identical. □

$$
\begin{aligned}
\mathcal{R}(\alpha^+) &= (\alpha, \top) & \mathcal{R}(\alpha^-) &= (\bot, \alpha) \\[6pt]
\mathcal{R}(\langle\to\rangle^+) &= (\top \to \bot, \top) & \mathcal{R}(\langle\to\rangle^-) &= (\bot, \bot \to \top) \\
\mathcal{R}(d^+) &= (\chi \to \bot, \top) & \mathcal{R}(d^-) &= (\bot, \chi \to \top) \\
\mathcal{R}(r^+) &= (\top \to \chi, \top) & \mathcal{R}(r^-) &= (\bot, \bot \to \chi) \\[6pt]
\mathcal{R}(\langle \mathtt{b} \rangle^+) &= (\text{bool}, \top) & \mathcal{R}(\langle \mathtt{b} \rangle^-) &= (\bot, \text{bool}) \\[6pt]
\mathcal{R}(\langle \mathtt{\{\}} \rangle^+) &= (\{\ell : \bot \mid \ell \text{ a label}\}, \top) & \mathcal{R}(\langle \mathtt{\{\}} \rangle^-) &= (\bot, \{\}) \\
\mathcal{R}(\langle \ell \rangle^+) &= (\{\ell' : \bot \mid \ell \neq \ell'\}, \top) & \mathcal{R}(\langle \ell \rangle^-) &= (\bot, \{\ell : \top\}) \\
\mathcal{R}(\ell^+) &= (\{\ell : \chi, \ell' : \bot \mid \ell \neq \ell'\}, \top) & \mathcal{R}(\ell^-) &= (\bot, \{\ell : \chi\})
\end{aligned}
$$

Figure 7.3: Mapping of $\Sigma$ to type pairs by $\mathcal{R}$

**Theorem 70.** *If* $t_1^+$ *and* $t_2^+$ *are equivalent, then* $\mathcal{E}^+(t_1^+)$ *and* $\mathcal{E}^+(t_2^+)$ *are the same language, and dually for negative type terms.*

*Proof.* The type terms $t_1^+, t_2^+$ are interpreted as types $\tau_1, \tau_2 \in \mathcal{T}$. Recall from Section 3.4.1 that two types $\tau_1$, $\tau_2$ are equal iff $\pi_k \tau_1 = \pi_k \tau_2$ for all $k$, where $\pi_k$ are the projection morphisms that truncate their arguments to a depth of at most $k$ nested type constructors.

We prove that, for all $k$, if $\pi_k t_1^+ = \pi_k t_2^+$ then $\mathcal{E}^+(t_1^+)$ and $\mathcal{E}^+(t_2^+)$ contain the same strings of length $\leqslant k$, by induction on $k$. The $k = 0$ case is trivial, and the $k + 1$ case proceeds by applying Lemma 69 and its dual repeatedly to ensure that any occurrences of $\mu$ are nested under $k + 1$ type constructors, and then doing structural induction on $t_1^+, t_2^+$. □

So, although $\mathcal{E}^+(t^+)$ is defined by examining the *syntax* of $t^+$, it respects equivalence between different type terms, making merely syntactic distinctions irrelevant.

### 7.2.2 Undoing the encoding

In this section, we build a function $\mathcal{R}$ as an inverse to $\mathcal{E}$. In principle, this can be used to convert type automata back to types for display to the user, by first converting them to NFAs, finding a regular expression describing the NFA using Kleene's algorithm, and finally using $\mathcal{R}$ to rebuild the type. In practice, this is massively more complicated than using the simpler algorithm described informally in Section 7.1.3. $\mathcal{R}$ is a technical ingredient in the proof of the representation theorem, rather than something that it is useful to implement.

$\mathcal{R}$ maps regular expressions over $\Sigma$ to *type pairs*, which are pairs $(t^+, t^-)$ of a positive and a negative type term. This mixture of positive and negative types is an essential detail of the encoding: regular languages do not distinguish the two while type terms do, and the most technically convenient thing to do is to have $\mathcal{R}$ produce both a negative and a positive type. We will find below that $\mathcal{R}$ produces $(t^+, \top)$ when given an automaton representing the positive type $t^+$, and similarly produces $(\bot, t^-)$ given an automaton for $t^-$.

Each of the symbols in $\Sigma$ is mapped to a type pair by $\mathcal{R}$, as shown in Fig. 7.3. This table has many cases, since $\Sigma$ is a fairly large alphabet, but a simple structure.

- Symbols marked with $^+$ are mapped to type pairs $(t^+, \top)$, while symbols marked with $^-$ are mapped to type pairs $(\bot, t^-)$.
- Symbols representing type constructors are mapped to the greatest or least type with that constructor according to polarity. For instance, $\mathcal{R}(\langle\to\rangle^+) = (\top \to \bot, \top)$ since $\top \to \bot$ is the least function type. Note that $\mathcal{R}(\langle\ell\rangle^+)$ is mapped to the least record type *not containing field* $\ell$, in accordance with the encoding of record types by $\mathcal{E}$ above.
- Symbols representing fields are mapped to a *context*, which is a type marking the use of a particular field by the particular type variable $\chi$ (which we use for no other purpose).

In order to interpret regular expressions as type terms, we must also interpret the three operations $+$, $\cdot$ and $^*$ (union, concatenation and Kleene star) as well as the elements of the alphabet $\Sigma$. We define these operations on type pairs, taking $+$ as meet/join

$$(t_1^+, t_1^-) + (t_2^+, t_2^-) = (t_1^+ \sqcup t_2^+, t_1^- \sqcap t_2^-)$$

and $\cdot$ as composition of contexts:

$$(t_1^+, t_1^-) \cdot (t_2^+, t_2^-) = \begin{cases} (\bot, \top) & \text{if } (t_2^+, t_2^-) = (\bot, \top) \\ \begin{pmatrix} [t_2^+/\chi^+, t_2^-/\chi^-]t_1^+, \\ [t_2^+/\chi^+, t_2^-/\chi^-]t_1^- \end{pmatrix} & \text{otherwise} \end{cases}$$

The constants $\emptyset$ and $\epsilon$ and are interpreted as $(\bot, \top)$ and $(\chi, \chi)$, the identities of $+$ and $\cdot$ respectively. We define $(t^+, t^-)^*$ such that:

$$(t^+, t^-)^* = \epsilon + (t^+, t^-) \cdot (t^+, t^-)^*$$

In the case where $\chi$ is guarded and appears covariantly on $t^+$, $t^-$, then such a $(t_*^+, t_*^-)$ can be found as:

$$(\mu\alpha.\chi \sqcup [\alpha/\chi]t^+,\ \mu\beta.\chi \sqcap [\beta/\chi]t^-)$$

In general, we use Bekič's technique for finding simultaneous fixed points (Section 2.1.8), and use the fixed point operators $\mu^+, \mu^-$ (as defined in Section 5.1.1) to relax the guardedness constraint. The resulting definition of $(t_*^+, t_*^-)^*$ is the following mouthful:

$$(t^+, t^-)^* = \begin{pmatrix} \mu^+\alpha.\chi \sqcup \begin{bmatrix} \alpha/\chi^+ \\ \mu^-\beta.\chi \sqcap [\alpha/\chi^+, \beta/\chi^-]t^-/\chi^- \end{bmatrix} t^+, \\ \mu^-\beta.\chi \sqcap \begin{bmatrix} \mu^+\alpha.\chi \sqcup [\alpha/\chi^+, \beta/\chi^-]t^+/\chi^+ \\ \beta/\chi^- \end{bmatrix} t^- \end{pmatrix}$$

Note that this simplifies into the definition above in the case where $\chi$ is guarded and covariant in both $t^+$ and $t^-$.

Since fixed points of type expressions can be found by iteration, we also have:

**Proposition 71.**

$$(t^+, t^-)^* \equiv \sum_k (t^+, t^-)^k$$

These definitions recover types from regular expressions over $\Sigma$. For instance, consider the regular expression $E = \mathcal{E}^+(\text{bool} \to \text{bool})$:

$$E = d^+ \cdot \langle b \rangle^- + r^+ \cdot \langle b \rangle^+ + \langle \to \rangle^+$$

We calculate $\mathcal{R}(E)$:

$$\begin{aligned}
\mathcal{R}(E) &= \mathcal{R}(d^+) \cdot \mathcal{R}(\langle b \rangle^-) + \mathcal{R}(r^+) \cdot \mathcal{R}(\langle b \rangle^+) + \langle \to \rangle^+ \\
&= (\chi \to \bot, \top) \cdot (\bot, \text{bool}) + (\top \to \chi, \top) \cdot (\text{bool}, \top) + (\top \to \bot, \top) \\
&\equiv (\text{bool} \to \bot, \top) + (\top \to \text{bool}, \top) + (\top \to \bot, \top) \\
&\equiv (\text{bool} \to \text{bool}, \top)
\end{aligned}$$

In general, $\mathcal{R}$ inverts $\mathcal{E}$:

**Theorem 72.** *For all* $t^+, t^-$*:*

$$\mathcal{R}(\mathcal{E}^+(t^+)) \equiv (t^+, \top) \qquad\qquad \mathcal{R}(\mathcal{E}^-(t^-)) = (\bot, t^-)$$

*Proof.* Induction on the height of the syntax tree of $t^+, t^-$. All cases are straightforward calculations like the above, except the difficult case of recursive types[2]. We consider the case of a positive recursive type $\mu\alpha.t^+$, since the negative case is dual:

$$\mathcal{R}(\mathcal{E}^+(\mu\alpha.t^+)) = \mathcal{R}(\mathcal{O}^+_\alpha(t^+))^* \cdot \mathcal{R}(\mathcal{E}^+([\bot/\alpha]t^+))$$

Since replacing $\alpha$ with $\bot$ does not change the height of a syntax tree, we may use the inductive hypothesis to see that $\mathcal{R}(\mathcal{E}^+([\bot/\alpha]t^+)) \equiv ([\bot/\alpha]t^+, \top)$. Let $(t^+_\alpha, t^-_\alpha)$ be $\mathcal{R}(\mathcal{O}^+_\alpha(t^+))$. By inspection of the definition of $\mathcal{O}^+$ and $\mathcal{R}$, $t^-_\alpha \equiv \top$.

Proposition 68 states $\mathcal{E}^+(t^+) \equiv \mathcal{O}^+_\alpha(t^+) \cdot \alpha^+ + \mathcal{E}^+([\bot/\alpha^+]t^+)$, so by Theorem 74 (proven in the next section) we can apply $\mathcal{R}$ to both sides, giving:

$$\begin{aligned}
\mathcal{R}(\mathcal{E}^+(t^+)) &\equiv \mathcal{R}(\mathcal{O}^+_\alpha(t^+)) \cdot \mathcal{R}(\alpha^+) + \mathcal{R}(\mathcal{E}^+([\bot/\alpha^+]t^+)) \\
(t^+, \top) &\equiv (t^+_\alpha, \top) \cdot (\alpha, \top) + ([\bot/\alpha]t^+, \top) \\
t^+ &\equiv [\alpha/\chi]t^+_\alpha \sqcup [\bot/\alpha]t^+
\end{aligned}$$

Using the simple definition of $^*$ (since $\chi$ is positive in $\mathcal{R}(\mathcal{O}^+_\alpha(t^+))$) then gives:

$$\begin{aligned}
\mathcal{R}(\mathcal{E}^+(\mu\alpha.t^+)) &= \mathcal{R}(\mathcal{O}^+_\alpha(t^+))^* \cdot \mathcal{R}(\mathcal{E}^+([\bot/\alpha]t^+)) \\
&\equiv (t^+_\alpha, \top)^* \cdot ([\bot/\alpha]t^+, \top) \\
&\equiv (\mu\alpha.\chi \sqcup [\alpha/\chi]t^+_\alpha, \top) \cdot ([\bot/\alpha]t^+, \top) \\
&\equiv (\mu\alpha.[\bot/\alpha]t^+ \sqcup [\alpha/\chi]t^+_\alpha, \top) \\
&\equiv (\mu\alpha.t^+, \top)
\end{aligned}$$

□

### 7.2.3 Simplifying types as languages

So far, we have shown that $\mathcal{E}$ preserves equivalence of types, and that $\mathcal{R}$ is its left inverse. The final ingredient in the representation theorem is to show that $\mathcal{R}$ preserves equivalence of regular expressions: that is, any two regular expressions over $\Sigma$ that are equal as languages are mapped to equivalent type pairs by $\mathcal{R}$. This means that we need not record the exact expression produced by $\mathcal{E}$, as any representation of the regular language will do, such as a finite automaton.

We do this by showing that type pairs form a Kleene algebra, which begins by showing that they form an idempotent semiring. The properties of $+$ are easily shown:

[2] As usual.

**Proposition 73.** *$+$ is associative, idempotent and commutative (up to $\equiv$), with identity $0$.*

*Proof.* These properties follow from the same properties of $\sqcup$, $\sqcap$. □

To show the relevant properties of $\cdot$, it is convenient to use some shorthand notation. We write $t$ for a type pair $(t^+, t^-)$, and write $[t]$ for the bisubstitution $[t^+/\chi^+, t^-/\chi^-]$. In this notation, we can write $\cdot$ more concisely:

$$t_1 \cdot t_2 = \begin{cases} 0 & \text{if } t_2 = 0 \\ [t_2]t_1 & \text{otherwise} \end{cases}$$

Note that $+$ and $\cdot$ as defined on bisubstitutions in Section 5.2.1 are very close to those defined for type terms:

$$[t_1 + t_2] = [t_1] + [t_2]$$
$$[t_1 \cdot t_2] = \begin{cases} [0] & \text{if } t_2 = 0 \\ [t_2] \cdot [t_1] & \text{otherwise} \end{cases}$$

Since this gives type pairs the structure of a Kleene algebra, we can use the completeness theorem (Section 2.2.1) to prove the representation theorem:

**Theorem 74.** *If $E_1$ and $E_2$ denote equal regular languages, then $\mathcal{R}(E_1) \equiv \mathcal{R}(E_2)$*

*Proof.* Since $E_1 = E_2$ is an equation of regular languages, $E_1 = E_2$ holds in all Kleene algebras by the completeness theorem (Section 2.2.1). The operations $+$, $\cdot$ and $^*$ on type pairs preserve $\equiv$, so we show that equivalence classes of $\equiv$ form a Kleene algebra.

First, we show that they form an idempotent semiring: $\cdot$ is associative with identity $\epsilon$ and zero $0$, and $+$ distributes over $\cdot$ (all up to $\equiv$).

The identity is trivial, and the zero is by definition. For associativity, we must show that $t_1 \cdot (t_2 \cdot t_3) \equiv (t_1 \cdot t_2) \cdot t_3$. First, consider the case when any of $t_1, t_2, t_3$ is $0$. In that case, both sides collapse to $0$, and the equation holds. Otherwise, the equation holds since composition of bisubstitutions is associative.

For left distributivity, we must show:

$$(t_1 + t_2) \cdot t_3 \equiv t_1 \cdot t_3 + t_2 \cdot t_3$$

First, we note that the equation holds trivially if any of $t_1$, $t_2$ or $t_3$ is $0$. Otherwise, assume none are $0$, in which case $t_1 + t_2$ must also be nonzero (since if $t_1^+ \sqcup t_2^+ \equiv \bot$, then $t_1^+ \equiv t_2^+ \equiv \bot$, and likewise for $\top$), and the result follows from bisubstitution distributivity (Proposition 39).

For right distributivity, we must show:

$$t_1 \cdot (t_2 + t_3) \equiv t_1 \cdot t_2 + t_1 \cdot t_3$$

As before, this holds trivially if any of $t_1$, $t_2$, $t_3$ are $0$, so we assume them nonzero whence also $t_2 + t_3$ is nonzero, and the equation follows from Proposition 38.

Finally, we note that the semiring is a $^*$-continuous Kleene algebra (Section 2.2.3) because Proposition 71 holds. □

Equal languages represent equivalent types, so any algorithm for simplifying finite automata may be used to simplify type automata. One standard 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 [Pot98b]. By proving a general representation theorem, we have removed the need to individually prove correctness of these algorithms as applied to types: that they are correct as applied to finite automata suffices.

The representation theorem allows other techniques: for instance, Ilie, Navarro and Yu give a simplification algorithm operating on NFAs [INY04], relying on Paige-Tarjan partition refinement [PT87]. Thanks to the representation theorem, we know that this algorithm correctly simplifies types, just because it correctly simplifies NFAs.

## 7.3 Simplifying typing schemes

The simplification techniques of the previous section preserve equivalence as types, and can therefore be used to simplify polar typing schemes by simplifying their component types. However, polar 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 4.2.1). 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

$$\phi(t_1^-, \dots t_n^-; t_1^+, \dots t_m^+)$$

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

$$\phi_{choose}(t_1^-, t_2^-;\ t_1^+) = [\,]\, t_1^- \to t_2^- \to t_1^+$$

A *simple instance* $P$ of $\phi$ is a typing scheme formed from $\phi$ where $t_i^-$ $(1 \leqslant i \leqslant n)$ is given as $\bigsqcap P_i^+$ for some finite set $P_i^+$ of type variables (taking $\bigsqcap \emptyset$ to be $\top$), while $t_j^+$ $(1 \leqslant j \leqslant m)$ is similarly given as $\bigsqcup P_j^-$ (taking $\bigsqcup \emptyset$ to be $\bot$). In this manner, the two typing schemes for *choose* are the simple instances $\phi_{choose}(\alpha, \alpha; \alpha)$ and $\phi_{choose}(\beta, \gamma; \beta \sqcup \gamma)$.

For a given simple instance $P$ of $\phi$, we say that there is a *flow edge* $i \rightsquigarrow j$ if $P_i^-$ and $P_j^+$ have a type variable in common. Note that both simple instances giving typing schemes for *choose* have the same two flow edges: $1 \rightsquigarrow 1$ and $2 \rightsquigarrow 1$.

In general, two simple instances of the same $\phi$ with the same flow edges are equivalent, allowing us to simplify polar typing schemes by relabelling flow edges with different variables. To prove this, we prove the following stronger result:

**Theorem 75.** *Given two simple instances* $P, Q$ *of a typing scheme*

$$\phi(t_1^-, \dots t_n^-; t_1^+, \dots t_m^+)$$

*where every flow edge of* $\mathsf{P}$ *is a flow edge of* $\mathsf{Q}$*, then*

$$\phi(P) \leqslant^{\forall} \phi(Q)$$

*Proof.* By definition of $\leqslant^{\forall}$, we seek some substitution $\rho$ such that $\rho(\phi(P)) \leqslant \phi(Q)$. Since $\phi$ is contravariant in negative occurrences of variables and covariant in positive ones, it suffices to find $\rho$ satisfying, for all $1 \leqslant i \leqslant n, 1 \leqslant j \leqslant m$:

$$\rho\left(\bigsqcup P_i^+\right) \leqslant \bigsqcup Q_i^+$$
$$\bigsqcap Q_j^- \leqslant \rho\left(\bigsqcap P_j^-\right)$$

which is equivalent to:

$$\rho(\alpha) \leqslant \bigsqcup Q_i^+ \quad \text{for } \alpha \in P_i^+$$
$$\bigsqcap Q_j^- \leqslant \rho(\alpha) \quad \text{for } \alpha \in P_j^-$$

We choose $\rho(\alpha)$ as follows:

$$\rho(\alpha) = \bigsqcap \left\{ \bigsqcup Q_i^+ \middle| \alpha \in P_i^+ \right\}$$

Trivially, this satisfies $\rho(\alpha) \leqslant \bigsqcup Q_i^+$ for $\alpha \in P_i^+$. For the other conditions, consider some $\alpha \in P_j^-$. For each $i$ such that $\alpha \in P_i^+$, we have a flow edge $i \rightsquigarrow j$ which by assumption must also appear in $Q$. Therefore, we must have some variable $\beta_{ij}$ in both $Q_i^+$ and $Q_j^-$, so:

$$\begin{aligned}\bigsqcap Q_j^- &\leqslant \bigsqcap \{\beta_{ij} | \alpha \in P_i^+\} \\ &\leqslant \bigsqcap \{Q_i^+ | \alpha \in P_i^+\} \\ &= \rho(\alpha)\end{aligned}$$

□

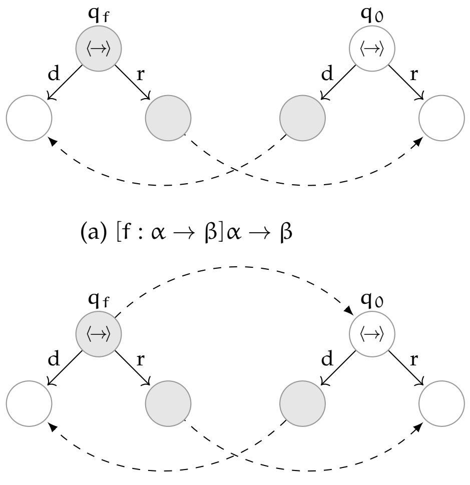
The upshot of this result is that the precise naming of variables does not matter, and only the placement of flow edges is important. This finally makes precise the intuition of Section 1.1 and Section 4.2.1, where it was noted that the purpose of type variables in MLsub typing schemes is merely to label data flow.

So, we adopt a representation of typing schemes which does away with explicit variables, and represents flow edges directly.

### 7.3.1 Scheme automata

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

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



(a) $[f : \alpha \to \beta]\alpha \to \beta$

(b) $[f : (\alpha \to \beta) \sqcap \gamma](\alpha \to \beta) \sqcup \gamma$

Figure 7.4: Two scheme automata

- 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 $x \in X$ (corresponding to $D^-(x)$ in a typing scheme $[D^-]t^+$), and a positive state $q_0$ (corresponding to $t^+$ in a typing scheme $[D^-]t^+$). The transition table $\delta$ retains the polarity condition of type automata.

The set $H(q)$ follows the same rules as type automata, but unlike type automata never contains type variables. Type variables are represented implicitly as flow edges, instead of appearing in $H(q)$ as they do in type automata.

To construct a scheme automaton from a typing scheme $[D^-]t^+$, we start by constructing type automata for $t^+$ and $D^-(x)$ (for each $x \in \mathrm{dom}(D^-)$). 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 $H(q)$. In Fig. 7.4a, 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.

### 7.3.2 Simplifying scheme automata

As well as simplifications using the representation theorem, scheme automata can also be simplified by removing redundant flow edges. Consider the two typing schemes whose scheme automata are shown in Figs. 7.4a and 7.4b. It is easy to see that

$$[f : (\alpha \to \beta) \sqcap \gamma](\alpha \to \beta) \sqcup \gamma \leqslant^{\forall} [f : \alpha \to \beta](\alpha \to \beta)$$

by instantiating $\gamma \mapsto (\alpha \to \beta)$. However, it is also the case that:

$$[f : \alpha \to \beta](\alpha \to \beta) \leqslant^{\forall} [f : (\alpha \to \beta) \sqcap \gamma](\alpha \to \beta) \sqcup \gamma$$

since $(\alpha \to \beta) \sqcap \gamma \leqslant (\alpha \to \beta) \leqslant (\alpha \to \beta) \sqcup \gamma$. Thus, both typing schemes are equivalent. As scheme automata, these differ only in the presence of a flow edge representing $\gamma$. Although both are equivalent, the scheme not mentioning $\gamma$ is shorter and easier to read. In order to concisely display typing schemes, it is useful to able to detect and remove redundant flow edges. We say that a flow edge $q^- \rightsquigarrow q^+$ is *admissable* if adding it results in an equivalent typing scheme. Admissable flow edges can be characterised by subtyping:

**Proposition 76.** *A flow edge* $q^- \rightsquigarrow q^+$ *is admissable iff* $t^- \leqslant t^+$*, where* $t^-, t^+$ *are the types represented by* $q^-, q^+$*.*

*Proof.* Let $\phi(t^-, t^+)$ be the typing scheme parameterised by the types represented by $q^-, q^+$, which is contravariant in its first parameter and covariant in its second. After adding the flow edge $q^- \rightsquigarrow q^+$, the resulting typing scheme is $\phi(t^- \sqcap \alpha, t^+ \sqcup \alpha)$ for fresh $\alpha$. Regardless of whether $t^+ \leqslant t^-$, it is always the case that $\phi(t^-, t^+) \leqslant^\forall \phi(t^- \sqcap \alpha, t^+ \sqcup \alpha)$, since $t^- \sqcap \alpha \leqslant t^-$ and $t^+ \leqslant t^+ \sqcup \alpha$. By expanding $\leqslant^\forall$, the converse holds iff we can find some $\rho$ such that:

$$\phi(\rho(t^- \sqcap \alpha), \rho(t^+ \sqcup \alpha)) \leqslant \phi(t^-, t^+)$$

which holds iff:

$$\begin{aligned} \rho(t^+) &\leqslant t^+ & \qquad t^- &\leqslant \rho(t^-) \\ \rho(\alpha) &\leqslant t^+ & \qquad t^- &\leqslant \rho(\alpha) \end{aligned}$$

If $t^- \not\leqslant t^+$, then no $\rho$ satisfying $t^- \leqslant \rho(\alpha) \leqslant t^+$ can exist, while if $t^- \leqslant t^+$ then $\rho = [t^-/\alpha]$ suffices. □

This suggests a straightforward heuristic algorithm for optimising the set of flow edges in a scheme automaton. First, we remove all of them. Then, we add them back one at a time, skipping any that are admissable. While the success of this heuristic depends greatly on the order flow edges are processed in, I found that a reverse postorder traversal (so that flow edges on child nodes are processed before parents) gave good results.

Of course, in order to implement this we must have an algorithm that decides subtyping problems of the form $t^- \leqslant t^+$ (recall that biunification operates on constraints $t^+ \leqslant t^-$, so is unsuitable). Such an algorithm is described in Chapter 8, as it forms the core of the subsumption algorithm.

### 7.3.3 Converting scheme automata to type automata

The two typing schemes for *choose* in Section 7.3.1 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.

The flow edges of a scheme automaton are the edges of a bipartite graph whose nodes are the negative and positive states. Each type variable $\alpha$ introduces a *biclique* of edges: that is, a set of negative and a set of positive states (the two *sides* of the biclique), with a flow edge between all pairs of the negative and positive states. To convert back from a scheme automaton, the flow edges must be decomposed into a union of such bicliques, each of which is then labelled with a type variable.

This problem, called *biclique decomposition* is NP-complete. However, we have found that a simple greedy heuristic works well in practice. For each

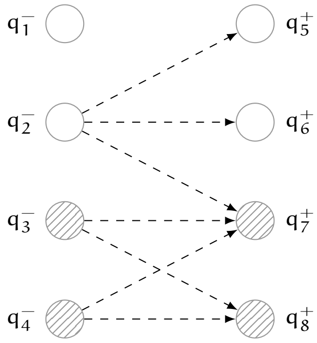


Figure 7.5: Example bipartite graph of flow edges with $\mathsf{Gr}(q_3^-)$ marked

state $q^-$ or $q^+$, we write $\mathsf{Flow}(q^-)$ or $\mathsf{Flow}(q^+)$ for the set of states to which there are flow edges:

$$\begin{aligned} \mathsf{Flow}(q^-) &= \{q^+ \mid q^- \rightsquigarrow q^+\} \\ \mathsf{Flow}(q^+) &= \{q^- \mid q^- \rightsquigarrow q^+\} \end{aligned}$$

We extend Flow to operate on sets Q of states of the same polarity:

$$\mathsf{Flow}(Q) = \bigcap_{q \in Q} \mathsf{Flow}(q)$$

Sets of negative and positive states $(Q^-, Q^+)$ form a biclique exactly when $\mathsf{Flow}(Q^-) = Q^+$ and $\mathsf{Flow}(Q^+) = Q^-$.

**Lemma 77.** *For any* $Q$, $(\mathsf{Flow}(Q), \mathsf{Flow}(\mathsf{Flow}(Q)))$ *forms a biclique.*

*Proof.* By construction, Flow is an antitone Galois connection (polarity), so

$$\mathsf{Flow}(\mathsf{Flow}(\mathsf{Flow}(Q))) = \mathsf{Flow}(Q)$$

by general properties of Galois connections. □

The *greedy biclique* $\mathsf{Gr}(q)$ of a state $q$ (of either polarity) is the biclique $(\mathsf{Flow}(\{q\}), \mathsf{Flow}(\mathsf{Flow}(\{q\})))$. For the example in Fig. 7.5, $\mathsf{Gr}(q)$ is as follows:

| | *States* | *Flow edges* |
|---|---|---|
| $\mathsf{Gr}(q_1^-)$ | $\{q_1^-\},\{\}$ | 0 |
| $\mathsf{Gr}(q_2^-) = \mathsf{Gr}(q_5^+) = \mathsf{Gr}(q_6^+)$ | $\{q_2^-\},\{q_5^+, q_6^+, q_7^+\}$ | 3 |
| $\mathsf{Gr}(q_3^-) = \mathsf{Gr}(q_4^-) = \mathsf{Gr}(q_8^+)$ | $\{q_3^-, q_4^-\},\{q_7^+, q_8^+\}$ | 4 |
| $\mathsf{Gr}(q_7^+)$ | $\{q_2^-, q_3^-, q_4^-\},\{q_7^+\}$ | 3 |

Our heuristic algorithm for biclique decomposition is to compute the size of $\mathsf{Gr}(q)$ for each $q$, greedily remove the largest biclique found assigning it a type variable, and iterate until no flow edges remain. Note that $\mathsf{Gr}(q)$ must in general be recomputed after each step, since e.g. $\mathsf{Gr}(q_7^+)$ changes after the flow edges of $\mathsf{Gr}(q_3^-)$ are removed, although this can be optimised by recomputing only those states whose flow edges have changed.

This algorithm is far from optimal. However, it does produce optimal results when the graph is a disjoint union of bicliques (with no two bicliques sharing a node in common). This class of easy biclique decomposition problems correspond to those typing automata in which no node is labelled by more than one variable, which include all of the ML type schemes. So, any typing scheme which can be written without use of $\sqcup$ and $\sqcap$ will be rendered as such with a minimal number of variables by this algorithm, although this algorithm will not in general introduce the minimal number of $\sqcup$ and $\sqcap$ when this number is above zero.

## 7.4 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 5.3.3 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 $\mathsf{biunify}(H; C)$ 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 head constructors, 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 $q', f$,

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

and by setting $H(q_1)$ to $H(q_1) \widetilde{\sqcup} H(q_2)$ when $q_1, q_2$ positive, or $H(q_1) \widetilde{\sqcap} H(q_2)$ when $q_1, q_2$ negative.

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 $t$, then the effect of $\mathrm{merge}(q_1, q_2)$ is to perform the atomic constraint elimination $\theta_{\alpha \leqslant t}$ or $\theta_{t \leqslant \alpha}$ as per Section 5.3.1. 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 $t^+ \leqslant t^-$, 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 7.6.

### 7.4.1 Termination and complexity

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

```
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 $q^+ \xrightarrow{f} q'^+, q^- \xrightarrow{f} q'^-$ do
            biunify($q'^+, q'^-$)
```

Figure 7.6: Biunification algorithm for scheme automata

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 about 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.

# 8 Deciding subsumption

*Seek simplicity and distrust it.*

—Alfred North Whitehead

So far, we have discussed type *inference* (constructing a type for an unannotated program) but not type *checking* (validating that a program matches a provided type). In order to support optional type annotations, we must be able to compare a user-provided type to an inferred type.

Suppose the programmer writes the following type annotation:

$$\mathsf{f} : \{\mathtt{x} : \alpha, \mathtt{y} : \alpha\} \to \{\mathtt{x} : \alpha \sqcup \mathrm{bool}\}$$

and gives the following implementation:

$$\mathsf{f} = \lambda x.\, x$$

In order to check whether the implementation matches the annotation, we must determine whether there is a typing derivation giving this typing scheme to $\mathsf{f}$. Type inference produces the following typing scheme for the implementation of $\mathsf{f}$:

$$\mathsf{f} : []\alpha \to \alpha$$

By the principality theorem (Proposition 66), we know that the typing schemes with which $\mathsf{f}$ can be typed are exactly the instances of the above. Thus, the implementation matches the annotation if and only if:

$$[\,]\alpha \to \alpha \leqslant^{\forall} [\,]\{\mathtt{x} : \alpha, \mathtt{y} : \alpha\} \to \{\mathtt{x} : \alpha \sqcup \mathrm{bool}\}$$

In order to check type annotations, we must decide the relation $\leqslant^{\forall}$.

In this chapter, we develop an algorithm for deciding subsumptions. While this was considered an intractable problem in previous frameworks, we see below that the algebraic construction of types (Chapter 3) makes this relatively straightforward.

## 8.1 Deciding the example

The tools we developed in Chapter 5 are not sufficient for the task. The annotation is not a constraint in the sense of those that we eliminated with bisubstitution. We require that $\mathsf{f}$ be polymorphic in $\alpha$: that is, we should have:

$$\mathsf{f} : \{\mathtt{x} : \tau, \mathtt{y} : \tau\} \to \{\mathtt{x} : \tau \sqcup \mathrm{bool}\}$$

for any type $\tau$. We are not searching for a particular $\alpha$, but making a statement true for all $\alpha$. Biunification, which attempts to solve constraints by producing a bisubstitution, does not respect this. Furthermore, biunification works only on constraints of the form $t^+ \leqslant t^-$, whereas here we need to compare two positive types.

So, we begin solving the problem by hand, before generalising the technique into an algorithm below. Expanding the definition of $\leqslant^\forall$, we seek to decide whether there exists $\rho$ such that:

$$\rho(\alpha \to \alpha) \leqslant \{x : \alpha, y : \alpha\} \to \{x : \alpha \sqcup \mathrm{bool}\}$$

By function subtyping, this is equivalently:

$$\begin{aligned} \{x : \alpha, y : \alpha\} &\leqslant \rho(\alpha) \\ \rho(\alpha) &\leqslant \{x : \alpha \sqcup \mathrm{bool}\} \end{aligned}$$

Thus, the subsumption holds iff we can choose some $\rho(\alpha)$ such that

$$\{x : \alpha, y : \alpha\} \leqslant \rho(\alpha) \leqslant \{x : \alpha \sqcup \mathrm{bool}\}$$

We can do this if and only if

$$\{x : \alpha, y : \alpha\} \leqslant \{x : \alpha \sqcup \mathrm{bool}\}$$

The previous statement implies this one by transitivity, and if this statement holds so does the previous, since we can choose $\rho(\alpha)$ to be $\{x : \alpha, y : \alpha\}$ or $\{x : \alpha\}$ or anything in between. This subtyping statement holds, since:

$$\{x : \alpha, y : \alpha\} \leqslant \{x : \alpha\} \leqslant \{x : \alpha \sqcup \mathrm{bool}\}$$

Thus, we have reduced the original *subsumption* problem to a *subtyping* problem, in which there is no $\rho$ to guess. The subsumption algorithm below follows the same pattern of recursively decomposing a subsumption problem, although in general it will produce many subtyping problems to be solved, all of the form $t^- \leqslant t^+$.

## 8.2 Deciding complex subtyping

So far, the only subtyping constraints we have looked at have been those of the form $t^+ \leqslant t^-$ that are handled by the biunification algorithm (Section 5.3.3). However, the subsumption algorithm below works by reducing a subsumption problem to a set of subtyping problems of the form $t^- \leqslant t^+$, for which biunification will not suffice. Biunification solves a harder problem than deciding subtyping: like unification, it must not just decide the truth of a constraint but produce a (bi)substitution solving it. Although merely deciding $t^- \leqslant t^+$ subtyping problems is simpler, the form of the constraint adds difficulties.

Positive types may involve $\sqcup$, while negative types may involve $\sqcap$. Any instance of $\sqcup$ appearing to the left of $\leqslant$ allows a constraint to be split into subconstraints, since:

$$\tau_1 \sqcup \tau_2 \leqslant \tau_3 \quad \text{iff} \quad \tau_1 \leqslant \tau_3 \wedge \tau_2 \leqslant \tau_3$$

Similarly, $\sqcap$ on the right of $\leqslant$ is easily handled. Biunification takes advantage of this, accepting constraints with $\sqcup$ only on the left and $\sqcap$ only on the right, and grinding them down into atomic subconstraints.

There is no similarly easy trick to handle $\sqcup$ appearing on the right. It is the case that:

$$\tau_1 \leqslant \tau_2 \vee \tau_1 \leqslant \tau_3 \quad \Longrightarrow \quad \tau_1 \leqslant \tau_2 \sqcup \tau_3$$

but the converse is not true. For instance,

$$\alpha \to \alpha \leqslant (\top \to \alpha) \sqcup (\alpha \to \bot)$$

while $\alpha \to \alpha \nleqslant \top \to \alpha$ and $\alpha \to \alpha \nleqslant \alpha \to \bot$. This has been a sticking point for previous work [TS96, Pot98b], which has generally had to settle for a sound but incomplete algorithm for subsumption. For illustration, upon facing a constraint $\tau_1 \leqslant \tau_2 \sqcup \tau_3$ it is sound but incomplete in general to proceed by deciding $\tau_1 \leqslant \tau_2$ and $\tau_1 \leqslant \tau_3$, since the results only allow you to say "yes" or "maybe" to the original question.

By contrast, the algorithms below precisely decide subsumption and $t^- \leqslant t^+$ subtyping. This is not due to the algorithms themselves, which are neither terribly complicated nor terribly novel, bearing a strong resemblance to previous work (e.g. Pottier's sound-but-incomplete entailment [Pot98b, p. 79] and Trifonov and Smith's sound-but-incomplete $\leqslant^{\forall}_{\mathsf{dec}}$ [TS96]).

Rather, it is due to the algebraic construction of types in Chapter 3, which give better reasoning properties about the subtyping relation than previous work. Since types are constructed as a coproduct of distributive lattices (see Section 3.2.3), Proposition 12 allows us to handle $\sqcup$ on the right when the join is between terms of different components. For instance, consider again Pottier's awkward example seen in Section 1.4.2:

$$(\bot \to \top) \to \bot \ \leqslant \ (\alpha \to \bot) \sqcup \alpha$$

Proposition 12 says that since $\alpha \to \bot$ and $\alpha$ are from different components (one is a function type, the other a type variable), this holds iff $(\bot \to \top) \to \bot \leqslant \alpha \to \bot$. In turn, by function subtyping that holds iff $\alpha \leqslant \bot \to \top$, which is false.

Thus, the construction of types using lattice coproducts (Section 3.2) and the treatment of type variables as indeterminates (Section 3.3) greatly simplify deciding complex subtyping relationships.

### 8.2.1 Reduced form and deterministic automata

Proposition 12 applies only to subtyping problems $\tau_1 \leqslant \tau_2$ when $\tau_1$ is a meet and $\tau_2$ a join of types constructed from distinct components. In order to use it, we must be able to bring polar types and type automata into this form.

Any polar type can be brought into this form, which we now term *reduced form*. Positive polar types are joins of constructed and recursive types. The $\mu$-binders can be gotten rid of by unrolling (guardedness ensures this eventually leaves constructed types), and the condition of distinct components can be ensured by merging subterms in the same component. For example, suppose we have the following positive type:

$$\{\ell_1 : \alpha\} \sqcup \{\ell_1 : \beta, \ell_2 : \beta\} \sqcup \mu\gamma.\alpha \to \gamma$$

This is not in reduced form, both because there are two members of the join in the same component (the two record types), and a recursive type. We bring it into reduced form by unrolling the recursive type and merging the records, like so:

$$\{\ell_1 : \alpha \sqcup \beta\} \sqcup \alpha \to (\mu\gamma.\alpha \to \gamma)$$

Likewise, a state $q$ of a type or scheme automaton is said to be in reduced form when $q$ has:

- exactly one outgoing $d$-transition and $r$-transition, if $\langle\rightarrow\rangle \in H(q)$
- exactly one outgoing $\ell$-transition for $\ell \in L$ if $\langle L \rangle \in H(q)$

States in reduced form correspond to polar types in reduced form (having at most one type from each component), so Proposition 12 can be applied to decide subtyping between types that they represent.

Rather than devise an algorithm for converting the states of an automaton to reduced form, we point out that one already exists: the classical algorithm for converting a nondeterministic finite automaton to a deterministic one (the *subset construction*). Since it is well-known that this results in an automaton accepting the same language, the representation theorem of Section 7.2 tells us that it also does not affect the type represented by the automaton.

A minor technicality is that the subset construction produces an automaton where states have at most one outgoing transition with a given label, whereas the definition of reduced form required exactly one. This is easily remedied by inserting the missing transitions, transitioning to a new state $q$ where $H(q)$ is empty and $q$ has no outgoing transitions. As a finite automaton, this transition adds no new accepted strings, so the representation theorem again tells us the transformation is valid.

For states $q$ in reduced form, we adopt the notation $r(q)$ for the unique successor of $q$ along an $r$-transition (if $\langle\rightarrow\rangle \in H(q)$) and likewise with other transition symbols. If $\langle\rightarrow\rangle \in q$, then $q$ represents a meet or a join (according to polarity), one of whose terms is a function type of domain $d(q)$ and range $r(q)$.

## 8.3 Subsumption algorithm

The subsumption algorithm compares scheme automata in reduced form. We note that in order to decide subsumption ($\leqslant^{\forall}$), it suffices to decide equivalence of typing schemes ($\equiv^{\forall}$):

**Proposition 78.** $[\Delta_1]\tau_1 \leqslant^{\forall} [\Delta_2]\tau_2$ *iff* $[\Delta_2]\tau_2 \equiv^{\forall} [\Delta_1 \sqcap \Delta_2]\tau_1 \sqcup \tau_2$

*Proof.* Since $[\Delta_2]\tau_2 \leqslant [\Delta_1 \sqcap \Delta_2]\tau_1 \sqcup \tau_2$, $[\Delta_2]\tau_2 \equiv^{\forall} [\Delta_1 \sqcap \Delta_2]\tau_1 \sqcup \tau_2$ iff $[\Delta_1 \sqcap \Delta_2]\tau_1 \sqcup \tau_2 \leqslant^{\forall} [\Delta_2]\tau_2$. Suppose $[\Delta_1]\tau_1 \leqslant^{\forall} [\Delta_2]\tau_2$, that is, $\rho([\Delta_1]\tau_1) \leqslant [\Delta_2]\tau_2$ for some $\rho$. Assuming $[\Delta_1]\tau_1$ and $[\Delta_2]\tau_2$ contain distinct sets of type variables (renaming if needed), that gives $\rho([\Delta_1 \sqcap \Delta_2]\tau_1 \sqcup \tau_2) = [\Delta_2]\tau_2$, and hence $[\Delta_1 \sqcap \Delta_2]\tau_1 \sqcup \tau_2 \leqslant^{\forall} [\Delta_2]\tau_2$.

Conversely, suppose $[\Delta_1 \sqcap \Delta_2]\tau_1 \sqcup \tau_2 \leqslant^{\forall} [\Delta_2]\tau_2$, that is, $\rho([\Delta_1 \sqcap \Delta_2]\tau_1 \sqcup \tau_2) \leqslant [\Delta_2]\tau_2$ for some $\rho$. We have

$$\rho([\Delta_1]\tau_1) \leqslant \rho([\Delta_1 \sqcap \Delta_2]\tau_1 \sqcup \tau_2) \leqslant [\Delta_2]\tau_2$$

and so $[\Delta_1]\tau_1 \leqslant [\Delta_2]\tau_2$. □

That is, we may form least-upper-bounds in the $\leqslant^{\forall}$ ordering by using $\sqcup$ on the type component and $\sqcap$ on the environment components of typing schemes.

The subsumption algorithm decides $[\Delta_1]\tau_1 \leqslant^{\forall} [\Delta_2]\tau_2$ essentially by computing $[\Delta_1 \sqcap \Delta_2]\tau_1 \sqcup \tau_2$, and checking whether this differs from $[\Delta_2]\tau_2$. Intuitively, to check the subsumption in the example above, we first represent the

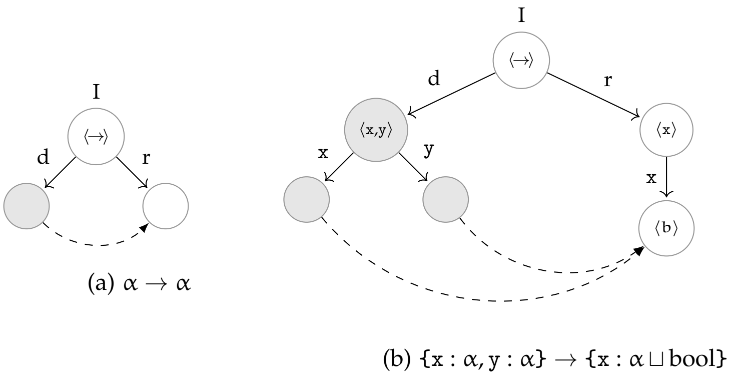

I
⟨→⟩
d
r
(a) $\alpha \to \alpha$
I
⟨→⟩
d
r
⟨x,y⟩
x
y
⟨x⟩
x
⟨b⟩
(b) $\{x : \alpha, y : \alpha\} \to \{x : \alpha \sqcup \mathrm{bool}\}$

Figure 8.1: Reduced-form scheme automata for subsumption example

two typing schemes as scheme automata in reduced form (Figs. 8.1a and 8.1b). We lay the diagram in Fig. 8.1a on top of that in Fig. 8.1b, and match up corresponding states, comparing their head constructors. If all of the corresponding head constructors are in the subtyping relation, we move onto the flow edges. Laying Fig. 8.1a on top of Fig. 8.1b creates a new flow edge between the states labelled ⟨x,y⟩ and ⟨x⟩, and so we check whether this new flow edge is *admissable* (see Section 7.3.2) using the algorithm in Section 8.4.

Given a subsumption problem $[D_1^-]t_1^+ \leqslant^\forall [D_2^-]t_2^+$, with the typing schemes represented as scheme automata in reduced form, we begin by calling:

$\mathsf{subsume}^+(q_1^+, q_2^+)$ where $q_1^+, q_2^+$ represent $t_1^+, t_2^+$
$\mathsf{subsume}^-(q_2^-, q_1^-)$ for each $x \in \mathrm{dom}\, D_2^-$, where $q_1^-, q_2^-$ represent $D_1^-(x), D_2^-(x)$

If this fails, we conclude that the subsumption does not hold. This is equivalent to a subtyping check:

$$[D_0^-]t_0^+ \leqslant [D_2^-]t_2^+$$

where $D_0^-, t_0^+$ are formed from $D_1^-, t_1^+$ by replacing all variable occurrences with $\bot$ and $\top$. Since $[D_0^-]t_0^+ \leqslant \rho([D_1^-]t_1^+)$ for any $\rho$, if this check fails then the subsumption cannot hold.

Otherwise, if this check succeeds, then the scheme automaton for $[D_1^- \sqcap D_2^-]t_1^+ \sqcup t_2^+$ must be the same as that for $[D_2^-]t_2^+$ except for flow edges. After $\mathsf{subsume}$ runs, the correspondence between states of both automata is stored in the tables $T^+, T^-$, so we find the flow edges to be added to $[D_1^- \sqcap D_2^-]t^+ \sqcup t_2^+$ as those pairs $(q_2^-, q_2^+)$ where:

$$(q_2^-, q_1^-) \in T^-$$
$$q_1^- \rightsquigarrow q_1^+$$
$$(q_1^+, q_2^+) \in T^+$$

If each of these new flow edges $(q_2^-, q_2^+)$ is admissable, then $[D_1^- \sqcap D_2^-]t_1^+ \sqcup t_2^+$ and $[D_2^-]t_2^+$ are equivalent, so the subsumption holds. We check admissability using the algorithm in the next section.

## 8.4 Deciding admissability of flow edges

Both the subsumption algorithm above and the algorithm for simplifying scheme automata in Section 7.3.2 require a decision procedure for subtyping

```
function subsume$^+(q_1^+, q_2^+)$
    if $(q_1^+, q_2^+) \notin T^+$ then
        $T \leftarrow T \cup \{(q_1^+, q_2^+)\}$
        if $\langle b \rangle \in H(q_1^+)$ then
            if $\langle b \rangle \notin H(q_2^+)$ then
                fail
        if $\langle \to \rangle \in H(q_1^+)$ then
            if $\langle \to \rangle \notin H(q_2^+)$ then
                fail
            subsume$^-(d(q_2^+), d(q_1^+))$
            subsume$^+(r(q_1^+), r(q_2^+))$
        if $\langle L_1 \rangle \in H(q_1^+)$ then
            if $\langle L_2 \rangle \notin H(q_2^+)$ for any $L_2 \subseteq L_1$ then
                fail
            subsume$^+(\ell(q_1^+), \ell(q_2^+))$ for $\ell \in L_2$
```

Figure 8.2: First part of subsumption algorithm (positive case, dual case subsume$^-$ elided)

problems of the form $t^- \leqslant t^+$, or equivalently (Proposition 76) an algorithm for deciding whether a flow edge $q^- \rightsquigarrow q^+$ is admissable.

The subtyping problems are represented as a pair of states $q^-, q^+$ of a scheme automaton. We write $q^- \leqslant q^+$ to mean that the flow edge $q^- \rightsquigarrow q^+$ is admissable, or equivalently that $t^- \leqslant t^+$, where $t^-, t^+$ are the types represented by $q^-, q^+$. Per Section 8.2.1 we assume the automaton to be in reduced form, so Proposition 12 then tells us that $q^- \leqslant q^+$ if and only if there is a subtyping relation in one of the components, that is, if one of the following conditions obtains:

- Both $H(q^-)$ and $H(q^+)$ contain $\langle b \rangle$.
- Both $H(q^-)$ and $H(q^+)$ contain $\langle \to \rangle$, and $d(q^+) \leqslant d(q^-), r(q^-) \leqslant r(q^+)$.
- Both $H(q^-)$ and $H(q^+)$ contain a record constructor, that of $H(q^-)$ has more fields, and $\ell(q^-) \leqslant \ell(q^+)$ for common fields.
- $q^- \rightsquigarrow q^+$ (that is, there is a flow edge between them and so they have a variable in common).

Checking whether $q^- \leqslant q^+$ is a straightforward recursion testing the above conditions. We memoise the recursion, both to increase performance on repeated subproblems, and to ensure termination in case of recursive types.

Since Proposition 76 of Section 7.3.2 showed that $q^- \leqslant q^+$ iff a flow edge $q^- \rightsquigarrow q^+$ is admissable, we may use the table of flow edges itself as the memoisation table, leading to the algorithm shown in Fig. 8.3.

## 8.5 Summary

This chapter was not especially long, and the algorithms it introduces are not unusually complicated. Neither are they entirely novel: the algorithm for deciding $t^- \leqslant t^+$ bears a strong resemblance to Pottier's entailment algorithm [Pot98b].

```
function admissable($q^-, q^+$)
    if $q^- \rightsquigarrow q^+$ then
        return TRUE
    else
        Insert flow edge $q^- \rightsquigarrow q^+$
        if $\langle b \rangle \in H(q^-) \cap H(q^+)$ then
            return TRUE
        if $\langle \rightarrow \rangle \in H(q^-) \cap H(q^+)$ then
            if admissable($d(q^+), d(q^-)$), admissable($r(q^-), r(q^+)$) then
                return TRUE
        if $\langle L_1 \rangle \in H(q^-)$, $\langle L_2 \rangle \in H(q^+)$, $L_1 \supseteq L_2$ then
            if admissable($\ell(q^-), \ell(q^+)$) for $\ell \in L_2$ then
                return TRUE
        Remove flow edge $q^- \rightsquigarrow q^+$
        return FALSE
```

Figure 8.3: Second part of subsumption algorithm: deciding flow edge admissability

Yet they resolve the problem of subsumption between polymorphic type schemes, a problem long considered intractable in previous formulations [TS96, Pot98b]. The work that enabled this was not done in this chapter, but in Chapter 3 when the lattice of types was built. In particular, the construction in Chapter 3 is such that Proposition 12 holds, which is what ensures the algorithms above are complete. Previous work defined types in a non-extensible way, for which Proposition 12 fails, causing incompleteness when these algorithms are used (see Section 10.3 for some further notes).

# 9 Extensions

*Structures are the weapons of the mathematician.*

—N. Bourbaki

Until this point, the type system being developed has remained spartan. Limiting the system to boolean, function and record types simplified the formal development and the description of the type inference algorithms, but real programming languages are more featureful.

This chapter demonstrates the generality of the system being developed by showing how some other type system features (user defined types, mutability, sums, etc.) fit into its framework. Little additional effort is required to integrate these features into type inference: indeed, most of them have already been implemented in the prototype compiler for MLsub. More speculative extensions, whose integration with MLsub is less obvious, are discussed in Section 11.1.

Recall from Chapter 3 (in particular, Section 3.2.3) that the lattice of types is built as a sum of components $C_i$:

$$\mathcal{T} = \sum_i (C_i)_\bot^\top$$

These components are all built from a small set of ingredients: constant lattices, finite products, and the functors $(-)^{\mathrm{op}}$ and $(-)^\top$. By combining these in different ways, we can develop a surprising number of useful features.

## 9.1 User-defined types

Before introducing more complex types, we first see how to name the ones that we already have, by introducing type definitions:

```
type id = int
```

There are two modes of use of these definitions:

- *Transparent* definitions, where the newly introduced name is equivalent to its definition, so that `ids` and `ints` can be used interchangeably.

- *Opaque* definitions, where the newly introduced name is a fresh type, incompatible with its definition.

Abstract data types can be expressed neatly by treating their definition as transparent inside the module defining them, and as opaque outside it.

Transparent definitions are easily implemented. In principle, they can be expanded in a preprocessing phase and ignored afterwards. In practice, this may cause an exponential increase in compile time as their definitions may be much larger than their names.

This issue, however, is exactly the same as that faced by implementations of, say, Standard ML or Haskell. The addition of subtyping does not make this any more difficult, and the standard solutions (lazy expansion of type aliases) apply. So, we do not discuss this issue further.

Opaque type definition cannot be preprocessed away. Instead, for each opaque type definition, we introduce a new component to the definition of the type lattice. For type definitions without parameters (like id above), this new component is simply $1$, giving such definitions the same status as type variables or the type bool.

Types with parameters are not much more difficult. To define a parameterised type ListPair[S, T], representing immutable lists whose elements are pairs of S and T, we simply add a component $A \times A$ to the definition of types. In general, the component corresponding to a type definition is a finite product of $A$ and $A^{\mathrm{op}}$, depending on whether the parameters are co- or contra-variant. (In the example ListPair[S, T], both parameters are covariant).

### 9.1.1 Variance and mutability

Most languages which support subtyping and user-defined types, from object-oriented languages like Java to functional ones like OCaml, support *invariant* parameters as well as co- and contra-variant ones. Invariant parameters are neither co- nor contra-variant. If a type ty[T] is invariant, then ty[X] is a subtype of ty[Y] only when X and Y are equal.

The classical examples of invariant parameters involve mutability. Suppose we have a type MList[T] of mutable lists whose elements are of type T, equipped with operations get and put to read and write elements of the list. The type of get applied such a list mentions T only in its return type, and is therefore covariant in T. However, the type of put applied to such a list mentions T only in its argument type, and is contravariant in T. The type MList[T] cannot therefore be said to be either co- or contra-variant in T.

Unlike co- and contra-variant parameters, invariant parameters cannot be encoded using the small set of ingredients we have allowed ourselves. This is not a simple omission: many parts of the system described in this thesis assume that all fields of all type constructors are either co- or contra-variant. For instance, the biunify algorithm produces a bisubstitution, acting differently on positive and negative occurrences of variables, while implicitly assuming that all occurrences of a variable are one or the other.

Rather than admit defeat here, I argue that invariance is a poor way to describe the parameter of MList, which makes it gratuitously difficult to typecheck quite reasonable pieces of code. Consider the following piece of code in Java (where ArrayList<T> plays the role of MList[T]):

```
void disableAll(ArrayList<Component> comps) {
  for (Component c : comps) {
    c.setEnabled(false);
  }
}
```

This function takes a list of Components (GUI controls) and disables them all. It seems reasonable to pass to this function an ArrayList<Button>, but this causes a type error. Due to invariance, ArrayList<Button> is not a subtype of ArrayList<Component>. Indeed, it would be unsound to make it so, since Checkboxes can be inserted into an ArrayList<Component>, but not into a ArrayList<Button>.

However, it is perfectly safe to pass the list of buttons to disableAll, which never inserts anything into its parameter list. To allow this usage in Java, we need to explicitly quantify over subtypes, as follows:

```
void disableAll(ArrayList<? extends Component> comps) {
  ...
}
```

This works, but requires some fairly heavy machinery: bounded quantification (the extends) and existential types (the ?). These annotations are not inferred, and must be manually specified.

A simpler approach is to avoid the use of invariance entirely, and use a technique used by Pottier, which he attributes to Cardelli [Pot98b]. We add a second type parameter, giving the type MList[(S,T)] which has *two* parameters, the contravariant S and the covariant T. The type schemes of the get and put operations are:

$$\begin{aligned} \mathtt{get} &: \forall\alpha.\mathtt{MList[(}\bot,\alpha\mathtt{)]} \to \mathrm{int} \to \alpha \\ \mathtt{put} &: \forall\alpha.\mathtt{MList[(}\alpha,\top\mathtt{)]} \to \mathrm{int} \to \alpha \to \mathrm{unit} \end{aligned}$$

The contravariant parameter appears in the type of put, describing the type of elements that may be inserted, while the covariant parameter appears in the type of get, describing the type of elements returned. The singleton function, which creates a new one-element list, has this type scheme:

$$\forall\alpha.\alpha \to \mathtt{MList[(}\alpha,\alpha\mathtt{)]}$$

Since all of the type parameters are either co- or contra-variant, this approach is compatible with our type inference algorithm, which infers this type for disableAll:

$$\mathtt{disableAll} : \mathtt{MList[(}\bot\mathtt{,Component)]} \to \mathrm{unit}$$

That is, disableAll requires a MList out of which you may take Components, but into which you need only be able to put $\bot$. The type MList[(Button, Button)] is a subtype of this type, since $\bot \leqslant \mathtt{Button} \leqslant \mathtt{Component}$, so this is as expressive as the version involving bounded quantification.

### 9.1.2 Type parameter notation

The trick of replacing each invariant type parameter with a pair of types ensures that principal type inference is possible with mutable collections, and gives useful types to functions like disableAll that use a mutable collection covariantly. However, it is awkward to double the number of type parameters in the program.

Happily, some mild syntactic sugar can smooth over this awkwardness. We allow a single type parameter in the surface syntax, and expand it as follows:

$$\begin{array}{rcl}
\texttt{MList[t]} & \mapsto & \texttt{MList[(t,t)]} \\
\texttt{MList[+t]} & \mapsto & \texttt{MList[(0,t)]} \\
\texttt{MList[-t]} & \mapsto & \texttt{MList[(t,0)]} \\
\texttt{MList[-}t_1\ \texttt{+}t_2\texttt{]} & \mapsto & \texttt{MList[(}t_1\texttt{,}t_2\texttt{)]}
\end{array}$$

The fourth case above is unusual, but necessary to assign a principal type for certain functions. For instance, if the `disableAll` method were modified to insert a `Button` into its argument, the inferred type of its argument would be `MList[Button,Component]`: that is, a list into which one may put a `Button`, but out of which one is only guaranteed to receive `Components` (Note that both homogeneous lists of `Buttons` and heterogeneous lists of different types of `Component` satisfy this specification).

The type 0 is a shorthand for either $\bot$ or $\top$, depending on whether it appears in a positive type (where only $\bot$ is allowed) or a negative one (the reverse). So, the positive type

$$t^+ = \texttt{MList[+bool]} \to \texttt{MList[+bool]}$$

is shorthand for

$$t^+ = \texttt{MList[(}\bot\texttt{,bool)]} \to \texttt{MList[(}\top\texttt{,bool)]}$$

Thus, the type of `disableAll` is displayed as follows:

$$\texttt{disableAll} : \texttt{MList[+Component]} \to \text{unit}$$

Superficially, this resembles the Java version using bounded quantification and wildcards, and is useful in much the same ways. However, by being based on simpler machinery (co- and contra-variance), this remains compatible with the type inference algorithm: the type for `disableAll` above is inferred, and no annotations are needed.

## 9.2 Sum types

So far, we have discussed product types (records) but not sums. Since sums are dual to products, the lattice of sum types is a simple modification of the lattice of record types. Record types were defined as follows:

$$\mathcal{L}^{(\mathcal{T}^\top)}$$

The lattice $\mathcal{T}^\top$ is the lattice of types plus an extra top element, so a record type is defined by a function mapping each possible label to either a type or this extra element, which is equivalently a partial function from labels to types. Since the extra element lies above everything else, a record types with an absent label is a supertype of the same record type with the label present – the usual depth subtyping rule.

To encode sum types, we use the same pattern but add an extra bottom element instead:

$$\mathcal{L}^{(\mathcal{T}_\bot)}$$

These types are similarly partial functions from labels to types, for which we adopt the following syntax (compare to {...}):

$$[\ell_1 : \tau_1 \mid \ell_2 : \tau_2 \mid \ldots \mid \ell_n : \tau_n]$$

Since the extra element representing absent labels lies below everything else, a sum type with an absent label is a subtype of the same sum type with the label present, so e.g. $[\ell_1 : \tau_1] \leqslant [\ell_1 : \tau_2 \mid \ell_2 : \tau_3]$ whenever $\tau_1 \leqslant \tau_2$.

The values of sum types are pairs of a label and a value $\ell\,\nu$, and semantically, the labels in a sum type represent possible cases. If more labels are present, then more cases are possible and less information is known, making the type a supertype. Sum types have a simple introduction form:

$$(\textsc{Lab}) \quad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \ell\, e : [\ell : \tau]}$$

Compare this to ($\textsc{Proj}$), the elimination form for records. The elimination form for sums, the `match` expression, is dually comparable to the introduction form for records:

$$(\textsc{Match}) \quad \frac{\Gamma, x_1 : \tau_1 \vdash e_1 : \tau \quad \ldots \quad \Gamma, x_n : \tau_n \vdash e_n : \tau}{\Gamma \vdash \texttt{match } e \texttt{ with } \ell_1\, x_1 \to e_1, \quad \ldots \quad \ell_n\, x_n \to e_n : \tau}$$

Type inference is done in the same way as for the previously-introduced types.

### 9.2.1 Tagged records

The type definitions of many functional programming languages are in the form of sums of products. While useful, this combination can sometimes be slightly awkward to deal with, since the outer sum structure must be eliminated before the inner product structure can be used. For instance, consider this definition of a simple arithmetic expression type, written in OCaml syntax:

```
type exp =
  | Const of loc * int
  | Var of loc * var
  | Add of loc * exp * exp
  | Sub of loc * exp * exp
  | Mul of loc * exp * exp
  | Div of loc * exp * exp
```

Each of these cases includes as its first field a source location. However, extracting the location from an expression is a surprisingly verbose operation:

```
let loc_of e = match e with
  | Const (loc, _) → loc
  | Var (loc, _) → loc
  | Add (loc, _, _) → loc
  | Sub (loc, _, _) → loc
  | Mul (loc, _, _) → loc
  | Div (loc, _, _) → loc
```

Since the product is nested inside the sum, the sum must be eliminated using `match` before the location can be projected out.

An alternative is possible. Instead of adding both record and sum types to the lattice of types, we add a single sum-of-product construction, as the composition of both:

$$\mathcal{L}^{\left(\mathcal{L}^{(\mathcal{T}_\perp)}\right)^\top}$$

Now, sum types must always be sums of records, of the general form:

$$[\ell_1 \{\ell_{1,1} : \tau_{1,1}, \ldots\} \mid \ell_2 \{\ell_{2,1} : \tau_{2,1}, \ldots\} \mid \ldots]$$

The general introduction form has both a label for the entire value (indicating which case of a sum type), as well as labelled fields, which we call a *tagged record*:

$$\ell\,\{\ell_1 : e_1, \ldots\}$$

The plain record type $\{\ldots\}$ becomes a shorthand for a sum mapping every possible label to the same record:

$$[\ell_1 : \{\ldots\} \mid \ell_2 : \{\ldots\} \mid \ldots]$$

Of course, an implementation should choose an optimised representation for this common case where all or most labels map to the same record.

The advantage of this approach is that the subtyping order induced by the definition above makes a sum-of-products a subtype of a simple product type having the fields common to all cases. For example:

$$\begin{gathered}[\ell_1 : \{\texttt{foo} : \mathrm{int}, \texttt{bar} : \mathrm{bool}\} \mid \ell_2 : \{\texttt{foo} : \mathrm{int}, \texttt{baz} : \mathrm{int} \to \mathrm{int}\}] \\ \leqslant \\ [\ell_1 : \{\texttt{foo} : \mathrm{int}\} \mid \ell_2 : \{\texttt{foo} : \mathrm{int}\}] \\ \leqslant \\ \{\texttt{foo} : \mathrm{int}\}\end{gathered}$$

This allows sums of records to be treated as plain records having the common fields, and in particular allows `loc_of` above to be implemented as:

```
let loc_of e = e.loc
```

### 9.2.2 Row and presence variables

The simple `match` statement above is easy to typecheck, but well short of the full-featured matching constructs available in real programming languages. One important feature it lacks is a *default case*, allowing a wildcard case which matches all unmentioned labels. The general syntax for such a match is:

$$\texttt{match}\ e\ \texttt{with}\ \ell_1\, x_1 \to e_1,\ \ \ldots\ \ \ell_n\, x_n \to e_n,\ x_* \to e_*$$

When $e$ evaluates to $\ell\, v$, and $\ell \neq \ell_i$ for any $1 \leqslant i \leqslant n$, then the result of the `match` statement is the result of evaluating the *default case* $e_*[v/x_*]$.

In languages like Standard ML, where all sum types must be predeclared and labels can be part of only one sum, such default cases pose few difficulties as they can simply be expanded to list the possible labels. However, when sum types are not predeclared and instead inferred structurally, it is more difficult to precisely type default cases.

For instance, consider this function:

$$f = \lambda x.\,\texttt{match}\ x\ \texttt{with}\ \ell_1\, x_1 \to \texttt{true},\ x_* \to x_*$$

Semantically, it makes sense to treat $f$ as having any of the following types:

$$\begin{aligned} f &: [\ell_1 : \top] \to \mathrm{bool} \\ f &: [\ell_1 : \top, \ell_2 : \mathrm{bool}] \to \mathrm{bool} \\ f &: [\ell_1 : \top, \ell_2 : \alpha] \to \mathrm{bool} \sqcup \alpha \end{aligned}$$

Assigning f a principal type which subsumes all of these is tricky. Below, we present one solution, a slight variant of Rémy's *row variables* [Rém94]. Instead of defining a single algebra of types $\mathcal{T}$, we simultaneously define two algebras, $\mathcal{T}$ (types) and $\mathcal{P}$ (presences). This moves us from ordinary first-order algebra to *multi-kinded* first-order algebra, but does not require deep changes to the theory. (Moving to a higher-kinded setting would require deeper changes, discussed briefly in Section 11.1.3). The algorithms previously presented require almost no changes to cope with a new kind.

The original definition of types with sum types was:

$$\mathcal{T} = (\mathcal{L}^{(\mathcal{T}_\perp)})^{\top}_{\perp} + \dots$$

where the "$\dots$" stands for the other components: functions, records, type variables and so on. The change is to separate sum types into separate types and presences:

$$\begin{aligned}\mathcal{T} &= (\mathcal{L}^{\mathcal{P}})^{\top}_{\perp} + \dots\\ \mathcal{P} &= \mathcal{T}_\perp + \mathsf{Free}(V_P)\end{aligned}$$

The difference is that as well as types, presences can be represented as *presence variables* drawn from the set $V_P = \{p_1, p_2, \dots\}$. This allows us to quantify over whether a label is present in a sum type.

We write the bottom element of $\mathcal{P}$ as `abs`, leaving $\perp$ to refer to the bottom element of $\mathcal{T}$. We leave the injection from $\mathcal{T}$ to $\mathcal{P}$ implicit. Thus, $\mathcal{P}$ contains distinct elements `abs`, $\perp$, $\top$, $p_1$, $p_1 \sqcup \mathrm{bool}$, and so on.

We quantify over presences (elements of $\mathcal{P}$) with $\pi$. Note that quantifying over all $\pi$ is different from using $p_1$: as per the construction of types in Chapter 3, presences variables appear in the lattice directly as indeterminates.

We use a similar syntax as for plain sum types, writing:

$$[\ell_1 : \pi_1 \mid \ell_2 : \pi_2 \mid \dots \mid \ell_n : \pi_n]$$

Labels not listed are assumed to be assigned presence `abs`. With this setup, we write new (Lab) and (Match) rules:

$$\begin{array}{rc}
(\textsc{Lab}) & \dfrac{\Gamma \vdash e : \tau}{\Gamma \vdash \ell\, e : [\ell : \tau]}\\[2ex]
(\textsc{MatchClosed}) & \dfrac{\Gamma \vdash e_1 : [\ell : \tau] \quad \Gamma, x : \tau \vdash e_2 : \tau'}{\Gamma \vdash \texttt{match}\ e_1\ \texttt{with}\ \ell\, x \to e_2 : \tau'}\\[2ex]
(\textsc{MatchOpen}) & \dfrac{\begin{array}{l}\Gamma \vdash e_1 : [\ell : \tau, (\ell_i : \tau_i \text{ for } \ell_i \neq \ell)]\\ \Gamma, x : \tau \vdash e_2 : \tau'\\ \Gamma, x : [\ell : \texttt{abs}, (\ell_i : \tau_i \text{ for } \ell_i \neq \ell)] \vdash e_3 : \tau'\end{array}}{\Gamma \vdash \texttt{match}\ e_1\ \texttt{with}\ \ell\, x \to e_2 \mid x \to e_3 : \tau'}
\end{array}$$

Expressions that match several different labels can be expanded to a nest of (MatchOpen) expressions followed by a single (MatchClosed). These rules give expressive types to functions that perform matching:

$$\begin{gathered}\lambda x.\texttt{match}\ x\ \texttt{with}\ \ell\, x \to \ell\, x + 1 \mid x \to x\\ :\\ [\ell : \texttt{int}, (\ell_i : p_i \text{ for } \ell_i \neq \ell)] \to [\ell : \texttt{int}, (\ell_i : p_i \text{ for } \ell_i \neq \ell)]\end{gathered}$$

This function takes a sum which may be $\ell n$ for some integer $n$, and is *polymorphic* in the presence of other labels. That is, if given an $[\ell : \texttt{int}, \ell_2 : \texttt{bool}]$, it returns an $[\ell : \texttt{int}, \ell_2 : \texttt{bool}]$. Labels other than $\ell$ are present in the result type if they are present in the input.

This presentation differs somewhat from Rémy's row and presence variables, by encoding the presence information using subtyping instead of using higher-kinded polymorphism. Rémy's presence variables are a pair of a type constructor and a type, where the type represents the type associated to the label, and the type constructor ranges over the identity (if the label is present), or the constant `abs` (if absent).

Due to its use of subtyping, the system presented here can express some types that Rémy's presence variables cannot, for instance, a function that takes two sum types and returns a sum containing label $\ell$ if either of the inputs did (using the $\sqcup$ operator on presence variables). On the other hand, due to the use of higher-kinded presence variables, variants of Rémy's system [Gar02] can express some types that the current system cannot, for instance, a sum type that may or may not contain a type at label $\ell$, but which if present must be `int`.

## 9.3 Complex function types

The function types studied so far have arity 1, that is, they map a single argument to a result. Such function types can be used to simulate multiple-argument functions with currying (encoding a two-argument function as $t_1 \to (t_2 \to t_3)$), but by directly supporting complex arities we can gain convenient support for named and optional arguments.

### 9.3.1 Multiple arguments

The simplest extension is to directly support multiple-argument functions. This entails replacing the single component $\mathcal{T}^{\mathrm{op}} \times \mathcal{T}$ used for functions with a component for each arity $n$ (up to some finite maximum, to preserve the finiteness condition):

$$C_n = (\mathcal{T}^{\mathrm{op}})^n \times \mathcal{T}$$

The application and abstraction rules are almost unchanged, and inference proceeds as before:

$$(\textsc{Abs}) \quad \frac{\Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e : \tau}{\Gamma \vdash \lambda(x_1, \ldots, x_n).e : (\tau_1, \ldots, \tau_n) \to \tau}$$

$$(\textsc{App}) \quad \frac{\Gamma \vdash e_f : (\tau_1, \ldots, \tau_n) \to \tau \quad \Gamma \vdash e_1 : \tau_1 \quad \ldots \quad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash e_f(e_1, \ldots, e_n) : \tau}$$

### 9.3.2 Named arguments

*Named arguments* are identified by a label, rather than by their position in the list of arguments. This improves readability, since it does not require the reader of some code to remember the order of arguments to each function. *Optional arguments* are also identified by a label, but the function definition contains a default value, and calls which omit this argument supply the default value instead.

Consider a function definition mixing all three types of arguments, for which we adopt this syntax:

```
fun f(a, b, ~x, ~y=42) → ...
```

The syntax `~x` is used to indicate a named parameter: `a` and `b` are positional (unnamed), while `x` and `y` are named (with `y` having a default value).

The general case of these complex function types looks like:

$$(t^-, \dots, \ell : t^-, \dots, ?\ell : t^-) \to t^+$$

That is, they have contravariant type parameters for each argument (positional, mandatory named and optional named), and a covariant result. Picking the right subtyping order for these types is a somewhat subjective exercise. It seems clear that a type with a mandatory named argument should be a subtype of the same type with the argument optional, but what should be done with an unexpected named argument? It could be silently ignored or flagged as a type error, and either choice would be sound.

I take the view that it should be an error, giving the following subtyping relations:

$$\begin{aligned}(\ell : \alpha) \to \beta &\not\leqslant\not\geqslant () \to \beta \\ (\ell : \alpha) \to \beta &\leqslant (?\ell : \alpha) \to \beta \\ () \to \beta &\leqslant (?\ell : \alpha) \to \beta\end{aligned}$$

This can be encoded using the same ingredients that we used for record and simple function types. We define a separate component $C_n$ to describe the complex function types with $n$ positional arguments[1]:

$$C_n = (\mathcal{T}^{\text{op}})^n \times \mathcal{L}^{(\mathcal{T}^{\text{op}})^\top} \times (2^{\mathcal{L}})^{\text{op}}$$

$C_n$ consists of three parts:

- $n$ contravariant types, for the positional parameters
- Some number of labelled contravariant types, for the named parameters
- A set of labels, specifying which named parameters are mandatory

The subtyping ordering this imposes ensures that adding more named parameters produces a subtype, but specifying that more of them are required produces a supertype. Therefore, adding a new optional parameter yields a subtype, making an optional parameter required yields a supertype, but adding a new mandatory parameter yields and unrelated type.

## 9.4 Effect systems

In most programming languages, whether evaluation of an expression causes side-effects is not tracked statically. This means that the high degree of assurance given by a static type system about the values produced by an expression does not extend to the side effects it may cause. Higher-order stateful functions such as memoisation operators do not have their inputs statically checked for suitability (it is in general incorrect to cache the result of a side-effecting procedure).

*Effect systems* remedy this, by extending the usual role of a type system to include tracking of side effects. There is extensive literature on the design of effect systems, varying wildly in the details of the sort of effects tracked. Our

[1] For $n$ less than some arbitrary finite maximum.

contribution to the area is not to design another such system, but merely to point out that the machinery developed in this thesis for type inference with subtyping can also be used to infer effects.

Usually, the effect tracked by an effect system for a piece of code is some over-approximation of the effects it may actually perform, allowing code to be marked with an effect `E` even if it only performs `E` conditionally, or not at all. This leads to an effect-system analogue of subtyping called *subeffecting*, involving a typing rule much like the standard subtyping rule that allows the collection of allowed effects to be freely increased.

Some form of subeffecting is useful in particular to type conditional expressions. Both branches of an `if`-expression often perform different effects, and subeffecting allows the entire expression to be annotated with the upper bound of both effects.

Attempting inference of effect sets carries with it the usual troubles of type inference and subtyping that have been the focus of this thesis. Once polymorphism is added (say, to allow a `map` function which works on both pure and side-effectful functions), the problem of effect inference is essentially a special case of the type inference problems studied in this thesis. This is not a new observation: Pottier's type inference scheme [Pot98b] is used to infer effects in Bauer and Pretnar's language `Eff` [Pre14].

Therefore, inferring types with effects is a reasonably straightforward extension of the current system. We add a new kind $\mathcal{E}$ of effects (just as a new kind $\mathcal{P}$ of presences was added in Section 9.2.2), and change the definition of function types from

$$\mathcal{T}^{\text{op}} \times \mathcal{T}$$

to

$$\mathcal{T}^{\text{op}} \times \mathcal{E} \times \mathcal{T}$$

Various simple effect systems can be described by a suitable choice of $\mathcal{E}$. Taking $\mathcal{E} = 2^{\mathcal{L}}$ gives the simplest effect system, where effects are described by a set of labels describing actions a function may perform. More complex effect systems are possible by e.g. defining effects to be a sum type which may refer to $\mathcal{T}$.

In general, the combination of subtyping, polymorphism and type inference available in MLsub makes experimenting with new type system features unusually easy.

# 10 Related work

*Such is the advantage of a well-made language, that its simplest notations often become the source of the most profound theories.*

—Pierre-Simon Laplace

Much work on type inference with subtyping, starting with the early work by Fuh and Mishra [FM90] and Mitchell [Mit91], infers *constrained types*. The core idea is simple: infer types expression-by-expression following the same pattern as Milner's Algorithm $\mathcal{W}$, but when a constraint arises (e.g. that a function's provided argument must be a subtype of what it accepts), attach it to the inferred type rather than eliminating it with unification. Many type system features can be integrated into this model, including lattice operations and recursive types [AW93].

The result of this process is a *constrained type*, a pair of a ML-style type expression and some set of associated subtyping constraints. Immediately, a number of questions arise:

- How do we define the subtyping order? As well as the supported type constructors (e.g. functions, records), should it have recursive types? Should it form a lattice? Is the order decidable?
- How should polymorphism be handled?
- Can we *simplify* constrained types, reducing an unwieldy inferred set of constraints into a shorter equivalent form?
- Can we solve *entailment* between constraint sets (whether one set of constraints implies another), as needed to check an inferred constrained type against a type signature?

The problem of entailment is superficially similar to simplification, but much more difficult. For simplification, heuristics are quite useful: a failure to optimally simplify a constrained type is not necessarily serious. Inexact heuristics are much worse for entailment, since they lead to programs being rejected if the compiler's heuristics cannot see why a valid type signature matches. Furthermore, simplification can choose which heuristics to apply, while entailment must handle arbitrary constraint sets.

In the following sections, we discuss each of these points in turn.

## 10.1 The subtyping order

One influential answer to the first question is the subtyping order of Amadio and Cardelli [AC93] whose types are defined as ground terms (i.e. without free variables) over the following syntax:

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

The recursive types definable with the fixed-point operator $\mu$ are exactly the *regular trees*, which are the possibly-infinite trees over the following syntax having only finitely many distinct subtrees:

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

These are equipped with a partial order $\leqslant$ which is bounded by $\bot$ and $\top$, where $\to$ is covariant in the range and contravariant in the domain[1].

Amadio and Cardelli give an algorithm for deciding subtyping between ground types in this system, but their algorithm takes exponential time if recursive types are nested in complex ways. Kozen et al. [KPS93] give a polynomial-time algorithm, while Palsberg and Smith [PS96] proved it equivalent to a system of constraints, and Palsberg and O'Keefe [PO95] proved it equivalent to a certain flow analysis.

For a type system based on constrained types to be useful, it is necessary to decide satisfiability of a set of constraints, since a type constrained by unsatisfiable constraints is more properly reported as a type error. Deciding whether a system of constraints (over the Amadio-Cardelli types or similar systems) is satisfiable was shown to be doable in polynomial time by Eifrig, Smith and Trifonov [EST95b] (see also Rehof's work [Reh98]).

This subtyping order forms a lattice, albeit not a distributive one. Additionally, it has two important features: it possesses recursive types, and it is *non-structural*.

### 10.1.1 Structural and non-structural subtyping

In some subtyping orders, the subtyping relation $t_1 \leqslant t_2 \to t_3$ implies that $t_1$ itself is a function type. If this implication holds, and extends to every parameterised type constructor (that is, subtyping relations hold only between applications of the same type constructor, or between unparameterised types), then the subtyping is said to be *structural*. (Structural subtyping is not to be confused with "structural typing". This unfortunate clash of names is explained in the footnote on page 33).

In other words, systems with structural subtyping have nontrivial subtyping relationships only between base (unparameterised) types, allowing $\texttt{int} \leqslant \texttt{real}$[2], but not $\texttt{List}[\alpha] \leqslant \texttt{Collection}[\alpha]$.

The advantage of structural subtyping is that type inference is made much easier, since constraints like $\alpha \leqslant t_2 \to t_3$ which arise during e.g. function applications can be eliminated by replacing all occurrences of $\alpha$ with $\beta \to \gamma$,

[1] The co/contravariant subtyping order for function types is one of the few points of general agreement between subtyping systems, although sadly even that is not universal [Cas95].

[2] $\texttt{int} \leqslant \texttt{real}$ is the standard example of a subtyping relation between basic types. However, it is generally a mistake to have this relation in a programming language, since the usual semantics of `real` (IEEE-754 floating-point arithmetic) are incompatible with integer arithmetic, disagreeing on the value of $5/3$, the validity of $x + 1 \neq x$, and the existence of infinity.

for fresh $\beta,\gamma$. Type inference proceeds much as ML's type inference, with subtyping only involved at atomic types [FM90, Mit91].

However, structural subtyping does not allow the usual subtyping order for records, which requires a record with more fields to be a subtype of one with fewer. Thus, structural subtyping cannot express subtyping as it is used in object-oriented programming, which relies on being able to pass objects of a subclass (possibly with extra methods) where objects of a superclass are required. Even the presence of least or greatest types makes the type system non-structural, since $\bot \leqslant t_1 \to t_2$ and yet $\bot$ is not a function type.

Structural subtyping is however a good fit for other applications. Simonet's Flow Caml system [Sim03] is an extension of the OCaml type system which uses structural subtyping to attach security labels tracking information flow to types. Bauer and Pretnar's language `Eff` [Pre14] is an ML-like language which uses structural subtyping to attach effect information to types. Both of these systems have a trivial subtyping relation for normal type constructors (functions, records, tuples, etc.), having nontrivial subtyping only on the leaf annotations (security levels for Flow Caml and effects for `Eff`).

### 10.1.2 Recursive types

Amadio and Cardelli's system has *equirecursive types*, which are recursive types which are implicitly equal to their unrollings. That is, $\mu\alpha.t$ and $t[\mu\alpha.t/\alpha]$ are equivalent types, and a term of one type may be used wherever a term of the other is expected, with no syntactic marker for the unrolling.

This is to be compared with *isorecursive types*, as found in ML, Haskell and other languages, which require explicit type declarations or rolling and unrolling operations for every recursive type. That is, the types that we write with $\mu$ are exactly equivalent to their unrollings: the typechecker rolls and unrolls them as needed (equirecursive). The recursively-defined types of ML and Haskell are merely isomorphic to their unrollings: an explicit operation (usually disguised as a constructor application or a pattern-match) must be used to convert between a type and its unrolling (isorecursive). See Pierce [Pie02] for more discussion of the distinction.

As pointed out in Section 2.1.6, many formulations of non-structural subtyping (such as the one in this thesis) cannot have principal types without supporting equirecursive types, since there are terms having many non-recursive types whose principal types are recursive. For this reason, equirecursive types are a more common feature of type systems with non-structural subtyping than of type systems in general.

## 10.2 Polymorphism

The next question is how to integrate polymorphism with subtyping. For the moment, we discuss *parametric polymorphism*, which allows functions to work at multiple types by using a type variable to stand for an arbitrary type, and requires that the function's behaviour be the same at all types. This is to be contrasted with *ad-hoc polymorphism*, discussed briefly below in Section 10.2.3, which allows a function to have different behaviour at different types.

There are two distinct ways to view a polymorphic type like $\forall\alpha.t$: in the ML style, we view it as its set of instances, while in the System F style, we view it as a function from types to types. In type systems without subtyping, the distinction matters little, but it has profound effects in the presence of subtyping (see Section 5.2.2).

### 10.2.1 ML-style polymorphism

The core ingredient in ML-style polymorphism is the *subsumption* relation, which relates polymorphic type schemes which have the same instances. This relation has been present since the beginnings of ML [DM82], but its extension to subtyping was introduced by Trifonov and Smith [TS96].

Their subsumption relation was defined in the style of a simulation: one type scheme subsumes another if for every instance of the second, there is a more general instance of the first. Section 4.2 discusses this relation under the name $\leqslant^{\forall}_{\text{sim}}$ and proves it equivalent to the simpler definition of $\leqslant^{\forall}$ used throughout this thesis (Proposition 31). However, since Trifonov and Smith treat type variables non-extensibly (by quantifying over ground types, see Section 1.4.2), the proof of Proposition 31 does not go through in their system and $\leqslant^{\forall}$ and $\leqslant^{\forall}_{\text{sim}}$ disagree.

Including type variables directly in the subtyping order is not a new idea. Brandt and Henglein [BH98] do just this, in their coinductive construction of a subtyping order. However, Brandt and Henglein emphasise the similarity of their construction to the ground types of Amadio and Cardelli [AC93], and the effect of including type variables directly on the subsumption relation does not appear to be well appreciated in the literature. See Section 3.3 for further discussion of this point.

### 10.2.2 System F-style polymorphism

Taking an alternative approach, polymorphic types may be considered as type-level functions that must be applied. The canonical example of such a style is System F, which was extended by Cardelli et. al. to System $F_{<:}$ [CMMS94] which includes *bounded quantifiers* $\forall\alpha \leqslant t_1.t_2$, which quantify over subtypes of a given type $t_1$.

There are several variants of System $F_{<:}$. Curien and Ghelli's System $F_{\leqslant}$ [CG90] is almost identical, differing only in a technicality of the equational theory. *F-bounded quantification* [CCH$^+$89] allows type variables to appear in their own bounds, admitting a form of recursive types. *Kernel* $F_{<:}$ restricts the subtyping order of $F_{<:}$, producing a simpler system.

It is a remarkable result of Pierce [Pie92] that the subtyping relation of $F_{<:}$ is undecidable, by reduction from the halting problem of two-counter Turing machines. The simpler Kernel $F_{<:}$ system has decidable subtyping.

Since these systems generalise plain System F, type inference is undecidable in all of them. Section 11.1.2 discusses some techniques that have been used to integrate some System F-style quantification into ML-style languages while maintaining good type inference, leaving their integration with MLsub as future work.

### 10.2.3 Ad-hoc polymorphism

In most of the subtyping systems discussed so far that support lattice operations (including MLsub), the following equation holds:

$$(t_1 \sqcup t_1') \to (t_2 \sqcap t_2') = (t_1 \to t_2) \sqcap (t_1' \to t_2')$$

That the type on the left is a subtype of that on the right is a consequence of the subtyping rule for function types, but there are systems where the type on the right is not a subtype of that on the left.

In a language supporting *ad-hoc polymorphism* by allowing e.g. the `typecase` construct, functions can be written that inspect the type of their argument. Such a function might check whether its argument was a $t_1$, in which case it returns a $t_2$, or a $t'_1$, in which case it returns a $t'_2$. This function would be of type $(t_1 \to t_2) \sqcap (t'_1 \to t'_2)$, but would not be of type $(t_1 \sqcup t'_1) \to (t_2 \sqcap t'_2)$ since it never returns something which is both a $t_2$ and a $t'_2$.

This makes type inference more difficult, since the compiler has fewer convenient equations to work with, but allows more precise types. A notable line of recent work along these lines is the work on *semantic subtyping* [Cas05, HP03], which defines the subtyping relation as set inclusion between denotations of types. Commonly, the denotation of a type is defined simply as the set of values having that type, which leads to a circularity: the subtyping relation is defined in terms of the set of values of a given type, while due to the $(\textsc{Sub})$ rule this set of values is defined in terms of the subtyping relation. Excising this circularity is a tricky business, done using a technical process called *bootstrapping*. Castagna and Frisch [CF05] give a good introduction to the approach.

Castagna and Xu integrated semantic subtyping with parametric polymorphism [CX11], including a careful treatment of type variables. The same authors along with Nguyễn and Abate worked on type inference [CNXA15], which was recently extended by Castagna, Petrucciani and Nguyễn [CPN16] to include typing polymorphic variants. However, the lack of a principality result for the system means that type inference may produce multiple solutions, and inferring a type for a program in general involves backtracking through them.

The set-theoretic interpretation has an implicit closed-world assumption, requiring any two empty types to be related by the subtyping relation. This gives a powerful subtyping relation, and the closed-world assumption allows reasoning with negation types. This set-theoretic reasoning perfectly fits the original application of processing XML queries [HP03], and gives expressive types to programs involving `typecase`-like constructs. However, since the closed-world assumption violates extensibility (Section 1.4), it is unclear how to reconcile this with the open-world, ML-style approach of the present work.

Another system that interprets types set-theoretically is the Coppo-Dezani-Venneri *intersection type discipline* [CDCV80], which includes the *intersection introduction* rule:

$$\frac{\Gamma \vdash e : \tau_1 \qquad \Gamma \vdash e : \tau_2}{\Gamma \vdash e : \tau_1 \cap \tau_2}$$

This rule allows two unrelated typing derivations for $\Gamma \vdash e : \tau_1$ and $\Gamma \vdash e : \tau_2$ to be combined into a single judgement. The principal types property holds: intuitively, if there are two different types for a term they can be combined with the above rule. The system also has a set-theoretic interpretation, thanks again to the intersection introduction rule, since the set of values typeable at type $\tau_1 \cap \tau_2$ is exactly the intersection of the sets of values typeable at types $\tau_1$ and $\tau_2$. The system does not have union types.

A remarkable property holds: a term of the untyped lambda calculus is typeable under the intersection type discipline if and only if it has a normal form. However, by the same result, type inference is undecidable [Roc88].

## 10.3 Simplification and entailment

Naively implementing type inference with constrained types tends to produce a large and uninformative set of constraints for any given program. In order to make them more palatable, it is necessary to *simplify* the constraints to an equivalent but shorter form.

Many useful techniques have been proposed to simplify constraints: replacing variables bound only above or only below with their bounds, collapsing cycles of subtyping constraints between variables, merging variables constrained in the same way, and removing "unreachable" constraints that do not affect the result [FA96, FF96, Pot98a, Pot01, FFSA98, AWP97]. This last technique was improved to keep track of the polarity of the constraints, checking positive and negative reachability separately, culminating in Eifrig, Smith and Trifonov's *garbage collection* algorithm [EST95a].

This work owes particular debt to Pottier's seminal thesis [Pot98b], which not only gives a thorough treatment of constrained types and their simplification, but also introduces the *mono-polarity invariant*, showing how positive and negative uses of type variables can be separated throughout inference. This invariant was the the inspiration for the biunification algorithm of Chapter 5.

### 10.3.1 Entailment

These simplification algorithms, while effective, are all heuristic in nature: they simplify a constrained type scheme to an equivalent but smaller one, but do not in general find the smallest equivalent scheme. The central issue is that deciding whether two constrained type schemes are equivalent proved problematic using non-structural types as previously proposed. In order to decide whether two constrained types are equivalent, one must decide whether one set of constraints implies another, a problem termed *entailment*. This is the constrained-types version of deciding subsumption, as studied in Chapter 8.

Most work on non-structural subtype entailment defines types along these lines:

$$\tau = \bot \mid \mathrm{bool} \mid \tau \to \tau \mid \top$$

With the usual subtyping order, these types form a lattice, but suffer the extensibility issues outlined in Section 1.4, due to the treatment of type variables as quantifying over ground types and the use of disjoint union to combine lattices (giving the set-theoretic instead of lattice-theoretic coproduct).

These issues make deciding entailment a remarkably difficult problem. While I recommend swapping the problem for an easier one (by defining types extensibly, as in Chapter 3), it is worthwhile to revisit previous work as many of the techniques developed there are directly applicable to the extensible formulation. In particular, I believe that one of Pottier's entailment algorithms [Pot98b, Fig. 8.2 on p. 79], which he shows sound-but-incomplete for the non-extensible definition of types, is in fact both sound and complete when types are defined extensibly.

Ultimately, the problem of non-structural subtype entailment for types defined non-extensibly (which I abbreviate to NSSE, following previous authors) is still open, neither solved nor shown undecidable. There are positive results in special cases: Henglein and Rehof [HR97] show that the problem is solvable in linear time if the constraints do not include type constructors (only variables), and is NP-complete if the subtyping order is structural (see Section 10.1.1). On the other hand, there are negative results in generalisations: Su et al. [SAN$^{+}$02] show that the full first-order theory of NSSE is undecidable. Explicitly, while NSSE amounts to deciding a ∀-sentence over the non-extensible definition of types and subsumption amounts to a ∀∃-sentence, Su et al. encode the Post correspondence problem in ∃∀∃∀∃∀ sentences over the same structure.

Henglein and Rehof [HR98] show that NSSE is PSPACE-hard by a reduction from NFA universality (whether an NFA accepts all strings). While the extensible formulation of types used in this thesis differs from that of NSSE, Henglein and Rehof's construction works in the current setting. Indeed, the approach in Chapter 8 to deciding $t^{-} \leqslant t^{+}$ subtyping problems (Fig. 8.3) began as an attempted converse to Henglein and Rehof's result, and the algorithm essentially checks NFA universality by converting to a DFA and checking DFA universality[3]. Rehof's thesis [Reh98] is an excellent resource for further detail on NSSE and related problems.

A line of work by Niehren and Priesnitz [NP01] culminating in Priesnitz's thesis [Pri04] also uses automata, but in a different way. They introduce *cap-set automata*, and show how NSSE can be characterised by word equations which map to cap-set automata. This approach expressed many variants of NSSE in a common framework and shed light on what made them difficult, but did not resolve decidability.

[3] Making this formal is tricky, due to the encoding of record types, but the algorithm is exactly DFA universality if no record types are used

# 11 Conclusions and future work

*I used to feel guilty in Cambridge that I spent all day playing games, while I was supposed to be doing mathematics.*

—John Horton Conway

The goal of this work was to extend the Hindley-Milner type system with subtyping, a goal shared by much previous work. However, previous attempts have always had one or more of the following caveats: lack of decidable type inference, lack of principal types, presence of explicit constraints in types, or lack of width subtyping. This work is the first attempt managing to avoid these caveats, preserving the pleasant properties of the Hindley-Milner type system, and even using the same typing rules (Chapter 4). The system has been implemented, including most of the extensions described in Chapter 9.

The primary tool by which this was achieved was not advanced or complicated algorithms, but attention to the minutiae of basic definitions. Thanks to the careful definition of polar type terms, the algorithm for biunification in Chapter 5 (and the version for automata in Chapter 7) are easy extensions of classical unification algorithms, while the algorithms for subsumption (Chapter 8) are straightforward thanks to the construction of the lattice of types in Chapter 3. By contrast, using the (admittedly shorter) definition of types used in previous work leads to a subsumption problem whose decidability has remained open for decades, and which has several counter-intuitive behaviours (see Section 1.4).

Generally, by choosing the most algebraically natural definitions rather than the most syntactically concise, the pitfalls that dogged previous attempts were avoided.

## 11.1 Future work

The classical Hindley-Milner type inference algorithm has formed the foundation of many functional programming languages, and has been extended with a huge variety of interesting language features. Indeed, one of its main strengths is how robustly it has integrated new features.

There is much future work to do in determining how smoothly these features integrate with the type system presented here, when biunificatation replaces unification and subtyping is pervasive. Chapter 9 began collecting the most low-hanging of these fruit, and some interesting next steps are explained below.

### 11.1.1 Advanced recursive types

MLsub supports recursive types though the $\mu$ operator (Section 5.1.1). Together with records and sum types (Section 9.2), this suffices to encode the basic algebraic datatypes found in all descendants of ML, such as this type of nonempty binary trees (written in Haskell's syntax):

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)
```

Among other modern languages, Haskell supports *non-regular* type definitions which cannot be expressed purely using $\mu$, such as this type of *perfect trees* (complete binary trees containing $2^k$ values):

```
data PTree a = Leaves a | Cons (PTree (a, a))
```

Above, a `PTree Int` is defined not just in terms of itself, but in terms of `PTree (Int, Int)`, `PTree ((Int,Int),(Int,Int))` and so on, a relationship which cannot be captured by the $\mu$ operator.

Yet more interesting type definitions are possible. For instance, Haskell also supports *indexed* types (or *generalised algebraic data types*), such as the following type of simple arithmetic expressions:

```
data Exp a where
    ConstI :: Int  -> Exp Int
    ConstB :: Bool -> Exp Bool
    Add    :: Exp Int -> Exp Int -> Exp Int
    Eq     :: Exp Int -> Exp Int -> Exp Bool
```

Here, the value constructors `ConstI`, `ConstB`, `Add` and `Eq` construct only the type specified on the right. For instance, `Exp Int` is inhabited by `ConstI` $n$ and `Add` $x$ $y$, but not by `ConstB` or `Eq`, while `Exp String` is not inhabited at all.

All of the recursive types definable in Haskell are *isorecursive*, rather than the *equirecursive* ones studied in this thesis (see Section 10.1.2 for more on the distinction). While extending the equirecursive $\mu$ operator to allow nonregular recursive types like `PTree` is a daunting task (and seems entirely impossible within the current formalism for recursive types), adding isorecursive nonregular types in the style of Haskell is relatively straightforward, since it amounts to adding opaque roll and unroll primitives for each type definition.

Indexed types/GADTs are another story. The type inference algorithms of this work rely on a strict separation between positive and negative types, which in turn relies on ensuring that each type parameter is either co- or contra-variant. This separation means that type inference need only consider subtyping constraints. The direction of subtyping constraints changes upon descent into contravariant positions, but it never collapses to an equality constraint. In Section 9.1.1, we see that this separation is not a limitation in normal type definitions: indeed, strictly enforcing it allows us to infer more precise types than systems that resort to invariant type parameters.

However, this trick does not work so neatly with GADTs. GADTs allow one to define type-level equality witnesses, like so:

```
data Eq a b where
  Refl :: Eq a a
```

The type $\texttt{Eq}\ \tau_1\ \tau_2$ is inhabited by Refl iff $\tau_1$ and $\tau_2$ are equal, and so typechecking necessarily involves equality constraints between types. Equality witnesses are so central to GADTs that, as Kiselyov shows [Kis10], most of the power of GADTs is available in a system with only conventional algebraic datatypes and equality witnesses as a primitive type.

Integrating the power of GADTs with the type system of this thesis is challenging, due to the surprisingly complex interactions between GADTs and subtyping. A useful survey of these subtleties was written by Scherer and Rémy [SR13].

### 11.1.2 First-class polymorphism

Like the Hindley-Milner calculus, MLsub supports polymorphic *definitions* but not polymorphic *abstractions*. That is, while a polymorphic identity function $\mathsf{id} : \forall\alpha.\alpha \to \alpha$ may be bound using let and then used at types bool → bool or int → int, there is no way to write a λ-expression abstracting over such a function and using it in multiple incompatible ways. For instance, the following program is well-typed in MLsub (and even in plain ML):

$$\begin{array}{l}\text{let } i = \lambda x.\, x \text{ in}\\ \text{if } ((i\ i)\ \texttt{true}) \text{ then } \texttt{true} \text{ else } \texttt{true}\end{array}$$

The identity function $i$ is used both of the following types:

$$\begin{array}{c}(\text{bool} \to \text{bool}) \to (\text{bool} \to \text{bool})\\ (\text{bool} \to \text{bool})\end{array}$$

These are both subsumed by the typing scheme $\alpha \to \alpha$ of $i$, so the program typechecks. However, suppose we try to abstract over $i$:

$$\lambda i.\ \text{if } ((i\ i)\ \texttt{true}) \text{ then } \texttt{true} \text{ else } \texttt{true}$$

MLsub infers a type for this (although plain ML gives an error), but the type is not particularly useful:

$$(\alpha \to ((\text{bool} \to \text{bool}) \sqcap \alpha)) \to \text{bool}$$

That is, the passed function must take an $\alpha$, and return something which is simultaneously an $\alpha$ and a bool → bool. Attempting to pass $\lambda x.x$ as the argument to this function results in a type error.

The inferred type above does not consider the possibility that $i$ may be of polymorphic type. To correctly type this abstraction, we need to be able to abstract over values of polymorphic type, passing $i$ with a type scheme $\forall\alpha.\alpha \to \alpha$ instead of a monomorphic type.

There have been many proposals for integrating *higher-rank types* with Hindley-Milner type inference, allowing functions which abstract over polymorphic arguments. The *semi-explicit polymorphism* of Garrigue and Rémy [GR97] (now used in OCaml) is a particularly simple proposal, but requires a type annotation everywhere a polymorphic type is introduced and a marker (which need not specify the type) everywhere one is eliminated. In OCaml, some ingenious syntactic punning means that these annotations are hidden in the definitions of record types, but this trick is unavailable in a language like MLsub with structural record types that need not be predeclared. Nonetheless, this proposal seems that it would integrate with MLsub with very little work, albeit at a high syntactic overhead.

At the other end of the scale is $\mathsf{ML}^\mathsf{F}$ by Le Botlan and Rémy, which requires remarkably few type annotations to type every term of System F, but complicates the type system significantly by adding two new types of quantifier and a form of subtyping. Unfortunately, it is not at all clear that $\mathsf{ML}^\mathsf{F}$ could be used in the present setting without serious work, since the subtyping order it defines is quite unrelated to that of this thesis.

Between these two extremes are a host of other proposals, varying in the annotation burden they place on the programmer, the complexity they add to the typing rules or the syntax of types, and the difficulty of implementation. Of particular note is the thread of proposals extending Haskell with higher-rank types [JVWS07, VWPJ08]. Dunfield and Krishnaswami's paper [DK13] tackling higher-rank types using bidirectional typechecking is also recommended, particularly for its excellent comparison of the various different proposals.

### 11.1.3 Module systems and higher-kinded types

As well as abstracting over polymorphic functions, languages with *higher-kinded polymorphism* allow the programmer to abstract over type constructors, allowing e.g. the same algorithm to be used with either linked lists or arrays. In the ML family, this is the role of the *module system*, which allows arbitrary parameterisation of *modules* (collections of type and data definitions) by other modules.

In the literature on module systems, there is a strong tradition of separating the *module language* from the *core language*, and placing only minimal requirements on the latter. The most important of these requirements are the ability to check subsumption (Chapter 8) and support for both opaque and transparent type definitions (Section 9.1), which are already satisfied by MLsub. So, it seems that much of the literature on module systems should apply to MLsub with little to no modification.

Some interesting recent work has focused on erasing the distinction between the core and module languages, allowing modules to be passed as values without friction. Rossberg's 1ML [Ros15] is an important contribution, unifying modules and ordinary records. However, this line of research may prove much more challenging to combine with the present work, since it does not easily fit into the lattice-based formalisation used throughout.

# Bibliography

[AB61] Smbat Abian and Arthur B Brown. A theorem on partially ordered sets with applications to fixed point theorems. *Canad. J. Math*, 13(78-83):2, 1961.

[AC93] Roberto M. Amadio and Luca Cardelli. Subtyping Recursive Types. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(4), 1993.

[AW93] Alexander Aiken and Edward L. Wimmers. Type inclusion constraints and type inference. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, FPCA '93, pages 31–41. ACM, 1993.

[AWP97] Alexander Aiken, Edward L Wimmers, and Jens Palsberg. Optimal representations of polymorphic types with subtyping. In *International Symposium on Theoretical Aspects of Computer Software*, pages 47–76. Springer, 1997.

[Bac02] Roland Backhouse. Galois connections and fixed point calculus. In *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, pages 89–150. Springer, 2002.

[Bau12] Andrej Bauer. On the failure of fixed-point theorems for chain-complete lattices in the effective topos. *Theoretical Computer Science*, 430:43–50, 2012.

[Bek84] Hans Bekič. Definable operations in general algebras, and the theory of automata and flowcharts. In C.B. Jones, editor, *Programming Languages and Their Definition*, volume 177 of *Lecture Notes in Computer Science*, pages 30–55. Springer Berlin Heidelberg, 1984.

[BH98] Michael Brandt and Fritz Henglein. Coinductive axiomatization of recursive type equality and subtyping. *Fundamenta Informaticae*, 33(4):309–338, 1998.

[BJ84] Hans Bekič and Cliff B. Jones. *Programming Languages and Their Definition: Selected Papers*, volume 177 of *LNCS*. Springer Berlin Heidelberg, 1984.

[Bof95] Maurice Boffa. Une condition impliquant toutes les identités rationnelles. *RAIRO-Theoretical Informatics and Applications-Informatique Théorique et Applications*, 29(6):515–518, 1995.

[Cas95] Giuseppe Castagna. Covariance and contravariance: conflict without a cause. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 17(3):431–447, 1995.

[Cas05] Giuseppe Castagna. Semantic subtyping: challenges, perspectives, and open problems. In *Italian Conference on Theoretical Computer Science*, pages 1–20. Springer, 2005.

[CC00] Jan Cederquist and Thierry Coquand. Entailment relations and distributive lattices. In Sam Buss, Petr Hajek, and Pavel Pudlak, editors, *Logic Colloquium '98: proceedings of the Annual European Summer Meeting of the Association for Symbolic Logic*, volume 13 of *Lecture Notes in Logic*, pages 110–123. AK Peters/Springer, 2000.

[CCH$^+$89] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C. Mitchell. F-bounded polymorphism for object-oriented programming. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, FPCA '89, pages 273–280. ACM, 1989.

[CDCV80] Mario Coppo, Mariangiola Dezani-Ciancaglini, and Betti Venneri. Principal type schemes and λ-calculus semantics. *To HB Curry: essays on combinatory logic, lambda calculus and formalism*, pages 535–560, 1980.

[CF05] Giuseppe Castagna and Alain Frisch. A gentle introduction to semantic subtyping. In *Proceedings of the 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP '05, pages 198–199. ACM, 2005.

[CG90] Pierre-Louis Curien and Giorgio Ghelli. Coherence of subsumption. In *Proceedings of the 15th Colloquium on Trees in Algebra and Programming*, pages 132–146. Springer Berlin Heidelberg, 1990.

[CMMS94] Luca Cardelli, Simone Martini, John C Mitchell, and Andre Scedrov. An extension of system F with subtyping. *Information and Computation*, 109(1):4–56, 1994.

[CNXA15] Giuseppe Castagna, Kim Nguyễn, Zhiwu Xu, and Pietro Abate. Polymorphic functions with set-theoretic types: Part 2: Local type inference and type reconstruction. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 289–302. ACM, 2015.

[Con71] John Horton Conway. *Regular Algebra and Finite Machines*. Chapman and Hall, London, 1971.

[CPN16] Giuseppe Castagna, Tommaso Petrucciani, and Kim Nguyễn. Set-theoretic types for polymorphic variants. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, pages 378–391. ACM, 2016.

[CX11] Giuseppe Castagna and Zhiwu 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.

[Dav55] Anne C. Davis. A characterization of complete lattices. *Pacific J. Math*, 5:311–319, 1955.

[DG56] J. De Groot. Non-archimedean metrics in topology. *Proceedings of the American Mathematical Society*, 7(5):948–953, 1956.

[DHP07] Brian A. Davey, Miroslav Haviar, and Hilary A. Priestley. Boolean topological distributive lattices and canonical extensions. *Applied Categorical Structures*, 15(3):225–241, 2007.

[DK13] Joshua Dunfield and Neelakantan R. Krishnaswami. Complete and easy bidirectional typechecking for higher-rank polymorphism. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 429–442. ACM, 2013.

[DM82] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '82, pages 207–212. ACM, 1982.

[DP02] Brian A. Davey and Hilary A. Priestley. *Introduction to lattices and order*. Cambridge University Press, 2002.

[Esc03] Martín H Escardó. Joins in the frame of nuclei. *Applied Categorical Structures*, 11(2):117–124, 2003.

[EST95a] Jonathan Eifrig, Scott Smith, and Valery 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.

[EST95b] Jonathan Eifrig, Scott Smith, and Valery 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.

[FA96] Manuel Fähndrich and Alex Aiken. Making set-constraint program analyses scale. Technical report, University of California at Berkeley, 1996.

[FF96] Cormac Flanagan and Matthias Felleisen. Modular and polymorphic set-based analysis: Theory and practice. Technical report, Rice University, 1996.

[FFSA98] Manuel Fähndrich, Jeffrey S. Foster, Zhendong Su, and Alexander Aiken. Partial online cycle elimination in inclusion constraint graphs. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pages 85–96. ACM, 1998.

[FM90] You-Chin Fuh and Prateek Mishra. Type inference with subtypes. *Theoretical Computer Science*, 73(2):155–175, 1990.

[Gar02] Jacques Garrigue. Simple type inference for structural polymorphism. In *The Ninth International Workshop on Foundations of Object-Oriented Languages*, 2002.

[GJ94] Mai Gehrke and Bjarni Jónsson. Bounded distributive lattices with operators. *Mathematica Japonica*, 40(2):207–215, 1994.

[GR97] Jacques Garrigue and Didier Rémy. Extending ML with semi-explicit higher-order polymorphism. In *International Symposium on Theoretical Aspects of Computer Software*, pages 20–46. Springer, 1997.

[Grä09] George Grätzer. *Lattice theory: First concepts and distributive lattices*. Courier Corporation, 2009.

[HM95] My Hoang and John 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.

[HP03] Haruo Hosoya and Benjamin C. Pierce. XDuce: A statically typed XML processing language. *ACM Trans. Internet Technol.*, 3(2):117–148, May 2003.

[HR97] Fritz Henglein and Jakob 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.

[HR98] Fritz Henglein and Jakob 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.

[INY04] Lucian Ilie, Gonzalo Navarro, and Sheng Yu. On NFA reductions. In Juhani Karhumäki, Hermann Maurer, Gheorghe Păun, and Grzegorz Rozenberg, editors, *Theory Is Forever*, volume 3113 of *Lecture Notes in Computer Science*, pages 112–124. Springer Berlin Heidelberg, 2004.

[Joh86] Peter T. Johnstone. *Stone spaces*. Cambridge University Press, 1986.

[JVWS07] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. Practical type inference for arbitrary-rank types. *Journal of Functional Programming*, 17(01):1–82, 2007.

[Kis10] Oleg Kiselyov. GADTs. `http://okmij.org/ftp/ML/first-class-modules/#GADT`, 2010.

[Koz90] Dexter Kozen. On Kleene algebras and closed semirings. In *Proceedings on Mathematical Foundations of Computer Science 1990*, MFCS '90, pages 26–47. Springer-Verlag New York, Inc., 1990.

[KPS93] Dexter Kozen, Jens Palsberg, and Michael 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.

[Kro91] Daniel Krob. Complete systems of $\mathcal{B}$-rational identities. *Theoretical Computer Science*, 89(2):207–343, 1991.

[KS12] Dexter Kozen and Alexandra Silva. Left-handed completeness. In Wolfram Kahl and Timothy G. Griffin, editors, *Proc. Conf. Relational and Algebraic Methods in Computer Science (RAMiCS 2012)*, volume 7560 of *Lecture Notes in Computer Science*, pages 162–178. Springer, 2012.

[LMM88] J-L. Lassez, M. J. Maher, and K. Marriott. Unification revisited. In *Foundations of Logic and Functional Programming*, pages 67–113. Springer, 1988.

[Mar76] George Markowsky. Chain-complete posets and directed sets with applications. *Algebra universalis*, 6(1):53–68, 1976.

[Mit91] John C Mitchell. Type inference with simple subtypes. *Journal of Functional Programming*, 1(03):245–285, 1991.

[ML78] Saunders Mac Lane. *Categories for the working mathematician*. Springer Science & Business Media, 1978.

[MM82] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(2):258–282, 1982.

[NP99] Joachim Niehren and Tim Priesnitz. Entailment of non-structural subtype constraints. In P.S. Thiagarajan and Roland Yap, editors, *Advances in Computing Science — ASIAN'99*, volume 1742 of *Lecture Notes in Computer Science*, pages 251–265. Springer Berlin Heidelberg, 1999.

[NP01] Joachim Niehren and Tim Priesnitz. Non-structural subtype entailment in automata theory. In Naoki Kobayashi and Benjamin C. Pierce, editors, *Theoretical Aspects of Computer Software*, volume 2215 of *Lecture Notes in Computer Science*, pages 360–384. Springer Berlin Heidelberg, 2001.

[Pie92] Benjamin C. Pierce. Bounded quantification is undecidable. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '92, pages 305–315. ACM, 1992.

[Pie02] Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002.

[PO95] Jens Palsberg and Patrick O'Keefe. A type system equivalent to flow analysis. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 367–378. ACM, 1995.

[Pot98a] François Pottier. A framework for type inference with subtyping. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, ICFP '98, pages 228–238. ACM, 1998.

[Pot98b] François Pottier. *Type inference in the presence of subtyping: from theory to practice*. PhD thesis, Université Paris 7, 1998.

[Pot01] François Pottier. Simplifying subtyping constraints: a theory. *Information and Computation*, 170(2):153–183, 2001.

[Pre14] Matija Pretnar. Inferring algebraic effects. *Logical Methods in Computer Science*, 10(3), 2014.

[Pri04] Tim Priesnitz. *Subtype Satisfiability and Entailment*. PhD thesis, Saarland University, 2004.

[PS96] Jens Palsberg and Scott Smith. Constrained types and their expressiveness. *ACM Trans. Program. Lang. Syst.*, 18(5):519–527, 1996.

[PT87] Robert Paige and Robert E Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989, 1987.

[Red64] Valentin N. Redko. On defining relations for the algebra of regular events. *Ukrainskii Matematicheskii Zhurnal*, 16:120–126, 1964.

[Reh98] Jakob Rehof. *The complexity of simple subtyping systems*. PhD thesis, University of Copenhagen, 1998.

[Rém94] Didier Rémy. Type inference for records in natural extension of ml. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-oriented Programming*, pages 67–95. MIT Press, 1994.

[Roc88] Simona Ronchi Della Rocca. Principal type scheme and unification for intersection type discipline. *Theoretical Computer Science*, 59(1):181–209, 1988.

[Ros15] Andreas Rossberg. 1ML — core and modules united (F-ing first-class modules). In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, pages 35–47. ACM, 2015.

[SAN+02] Zhendong Su, Alexander Aiken, Joachim Niehren, Tim Priesnitz, and Ralf 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.

[Sch12] Bernd S. W. Schröder. The fixed point property for ordered sets. *Arabian Journal of Mathematics*, 1(4):529–547, 2012.

[Sim03] Vincent Simonet. Flow Caml in a nutshell. In *Proceedings of the first APPSEM-II workshop*, pages 152–165, 2003.

[SR13] Gabriel Scherer and Didier Rémy. GADTs meet subtyping. In *Programming Languages and Systems: 22nd European Symposium on Programming, ESOP 2013*, pages 554–573. Springer Berlin Heidelberg, 2013.

[Tar55] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific journal of Mathematics*, 5(2):285–309, 1955.

[TS96] Valery Trifonov and Scott Smith. Subtyping constrained types. In Radhia Cousot and David A. Schmidt, editors, *Static Analysis*, volume 1145 of *Lecture Notes in Computer Science*, pages 349–365. Springer Berlin Heidelberg, 1996.

[VWPJ08] Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones. FPH: First-class polymorphism for Haskell. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, pages 295–306. ACM, 2008.

[WF94] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and computation*, 115(1):38–94, 1994.

[Win83] Glynn Winskel. A representation of completely distributive algebraic lattices. Technical report, Carnegie Mellon University, 1983.