The Simple Essence of Algebraic Subtyping

Principal Type Inference with Subtyping Made Easy (Functional Pearl)

LIONEL PARREAUX, EPFL, Switzerland

MLsub extends traditional Hindley-Milner type inference with subtyping while preserving compact principal types, an exciting new development. However, its specification in terms of *biunification* is difficult to understand, relying on the new concepts of bisubstitution and polar types, and making use of advanced notions from abstract algebra. In this paper, we show that these are in fact not essential to understanding the mechanisms at play in MLsub. We propose an alternative algorithm called *Simple-sub*, which can be implemented efficiently in under 500 lines of code (including parsing, simplification, and pretty-printing), looks more familiar, and is easier to understand.

We present an experimental evaluation of Simple-sub against MLsub on a million randomly-generated well-scoped expressions, showing that the two systems agree. The mutable automaton-based implementation of MLsub is quite far from its algebraic specification, leaving a lot of space for errors; in fact, our evaluation uncovered several bugs in it. We sketch more straightforward soundness and completeness arguments for Simple-sub, based on a syntactic specification of the type system.

This paper is meant to be light in formalism, rich in insights, and easy to consume for prospective designers of new type systems and programming languages. In particular, no abstract algebra is inflicted on readers.

CCS Concepts: • **Software and its engineering** → *Functional languages*; *Polymorphism.*

Additional Key Words and Phrases: type inference, subtyping, principal types

ACM Reference Format:
Lionel Parreaux. 2020. The Simple Essence of Algebraic Subtyping: Principal Type Inference with Subtyping Made Easy (Functional Pearl). *Proc. ACM Program. Lang.* 4, ICFP, Article 124 (August 2020), 29 pages. https://doi.org/10.1145/3409006

1 INTRODUCTION

The ML family of languages, which encompasses Standard ML, OCaml, and Haskell, have been designed around a powerful "global" approach to type inference, rooted in the work of Hindley [1969] and Milner [1978], later closely formalized by Damas and Milner [1982]. In this approach, the type system is designed to be simple enough that types can be unambiguously inferred from terms without the help of any type annotations. That is, for any well-typed unannotated term, it is always possible to infer a *principal type* which subsumes all other types that can be assigned to this term. For instance, the term $\lambda x.\ x$ can be assigned types `bool` → `bool` and `int` → `int`, but both of these are subsumed by the polymorphic type $\forall \alpha.\ \alpha \to \alpha$, also written 'a → 'a, which is the principal type of this term.

This "***Hindley-Milner***" (HM) type inference approach contrasts with more restricted "***local***" approaches to type inference, found in languages like Scala, C#, and Idris, which often require the types of variables to be annotated explicitly by programmers. On the flip side, abandoning the principal type property allows these type systems to be more expressive, and to support features

Author's address: Lionel Parreaux, lionel.parreaux@epfl.ch, EPFL, BC 214 (Bâtiment BC), Lausanne, CH-1015, Switzerland.

2475-1421/2020/8-ART124
https://doi.org/10.1145/3409006

like object orientation and dependent types. Even ML languages like OCaml and Haskell have adopted type system features which, when used, break the principal type property[1] and sometimes require explicit type annotations. Supporting principal types is a delicate tradeoff.

Subtyping is an expressive approach allowing types to be structured into hierarchies — usually a subtyping *lattice* — with the property that types can be *refined* or *widened* implicitly following this hierarchy. This lets one express the fact that some types are more precise (contain more information) than others, but still have a compatible runtime representation, so that no coercions between them are needed. For instance, in a system where the type '`nat`' is a subtype of '`int`', one can transparently use a '`nat list`' in place where an '`int list`' is expected, without having to apply a coercion function on all the elements of the list. Subtyping can be emulated using somewhat heavy type system machinery (which both OCaml and Haskell do, to some extent[2]), but first-class support for subtyping gives the benefit of simpler type signatures and better type inference.

While subtyping is a staple of object-oriented programming (being used to mirror class inheritance hierarchies), it is by no means limited to that paradigm, and has found vast domains of applications in functional programming too, including refinement types for ML data types [Freeman and Pfenning 1991], lightweight verification [Rondon et al. 2008; Rushby et al. 1998; Vazou et al. 2014], full dependent types [Hutchins 2010], first-class modules [Amin et al. 2016; Rossberg 2015], polymorphic variants [Castagna et al. 2016], XML transformations [Hosoya et al. 2005], and higher-rank polymorphism [Dunfield and Krishnaswami 2013; Odersky and Läufer 1996].

For a long time, it was widely believed that implicit subtyping got in the way of satisfactory global type inference. Indeed, previous approaches to inferring subtypes failed to support principal types, or resulted in the inference of large types containing sets of unwieldy constraints, making them difficult to understand by programmers.

MLsub was introduced by Dolan and Mycroft [2017] as an ML-style type system supporting subtyping, polymorphism, and global type inference, while still producing compact principal types. Here, *compact* refers to the fact that the inferred types are relatively simple type expressions without any visible constraints, making them easy to read and understand. This was achieved by carefully designing the semantics of the subtyping lattice using an *algebra-first* approach, also referred to as ***algebraic subtyping*** [Dolan 2017].

However, the specification of MLsub's type inference algorithm as *biunification* is difficult to understand for experts and non-experts alike. On the surface, it looks more complicated than the *algorithm W* traditionally used for HM type systems, requiring additional concepts such as bisubstitution, polar types, and advanced notions from abstract algebra. Although its elegant presentation will appeal to mathematically-minded researchers, experience has shown that grasping an understanding of the approach complete enough to reimplement the algorithm required reading Dolan's thesis in full, and sometimes more [Courant 2018].

Thankfully, it turns out that the essence of algebraic subtyping can be captured by a much simpler algorithm, ***Simple-sub***, which is also more efficient than biunification (or at least, than the basic syntax-driven form of biunification used as a specification for MLsub[3]). In this paper, we show that inferring MLsub types is surprisingly easy, and can be done in under 300 lines of Scala code,

[1] OCaml has been quite conservative with such principality-breaking features, notable exceptions being GADTs and overloaded record fields, but Haskell has been adopting them more liberally in order to increase expressiveness.

[2] For instance, OCaml uses row types to make object types and polymorphic variants more flexible, avoiding the need for explicit coercions, somewhat similarly to implicit subtyping [Pottier 1998, Chapter 14.7]; and Haskell can use type classes to emulate subtyping, as exemplified by the *lens* and *optics* libraries, which are both designed around a subtyping analogy.

[3] Operationally speaking, Simple-sub has many similarities to the graph-based implementation of MLsub, though the two algorithms are still quite different — in particular, MLsub separates positive nodes from negative nodes in its constraint graph (while there is no such separation in Simple-sub), and MLsub generates many more type variables than Simple-sub.

with an additional 200 lines of code for simplification and pretty-printing. Simple-sub is available online at https://github.com/LPTK/simple-sub. While the implementation we present is written in Scala, it is straightforward to translate into any other functional programming languages.

Our main contribution is to recast MLsub into a simpler mold and to disentangle its algebraic construction of types from the properties actually needed by the type inference and type simplification processes. We believe this will allow future designers of type systems and programming languages to benefit from some of the great insights of the approach, without having to delve too deeply into the theory of algebraic subtyping. In the rest of this paper, we:

- present MLsub and the *algebraic subtyping* discipline on which it is based (Section 2);
- describe our new Simple-sub type inference algorithm (Section 3);
- explain how to perform basic simplification on the types inferred by Simple-sub (Section 4);
- sketch the correctness proofs of Simple-sub through *soundness* and *completeness* theorems, formalizing the minimal subtyping relation needed to carry out these proofs. (Section 5);
- describe our experimental evaluation: we verified that Simple-sub and MLsub agreed on the results of type inference for over a million automatically-generated programs (Section 6).

2 ALGEBRAIC SUBTYPING AND MLsub

Let us first describe MLsub and the algebraic subtyping philosophy that underlies its design.

2.1 Background on Algebraic Subtyping

There are at least three major schools of thought on formalizing subtyping. *Syntactic approaches*, as in this paper, use direct specifications (usually given as inference rules) for the subtyping relationship, closely following the syntax of types. *Semantic approaches* [Frisch et al. 2008] view types as sets of values which inhabit them, and define the subtyping relationship as set inclusion between these sets. *Algebraic approaches* [Dolan 2017] define types as abstract elements of a distributive lattice, whose algebraic properties are carefully chosen to yield good properties, such as "extensibility" and principal types.

Syntactic approaches somehow pay for their simplicity by forcing type system designers to consider the consequences and interactions of all their inference rules, having to manually verify that they result in a subtyping relationship with the desired algebraic properties.

The semantic approach is probably the most intuitive, and is also very powerful; however, it suffers from difficulties related to polymorphism — an understanding of type variables as *ranging* over ground types can lead to paradoxes and a lack of extensibility [Dolan 2017].

As a response to these perceived shortcomings, Dolan argues that *algebra* (not *syntax*) should come first, in order to guarantee from the start that type systems are well-behaved, as opposed to ensuring it as a sort of afterthought. In particular, he emphasizes the concept of *extensibility* of type systems, the idea being that existing programs should remain well-typed when new type forms are added. While the practical usefulness of this notion of extensibility is unclear,[4] the general approach of making the subtyping lattice *distributive* does helps to simplify types aggressively and to construct algorithms for checking subsumption effectively (i.e., checking whether one type signature is as general as another).

In the rest of this section, we explain the basics of MLsub and its static semantics, to set the stage for the presentation of Simple-sub in Section 3.

[4]We have some doubts that the extensibility property of algebraic subtyping is as useful as advertised by its author, besides its simplifying consequences. First, practical programming languages already have extensible type systems by design, since they allow user-defined data types, and the subtyping paradoxes which arise from closed-world formal calculi do not typically arise or cause troubles in these settings. Second, extending the core type semantics of a programming language happens exceedingly rarely, and thus is probably not a scenario worth optimizing for, when designing a type system.

2.2 Basics of MLsub

2.2.1 Term Language. The term syntax of MLsub is given in Figure 1. We make some simplifications compared to the original MLsub presentation: first, we omit boolean literals and if-then-else, as they can easily be typed as primitive combinators; second, we use only one form of variables x (MLsub distinguished between lambda-bound and let-bound variables, for technical reasons).

$$t \;::=\; x \mid \lambda x.\, t \mid t\ t \mid \{\, l_0 = t\,;\ \ldots\,;\ l_n = t\,\} \mid t.l \mid \text{let rec } x = t \text{ in } t$$

Fig. 1. Syntax of MLsub terms, for which we want to infer types.

2.2.2 Type Language. The type syntax of MLsub, summarized in Figure 2, consists in primitive types (such as int and bool) function types, record types, type variables, top $\top$ (the type of all values — supertype of all types), bottom $\bot$ (the type of no values — subtype of all types), type union $\sqcup$, type intersection $\sqcap$, and recursive types $\mu\alpha.\,\tau$.

$$\tau \;::=\; \text{primitive} \mid \tau \to \tau \mid \{\, l_0 : \tau\,;\ \ldots\,;\ l_n : \tau\,\} \mid \alpha \mid \top \mid \bot \mid \tau \sqcup \tau \mid \tau \sqcap \tau \mid \mu\alpha.\,\tau$$

Fig. 2. Syntax of MLsub types.

2.2.3 Type System. The declarative type system of MLsub is given in Figure 3. It is mostly as presented by Dolan and Mycroft [2017]. We support recursive let bindings explicitly for clarity, though recursion could also be factored out into a combinator [Damas and Milner 1982]. We write $\overline{E}^{i}$ for a repetition of elements E indexed by i. Of particular interest are the rules T-Sub, which takes a term from a subtype to a supertype implicitly (without a term-level coercion), T-Let, which types x in its recursive right-hand side in a monomorphic way, but types x in its body polymorphically, and T-Var, which instantiates polymorphic types using the substitution syntax $[\tau_0/\alpha_0]\tau$.

We appeal to the reader's intuition and leave the precise definition of subtyping unspecified for now, as it requires some technicalities around recursive types. In contrast to MLsub, which gives an algebraic account of subtyping, we will present a syntactic subtyping system in Section 5.1.

$$\text{T-Lit}\ \frac{}{\Gamma \vdash n : \text{int}} \qquad \text{T-Var}\ \frac{\Gamma(x) = \overline{\forall\alpha_i.}^{\,i}\ \tau}{\Gamma \vdash x : \overline{[\tau_i/\alpha_i]}^{\,i}\,\tau} \qquad \text{T-Abs}\ \frac{\Gamma,\, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda x.\, t : \tau_1 \to \tau_2} \qquad \text{T-App}\ \frac{\Gamma \vdash t_0 : \tau_1 \to \tau_2 \quad \Gamma \vdash t_1 : \tau_1}{\Gamma \vdash t_0\ t_1 : \tau_2}$$

$$\text{T-Rcd}\ \frac{\overline{\Gamma \vdash t_i : \tau_i}^{\,i}}{\Gamma \vdash \{\, \overline{l_i = t_i}^{\,i}\,\} : \{\, \overline{l_i : \tau_i}^{\,i}\,\}} \qquad \text{T-Proj}\ \frac{\Gamma \vdash t : \{\, l : \tau\,\}}{\Gamma \vdash t.l : \tau} \qquad \text{T-Sub}\ \frac{\Gamma \vdash t : \tau_1 \quad \tau_1 \le \tau_2}{\Gamma \vdash t : \tau_2}$$

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

$$\text{succ} : \text{int} \to \text{int} \quad \text{iszero} : \text{int} \to \text{bool} \quad \text{true} : \text{bool} \quad \text{false} : \text{bool} \quad \text{if} : \forall\alpha.\ \text{bool} \to \alpha \to \alpha \to \alpha$$

Fig. 3. Declarative typing rules of MLsub (and Simple-sub).

2.3 Informal Semantics of Types

While most MLsub type forms are usual and unsurprising, two kinds of types require our special attention: set-theoretic types (especially unions and intersections), and recursive types.

2.3.1 Set-Theoretic Types. To a first approximation, union and intersection types can be understood in set-theoretic terms: $\tau_0 \sqcup \tau_1$ (resp. $\tau_0 \sqcap \tau_1$) represents the type of values that are *either* (resp. *both*) of type τ_0 *or* (resp. *and*) of type τ_1.

MLsub uses these types to indirectly constrain type variables: When a type variable α is supposed to be a *subtype* of some type τ (i.e., values of type α may be used at type τ), MLsub substitutes all occurrences of α in input position with $\alpha \sqcap \tau$, making sure that any arguments passed in as α values are also τ values. Similarly, when α is supposed to be a *supertype* of some τ (i.e., values of type τ may be used at type α), MLsub substitutes all occurrences of α in output position with $\alpha \sqcup \tau$.

As an example, one type inferred for the term $\lambda x.\ \{\ \mathsf{L} = x - 1\ ;\ \mathsf{R} = x\ \}$ could be $\alpha \sqcap \mathsf{int} \to \{\ \mathsf{L} : \mathsf{int}\ ;\ \mathsf{R} : \alpha\ \}$. This type reflects the fact that the original argument, of some type α, is returned in the R field of the result record (as the input of the function ends up in that position), but also that this argument should be able to be treated as an int, expressed via the type intersection $\alpha \sqcap \mathsf{int}$ on the left-hand side of the function type. Keeping track of the precise argument type α is important: it could be later substituted with a more specific type than int, such as $\alpha = \mathsf{nat}$, which would give us $\mathsf{nat} \to \{\ \mathsf{L} : \mathsf{int}\ ;\ \mathsf{R} : \mathsf{nat}\ \}$. On the other hand, there may be type signatures where α becomes undistinguishable from int. For instance, consider the term '$\lambda x.$ if true then $x - 1$ else x', whose simplified inferred type would be just $\mathsf{int} \to \mathsf{int}$, as the seemingly-more precise type $\alpha \sqcap \mathsf{int} \to \alpha \sqcup \mathsf{int}$ does not actually contain more information (we expand on this in Section 4.3.1).

The beauty of MLsub is that this sort of reasoning scales to arbitrary flows of variables and higher-order functions; for instance, the previous example can be generalized by passing in a function f to stand for the $\cdot - 1$ operation, as in $\lambda f.\ \lambda x.\ \{\ \mathsf{L} = f\ x\ ;\ \mathsf{R} = x\ \}$ whose type can be inferred as $(\beta \to \gamma) \to \alpha \sqcap \beta \to \{\ \mathsf{L} : \gamma\ ;\ \mathsf{R} : \alpha\ \}$. Applying this function to argument $(\lambda x.\ x - 1)$ yields the same type (after simplification) as in the example of the previous paragraph.

2.3.2 Recursive Types. A recursive type $\mu\alpha.\ \tau$ represents a type we can unroll as many times as we want; for instance, $\mu\alpha.\ (\top \to \alpha)$, which we write just $\mu\alpha.\ \top \to \alpha$, is equivalent to $\top \to \mu\alpha.\ \top \to \alpha$, which is equivalent to $\top \to \top \to \mu\alpha.\ \top \to \alpha$, etc., and is the type of a function that can be applied to any arguments (any subtypes of $\top$) indefinitely. A recursive type is conceptually infinite — if we unrolled the above fully, it would unfold as an infinitely-deep tree $\top \to \top \to \top \to \ldots$.

If this sounds confusing, that's perfectly fine. We will get some deeper intuition on recursive types and *why* we need them in Section 3.4.1, and we will give them a formal treatment in Section 5.1. The high-level idea is that recursive types are sometimes necessary to give principal types to MLsub terms. The example given by Dolan [2017] is that of the term $Y(\lambda f.\ \lambda x.\ f)$ where Y is the call-by-value Y combinator. This term represents a function which ignores its parameter and returns itself. It can be given type $\top \to \top$ as well as $\top \to \top \to \top$, and $\top \to \top \to \top \to \top$, etc. Its principal type is the recursive type shown in the previous paragraph.

2.3.3 Typing Surprises. It is worth noting that inferring recursive types can lead to typing terms which *appear* ill-typed, and would in fact not be well-typed in ML.[5] This kind of surprises can also arise simply due to subtyping. For instance, in MLsub, the strange-looking term $\lambda x.\ x\ x$, which takes a function in parameter and applies it to itself, can be typed as $\forall \alpha, \beta.\ (\alpha \to \beta) \sqcap \alpha \to \beta$.

[5]Interestingly, the OCaml compiler supports recursive types, but it only allows them as part of object types by default, because they can otherwise lead to surprises — in some cases inferring recursive types instead of reporting obvious errors. In a practical language based on MLsub, it would be possible to have similar restrictions on the inference of recursive types.

Indeed, if the input x passed to the function has type $(\alpha \to \beta) \sqcap \alpha$, that means it has type $\alpha \to \beta$ (a function taking an α argument) *and* type α, meaning that it can be passed as an argument to itself.

2.4 Expressiveness

There is an important caveat to add to the definition of types we gave above: these types cannot actually be used freely within type expressions. The true syntax of MLsub is segregated between *positive* and *negative* types. In particular, unions are positive types (and may not appear in negative position) and intersections are negative types (and may not appear in positive position).

2.4.1 Polarity of Type Positions. Positive positions correspond to the types that a term *outputs*, while negative positions correspond to the types that a term *takes in* as input. For instance, in $(\tau_0 \to \tau_1) \to \tau_2$, type τ_2 is in positive position since it is the output of the main function, and the function type $(\tau_0 \to \tau_1)$ is in negative position, as it is taken as an input to the main function. On the other hand, τ_1, which is returned *by the function taken as input* is in negative position (since it is provided by callers via the argument function), and τ_0 is in positive position (since it is provided by the main function when calling the argument function).

2.4.2 Consequence of the Polarity Restriction. These *polarity* restrictions mean that the full syntax of types we saw above cannot actually be used as is; programmers cannot write, in their own type annotations, types that violate the polarity distinction. In fact, in MLsub, one *cannot* express the type of a function which takes "either an integer or a string": this type would have been $\mathsf{int} \sqcup \mathsf{string} \to \tau$, but $\mathsf{int} \sqcup \mathsf{string}$ is illegal in negative position. On the other hand, MLsub may assign the legal type $\tau \to \mathsf{int} \sqcup \mathsf{string}$ to functions which may return either an integer or a string... which is not a very useful type, since we cannot do anything useful with a $\mathsf{int} \sqcup \mathsf{string}$ value in MLsub.

What this all comes down to, perhaps surprisingly, is that the MLsub language is fundamentally no more expressive than a structurally-typed Java! To understand this, recall that Java allows users to quantify types using type variables, and also allows bounding these type variables with subtypes and supertypes.[6] The insight is that unions and intersections, when used at the appropriate polarity, are only a way of indirectly bounding type variables. For instance, the MLsub type:

$$\alpha \sqcap \mathsf{nat} \sqcap \mathsf{odd} \to \{\ \mathsf{L} : \mathsf{int}\ ;\ \mathsf{R} : \alpha \sqcup \mathsf{prime}\ \}$$

is equivalent to the Java-esque type (where type parameters are written between $\langle\cdot\rangle$):

$$\langle \alpha\ \ \textbf{super}\ \ \mathsf{prime}\ \ \textbf{extends}\ \ \mathsf{nat}\ \&\ \mathsf{odd} \rangle (\alpha) \to \{\ \mathsf{L} : \mathsf{int}\ ;\ \mathsf{R} : \alpha\ \}$$

meaning that α should be a *supertype* of prime and also a *subtype* of both nat and odd.

Moreover, MLsub's recursive types are expressible via F-bounded polymorphism, which Java also supports, allowing one to bound a type variable with a type expression that contains occurrences of the type variable itself.

2.5 Essence of MLsub Type Inference

Reading Dolan [2017]; Dolan and Mycroft [2017], one could be led to think that MLsub is all about:

- supporting unions and intersections in the type language, forming a distributive lattice;
- a new algorithm called biunification as an alternative to traditional unification;
- separating the syntax of types between positive and negative types.

However, we argue that this is not the most helpful way of understanding the processes at work in this type inference algorithm; instead, we argue that the essence of the approach is:

[6] For more details on these Java features, see https://docs.oracle.com/javase/tutorial/java/generics/bounded.html and https://docs.oracle.com/javase/tutorial/java/generics/lowerBounded.html.

```
enum Term {
  case Lit (value: Int)                          // as in: 27
  case Var (name: String)                        // as in: x
  case Lam (name: String, rhs: Term)             // as in: λx. t
  case App (lhs: Term, rhs: Term)                // as in: s t
  case Rcd (fields: List[(String, Term)])        // as in: { a : 0; b : true; ... }
  case Sel (receiver: Term, fieldName: String)   // as in: t.a
  case Let (isRec: Boolean, name: String, rhs: Term, body: Term)  }
```

Fig. 4. Scala syntax for MLsub terms (using the *enum*[8] keyword for defining algebraic data types).

- making the semantics of types simple enough that all inferred subtyping constraints can be reduced to constraints on type variables, which can be recorded efficiently (for instance using mutation, as done in this paper and in MLsub's actual implementation);
- using intersection, union, and recursive types to express compact type signatures where type variables are indirectly constrained, avoiding the need for separate constraint specifications.

3 THE SIMPLE-SUB TYPE INFERENCE ALGORITHM

We now present Simple-sub. We start with the internal Scala syntax used in the algorithms (Section 3.1); we then describe the basic mechanisms of type inference, at first omitting let bindings for simplicity (Section 3.2); we show how to produce user-facing type representations from the results of type inference (Section 3.3); we discuss some insights on recursive types and unroll an example of type inference (Section 3.4); and we explain how to support let polymorphism and recursive let bindings (Section 3.5). Finally, we summarize the full Simple-sub algorithm (Section 3.6).

3.1 Simple-sub Syntax

3.1.1 Term Syntax. The Scala implementation of the term syntax is shown in Figure 4. It is derived directly from Figure 1, except that we add a construct for integer literals.

As mentioned before, there is no need for an if-then-else feature, since we can just add one as a combinator: our parser actually parses code of the form "if e_0 then e_1 else e_2" as if it were written "if e_0 e_1 e_2," and we perform type checking starting from a context which assigns to the 'if' identifier the type $\forall\alpha.\ \text{bool} \to \alpha \to \alpha \to \alpha$.[7]

```
enum SimpleType {
  case Variable  (st: VariableState)
  case Primitive (name: String)
  case Function  (lhs: SimpleType, rhs: SimpleType)
  case Record    (fields: List[(String, SimpleType)])  }
```

3.1.2 Type Syntax. We start from the realization that union, intersection, top, bottom, and recursive types are all not really core to the type inference approach. Therefore, we focus on type variables,

[7] As explained by Dolan and Mycroft [2017], this type is just as general as the more natural $\forall\alpha, \beta.\ \text{bool} \to \alpha \to \beta \to \alpha \sqcup \beta$.

basic type constructors (primitive types), function types, and record types as the cornerstone of the algorithm. Their Scala syntax is shown above.

The state of a type variable is simply given by all the bounds that are recorded for it:

```
class VariableState(var lowerBounds: List[SimpleType],
                    var upperBounds: List[SimpleType])
```

Notice that we use mutable variables (`var` instead of `val`) in order to hold the current state of the algorithm — these lists will be mutated as the algorithm proceeds.

All we need to do in order to perform type inference now is to find all subtyping constraints entailed by a given program, and to propagate these constraints until they reach type variables, which we can constrain by mutating their recorded bounds.

3.2 Basic Type Inference

3.2.1 Typing. The core function for type inference is `typeTerm`, which assigns a type to a given term in some context of type `Ctx`; it is complemented by a `constrain` function to imperatively constrain one type to be a subtype of another, raising an error if that is not possible:

```
type Ctx = Map[String, SimpleType]

def typeTerm(term: Term)(implicit ctx: Ctx): SimpleType = ...

def constrain(lhs: SimpleType, rhs: SimpleType): Unit = ...
```

Above, we made the `ctx` parameter in `typeTerm` implicit so it does not have to be passed explicitly into each recursive call if it does not change.

We make use of two small helper functions, `freshVar` and `err`, to generate new type variables and raise errors, respectively:

```
def freshVar: Variable =
  Variable(new VariableState(Nil, Nil))

def err(msg: String): Nothing =
  throw new Exception("type error: " + msg)
```

Remember that `VariableState` is a `class` and not a `case class` (also called *data class*). This means that each `VariableState` instance, created with `new`, is unique and distinct from other instances.

Now that we have established the necessary premises, we can start writing the core of the basic type inference algorithm:

```
def typeTerm(term: Term)(implicit ctx: Ctx): SimpleType = term match {

  // integer literals:
  case Lit(n) => Primitive("int")
```

Below, the `ctx.getOrElse(k, t)` function is used to access the `ctx` map at a given key `k`, specifying a thunk `t` to execute in case that key is not found:

```
  // variable references:
  case Var(name) => ctx.getOrElse(name, err("not found: " + name))

  // record creations:
  case Rcd(fs) => Record(fs.map { case (n, t) => (n, typeTerm(t)) })
```

In order to type a lambda abstraction, we create a fresh variable to represent the parameter type, and we type the body of the lambda in the current context extended with a binding for this parameter, where `name -> param` is another syntax for the pair `(name, param)`:

```
  // lambda abstractions:
  case Lam(name, body) =>
    val param = freshVar
    Function(param, typeTerm(body)(ctx + (name -> param)))
```

To type applications, we similarly introduce a fresh variable standing for the result type of the function that we are applying:

```
  // applications:
  case App(f, a) =>
    val res = freshVar
    constrain(typeTerm(f), Function(typeTerm(a), res))
    res
```

Finally, record accesses result in a constraint that the receiver on the left-hand side of the selection is a record type with the appropriate field name:

```
  // record field selections:
  case Sel(obj, name) =>
    val res = freshVar
    constrain(typeTerm(obj), Record((name -> res) :: Nil))
```

```
    res
}
```

As one can observe, the basic MLsub type inference algorithm so far is quite similar to the traditional algorithm W for HM-style type inference.

3.2.2 Constraining. The first thing to diverge from algorithm W is the handling of constraints.

First, note that type variable bounds, which are updated using mutation, may very well begin to form cycles as type inference progresses. We have to account for this by using a cache parameter (initially empty) which remembers all the type comparisons that have been or are being made. This not only prevents us from falling into infinite recursion, but also avoids repeating identical work (i.e., solving some of the sub-constraints more than once), which is important to avoid making the algorithm exponential in complexity [Pierce 2002, Chapter 21.10].

```
def constrain(lhs: SimpleType, rhs: SimpleType)
      (implicit cache: MutSet[(SimpleType, SimpleType)]): Unit = {
  if (cache.contains(lhs -> rhs)) return () else cache += lhs -> rhs
```

The next step is to match each possible pair of basic types which can be constrained successfully, and propagate the constraints accordingly:

```
  (lhs, rhs) match {

    case (Primitive(n0), Primitive(n1)) if n0 == n1 =>
      () // nothing to do

    case (Function(l0, r0), Function(l1, r1)) =>
      constrain(l1, l0); constrain(r0, r1)

    case (Record(fs0), Record(fs1)) =>
      fs1.foreach { case (n1, t1) =>
        fs0.find(_._1 == n1) match {
          case None => err("missing field: " + n1 + " in " + lhs)
          case Some((_, t0)) => constrain(t0, t1) }}
```

Function types are constrained according to the usual rules of contra- and co-variance of parameter and result types (respectively), and record types according to the usual width and depth subtyping.

The case for type variables appearing on the left- or right-hand side is interesting, as this is when we finally mutate the bounds of variables. **Crucially**, after adding the corresponding upper or lower bound to the variable state, we need to iterate[9] over the *existing* opposite bounds of the variable being constrained, in order to make sure that they *become consistent* with the new bound:

[9]Scala syntax `(foo(a, _, c))` is a shorthand for the lambda abstraction `(x => foo(a, x, c))`.

```
    case (Variable(lhs), rhs) =>
      lhs.upperBounds = rhs :: lhs.upperBounds
      lhs.lowerBounds.foreach(constrain(_, rhs))

    case (lhs, Variable(rhs)) =>
      rhs.lowerBounds = lhs :: rhs.lowerBounds
      rhs.upperBounds.foreach(constrain(lhs, _))
```

Note that there is something deeply non-trivial happening here: we install the new upper bound `rhs` *before* recursing into the `lhs.lowerBounds`. This turns out to be essential — without it, recursive constraining calls which would get back to `lhs` would miss this new upper bound, failing to constrain it. Another subtlety is that *new bounds* may very well appear in `lhs.upperBounds` and `lhs.lowerBounds` while we are recursing. This subtlety is briefly discussed further in Section 5.3.1.

Finally, if all other options have failed, we report an error that the two types cannot be constrained:

```
    case _ => err("cannot constrain " + lhs + " <: " + rhs)
  }
}
```

In this subsection, we saw the core of the Simple-sub type inference algorithm, which distills what we argue is the "simple essence" of Dolan's type inference approach. However, we are not quite done yet. We still need to produce user-readable type representations (next section) and to support polymorphism and recursion through let bindings (Section 3.5).

3.3 User-Facing Types Representations

Where did union, intersection, top, bottom, and recursive types go? It turns out these are not really core to the type inference approach, and are more like emergent properties of type pretty-printing and simplification. They only come up once we try to display friendly type expressions to users, after type inference has done the gist of its job.

3.3.1 Constraining Type Variables Indirectly. Remember that we have been constraining type variables, but that type variable constraints are not part of the syntax that MLsub and Simple-sub are supposed to output. The "trick" is to indirectly encode these constraints through the use of union and intersection types (recall the examples given in Section 2.3.1).

3.3.2 Targeted Type Syntax. In order to produce user-friendly type representations in the tradition of MLsub, we target the type syntax tree presented in Figure 5.

3.3.3 Type Coalescing Algorithm. In order to produce immutable `Type` values from our inferred `SimpleType` internal representation, we need to go through a process we refer to as *type coalescing*, whose goal is to replace the positive occurrences of type variables with a union of their lower bounds, and their negative occurrences with an intersection of their upper bounds.

We define a `PolarVariable` type synonym which associates a type variable state with a polarity. The algorithm starts by initializing an empty mutable map called `recursive`, whose goal is to remember which type variables refer to themselves through their bounds, assigning them a fresh type variable which will be used when constructing the corresponding `RecursiveType` value:

```
enum Type {
  case Top
  case Bot
  case Union (lhs: Type, rhs: Type)
  case Inter (lhs: Type, rhs: Type)
  case FunctionType  (lhs: Type, rhs: Type)
  case RecordType    (fields: List[(String, Type)])
  case RecursiveType (name: String, body: Type)
  case TypeVariable  (name: String)
  case PrimitiveType (name: String)  }
```

Fig. 5. The type syntax targeted as the end result of type inference.

```
type PolarVariable = (VariableState, Boolean) // 'true' means 'positive'

def coalesceType(ty: SimpleType): Type = {
  val recursive: MutMap[PolarVariable, TypeVariable] = MutMap.empty
```

The `(VariableState, Boolean)` keys of the `recursive` map include the polarity in the right-hand side to make sure we produce only *polar* recursive types (those whose variables occur with the same polarity as the types themselves); this turns out to be necessary for principality [Dolan 2017].

Then, we define a worker function `go` which calls itself recursively in a straightforward manner, but makes sure to keep track of the type variables that are currently being coalesced, and to keep track of the current polarity — whether we are coalescing a positive (`polar == true`) or negative (`polar == false`) type position:

```
  def go(ty: SimpleType, polar: Boolean)(inProcess: Set[PolarVariable]): Type
       = ty match {

    case Primitive(n) => PrimitiveType(n)

    case Function(l, r) =>
      FunctionType(go(l, !polar)(inProcess), go(r, polar)(inProcess))

    case Record(fs) =>
      RecordType(fs.map(nt => nt._1 -> go(nt._2, polar)(inProcess)))
```

The interesting case is the following.[10] In the code below, `vs.uniqueName`, is an attribute defined in the `VariableState` class, which holds a name unique to `vs`.

```
    case Variable(vs) =>
      val vs_pol = vs -> polar
      if (inProcess.contains(vs_pol))
        recursive.getOrElseUpdate(vs_pol, TypeVariable(freshVar.uniqueName))
      else {
        val bounds = if (polar) vs.lowerBounds else vs.upperBounds
        val boundTypes = bounds.map(go(_, polar)(inProcess + vs_pol))
        val mrg = if (polar) Union else Inter
        val res = boundTypes.foldLeft(TypeVariable(vs.uniqueName))(mrg)
        recursive.get(vs_pol).fold(res)(RecursiveType(_, res))
      }
```

We first check whether the variable is already being coalesced. If it is, we look up the 'recursive' map: if this map already contains an entry for the variable, we simply return it; otherwise, we create a new fresh `TypeVariable` and update the map using `getOrElseUpdate`.

If we are not coalescing a recursive variable occurrence, we look into the bounds of the variable. Depending on the current polarity, we recurse into the lower or upper bounds. Then, if the `recursive` map now contains an entry for the variable, it means the variable was recursive. In this case, we wrap the result in `RecursiveType` with the variable found in the map.

We conclude the algorithm by calling `go` on the top-level type `ty` with an empty `inProcess`:

```
  }

  go(ty, true)(Set.empty)
}
```

This algorithm does produces some unnecessary variables (variables which could be removed to simplify the type expression); we see how to simplify type representations in Section 4.

3.4 Examples

Now is a good time to pause and exemplify some crucial aspects of Simple-sub.

3.4.1 Recursive Types. The reason for having recursive types in the user-facing type syntax has now become quite obvious: we need them in order to "tie the knot" when we are trying to coalesce type variables which appears in the coalescence of their own bounds.

For instance, consider the possible inferred representation `Function(Variable(s), Variable(t))`, where `s = new VariableState(Nil, Nil)` and `t = new VariableState(Nil, Function(Variable(s),`

[10] In Scala, `opt.fold(t)(f)` folds the `opt` option by applying a function `f` on the contained value, or by evaluating a default thunk `t` if the option is empty. The `map.getOrElseUpdate(k, t)` method of `MutMap` looks up a key `k` in `map`; if the key is not found, it evaluates the thunk `t` and update `map` with the value; otherwise, it returns the value associated with the key.

`Variable(t)) :: Nil`). Notice that there is a cycle in the upper bounds of `t`; therefore, the coalescing algorithm turns this `SimpleType` representation into the user-facing type `Function(Top, Recursive("t", Function(Top, TypeVariable("t"))))`, which corresponds to $\top \to (\mu\alpha.\top \to \alpha)$.

3.4.2 Example of Type Inference. To facilitate our understanding of the typing and coalescing algorithms, we now unroll the execution of a type inference run. Consider the term twice $= \lambda f.\ \lambda x.\ f\ (f\ x)$, which takes a function f and some x in parameter, and applies f twice on x.

```
typeTerm(λf. λx. f (f x))(empty)
| typeTerm(λx. f (f x))(Map(f ↦ α)) // α fresh
| | typeTerm(f (f x))(Map(f ↦ α, x ↦ β)) // β fresh
| | | typeTerm(f)(Map(f ↦ α, x ↦ β)) = α
| | | typeTerm(f x)(Map(f ↦ α, x ↦ β))
| | | | typeTerm(f)(Map(f ↦ α, x ↦ β)) = α
| | | | typeTerm(x)(Map(f ↦ α, x ↦ β)) = β
| | | | constrain(α, Function(β, γ)) // γ fresh
| | | | | α.upperBounds = Function(β, γ) :: α.upperBounds
| | | = γ
| | | constrain(α, Function(γ, δ)) // δ fresh
| | | | α.upperBounds = Function(γ, δ) :: α.upperBounds
| | = δ
| = Function(β, δ)
= Function(α, Function(β, δ))
```

After this process, we end up with two upper bounds on α, namely `Function(`β`, `γ`)` and `Function(`γ`, `δ`)`. We next see how the type coalescing algorithm unrolls from this inferred `SimpleType` representation. We omit the details of some of the less interesting sub-executions, and by a slight abuse of notation we use α to denote α`.uniqueName`:

```
coalesceType(Function(α, Function(β, δ)))
| go(Function(α, Function(β, δ)), true)(empty)
| | go(α, false)(empty)
| | | val bounds = Function(β, γ) :: Function(γ, δ) :: Nil
| | | val boundTypes
| | | | go(Function(β, γ), false)(Set(α ↦true)) = β → γ
| | | | go(Function(γ, δ), false)(Set(α ↦true)) = γ → δ
| | | = β → γ :: γ → δ :: Nil
| | = α ⊓ (β → γ) ⊓ (γ → δ)
| | go(Function(β, δ), true)(empty)
| | | go(β, false)(empty) = β
| | | go(δ, true)(empty) = δ
| | = β → δ
| = α ⊓ (β → γ) ⊓ (γ → δ) → β → δ
= α ⊓ (β → γ) ⊓ (γ → δ) → β → δ
```

Finally, we will see in Section 4 that this type can be compacted to $\alpha \sqcap (\beta \sqcup \gamma \to \gamma \sqcap \delta) \to \beta \to \delta$, and then simplified to $(\beta \sqcup \gamma \to \gamma) \to \beta \to \gamma$, since α occurs only negatively (thus can be removed) and δ and γ co-occur negatively (thus can be merged into a single variable).

3.5 Let Polymorphism and Recursion

3.5.1 Let Polymorphism. In traditional ML languages, local let bindings may be assigned polymorphic types. This requires keeping track of generalized *typing schemes* which are to be *instantiated* with fresh variables on every use, and making sure that we are not generalizing those type variables which occur in the environment, which would be unsound.

One way of determining which type variables to generalize is to scan the current environment, looking for references to the type variables in question. However, that is quite inefficient (it adds a linear-time operation in an important part of the algorithm).

Efficient generalization in ML. A better approach is to use levels. The idea is that all fresh type variables created inside the right-hand side of a let binding are first assigned a higher level, which indicates that they should be generalized. However, the level of a variable is lowered when the variable "escape" through a constraint into the enclosing environment, preventing its future generalization (see the web article by Kiselyov [2013] for an excellent resource on the subject).

Simple-sub typing with levels. We can use the same idea to achieve let polymorphism in Simple-sub, though we have to be a little more careful, because we do not merely *unify* type variables as in ML, but instead we constrain their bounds. Our idea is to make sure that lower-level type variables never refer to higher-level ones through their bounds, and to enforce that property by duplicating type structures as needed, when it would otherwise be violated by the addition of a bound.

We first need to add a `lvl` field to type variable states:

```
class VariableState(val level: Int,
                    var lowerBounds: List[SimpleType],
                    var upperBounds: List[SimpleType])
```

and to update `freshVar` correspondingly:

```
def freshVar(implicit lvl: Int): Variable =
  Variable(new VariableState(lvl, Nil, Nil))
```

Next, we add an implicit `lvl` parameter to the `typeTerm` function, and we make sure to type the right-hand sides of let bindings with a higher level than the current one:

```
def typeTerm(trm: Term)(implicit ctx: Ctx, lvl: Int): SimpleType = trm match {
  ...
  // non-recursive let bindings:
  case Let(false, nme, rhs, bod) =>
    val rhs_ty = typeTerm(rhs)(ctx, lvl + 1) // incremented level!
    typeTerm(bod)(ctx + (nme -> PolymorphicType(lvl, rhs_ty)), lvl)
  ...
}
```

```
sealed trait TypeScheme {
  // to be implemented in SimpleType and TypeScheme:
  def instantiate(implicit lvl: Int): SimpleType
  def level: Int
}
enum SimpleType extends TypeScheme {
  case Variable  (s: VariableState)
  case Primitive (name: String)
  case Function  (lhs: SimpleType, rhs: SimpleType)
  case Record    (fields: List[(String, SimpleType)])
  // the following members are required to implement TypeScheme:
  def instantiate(implicit lvl: Int) = this
  lazy val level = this match {
    case Function(lhs, rhs) => max(lhs.level, rhs.level)
    case Record(fields)     => fields.map(_._2.level).maxOption.getOrElse(0)
    case Variable(vs)       => vs.level
    case Primitive(_)       => 0
  }
}
case class PolymorphicType(level: Int, body: SimpleType) extends TypeScheme {
  def instantiate(implicit lvl: Int) = body.freshenAbove(level)
}
class VariableState(val level: Int,
                    var lowerBounds: List[SimpleType],
                    var upperBounds: List[SimpleType])
```

Fig. 6. Final definitions of Simple-sub's internal type representation.

Notice that in the context used to type the body of the let binding, we wrap the right-hand side type into a `PolymorphicType` wrapper, which is defined at the end of Figure 6. A polymorphic type wraps a simple type `body`, but additionally remembers above which `level` the type variables that appear in `body` are to be considered universally quantified. Its `instantiate(lvl)` method copies `body`, replacing the type variables above `level` with fresh variables at level `lvl` (a task performed by `freshenAbove`, whose implementation is too boring to warrant taking space in this paper).

In order to make `PolymorphicType` and `SimpleType` type-compatible, we create a common base trait[11] `TypeScheme`, as shown in Figure 6. This trait contains two abstract methods: one to instantiate the type at a given level, and one to compute the level of the type. The latter is implemented in `SimpleType` by a field which is lazily evaluated, to avoid needless recomputation; this field is used to remember the maximum level of any type variables contained in the type.

Finally, we adapt `typeTerm` to instantiate the types associated with variable names in `ctx`:

```
type Ctx = Map[String, TypeScheme]
```

[11] In Scala, a `sealed trait` is like an interface which can only be implemented by types defined in the same file.

```
def typeTerm(trm: Term)(implicit ctx: Ctx, lvl: Int): SimpleType = trm match {
  ...

  case Var(name) => ctx.getOrElse(name, err("not found: " + name)).instantiate

  ...
}
```

Constraining with levels. The next step is to make sure that variables of higher level do not escape into the bounds of variables of lower level. We do that by adding guards in the constraining algorithm, preventing it from happening:

```
def constrain(lhs: SimpleType, rhs: SimpleType)
    ...

    case (Variable(lhs), rhs0) if rhs.level <= lhs.level => // new guard here
      lhs.upperBounds = rhs :: lhs.upperBounds
      lhs.lowerBounds.foreach(constrain(_, rhs))

    case (lhs0, Variable(rhs)) if lhs.level <= rhs.level => // new guard here
      rhs.lowerBounds = lhs :: rhs.lowerBounds
      rhs.upperBounds.foreach(constrain(lhs, _))
```

Naturally, we also need to handle the cases where there is a level violation. In such cases, we make a copy of the problematic type up to its type variables of wrong level using the `extrude` function, which returns a type at the right level that mirrors the structure of the original type:

```
    case (lhs @ Variable(_), rhs0) =>
      val rhs = extrude(rhs0, false)(lhs.level, MutMap.empty)
      constrain(lhs, rhs)

    case (lhs0, rhs @ Variable(_)) =>
      val lhs = extrude(lhs0, true)(rhs.level, MutMap.empty)
      constrain(lhs, rhs)

    ...
```

The `extrude` function is defined in Figure 7. Its goal is to make a copy of the problematic type such that the copy has the requested level *and* is a subtype (if `pol == false`) or a supertype (if `pol == true`) of the original type. `extrude` recursively traverses its argument type tree up to its subtrees of acceptable levels, and when it finds a type variable `vs` with the wrong level, it creates a copy `nvs` of the faulty type variable at the requested level `lvl`, and registers `nvs` as a bound of `vs`. This works

```
def extrude(ty: SimpleType, pol: Boolean)
      (implicit lvl: Int, C: MutMap[VariableState, VariableState]): SimpleType
      = if (ty.level <= lvl) ty else ty match {
  case Primitive(_)    => ty
  case Function(l, r)  => Function(extrude(l, !pol), extrude(r, pol))
  case Record(fs)      => Record(fs.map(nt => nt._1 -> extrude(nt._2, pol)))
  case Variable(vs)    => C.getOrElse(vs, {
    val nvs = freshVar
    C += vs -> nvs
    if (pol) {
      vs.upperBounds = nvs :: vs.upperBounds
      nvs.lowerBounds = vs.lowerBounds.map(extrude(_, pol))
    } else {
      vs.lowerBounds = nvs :: vs.lowerBounds
      nvs.upperBounds = vs.upperBounds.map(extrude(_, pol))
    }
    nvs
  })
}
```

Fig. 7. Type extrusion algorithm.

because `nvs` has a *lower* level than `vs`, satisfying the invariant on levels. There is an important subtlety, here: we *also* have to recursively extrude the opposite bounds of `vs` to place them in the `nvs` copy, but these bounds may form a cycle. To avoid going into an infinite extrusion loop, we keep a cache `C` of the variables already being extruded. So `extrude` can copy entire potentially-cyclic bound graphs, at the same time as the type trees in which these graphs are rooted.

Let polymorphism in MLsub. In contrast to the approach presented here, Dolan uses an equivalent "lambda-lifted" type system, which associates to let-bound variables entire typing environments, in the typing context. While this can make for a slicker specification, it is rather counter-intuitive and thus harder to understand, creates many useless type variables (which need to be simplified later), and needlessly duplicates constraints, which causes inefficiencies [Pottier 1998, Chapter 16.2].

3.5.2 Recursive Let Bindings. Finally, supporting recursive let bindings is done in the usual way, by typing the right-hand side of the let binding with, in the context, a name bound to a type variable which is later checked to be a supertype of the actual right-hand side type (see Figure 8).

3.6 Summary

We summarize the final typing and constraining algorithms in Figures 8 and 9, respectively.

Overall, Simple-sub looks more like traditional type inference algorithms than Dolan's biunification, and it completely eschews the complexities of bisubstitution and polar types. Yet, as we confirm experimentally in Section 6, both algorithms produce equivalent results. This shows that bisubstitution and polar types are not, in fact, essential to type inference with subtyping and principal types in the style of MLsub.

```
def typeTerm(trm: Term)(implicit ctx: Ctx, lvl: Int): SimpleType = trm match {
  case Lit(n)    => Primitive("int")
  case Var(name) => ctx.getOrElse(name, err("not found: " + name)).instantiate
  case Rcd(fs)   => Record(fs.map { case (n, t) => (n, typeTerm(t)) })
  case Lam(name, body) =>
    val param = freshVar
    Function(param, typeTerm(body)(ctx + (name -> param), lvl))
  case App(f, a) =>
    val res = freshVar
    constrain(typeTerm(f), Function(typeTerm(a), res))
    res
  case Sel(obj, name) =>
    val res = freshVar
    constrain(typeTerm(obj), Record((name -> res) :: Nil))
    res
  case Let(isrec, nme, rhs, bod) =>
    val rhs_ty = if (isrec) {
      val exp = freshVar(lvl + 1)
      val inf = typeTerm(rhs)(ctx + (nme -> exp), lvl + 1)
      constrain(inf, exp)
      exp
    } else typeTerm(rhs)(ctx, lvl + 1)
    typeTerm(bod)(ctx + (nme -> PolymorphicType(lvl, rhs_ty)), lvl)
}
```

Fig. 8. Full specification of term typing in Simple-sub.

4 SIMPLIFYING TYPES

As it is, the algorithm shown in the previous section infers types which often contain redundancies in their structures, as well as type variables which could be removed or unified. An important component of type inference when subtyping is involved is to simplify the types inferred, so as to make them compact and easy to comprehend [Pottier 1998]. If we did not perform any simplification, the inferred types could grow linearly with the size of the program!

In this section, we explore the design space and tradeoffs of type simplification (Section 4.1); we recall how MLsub performs automaton-based simplification (Section 4.2); we explain the ideas behind Simple-sub's more basic approach to simplification, which turns out to be sufficient most of the time — and sometimes better (Section 4.3); and we describe an intermediate representation to facilitate the application of these ideas (Section 4.4).

4.1 Type Simplification Tradeoffs

Part of the appeal of algebraic subtyping is that it produces *compact* principal types, which are easy to read, unlike previous approaches to subtype inference. However, this comes at a cost: it requires making simplifying assumptions about the semantics of types. These assumptions hold in MLsub, but may not hold in languages with more advanced features.

```
def constrain(lhs: SimpleType, rhs: SimpleType)
    (implicit cache: MutSet[(SimpleType,SimpleType)] = MutSet.empty): Unit = {
  if (cache.contains(lhs -> rhs)) return () else cache += lhs -> rhs
  (lhs, rhs) match {
    case (Primitive(n0), Primitive(n1)) if n0 == n1 => ()
    case (Function(l0, r0), Function(l1, r1)) =>
      constrain(l1, l0); constrain(r0, r1)
    case (Record(fs0), Record(fs1)) =>
      fs1.foreach { case (n1, t1) =>
        fs0.find(_._1 === n1) match {
          case None => err("missing field: " + n1 + " in " + lhs)
          case Some((_, t0)) => constrain(t0, t1) }}
    case (Variable(lhs), rhs0) if rhs.level <= lhs.level =>
      lhs.upperBounds = rhs :: lhs.upperBounds
      lhs.lowerBounds.foreach(constrain(_, rhs))
    case (lhs0, Variable(rhs)) if lhs.level <= rhs.level =>
      rhs.lowerBounds = lhs :: rhs.lowerBounds
      rhs.upperBounds.foreach(constrain(lhs, _))
    case (lhs @ Variable(_), rhs0) =>
      val rhs = extrude(rhs0, false)(lhs.level, MutMap.empty)
      constrain(lhs, rhs)
    case (lhs0, rhs @ Variable(_)) =>
      val lhs = extrude(lhs0, true)(rhs.level, MutMap.empty)
      constrain(lhs, rhs)
    case _ => err("cannot constrain " + lhs + " <: " + rhs)
  }}
```

Fig. 9. Full specification of subtype constraining in Simple-sub.

For instance, MLsub considers the types $(\mathsf{int} \to \mathsf{int}) \sqcap (\mathsf{nat} \to \mathsf{nat})$ and $\mathsf{int} \sqcup \mathsf{nat} \to \mathsf{int} \sqcap \mathsf{nat}$ to be equivalent, although the latter intuitively contains strictly *less* information. This assumption is sound, because MLsub programs cannot distinguish between the two types — program which works with one will also work with the other. However, the equivalence would not hold in a language which, for example, used intersection types to encode overloading.

As another example, MLsub does not distinguish between the types $\{\ \mathsf{tag} : 0\ ;\ \mathsf{payload} : \mathsf{str}\ \} \sqcup \{\ \mathsf{tag} : 1\ ;\ \mathsf{payload} : \mathsf{int}\ \}$ and $\{\ \mathsf{tag} : 0 \sqcup 1\ ;\ \mathsf{payload} : \mathsf{str} \sqcup \mathsf{int}\ \}$, where 1 and 2 denote singleton literal types (trivial to add to our type system). But in languages like TypeScript which support flow typing [Pearce 2013; Tobin-Hochstadt and Felleisen 2010], the former holds more information, since the different types of payload could be extracted separately by first matching on the tag.[12]

[12]Some approaches achieve complete type inference in the face of this more advanced form of reasoning [Castagna et al. 2016], but they typically lack the principal type property (generating a set of possible types instead of a single, most-general type), and they naturally do not enable the same level of simplification.

These simplifying assumptions are not *necessary* for principal type inference — they are merely a requirement of MLsub's simplification and subsumption checking approaches (note that subsumption checking is outside the scope of this paper). While they are *implied* by Dolan's algebraic construction of subtyping, making them inescapable in his system, these assumptions can actually be separated from the type inference specification — we see a syntactic interpretation of subtyping in Section 5 which does *not* imply them, the understanding being that the system can be completed with more rules as desired, to achieve the simplification potential described in this section.

4.2 Type Simplification in MLsub

Thanks to the simplifying assumptions described in the previous subsection, MLsub can represent types as finite-state automata, where the states are type variables and where the edges, which are labelled, represent relations between these type variables. There are four sorts of labels on any edge between two type variables α and β: an "*is-a*" label indicate that α is a subtype of β; a "*consumes*" label indicate that α is a function which takes some β in parameter; a "*produces*" label indicate that α is a function which returns some β as a result; and finally, a "*contains-L*" label indicate that α is a record which contains a field named L of type β. The starting state of the automaton represents the root of the type expression.

This clever representation allows one to simplify types by reusing well-known existing techniques from the domain of automata theory: type automata can be made deterministic ("*is-a*"-labelled edges are seen as ϵ edges, so type automata are initially non-deterministic) and then minimized, to achieve simplification. However, this is a quite heavy and expensive approach. We found that in practice, a more straightforward simplification algorithm was often sufficient. We describe such an algorithm in the rest of this section.

4.3 Type Simplification in Simple-sub

Our simplification approach hinges on two main ideas: *co-occurrence analysis* and *hash consing*.

4.3.1 Co-occurrence Analysis. Co-occurrence analysis looks at every variable that appears in a type in both positive and negative positions, and records along which other variables and types it always occurs. A variable v occurs along a type τ if it is part of the same type union $\ldots \sqcup v \sqcup \ldots \sqcup \tau \sqcup \ldots$ or part of the same type intersection $\ldots \sqcap v \sqcap \ldots \sqcap \tau \sqcap \ldots$

Based on this information, we can perform three kinds of simplification:

Removal of polar variable. First, we want to remove type variables which appear *only* positively (or negatively) in a type expression. For instance, consider the type inferred for $\lambda x.x + 1$, which is currently $\alpha \sqcap \mathsf{int} \to \mathsf{int}$ (because the typing of lambda expressions always assigns a type variable to the parameter). The variable α in this type is redundant since it only occurs in negative position — whichever α the caller may pick, the function will still require the argument to be an int, and it will still produce an int as a result. So we can simply remove α and obtain the simplified type $\mathsf{int} \to \mathsf{int}$.

As another example, the type of a function which uses its argument as an int but never terminates, $\mathsf{int} \to \alpha$, can be simplified to $\mathsf{int} \to \bot$.

Unification of indistinguishable variables. We have previously mentioned that a type such as $\mathsf{bool} \to \alpha \to \beta \to \alpha \sqcup \beta$ (the natural type of if-then-else) is equivalent to the simpler type $\mathsf{bool} \to \alpha \to \alpha \to \alpha$. This is true because the positive occurrences of the variables α and β are "indistinguishable" — whenever an α is produced, a β is also produced. Therefore, we cannot distinguish the two variables, and they can be unified.

Based on the result of the co-occurrence analysis, we can unify all those variables that always occur together either in positive or in negative positions (or both).

Flattening of "variable sandwiches". What we call a "variable sandwich" is an inferred type variable v which contains some τ both as an upper bound and as a lower bound, i.e., $v \le \tau$ and $v \ge \tau$. This means that v is equivalent to τ. In a coalesced type, this will transpire as v co-occurring with τ both positively and negatively. So we can use the result of co-occurrence analysis to remove variables which are sandwiched between two identical bounds. As an example, we simplify the type $\alpha \sqcap \mathsf{int} \to \alpha \sqcup \mathsf{int}$ to just $\mathsf{int} \to \mathsf{int}$.

Conceptually, this idea subsumes polar variable removal, which was explained above. Indeed, if a variable never occurs positively, it conceptually occurs both positively and negatively along with the type $\bot$, so we can replace that variable with $\bot$ (i.e., remove it from all type unions).

4.3.2 Hash Consing. Simple-sub's other simplification approach, *hash consing*, deals with removing duplicated structures in coalesced type expressions.

Consider the following recursive term:

$$\mathsf{let}\ f = \lambda x.\ \{\ \mathsf{L} = x\ ;\ \mathsf{R} = f\,x\ \}\ \mathsf{in}\ f$$

The coalesced type inferred for this term would be:

$$\alpha \to \{\ \mathsf{L} : \alpha\ ;\ \mathsf{R} : \mu\beta.\ \{\ \mathsf{L} : \alpha\ ;\ \mathsf{R} : \beta\ \}\ \}$$

Notice that there is an outer record layer that is redundant. We would like to instead infer:

$$\alpha \to \mu\beta.\ \{\ \mathsf{L} : \alpha\ ;\ \mathsf{R} : \beta\ \}$$

This can be done by performing hash consing on the types being coalesced, in the `coalesceType` function: instead of simply remembering which *variables* are in the process of being coalesced, we can remember whole type expressions; when we reach a type expression which is already being coalesced, we introduce a recursive type variable in this position, removing the redundant outer layer of types like the above.

Interestingly, MLsub does not currently perform a comparable simplification, so Simple-sub infers simpler types in examples like the one above.

4.4 An Intermediate Representation for Simplification

The above two approaches do not work very well out of the box. First, we cannot perform them on non-coalesced types, since co-occurrence analysis would miss information which only becomes apparent after the bounds are flattened. For instance, if we inferred a type variable α with upper bounds $\tau_0 \to \tau_1$ and $\tau_2 \to \tau_3$, only after we flatten these bounds and merge the function types into $\tau_0 \sqcup \tau_2 \to \tau_1 \sqcap \tau_2$ do we notice the co-occurrence of τ_0, τ_2 and τ_1, τ_3. Second, it is awkward to perform the normalization steps necessary for this sort of function type merging on the final coalesced type representation, which is syntactically too loose (it can represent types which do not correspond to inferred types, for instance merging unions and intersections).

```
case class CompactType(vars:   Set[TypeVariable],
                       prims:  Set[PrimType],
                       rcd:    Option[SortedMap[String, CompactType]],
                       fun:    Option[(CompactType, CompactType)])
case class CompactTypeScheme(cty:      CompactType,
                             recVars:  Map[TypeVariable, CompactType])
```

$$\text{S-Refl}\ \frac{}{\tau \le \tau} \qquad \text{S-Trans}\ \frac{\Sigma \vdash \tau_0 \le \tau_1 \quad \Sigma \vdash \tau_1 \le \tau_2}{\Sigma \vdash \tau_0 \le \tau_2} \qquad \text{S-Weaken}\ \frac{H}{\Sigma \vdash H} \qquad \text{S-Assum}\ \frac{\Sigma, \triangleright H \vdash H}{\Sigma \vdash H} \qquad \text{S-Hyp}\ \frac{H \in \Sigma}{\Sigma \vdash H}$$

$$\text{S-Rec}\ \frac{}{\mu\alpha.\,\tau \equiv [\mu\alpha.\,\tau/\alpha]\tau} \qquad \text{S-Or}\ \frac{\forall i, \exists j,\ \Sigma \vdash \tau_i \le \tau'_j}{\Sigma \vdash \bigsqcup_i \tau_i \le \bigsqcup_j \tau'_j} \qquad \text{S-And}\ \frac{\forall i, \exists j,\ \Sigma \vdash \tau_j \le \tau'_i}{\Sigma \vdash \bigsqcap_j \tau_j \le \bigsqcap_i \tau'_i} \qquad \text{S-Fun}\ \frac{\triangleleft\Sigma \vdash \tau_0 \le \tau_1 \quad \triangleleft\Sigma \vdash \tau_2 \le \tau_3}{\Sigma \vdash \tau_1 \to \tau_2 \le \tau_0 \to \tau_3}$$

$$\text{S-Rcd}\ \frac{}{\{\overline{l_i : t_i}^{\,i}\} \equiv \bigsqcap_i \{l_i : t_i\}} \qquad \text{S-Depth}\ \frac{\triangleleft\Sigma \vdash \tau_1 \le \tau_2}{\Sigma \vdash \{l : \tau_1\} \le \{l : \tau_2\}}$$

$$\triangleleft(H_0, H_1) = \triangleleft H_0, \triangleleft H_1 \qquad \triangleleft(\triangleright H) = H \qquad \triangleleft(\tau_0 \le \tau_1) = \tau_0 \le \tau_1$$

Fig. 10. Declarative subtyping rules of Simple-sub. These only cover *part* of the relationships present in Dolan's algebraic construction of types [Dolan 2017]. More subtyping rules can be added to give desirable properties to the system (such as distributivity of unions, intersections, and type constructors), but they are not strictly required for principal type inference. Note that by convention, we consider that an empty union is $\bot$ and an empty intersection is $\top$, so these rules cover things like int $\le \top$.

For these reasons, we introduce an intermediate `CompactType` representation between `SimpleType` and `Type`, in which to perform simplification more easily. The `CompactType` representation, shown above, corresponds to a normalized representation of types where all the non-recursive variable bounds are coalesced. The `recVars` field of `CompactTypeScheme` records the bounds of recursive type variables (which we cannot coalesce, as they are cyclic).

The `compactType` function to convert a `SimpleType` into a `CompactTypeScheme` is straightforward and looks like the `coalesceType` function shown earlier. The `simplifyType` function is slightly more complicated, as it has to perform a co-occurrence analysis pass followed by a rewriting pass. Finally, hash consing is done as part of the `coalesceCompactType` function. We do not show the implementations of these functions here for lack of space, but they can be seen in the code associated with the paper.

5 FORMALIZATION OF SIMPLE-SUB

So far, we have appealed to an intuitive understanding of subtyping, eschewing a more explicit characterization. In this section, we make our intuition more formal by giving a syntactic account of the minimal subtyping relationship required to make the type inference algorithm of Section 3 sound and complete. We state the corresponding theorems and sketch how to carry out their proofs. The complete proofs are outside the scope of this (already quite long) paper.

We restrict ourselves to the non-let-polymorphic version of Simple-sub for simplicity.[13]

5.1 A Syntax-First Definition of Subtyping

Figure 10 presents the *minimal subtyping rules* necessary to perform sound and complete type inference in Simple-sub. The general subtyping judgement has the form $\Sigma \vdash \tau_0 \le \tau_1$ and includes a subtyping context Σ made of subtyping hypotheses of the form $\tau_2 \le \tau_3$, possibly prefixed with a $\triangleright$ symbol. We use $\Sigma \vdash \tau_0 \equiv \tau_1$ as a shorthand for $\Sigma \vdash \tau_0 \le \tau_1 \wedge \Sigma \vdash \tau_1 \le \tau_0$. When Σ is empty, we omit the $\Sigma \vdash$ and just write $\tau_0 \le \tau_1$ and $\tau_0 \equiv \tau_1$.

[13] Our implementation of let polymorphism can be proven correct separately, by showing it has the same effect as duplicating the let-bound definition at each of its use sites (which is the semantics of let polymorphism [Damas and Milner 1982]).

Note that Figure 10 only presents a subset of all the rules one may want in an actual system. In particular, type simplification and subsumption checking (to determine whether one type signature is at least as general as another) require rules to merge type constructors like function types, so that for instance the equivalence $(\tau_0 \to \tau_1) \sqcap (\tau_2 \to \tau_3) \equiv \tau_0 \sqcup \tau_1 \to \tau_2 \sqcap \tau_3$ holds (see the related discussion in Section 4.1). On the other hand, we do not expect rules like distributivity of unions over intersections to be actually useful in a pure MLsub-style system, since unions and intersections are normally kept separate (unions occurring strictly positively, and intersections strictly negatively); however, they could come in useful in a generalized system.

5.1.1 Subtyping Recursive Types. A consequence of our syntactic account of subtyping is that we do *not* define types as some fixed point over a generative relation, as done in, e.g., [Dolan 2017; Pierce 2002]. Instead, we have to account for the fact that we manipulate *finite* syntactic type trees, in which recursive types have to be manually unfolded to derive things about them. This is the purpose of the S-Rec rule, which substitutes a recursive types within itself to expose one layer of its underlying definition. However, as remarked by Amadio and Cardelli [1993], the S-Rec rule alone is not sufficient to derive valid inequalities like $\mu\alpha_0.\, \tau \to \tau \to \alpha_0 \;\le\; \mu\alpha_1.\, \tau \to \alpha_1$ because these types, although equivalent, never unfold to the precise same syntactic representation. This motivates the next paragraph.

5.1.2 Subtyping Hypotheses. We make use of the environment Σ to store subtyping hypotheses, to be leveraged later using the S-Hyp rule. We have to be careful not to allow the use of a hypothesis right after assuming it, which would obviously make the system unsound. In the specification of their constraint solving algorithm, Hosoya et al. [2005] use two distinct judgments $\vdash$ and $\vdash'$ to distinguish from places where the hypotheses can or cannot be used. We take a different, but related approach. Our S-Assum subtyping rule resembles the Löb rule described by Appel et al. [2007], which uses the "later" modality $\triangleright$ in order to delay the applicability of hypotheses — by placing this symbol in front of the hypothesis being assumed, we prevent its immediate usage by S-Hyp. We eliminate $\triangleright$ when passing through a function or record constructor, using $\triangleleft$ to remove all $\triangleright$ occurrences from the set of hypotheses, thereby unlocking them for use by S-Hyp.

These precautions reflect the "guardedness" restrictions used by Dolan [2017] on recursive types, which prevents usages of α that are not guarded by $\to$ or $\{\ldots\}$ in a recursive type $\mu\alpha.\, \tau$. By contrast, our restriction is not a syntactic one, and contrary to Dolan we do allow types like $\mu\alpha.\, \alpha$ — this type unfolds into itself by S-Rec — and $\mu\alpha.\, \alpha \sqcap \alpha$, about which no useful assumptions can be leveraged, since they never go through a function or record constructor.

5.1.3 Example. As an example, let us try to derive the following inequality, which states that the type of a function taking two curried τ arguments an arbitrary number of times is a *special case* of the type of a function taking a single τ argument an arbitrary number of times:

$$\mu\alpha_0.\, \tau \to \tau \to \alpha_0 \;\le\; \mu\alpha_1.\, \tau \to \alpha_1$$

To facilitate the development, we use the shorthands $\tau_0 = \mu\alpha_0.\, \tau \to \tau \to \alpha_0$; $\tau_1 = \mu\alpha_1.\, \tau \to \alpha_1$; and $H = \tau_0 \;\le\; \tau_1$. We start by deriving that the respective unfoldings of the recursive types are subtypes; that is, that $\tau \to \tau \to \tau_0 \;\le\; \tau \to \tau_1$ (1). Note that for conciseness, we omit the

applications of S-Weaken in the derivations below:

$$\textsc{Fun}\ \frac{\textsc{Refl}\ \dfrac{}{H \vdash \tau \le \tau} \qquad \dfrac{\textsc{Fun}\ \dfrac{\textsc{Refl}\ \dfrac{}{H \vdash \tau \le \tau} \qquad \dfrac{(\tau_0 \le \tau_1) \in H}{H \vdash \tau_0 \le \tau_1}\ \textsc{Hyp}}{H \vdash \tau \to \tau_0 \ \le\ \tau \to \tau_1} \qquad \dfrac{}{H \vdash \tau \to \tau_1 \le \tau_1}\ \textsc{Rec}}{H \vdash \tau \to \tau_0 \ \le\ \tau_1}\ \textsc{Trans}}{\triangleright H \vdash \tau \to \tau \to \tau_0 \ \le\ \tau \to \tau_1} \quad (1)$$

Then, we simply have to fold back the unfolded recursive types, using Rec and Trans:

$$\textsc{Assum}\ \frac{\textsc{Trans}\ \dfrac{\textsc{Trans}\ \dfrac{\textsc{Rec}\ \dfrac{}{\triangleright H \vdash \tau_0 \ \le\ \tau \to \tau \to \tau_0} \qquad (1)}{\triangleright H \vdash \tau_0 \ \le\ \tau \to \tau_1} \qquad \dfrac{}{\triangleright H \vdash \tau \to \tau_1 \ \le\ \tau_1}\ \textsc{Rec}}{\triangleright H \vdash \tau_0 \le \tau_1}}{\tau_0 \ \le\ \tau_1}$$

5.2 Simplified Algorithms and Mutability

For ease of formal reasoning, we use a simpler definition of type coalescing, shown in Figure 11. In this definition, we refer to `TypeVariable(vs.uniqueName)` as α_{vs}, and we use α^+_{vs} and α^-_{vs} to denote two additional (distinct) unique names, to be used as positive and negative recursive occurrence binders — they serve the purpose of `freshVar.uniqueName` in the version of type coalescing shown in Section 3.3. Similarly, it is possible to give a simpler (but less efficient) definition of the `constrain` function using immutable data structures instead of mutable ones, which is easily proven equivalent; we do not show this simpler version here for lack of space, but assume its existence.

The only mutability left in the simplified algorithms is the mutability of type variable bounds. We refer to these bounds collectively as σ, which maps each VariableState instance to its current upper and lower bounds. We write $\text{foo(args)}_\sigma \Downarrow_{\sigma'} \text{res}$ to denote the execution of some function foo given bounds σ and having the effect of producing the new bounds σ'. We use the shorthand lb^{vs}_σ for $\text{vs.lowerBounds}_\sigma$ and ub^{vs}_σ for $\text{vs.upperBounds}_\sigma$.

$$\begin{aligned}
\mathbf{E}^{\phi}_{\sigma}[\![\text{Primitive(n)}]\!]\, C &= \text{n} \\
\mathbf{E}^{\phi}_{\sigma}[\![\text{Function(s, t)}]\!]\, C &= \mathbf{E}^{\neg\phi}_{\sigma}[\![\text{s}]\!]\, C \to \mathbf{E}^{\phi}_{\sigma}[\![\text{t}]\!]\, C \\
\mathbf{E}^{\phi}_{\sigma}[\![\text{Record(fs)}]\!]\, C &= \textstyle\bigsqcap_{(\text{n,t}) \in \text{fs}} \{\, \text{n} : \mathbf{E}^{\phi}_{\sigma}[\![\text{t}]\!]\, C \,\} \\
\mathbf{E}^{-}_{\sigma}[\![\text{Variable(vs)}]\!]\, C &= \alpha^-_{vs} && \text{if } (\text{vs}, -) \in C \\
\mathbf{E}^{-}_{\sigma}[\![\text{Variable(vs)}]\!]\, C &= \mu\alpha^-_{vs}.\, \alpha_{vs} \cap \textstyle\bigsqcap_{\text{u} \in \text{ub}^{vs}_\sigma} \mathbf{E}^{-}_{\sigma}[\![\text{u}]\!]\, (C \cup \{(\text{vs}, -)\}) && \text{if } (\text{vs}, -) \notin C \\
\mathbf{E}^{+}_{\sigma}[\![\text{Variable(vs)}]\!]\, C &= \alpha^+_{vs} && \text{if } (\text{vs}, +) \in C \\
\mathbf{E}^{+}_{\sigma}[\![\text{Variable(vs)}]\!]\, C &= \mu\alpha^+_{vs}.\, \alpha_{vs} \cup \textstyle\bigsqcup_{\text{l} \in \text{lb}^{vs}_\sigma} \mathbf{E}^{+}_{\sigma}[\![\text{l}]\!]\, (C \cup \{(\text{vs}, +)\}) && \text{if } (\text{vs}, +) \notin C
\end{aligned}$$

Fig. 11. Type coalescing, where the metavariable ϕ is either + or − and $\neg(+) = -$ and $\neg(-) = +$.

$$\begin{aligned}
\mathcal{E}^{-}_{\sigma}[\![\text{Variable(vs)}]\!]\, C &= \mu\alpha^-_{vs}.\, \textstyle\bigsqcap_{\text{u} \in \text{ub}^{vs}_\sigma} \mathcal{E}^{-}_{\sigma}[\![\text{u}]\!]\, (C \cup \{(\text{vs}, -)\}) && \text{if } (\text{vs}, -) \notin C \\
\mathcal{E}^{+}_{\sigma}[\![\text{Variable(vs)}]\!]\, C &= \mu\alpha^+_{vs}.\, \mathcal{E}^{-}_{\sigma}[\![\text{Variable(vs)}]\!]\, C \cup \textstyle\bigsqcup_{\text{l} \in \text{lb}^{vs}_\sigma} \mathcal{E}^{+}_{\sigma}[\![\text{l}]\!]\, (C \cup \{(\text{vs}, +)\}) && \text{if } (\text{vs}, +) \notin C
\end{aligned}$$

Fig. 12. *Unifying* type coalescing. All other cases are exactly like in Figure 11, and are omitted.

5.3 Soundness and Completeness

Our theorems of interest are the *soundness* and *completeness* of Simple-sub:

Theorem 1 (Soundness). *Simple-sub only yields types which comply with the declarative type system: if* $\mathsf{typeTerm}(t)(\mathsf{empty})_\emptyset \Downarrow_\sigma \mathsf{st}$ *for some* st *and* σ*, then there exists a type* τ *such that* $\vdash t : \tau$.

Theorem 2 (Completeness). *Simple-sub always finds principal type schemes: if* $\vdash t : \tau$*, then* $\mathsf{typeTerm}(t)(\mathsf{empty})_\emptyset \Downarrow_\sigma \mathsf{st}$ *for some* st *and* σ*, and* $\mathbf{E}^+_\sigma[\![\mathsf{st}]\!] \leq^\forall \tau$.

5.3.1 Soundness. As usual, proving the theorem requires proving a more general lemma.

We use *unifying* type coalescing (Figure 12) — a variant of type coalescing which allows proving the soundness lemmas more easily. The crucial property of unifying coalescing is that it instantiates each type variable α_{vs} in a way that makes the positive coalescing of vs a subtype of its negative coalescing, as long as all lower bounds of vs are subtypes of all its upper bounds — i.e., its bounds are *consistent*. We also denote by $\vdash_{\mathsf{cons}} \sigma$ the fact that the bounds of all variables in σ are consistent.

Lemma 1 (Soundness — General). *Assuming* $\vdash_{\mathsf{cons}} \sigma$ *and* $\mathtt{typeTerm(}t\mathtt{)(ctx)}_\sigma \Downarrow_{\sigma'} \mathtt{st}$*, then* $\vdash_{\mathsf{cons}} \sigma'$ *and* $\mathcal{E}^-_{\sigma'}[\![\mathsf{ctx}]\!] \vdash t : \mathcal{E}^-_{\sigma'}[\![\mathsf{st}]\!]$.

The proof is by induction on the executions of `typeTerm`, assuming that `typeTerm` terminates successfully. In the process, we need a number of auxiliary lemmas, most of which we do not show here. The core of the proof actually resides in the proof of *sound constraining* (Lemma 2).

Lemma 2 (Soundness of Constraining). *When it succeeds, constraining in consistent bounds ensures that the bounds remain consistent and the coalescing of the constrained types become subtypes: if* $\vdash_{\mathsf{cons}} \sigma$ *and* $\mathsf{constrain}(\mathsf{st}_0, \mathsf{st}_1)_\sigma \Downarrow_{\sigma'} ()$*, then* $\vdash_{\mathsf{cons}} \sigma'$ *and* $\mathcal{E}^-_{\sigma'}[\![\mathsf{st}_0]\!] \leq \mathcal{E}^+_{\sigma'}[\![\mathsf{st}_1]\!]$.

This is proven by induction on executions of the constraining calls. We actually need a stronger induction hypothesis, which relates the subtyping context Σ with the constraining cache, talks precisely about the bounds introduced by each call, and states that existing subtyping relations between coalesced types are not altered by further constraining calls. The `Variable` cases are quite subtle; when we insert the new bound into the variable's state, we temporarily break the consistency of the variable's bounds, but we restore it as an effect of the following recursive calls to propagate the bound. To apply the induction on these recursive calls, we need to loosen the "consistent bounds" premise of the hypothesis, making an exception for those variables which appear as part of the current constraining cache, thus allowing the calls to happen in partially-broken bounds.

5.3.2 Completeness. Completeness is proven through the following more general lemma:

Lemma 3 (Completeness — General). *Assuming* $\Gamma \vdash t : \tau$*, then for all* ctx*,* σ*, and type-variable substitution* ρ*, if* $\rho(\mathbf{E}^-_\sigma[\![\mathsf{ctx}]\!]) \equiv \Gamma$ *then* $\mathsf{typeTerm}(t)(\mathsf{ctx})_\sigma \Downarrow_{\sigma'} \mathsf{st}$ *for some* st *and* σ'*, and there exists a substitution* ρ' *such that* $\rho'(\mathbf{E}^-_{\sigma'}[\![\mathsf{ctx}]\!]) \equiv \Gamma$ *and* $\rho'(\mathbf{E}^+_{\sigma'}[\![\mathsf{st}]\!]) \leq \tau$.

Remark that "there exists a ρ' such that $\rho'(\tau_0) \leq \tau_1$" is equivalent to "$\tau_0 \leq^\forall \tau_1$" by definition of the subsumption relation $\leq^\forall$. Again, the core of the proof resides in lemmas about constraining.

Lemma 4 (Termination of Constraining). *For all* st_0, st_1*, and* σ*, either* $\mathsf{constrain}(\mathsf{st}_0, \mathsf{st}_1)_\sigma$ *hits an* `err(...)` *case and fails, or there exists a* σ' *such that* $\mathsf{constrain}(\mathsf{st}_0, \mathsf{st}_1)_\sigma \Downarrow_{\sigma'} ()$.

Lemma 5 (Completeness of Constraining). *Constraining succeeds on types whose coalesced forms are subtypes, and it does not alter existing subtyping relationships: for all* st_0, st_1, ρ*, and* σ*, if* $\rho(\mathbf{E}^-_\sigma[\![\mathsf{st}_0]\!]) \leq \mathbf{E}^+_\sigma[\![\mathsf{st}_1]\!]$*, then* $\mathsf{constrain}(\mathsf{st}_0, \mathsf{st}_1)_\sigma \Downarrow_{\sigma'} ()$ *for some* σ' *— i.e., constraining does not yield an error — and for all* st_2, st_3, σ' *such that* $\mathsf{constrain}(\mathsf{st}_2, \mathsf{st}_3)_\sigma \Downarrow_{\sigma'} ()$*, then* $\rho(\mathbf{E}^-_{\sigma'}[\![\mathsf{st}_0]\!]) \leq \mathbf{E}^+_{\sigma'}[\![\mathsf{st}_1]\!]$.

We prove Lemmas 3 and 5 by induction on typing and subtyping derivations, respectively. The rule S-Trans causes some trouble: in case the subtyping derivation used it, we get premises which refer to derivations on which we cannot apply the induction, because they do not correspond to recursive `constrain` calls. S-Trans can be removed from the system and proven from the other rules *only in an empty subtyping context*; indeed, Σ could in principle include transitivity-breaking hypotheses, such as $(\top \leq \bot) \in \Sigma$. But the subtyping context, which will mirror the constraining cache, will not be empty in the actual inductive proof of (a stronger version of) Lemma 5. The solution is to show that no transitivity-breaking assumptions are ever introduced in the subtyping context during successful constraining, and that the input subtyping relation can always be derived without using S-Trans; we do this by strengthening the induction hypothesis accordingly.

6 EXPERIMENTAL EVALUATION

To evaluate the strengths of both the Simple-sub and MLsub implementations, we ran them on 1,313,832 randomly-generated expressions of varying sizes, of which 515,509 turned out to be well-typed and 798,321 turned out to be ill-typed.

Subsumption checking. MLsub provides a *subsumption* checker, whose goal is to determine if one inferred signature is at least as general as another (i.e., it tests for $\leq^{\forall}$). We used MLsub's subsumption checker to verify that both algorithms produced equivalent result, checking that mutual subsumption held between the two inferred type expressions.

Generated expressions. We considered random expressions making use of integer literals, lambdas, applications, record creations, field selections, recursive let bindings, and non-recursive let bindings. We used at most five different variable names and at most three different field names per expression. We stochastically generated well-scoped expressions without shadowing, using a depth-first search with an exponential decrease in probability of recursing into non-leaf subexpression. This generated 1,313,832 expressions of sizes ranging from 1 to 23, the majority (about a million) being in the 10—15 range. The code used for generating and testing these expressions can be found online in the *mlsub-compare* branch of the repository, as well as in the archived artifact of this paper.

Bugs found in MLsub. We found several bugs in MLsub: a variable shadowing bug — the expression `let rec x = (let y = x in (fun x -> y)) in x` gets assigned the wrong type `a -> a` because of the shadowing of let-bounds variable `x`;[14] a type comparison bug due to a typo (which had already been fixed in another branch of the project); and a simplification bug, giving (for instance) to the record expression `{u = 0; v = {w = {w = 0}}}` the wrong type `{u : int, v : (rec a = {w : a})}`. Because of the latter bug, to carry out the tests successfully, we had to disable MLsub's simplifier (but the Simple-sub simplifier was still enabled).

Summary. We were able to make sure that Simple-sub and MLsub agreed on type inference for more than a million randomly-generated expressions. Systematic testing turned out to be a surprisingly useful tool for detecting small mistakes which would have otherwise gone unnoticed.

7 CONCLUSIONS

Algebraic subtyping and its realization in MLsub demonstrated a very promising new technique for inferring compact principal types in the presence of implicit subtyping. In this paper, we presented a simpler foundational view of MLsub's type system, which does not require notions of bisubstitution or polar types, and slightly departs from the focus on *algebra first*. This new understanding yields a simpler and more approachable specification and implementation. We showed the design of

[14] We found this because, in an earlier testing attempt, we generated expressions which contained shadowing.

Simple-sub, which achieves principal type inference and reasonable simplification in just 500 lines of code, to serve as an inspiration to future type systems and programming languages designers.

8 ACKNOWLEDGMENTS

I would like to thank Stephen Dolan and Alan Mycroft for their responsiveness and help in testing MLsub, and for their feedback on the paper; Paolo G. Giarrusso for his insightful suggestions; the paper's shepherd Tom Schrijvers; and the anonymous reviewers for their useful comments.

REFERENCES

Roberto M. Amadio and Luca Cardelli. 1993. Subtyping Recursive Types. *ACM Trans. Program. Lang. Syst.* 15, 4 (Sept. 1993), 575–631. https://doi.org/10.1145/155183.155231

Nada Amin, Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. 2016. *The Essence of Dependent Object Types.* Springer International Publishing, Cham, 249–272. https://doi.org/10.1007/978-3-319-30936-1_14

Andrew W. Appel, Paul-André Melliès, Christopher D. Richards, and Jérôme Vouillon. 2007. A Very Modal Model of a Modern, Major, General Type System. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Nice, France) *(POPL '07)*. Association for Computing Machinery, New York, NY, USA, 109–122. https://doi.org/10.1145/1190216.1190235

Giuseppe Castagna, Tommaso Petrucciani, and Kim Nguyen. 2016. Set-theoretic types for polymorphic variants. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP 2016)*. Association for Computing Machinery, Nara, Japan, 378–391. https://doi.org/10.1145/2951913.2951928

Nathanaël Courant. 2018. Safely typing algebraic effects (Gagallium Blog). http://gallium.inria.fr/blog/safely-typing-algebraic-effects/. Accessed: 2020-06-30.

Luis Damas and Robin Milner. 1982. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '82)*. Association for Computing Machinery, Albuquerque, New Mexico, 207–212. https://doi.org/10.1145/582153.582176

Stephen Dolan. 2017. *Algebraic subtyping.* Ph.D. Dissertation.

Stephen Dolan and Alan Mycroft. 2017. Polymorphism, subtyping, and type inference in MLsub. *ACM SIGPLAN Notices* 52, 1 (Jan. 2017), 60–72. https://doi.org/10.1145/3093333.3009882

Joshua Dunfield and Neelakantan R. Krishnaswami. 2013. Complete and Easy Bidirectional Typechecking for Higher-Rank Polymorphism. *SIGPLAN Not.* 48, 9 (Sept. 2013), 429–442. https://doi.org/10.1145/2544174.2500582

Tim Freeman and Frank Pfenning. 1991. Refinement types for ML. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*. 268–277.

Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. 2008. Semantic Subtyping: Dealing Set-Theoretically with Function, Union, Intersection, and Negation Types. *J. ACM* 55, 4, Article 19 (Sept. 2008), 64 pages. https://doi.org/10.1145/1391289.1391293

Roger Hindley. 1969. The Principal Type-Scheme of an Object in Combinatory Logic. *Trans. Amer. Math. Soc.* 146 (1969), 29–60. https://doi.org/10.2307/1995158 Publisher: American Mathematical Society.

Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. 2005. Regular Expression Types for XML. *ACM Trans. Program. Lang. Syst.* 27, 1 (Jan. 2005), 46–90. https://doi.org/10.1145/1053468.1053470

DeLesley S. Hutchins. 2010. Pure Subtype Systems. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Madrid, Spain) *(POPL '10)*. Association for Computing Machinery, New York, NY, USA, 287–298. https://doi.org/10.1145/1706299.1706334

Oleg Kiselyov. 2013. Efficient generalization with levels (Okmij Blog). http://okmij.org/ftp/ML/generalization.html#levels. Accessed: 2020-06-30.

Robin Milner. 1978. A theory of type polymorphism in programming. *J. Comput. System Sci.* 17, 3 (Dec. 1978), 348–375. https://doi.org/10.1016/0022-0000(78)90014-4

Martin Odersky and Konstantin Läufer. 1996. Putting Type Annotations to Work. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg Beach, Florida, USA) *(POPL '96)*. Association for Computing Machinery, New York, NY, USA, 54–67. https://doi.org/10.1145/237721.237729

David J. Pearce. 2013. Sound and Complete Flow Typing with Unions, Intersections and Negations. In *Verification, Model Checking, and Abstract Interpretation (Lecture Notes in Computer Science)*, Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni (Eds.). Springer, Berlin, Heidelberg, 335–354. https://doi.org/10.1007/978-3-642-35873-9_21

Benjamin C. Pierce. 2002. *Types and programming languages.* MIT press.

François Pottier. 1998. *Type Inference in the Presence of Subtyping: from Theory to Practice.* Research Report RR-3483. INRIA. https://hal.inria.fr/inria-00073205

Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. 2008. Liquid Types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. Association for Computing Machinery, New York, NY, USA, 159–169. https://doi.org/10.1145/1375581.1375602

Andreas Rossberg. 2015. 1ML – Core and Modules United (F-Ing First-Class Modules). In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming* (Vancouver, BC, Canada) *(ICFP 2015)*. Association for Computing Machinery, New York, NY, USA, 35–47. https://doi.org/10.1145/2784731.2784738

John Rushby, Sam Owre, and Natarajan Shankar. 1998. Subtypes for Specifications: Predicate Subtyping in PVS. *IEEE Trans. Softw. Eng.* 24, 9 (Sept. 1998), 709–720. https://doi.org/10.1109/32.713327

Sam Tobin-Hochstadt and Matthias Felleisen. 2010. Logical Types for Untyped Languages. *SIGPLAN Not.* 45, 9 (Sept. 2010), 117–128. https://doi.org/10.1145/1932681.1863561

Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2014. Refinement Types for Haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming* (Gothenburg, Sweden) *(ICFP '14)*. Association for Computing Machinery, New York, NY, USA, 269–282. https://doi.org/10.1145/2628136.2628161