Intersection Types and Computational Effects *

Rowan Davies
Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213, U.S.A.
rowan@cs.cmu.edu

Frank Pfenning
Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213, U.S.A.
fp@cs.cmu.edu

ABSTRACT
We show that standard formulations of intersection type systems are unsound in the presence of computational effects, and propose a solution similar to the value restriction for polymorphism adopted in the revised definition of Standard ML. It differs in that it is not tied to **let**-expressions and requires an additional weakening of the usual subtyping rules. We also present a bi-directional type-checking algorithm for the resulting language that does not require an excessive amount of type annotations and illustrate it through some examples. We further show that the type assignment system can be extended to incorporate parametric polymorphism. Taken together, we see our system and associated type-checking algorithm as a significant step towards the introduction of intersection types into realistic programming languages. The added expressive power would allow many more properties of programs to be stated by the programmer and statically verified by a compiler.

Categories and Subject Descriptors
F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs—*type structure*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*polymorphism*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs

General Terms
Languages,Theory,Verification

1. INTRODUCTION
The advantages of statically typed programming languages are well known, and have been described many times (see, for example, [3]). However, conventional type systems for realistic programming languages can not express, and therefore not check, many interesting program properties. In prior research we have designed an extension of ML's type system to capture invariants of data structures. In the resulting language of *refinement types* [7, 5, 4] we can define subtypes of data types, essentially via regular tree grammars. Experiments with refinement types and related work on soft types [1, 21] have demonstrated the utility of the additional expressive power to catch more programmer errors and give stronger guarantees at module boundaries. For practical purposes, refinements require at least some form of intersection types, because a given function may have more than one property. As a simple example (elaborated in Section 5), consider the types `nat` of natural numbers and `pos` of positive natural numbers where $\mathsf{pos} \leq \mathsf{nat}$. Then the function `double` maps natural numbers to natural numbers ($\mathsf{nat} \rightarrow \mathsf{nat}$), but also maps positive numbers to positive numbers ($\mathsf{pos} \rightarrow \mathsf{pos}$) and hence has type $(\mathsf{nat} \rightarrow \mathsf{nat}) \wedge (\mathsf{pos} \rightarrow \mathsf{pos})$.

In this paper we demonstrate that general intersection types are unsound in the presence of computational effects and make two major contributions towards the use of intersection types in practical programming languages:

1. We propose a simple type assignment system for a core functional language with mutable references and intersection types and prove that it is sound, and
2. we design a corresponding source language that permits bi-directional type-checking without being prohibitively verbose.

We illustrate the resulting core language with some small examples. Our restriction is similar to the *value restriction* employed in ML [10] in order to avoid unsound uses of parametric polymorphism (see [17, 20]). However, in addition to a value restriction on the introduction of intersections, we also need to discard the distributivity law for subtyping, leading to a system which is overall significantly simpler than general intersection types without a noticeable loss in expressive power or accuracy.

Refinement types differ from intersection types in that the intersection $A \wedge B$ can only be formed if A and B are specializations of the same simple type. We do not impose this additional requirement here, since it is orthogonal to both soundness in the presence of effects and the issue of

*This work was sponsored in part by the Advanced Research Projects Agency CSTO under the title "The Fox Project: Advanced Languages for Extensible Systems", ARPA Order No. C533.

bi-directional type checking. So our results apply, for example, to operator overloading and even self-application. On the other hand, we have chosen an inclusion interpretation of subtyping which allows us to give an untyped operational semantics without explicit coercions. This is sufficient for refinement types and could easily be extended to a coercive interpretation of subtyping and intersections [2, 15].

Finally, we show how our type assignment system can be extended to include a value restricted form of parametric polymorphism. However, we do not show how to extend our source language, since a generalization of bi-directional type checking to *local type inference* [13] in the presence of polymorphism, subtyping, intersections, and a value restriction does not appear straightforward.

We close the introduction with a simple example that illustrates the unsoundness of intersection in the presence of mutable references. Similar counterexamples can be constructed for other computational effects such as exceptions. Assume, as above, that $\mathsf{pos} \leq \mathsf{nat}$. We work with a representation of natural numbers as bit strings so that ϵ represents 0 and $\epsilon 1$ represents 1.

$$
\begin{array}{l}
\mathbf{let}\ x = \mathbf{ref}(\epsilon\, 1) : \mathsf{nat}\ \mathbf{ref}\ \wedge\ \mathsf{pos}\ \mathbf{ref} \\
\mathbf{in\ let}\ y = (x := \epsilon) \\
\mathbf{in\ let}\ z = !\ x \\
\mathbf{in}\ z : \mathsf{pos}
\end{array}
$$

In this example, we create a new cell with initial contents $\epsilon\, 1$ and assign the type $\mathsf{nat}\,\mathbf{ref}\ \wedge\ \mathsf{pos}\ \mathbf{ref}$. Certainly, both of these are valid types for x, since the contents of the cell is both of type nat and type pos. Then we assign ϵ to x, which is well-typed since x has type $\mathsf{nat}\ \mathbf{ref}$, among others. Then we read the contents, requiring the result to have type pos, which is valid since x has type $\mathsf{pos}\ \mathbf{ref}$, among others. During evaluation, however, z will be bound to ϵ, so the type system is unsound: the whole expression has type pos, but evaluates to ϵ which represents zero and does not have type pos.

The remainder of this paper is organized as follows. In Section 2 we show that a value restriction on intersection introduction leads to a sound type assignment system for a small functional language including mutable references. In Section 3 we present a corresponding source language and a bi-directional type-checking algorithm. Our type system is generalized to include parametric polymorphism in Section 4 and illustrated by various examples in Section 5. We conclude with some remarks about future work in Section 6.

2. A VALUE RESTRICTION FOR INTERSECTION TYPES

In this section we present a small language with functions, mutable references, and intersection types. We also include an example datatype bits for strings of bits, along with two subtypes nat for natural numbers (bit-strings without leading zeroes) and pos for positive natural numbers. We place a value restriction on the introduction of intersections, omit the problematic distributivity rule from subtyping, and then show that this leads to a sound system by proving an appropriate progress and type preservation theorem. An analysis of the proof gives some insight as to why each of our restrictions is required.

The term language in this section does not contain types for several reasons:

1. the typing rules are maximally general, admitting as many programs as possible,
2. the progress theorem demonstrates that an *untyped* operational semantics is sound (i.e., types may, but need not be carried at runtime), and
3. formulating reduction rules directly on terms with some type annotations as in Section 3 is awkward at best.

Type inference for this language is most likely undecidable, and principal types do not exist. Therefore we present a more practical source language which includes some type information and an associated bi-directional type checking algorithm in Section 3. The untyped terms in this section can be obtained simply by erasure which would naturally be part of the compilation process (see Theorem 7). We avoid coercions by considering only subtype relations which are inclusions, but this is not an essential restriction of our approach.

2.1 Syntax

The syntax is relatively standard for a call-by-value language in the ML family. We allow general fixed-points with eager unrolling, which means we should distinguish two kinds of variables: those bound in λ, **let** and **case** expressions which stand for values (denoted by x), and those bound in **fix** expressions which stand for arbitrary terms (denoted by u). As proposed by Leroy [9], we can also easily admit a "by name" **let** expression. We further use identifiers l to address cells in the store during evaluation.

We represent natural numbers as bit-strings in standard form, with the least significant bit rightmost and no leading zeroes. We view $\mathsf{0}$ and $\mathsf{1}$ as constructors written in postfix form, and ϵ stands for the empty string. For example, 6 would be represented as $\epsilon\mathsf{110}$.

$$
\begin{array}{lrl}
\text{Types} & A ::= & A_1 \rightarrow A_2 \mid A\ \mathbf{ref} \mid \mathbf{unit} \\
 & \mid & \mathsf{bits} \mid \mathsf{nat} \mid \mathsf{pos} \mid A_1 \wedge A_2 \\
\text{Terms} & M ::= & x \mid \lambda x.\, M \mid M_1\, M_2 \\
 & \mid & \mathbf{let}\ x = M_1\ \mathbf{in}\ M_2 \\
 & \mid & u \mid \mathbf{fix}\ u.\, M \\
 & \mid & l \mid \mathbf{ref}\ M \mid !\ M \mid M_1 := M_2 \mid () \\
 & \mid & \epsilon \mid M\,\mathsf{0} \mid M\,\mathsf{1} \\
 & \mid & \mathbf{case}\ M\ \mathbf{of}\ \epsilon \Rightarrow M_1 \mid x\,\mathsf{0} \Rightarrow M_2 \mid y\,\mathsf{1} \Rightarrow M_3
\end{array}
$$

We use A, B for types and M, N for terms. We write $\{M'/x\}M$ for the result of substituting M' for x in M, renaming bound variables as necessary to avoid the capture of free variables in M'.

We distinguish the following terms as *values*:

$$
\text{Values}\quad V \quad ::= \quad x \mid \lambda x.\, M \mid l \mid () \mid \epsilon \mid V\,\mathsf{0} \mid V\,\mathsf{1}
$$

For type-checking, we need to assign types to variables and cells in contexts Γ and Δ, respectively. Moreover, during execution of a program we need to maintain a store C.

$$\begin{array}{rrcl} \text{Variable Contexts} & \Gamma & ::= & \cdot \mid \Gamma, x{:}A \mid \Gamma, u{:}A \\ \text{Cell Contexts} & \Delta & ::= & \cdot \mid \Delta, l{:}A \\ \text{Store} & C & ::= & \cdot \mid C, (l = V) \\ \text{Program States} & P & ::= & C \triangleright M \end{array}$$

We assume that variables x, u and cells l can be declared at most once in a context or store. We omit leading $\cdot$'s from contexts, and write Γ, Γ' for the result of appending two variable disjoint contexts (and similarly for cell contexts and stores).

2.2 Subtyping

The subtyping judgment for this language has the form:

$A \leq B$ Type A is a subtype of B.

It is defined by the following rules, which are standard except for the omission of distributivity (see below) and the addition of subtyping for the base types bits, nat and pos. Note that function types are contra-variant in the argument and co-variant in the result, while the **ref** type constructor is non-variant.

$$\frac{}{\mathsf{pos} \leq \mathsf{nat}} \qquad \frac{}{\mathsf{nat} \leq \mathsf{bits}}$$

$$\frac{}{A \leq A} \qquad \frac{A_1 \leq A_2 \quad A_2 \leq A_3}{A_1 \leq A_3}$$

$$\frac{}{A_1 \wedge A_2 \leq A_1} \qquad \frac{}{A_1 \wedge A_2 \leq A_2}$$

$$\frac{A \leq B_1 \qquad A \leq B_2}{A \leq B_1 \wedge B_2}$$

$$\frac{B_1 \leq A_1 \qquad A_2 \leq B_2}{A_1 \to A_2 \leq B_1 \to B_2}$$

$$\frac{A \leq B \qquad B \leq A}{A\ \mathbf{ref} \leq B\ \mathbf{ref}}$$

As mentioned earlier, we omit the following distributivity rule since it leads to unsoundness when functions involve effects:

$$\frac{}{(A \to B) \wedge (A \to B') \leq A \to (B \wedge B')}$$

A counterexample is given by $(\lambda x.\ \mathbf{ref}\ (\epsilon\, 1))\ ()$ which would have type $(\mathsf{nat}\,\mathbf{ref}) \wedge (\mathsf{pos}\,\mathbf{ref})$ since

$$(\lambda x.\ \mathbf{ref}\,\epsilon) : (\mathbf{unit} \to \mathsf{nat}\,\mathbf{ref}) \wedge (\mathbf{unit} \to \mathsf{pos}\,\mathbf{ref})$$

and the above rule allows us to infer

$$\begin{array}{c} (\mathbf{unit} \to \mathsf{nat}\,\mathbf{ref}) \wedge (\mathbf{unit} \to \mathsf{pos}\,\mathbf{ref}) \\ \leq \mathbf{unit} \to (\mathsf{nat}\,\mathbf{ref} \wedge \mathsf{pos}\,\mathbf{ref}) \end{array}$$

In other words, distributivity would allow us to circumvent the value restriction on the introduction of intersections given in the next subsection. It is somewhat surprising, however, that it is possible to simply drop the rule and obtain a sensible system. The loss in expressive power that results appears to be minimal in practice, since often when we want to coerce type $(A \to B) \wedge (A \to B')$ to type $A \to (B \wedge B')$ we can replace the first type by the second throughout the type derivation (possibly using η-expansion to satisfy the value restriction on intersection introduction, similar to [20]). Dropping distributivity has the interesting effect that the subtyping rules for various type constructors ($\to$, **ref**, $\wedge$) are now orthogonal. This persists, even when further constructors such as products or lists are added.

The rules above do not immediately yield an algorithm for deciding subtyping. We thus present the following algorithmic subtyping judgment, and show that it is equivalent to the above. We use the notation A^o for an *ordinary* type, namely one that is not an intersection, although it may contain embedded intersections. Due to the absence of distributivity, our subtyping algorithm is quite different to previous such algorithms proposed for intersection types, such as that of Reynolds [14].

$A \trianglelefteq B$ Type A is algorithmically a subtype of type B.

$$\frac{}{\mathsf{pos} \trianglelefteq \mathsf{pos}} \qquad \frac{}{\mathsf{pos} \trianglelefteq \mathsf{nat}} \qquad \frac{}{\mathsf{pos} \trianglelefteq \mathsf{bits}}$$

$$\frac{}{\mathsf{nat} \trianglelefteq \mathsf{nat}} \qquad \frac{}{\mathsf{nat} \trianglelefteq \mathsf{bits}} \qquad \frac{}{\mathsf{bits} \trianglelefteq \mathsf{bits}}$$

$$\frac{B_1 \trianglelefteq A_1 \qquad A_2 \trianglelefteq B_2}{A_1 \to A_2 \trianglelefteq B_1 \to B_2} \qquad \frac{A \trianglelefteq B \qquad B \trianglelefteq A}{A\ \mathbf{ref} \trianglelefteq B\ \mathbf{ref}}$$

$$\frac{}{\mathbf{unit} \trianglelefteq \mathbf{unit}}$$

$$\frac{A_1 \trianglelefteq B^o}{A_1 \wedge A_2 \trianglelefteq B^o} \qquad \frac{A_2 \trianglelefteq B^o}{A_1 \wedge A_2 \trianglelefteq B^o}$$

$$\frac{A \trianglelefteq B_1 \qquad A \trianglelefteq B_1}{A \trianglelefteq B_1 \wedge B_2}$$

We now prove some simple lemmas needed to show that algorithmic and declarative subtyping coincide.

Lemma 1 (Algorithmic Subtyping). *The algorithmic subtyping judgment satisfies:*

1. *If $A \trianglelefteq B$ then $A \wedge A' \trianglelefteq B$ and $A' \wedge A \trianglelefteq B$.*
2. *$A \trianglelefteq A$.*

3. If $A_1 \trianglelefteq A_2$ and $A_2 \trianglelefteq A_3$ then $A_1 \trianglelefteq A_3$.

PROOF. By simple inductions on given types or derivations. Reflexivity (2) requires monotonicity (1). □

THEOREM 2. *$A \trianglelefteq B$ if and only if $A \leq B$.*

PROOF. In each direction, by induction on the given derivation, using the properties in the preceding lemma. □

2.3 Typing

The typing judgment for terms has the form:

$\Delta; \Gamma \vdash M : A$	Term M has type A in cell context Δ and variable context Γ.

The typing rules are given in Figure 1. These rules are standard for functions, definitions, fixed points, references, and intersection types, with the exception that the introduction rule for intersections is restricted to values.

There are three typing rules for **case**, depending on whether the subject can be shown to have type bits, nat, or pos. This illustrates that in a typical use of intersections as refinements, we derive introduction as well as elimination rules for each type. Note that the branch for ϵ does not need to be checked when the case subject has type pos. We omit formal statement and proof of the trivial exchange, weakening, and contraction properties.

This type system does not admit principal types. For example, the term $\mathbf{ref}\,(\epsilon\,1)$ has the types $\mathsf{bits}\,\mathbf{ref}$, $\mathsf{nat}\,\mathbf{ref}$ and $\mathsf{pos}\,\mathbf{ref}$, but none of these is a subtype of the others. Even if we add parametric polymorphism (as in Section 4), this term has no principal type.

The value restriction on intersection introduction does not appear to result much loss of expressive power, particularly since non-values may be η-expanded to values, just as in [20]. This only alters the semantics of the program when effects are involved.

LEMMA 3 (SUBSTITUTION).

1. *If $\Delta; \Gamma \vdash V : A$ and $\Delta; (\Gamma, x{:}A) \vdash N : B$ then $\Delta; \Gamma \vdash \{V/x\}N : B$.*
2. *If $\Delta; \Gamma \vdash M : A$ and $\Delta; (\Gamma, u{:}A) \vdash N : B$ then $\Delta; \Gamma \vdash \{M/u\}N : B$.*

PROOF. By a standard induction on the typing derivation for N. □

Stores are typed using the following judgment:

$\Delta \vdash C : \Delta'$	Store C satisfies cell context Δ' when checked against cell context Δ.

The rules simply type each value under an empty variable context.

$$\frac{}{\Delta \vdash \cdot : \cdot} \qquad \frac{\Delta \vdash C' : \Delta' \qquad \Delta; \cdot \vdash V : A}{\Delta \vdash (C', l = V) : (\Delta', l{:}A)}$$

The following judgment defines typing of program states:

$\vdash (C \triangleright M) : (\Delta \triangleright A)$	Program state $(C \triangleright M)$ satisfies cell context Δ and type A.

It is defined directly from the previous typing judgments:

$$\frac{\Delta \vdash C : \Delta \qquad \Delta; \cdot \vdash M : A}{\vdash (C \triangleright M) : (\Delta \triangleright A)}$$

2.4 Reduction Semantics

We now present a reduction style semantics for our language, roughly following Wright and Felleisen [22]. We start by defining *evaluation contexts*, namely expressions with a hole $[\,]$ within which a reduction may occur:

$$\begin{array}{rcl} E & ::= & [\,] \mid E\,M \mid V\,E \\ & & \mid \mathbf{let}\ x = E\ \mathbf{in}\ M \\ & & \mid \mathbf{ref}\,E \mid \mathbf{!}\,E \mid E \mathbin{\mathbf{:=}} M \mid V \mathbin{\mathbf{:=}} E \\ & & \mid E\,\mathsf{0} \mid E\,\mathsf{1} \\ & & \mid \mathbf{case}\ E\ \mathbf{of}\ \epsilon \Rightarrow M_1 \ \mid\ x\,\mathsf{0} \Rightarrow M_2 \ \mid\ y\,\mathsf{1} \Rightarrow M_3 \end{array}$$

We write $C \triangleright M \longmapsto C' \triangleright M'$ for a one-step computation, defined by the reduction rules in Figure 2. We maintain the invariant that M does not contain free variables x or u and that all cells l in M are defined in C.

Critical in the proof of progress are the following inversion properties. These are generalizations of simpler properties in languages without subtyping, intersections, or effects. They are stated at a level of generality where each can be proved directly by inducting on the given typing derivation.

LEMMA 4 (TYPING INVERSION).

1. *If $\Delta; \cdot \vdash V : A$ and $A \trianglelefteq B_1 \to B_2$ then $V = \lambda x.\ M$ and $\Delta; (x{:}B_1) \vdash M : B_2$.*
2. *If $\Delta; \cdot \vdash V : A$ and $A \trianglelefteq B\ \mathbf{ref}$ then $V = l$ and there exists a B' such that $l{:}B'$ in Δ, $B' \trianglelefteq B$, and $B \trianglelefteq B'$.*
3. *If $\Delta; \cdot \vdash V : A$ and $A \trianglelefteq \mathsf{bits}$ then we have one of the following cases:*
   - *(a) $V = \epsilon$*
   - *(b) $V = (V_0\ \mathsf{0})$ and $\Delta; \cdot \vdash V_0 : \mathsf{bits}$*
   - *(c) $V = (V_1\ \mathsf{1})$ and $\Delta; \cdot \vdash V_1 : \mathsf{bits}$*
4. *If $\Delta; \cdot \vdash V : A$ and $A \trianglelefteq \mathsf{nat}$ then we have one of the following cases:*
   - *(a) $V = \epsilon$*

$$\frac{x{:}A \text{ in } \Gamma}{\Delta; \Gamma \vdash x : A}\,\mathsf{tp\_var} \qquad \frac{\Delta; (\Gamma, x{:}A) \vdash M : B}{\Delta; \Gamma \vdash \lambda x.\ M : A \to B}\,\mathsf{tp\_lam} \qquad \frac{\Delta; \Gamma \vdash M : A \to B \quad \Delta; \Gamma \vdash N : A}{\Delta; \Gamma \vdash M\, N : B}\,\mathsf{tp\_app}$$

$$\frac{\Delta; \Gamma \vdash M : A \qquad \Delta; (\Gamma, x{:}A) \vdash N : B}{\Delta; \Gamma \vdash \mathbf{let}\ x = M\ \mathbf{in}\ N : B}\,\mathsf{tp\_let} \qquad \frac{u{:}A \text{ in } \Gamma}{\Delta; \Gamma \vdash u{:}A}\,\mathsf{tp\_var}' \qquad \frac{\Delta; (\Gamma, u{:}A) \vdash M : A}{\Delta; \Gamma \vdash \mathbf{fix}\ u.\ M : A}\,\mathsf{tp\_fix}$$

$$\frac{l{:}A \text{ in } \Delta}{\Delta; \Gamma \vdash l : A\,\mathbf{ref}}\,\mathsf{tp\_cell} \qquad \frac{\Delta; \Gamma \vdash M : A}{\Delta; \Gamma \vdash \mathbf{ref}\ M : A\,\mathbf{ref}}\,\mathsf{tp\_ref} \qquad \frac{\Delta; \Gamma \vdash M : A\,\mathbf{ref}}{\Delta; \Gamma \vdash \mathbf{!}\, M : A}\,\mathsf{tp\_get}$$

$$\frac{\Delta; \Gamma \vdash M : A\,\mathbf{ref} \quad \Delta; \Gamma \vdash N : A}{\Delta; \Gamma \vdash M \mathbin{\mathbf{:=}} N : \mathbf{unit}}\,\mathsf{tp\_set} \qquad \frac{}{\Delta; \Gamma \vdash () : \mathbf{unit}}\,\mathsf{tp\_unit}$$

$$\frac{}{\Delta; \Gamma \vdash \epsilon : \mathsf{nat}}\,\mathsf{tp\_e}$$

$$\frac{\Delta; \Gamma \vdash M : \mathsf{pos}}{\Delta; \Gamma \vdash M\ 0 : \mathsf{pos}}\,\mathsf{tp\_z1} \qquad \frac{\Delta; \Gamma \vdash M : \mathsf{bits}}{\Delta; \Gamma \vdash M\ 0 : \mathsf{bits}}\,\mathsf{tp\_z2}$$

$$\frac{\Delta; \Gamma \vdash M : \mathsf{nat}}{\Delta; \Gamma \vdash M\ 1 : \mathsf{pos}}\,\mathsf{tp\_o1} \qquad \frac{\Delta; \Gamma \vdash M : \mathsf{bits}}{\Delta; \Gamma \vdash M\ 1 : \mathsf{bits}}\,\mathsf{tp\_o2}$$

$$\frac{\Delta; \Gamma \vdash M : \mathsf{bits} \quad \Delta; \Gamma \vdash M_1 : A \quad \Delta; (\Gamma, x{:}\mathsf{bits}) \vdash M_2 : A \quad \Delta; (\Gamma, y{:}\mathsf{bits}) \vdash M_3 : A}{\Delta; \Gamma \vdash \mathbf{case}\ M\ \mathbf{of}\ \epsilon \Rightarrow M_1 \mid x\ 0 \Rightarrow N_2 \mid y\ 1 \Rightarrow M_3 : A}\,\mathsf{tp\_case1}$$

$$\frac{\Delta; \Gamma \vdash M : \mathsf{nat} \quad \Delta; \Gamma \vdash M_1 : A \quad \Delta; (\Gamma, x{:}\mathsf{pos}) \vdash M_2 : A \quad \Delta; (\Gamma, y{:}\mathsf{nat}) \vdash M_3 : A}{\Delta; \Gamma \vdash \mathbf{case}\ M\ \mathbf{of}\ \epsilon \Rightarrow M_1 \mid x\ 0 \Rightarrow N_2 \mid y\ 1 \Rightarrow M_3 : A}\,\mathsf{tp\_case2}$$

$$\frac{\Delta; \Gamma \vdash M : \mathsf{pos} \quad \Delta; (\Gamma, x{:}\mathsf{pos}) \vdash M_2 : A \quad \Delta; (\Gamma, y{:}\mathsf{nat}) \vdash M_3 : A}{\Delta; \Gamma \vdash \mathbf{case}\ M\ \mathbf{of}\ \epsilon \Rightarrow M_1 \mid x\ 0 \Rightarrow N_2 \mid y\ 1 \Rightarrow M_3 : A}\,\mathsf{tp\_case3}$$

$$\frac{\Delta; \Gamma \vdash V : A \qquad \Delta; \Gamma \vdash V : B}{\Delta; \Gamma \vdash V : A \wedge B}\,\mathsf{tp\_conj} \qquad \frac{\Delta; \Gamma \vdash M : A \qquad A \trianglelefteq B}{\Delta; \Gamma \vdash M : B}\,\mathsf{tp\_subs}$$

Figure 1: Typing Rules

$$
\begin{array}{rcll}
C \rhd E[(\lambda x.\, M)\, V] & \longmapsto & C \rhd E[\{V/x\}M] & \\
C \rhd E[\mathbf{let}\ x = V\ \mathbf{in}\ M] & \longmapsto & C \rhd E[\{V/x\}M] & \\
C \rhd E[\mathbf{fix}\ u.\, M] & \longmapsto & C \rhd E[\{\mathbf{fix}\ u.\, M/u\}M] & \\
C \rhd E[(\mathbf{ref}\ V)] & \longmapsto & C, (l = V) \rhd E[l] & (l \text{ not in } C \text{ or } E) \\
C_1, (l = V), C_2 \rhd E[!\, l] & \longmapsto & C_1, (l = V), C_2 \rhd E[V] & \\
C_1, (l = V_1), C_2 \rhd E[l := V_2] & \longmapsto & C_1, (l = V_2), C_2 \rhd E[()] & \\
C \rhd E[\mathbf{case}\ \epsilon\ \mathbf{of}\ \epsilon \Rightarrow M_1 \mid x\ 0 \Rightarrow M_2 \mid y\ 1 \Rightarrow M_3] & \longmapsto & C \rhd E[M_1] & \\
C \rhd E[\mathbf{case}\ V\ 0\ \mathbf{of}\ \epsilon \Rightarrow M_1 \mid x\ 0 \Rightarrow M_2 \mid y\ 1 \Rightarrow M_3] & \longmapsto & C \rhd E[\{V/x\}M_2] & \\
C \rhd E[\mathbf{case}\ V\ 1\ \mathbf{of}\ \epsilon \Rightarrow M_1 \mid x\ 0 \Rightarrow M_2 \mid y\ 1 \Rightarrow M_3] & \longmapsto & C \rhd E[\{V/y\}M_3] &
\end{array}
$$

Figure 2: Reduction Rules

(b) $V = (V_0\ 0)$ *and* $\Delta; \cdot \vdash V_0 : \mathsf{pos}$

(c) $V = (V_1\ 1)$ *and* $\Delta; \cdot \vdash V_1 : \mathsf{nat}$

5. *If* $\Delta; \cdot \vdash V : A$ *and* $A \trianglelefteq \mathsf{pos}$ *then we have one of the following cases:*

   (a) $V = (V_0\ 0)$ *and* $\Delta; \cdot \vdash V_0 : \mathsf{pos}$

   (b) $V = (V_1\ 1)$ *and* $\Delta; \cdot \vdash V_1 : \mathsf{nat}$

We are now ready to prove our main theorem, namely that our type system with mutable references and value-restricted intersections satisfies progress and type preservation, i.e., that programs can't go wrong as in the example in the introduction.

THEOREM 5 (PROGRESS AND TYPE PRESERVATION). *If* $\vdash (C \rhd M) : (\Delta \rhd A)$ *then either*

1. *M is a value.*
2. $(C \rhd M) \longmapsto (C' \rhd M')$ *for some* C', M' *and* Δ' *satisfying* $\vdash (C' \rhd M') : (\Delta, \Delta' \rhd A)$.

PROOF. By induction on the typing derivation for M.

- The case for subsumption is immediate, using the induction hypothesis.
- The case for intersection introduction is trivial: the value restriction forces M to be a value.
- For the remaining cases the typing rule matches the top term constructor of M.
- The cases for the typing rules corresponding to $\lambda x.\, M$, l, $()$ and ϵ are trivial, since they are values.
- The case for the typing rule corresponding to **fix** is easy, since we can apply the substitution lemma to construct the required typing derivation.
- In the other cases, we apply the induction hypothesis to the subderivations for appropriate subterms N_i of M which are in evaluation positions i.e. $M = E[N_i]$ (in each case, there is at least one).
- Then, if for some N_i the induction hypothesis yields $(C \rhd N_i) \longmapsto (C' \rhd N_i')$ with $(C' \rhd N_i') : (\Delta, \Delta' \rhd B)$ then we can construct the required reduction and typing derivation for M.
- Otherwise, each immediate subterm N_i with $M = E[N_i]$ is a value.
- Then we apply the appropriate clause of the preceding inversion lemma, using reflexivity of algorithmic subtyping. In each case we find that M can be reduced and we can construct the required typing for the result of reduction, using the substitution lemma in some cases.

□

All of our restrictions are needed in this proof:

- The case of $E[!\, l]$ requires subtyping for A **ref** to be co-variant.
- The case of $E[l := V]$ requires subtyping for A **ref** to be contra-variant. With the previous point it means it must be non-variant.
- The value restriction is needed because otherwise the induction hypothesis is applied to the premises of the intersection introduction rule

  $$\frac{\Delta; \cdot \vdash M : A_1 \qquad \Delta; \cdot \vdash M : A_2}{\Delta; \cdot \vdash M : A_1 \wedge A_2}$$

  which yields that for some C_1, M_1 and Δ_1

  $$(C \rhd M) \longmapsto (C_1 \rhd M_1) \quad \text{and } \vdash (C_1 \rhd M_1) : (\Delta, \Delta_1 \rhd A_1)$$

  and also that for some C_2, M_2 and Δ_2

  $$(C \rhd M) \longmapsto (C_2 \rhd M_2) \quad \text{and } \vdash (C_2 \rhd M_2) : (\Delta, \Delta_2 \rhd A_2)$$

  Even if we show that evaluation is deterministic (which shows $M_1 = M_2 = M'$ and $C_1 = C_2 = C'$), we have no way to reconcile Δ_1 and Δ_2 to a Δ' such that

  $$\vdash (C' \rhd M') : (\Delta, \Delta' \rhd A_1 \wedge A_2)$$

  because a new cell allocated in C_1 and C_2 may be assigned a different type in Δ_1 and Δ_2. It is precisely this observation which gives rise to the counterexample in the introduction.
- The absence of distributivity is critical in the inversion property for values $V : A$ for $A \trianglelefteq B_1 \to B_2$ which relies on the property that if $A_1 \wedge A_2 \trianglelefteq B_1 \to B_2$ then either $A_1 \trianglelefteq B_1 \to B_2$ or $A_2 \trianglelefteq B_1 \to B_2$.

The analysis above indicates that if we fix the cells in the store and disallow new allocations by removing the **ref** M construct, the language would be sound even without a value restriction as long as the **ref** type constructor is non-variant.

Overall, this proof is not much more difficult than the case without intersection types, but this is partially because we have set up our definitions very carefully.

3. BIDIRECTIONAL TYPE CHECKING

In the previous section we presented a pure type assignment system. This language is not directly suitable for programming: for example, type inference is impractical and principal types do not exist. In this section we present the core of a programming language which allows the programmer to specify types, along with an algorithm for type checking.

Previous proposals for explicitly typed languages with intersection types have required types to be given for each bound variable. This leads to a problem, since the programmer may wish to associate different types with a variable in different branches of the intersection introduction rule. This problem was solved by Reynolds [14] by introducing a type declaration including many alternative types. This was extended by Pierce [12] to a special form binding a type variable to one of a set of alternative types. Neither of these solutions is completely satisfactory. For example, there is still no way to directly specify that a curried function has a type like $(\mathsf{pos} \to \mathsf{nat} \to \mathsf{pos}) \wedge (\mathsf{nat} \to \mathsf{pos} \to \mathsf{pos})$. Another solution appears in [19] and uses an intersection introduction rule that explicitly includes two terms that are identical except for type information. This solution is used for a compiler intermediate language, and does not seem suitable for directly programming in, since maintaining nearly identical code fragments would be very tedious.

Our proposal instead distinguishes terms for which a type can be synthesized from terms which can be checked against a given type. This schema can type only normal forms, so we include an explicit type ascription $C : A$ which checks C against A and then yields A as the synthesized type for the whole expression. Unlike a cast operation in some dynamically typed languages, however, this ascription incurs no run-time overhead. In practice, we mostly ascribe types to function definitions which is a small step from ML where "good style" already suggests explicit ascription of signatures to structures.

3.1 Syntax

We use the same types as in the previous section. We omit locations from the language of terms, since they are created by the evaluation of terms $\mathbf{ref}\, C$ and should not directly occur in program source.

$$\begin{array}{ll}
\text{Inferable} & I ::= x \mid u \mid I\,C \mid \mathbf{!}\,I \mid I \mathbin{\mathbf{:=}} C \mid () \\
& \quad \mid \epsilon \mid I\,0 \mid I\,1 \\
& \quad \mid C{:}A \\
\text{Checkable} & C ::= I \mid \lambda x.\, C \mid \mathbf{let}\ x = I\ \mathbf{in}\ C \\
& \quad \mid \mathbf{fix}\ u.\, C \mid \mathbf{ref}\ C \\
& \quad \mid \mathbf{case}\ I\ \mathbf{of}\ \epsilon \Rightarrow C_1 \ \mid\ x\,0 \Rightarrow C_2 \ \mid\ y\,1 \Rightarrow C_3
\end{array}$$

We distinguish the following terms as values:

$$\begin{array}{lll}
\text{Inferable} & I_v & ::= \ x \mid () \mid \epsilon \mid I_v\,0 \mid I_v\,1 \mid C_v{:}A \\
\text{Checkable} & C_v & ::= \ I_v \mid \lambda x.\, C
\end{array}$$

We write I_{nv} for an inferable term not of the form I_v.

3.2 Typing

The typing judgments for inferable and checkable terms have the form:

$\Gamma \vdash I \uparrow A$	Term I has A as an inferable type.
$\Gamma \vdash C \downarrow A$	Term C checks against type A.

It is our intention that the rules for these judgments be interpreted algorithmically: given Γ and I, we can construct all derivations of $\Gamma \vdash I \uparrow A$ (also constructing A for each derivation); given Γ, C, and A we can check whether there is a derivation of $\Gamma \vdash C \downarrow A$. In an implementation, this could lead to unacceptable non-determinism and further transformations are required to obtain an efficient algorithm. We are currently investigating an algorithm which synthesizes all types of an inferable term and tracks applicable ones through the use of boolean constraints. In any case, the system is much simpler than full intersection types previously implemented in [6, 4] which have already demonstrated feasibility in most practical cases.

The following judgment non-deterministically extracts ordinary types from a type (possibly an intersection).

$A \uparrow B^o$ Type A has ordinary type B^o as a conjunct.

$$\frac{}{A^o \uparrow A^o}\ \mathsf{cnjct\_ord}$$

$$\frac{A_1 \uparrow B_1^o}{(A_1 \wedge A_2) \uparrow B_1^o}\ \mathsf{cnjct\_left}$$

$$\frac{A_2 \uparrow B_2^o}{(A_1 \wedge A_2) \uparrow B_2^o}\ \mathsf{cnjct\_right}$$

The type checking rules are given in Figure 3. Note that there are two rules to check terms I against a type only to avoid overlap with the rule to check terms C_v against a conjunction.

3.3 Soundness

We now show that the bi-directional checking algorithm is sound with respect to the type assignment system in the previous section. Completeness is somewhat trickier: while we can often directly annotate programs which are well-typed according to the type assignment system in Section 2, we may sometimes need to restructure it by lifting local function definitions. For example the following term

$$\lambda x.\ \mathbf{let}\ f = \lambda y.\ x\ \mathbf{in}\ f\,x$$

may be assigned the type $(\mathsf{nat} \to \mathsf{nat}) \wedge (\mathsf{pos} \to \mathsf{pos})$ but there is no annotation that we can add for the type of f that

$$\frac{x{:}A \text{ in } \Gamma}{\Gamma \vdash x \uparrow A}\,\textsf{ti\_var} \qquad \frac{u{:}A \text{ in } \Gamma}{\Gamma \vdash u \uparrow A}\,\textsf{ti\_var}'$$

$$\frac{\Gamma \vdash I \uparrow A \quad A \uparrow B_1 \to B_2 \quad \Gamma \vdash C \downarrow B_1}{\Gamma \vdash I\, C \uparrow B_2}\,\textsf{ti\_app} \qquad \frac{\Gamma \vdash C \downarrow A}{\Gamma \vdash (C : A) \uparrow A}\,\textsf{ti\_type}$$

$$\frac{\Gamma \vdash I \uparrow A \qquad A \uparrow B\,\mathbf{ref}}{\Gamma \vdash \mathbf{!}\,I \uparrow B}\,\textsf{ti\_get} \qquad \frac{\Gamma \vdash I \uparrow A \quad A \uparrow B\,\mathbf{ref} \quad \Gamma \vdash C \downarrow B}{\Gamma \vdash I \mathbin{\mathbf{:=}} C \uparrow \mathbf{unit}}\,\textsf{ti\_set} \qquad \frac{}{\Gamma \vdash () \uparrow \mathbf{unit}}\,\textsf{ti\_unit}$$

$$\frac{}{\Gamma \vdash \epsilon \uparrow \textsf{nat}}\,\textsf{ti\_e}$$

$$\frac{\Gamma \vdash I \uparrow \textsf{pos}}{\Gamma \vdash I\ 0 \uparrow \textsf{pos}}\,\textsf{ti\_z1} \qquad \frac{\Gamma \vdash I \uparrow \textsf{nat}}{\Gamma \vdash I\ 0 \uparrow \textsf{bits}}\,\textsf{ti\_z2} \qquad \frac{\Gamma \vdash I \uparrow \textsf{bits}}{\Gamma \vdash I\ 0 \uparrow \textsf{bits}}\,\textsf{ti\_z3}$$

$$\frac{\Gamma \vdash I \uparrow \textsf{pos}}{\Gamma \vdash I\ 1 \uparrow \textsf{pos}}\,\textsf{ti\_o1} \qquad \frac{\Gamma \vdash I \uparrow \textsf{nat}}{\Gamma \vdash I\ 1 \uparrow \textsf{pos}}\,\textsf{ti\_o2} \qquad \frac{\Gamma \vdash I \uparrow \textsf{bits}}{\Gamma \vdash I\ 1 \uparrow \textsf{bits}}\,\textsf{ti\_o3}$$

$$\frac{\Gamma \vdash I_v \uparrow A \qquad A \trianglelefteq B^{\circ}}{\Gamma \vdash I_v \downarrow B^{\circ}}\,\textsf{tc\_subsv} \qquad \frac{\Gamma \vdash I_{nv} \uparrow A \qquad A \trianglelefteq B}{\Gamma \vdash I_{nv} \downarrow B}\,\textsf{tc\_subsnv} \qquad \frac{\Gamma \vdash C_v \downarrow A \qquad \Gamma \vdash C_v \downarrow B}{\Gamma \vdash C_v \downarrow A \wedge B}\,\textsf{tc\_conj}$$

$$\frac{\Gamma, x{:}A \vdash C \downarrow B}{\Gamma \vdash \lambda x.\, C \downarrow A \to B}\,\textsf{tc\_lam} \qquad \frac{\Gamma \vdash I \uparrow A \qquad \Gamma, x{:}A \vdash C \downarrow B}{\Gamma \vdash \mathbf{let}\ x = I\ \mathbf{in}\ C \downarrow B}\,\textsf{tc\_let} \qquad \frac{\Gamma, u{:}A \vdash C \downarrow A}{\Gamma \vdash \mathbf{fix}\ u.\, C \downarrow A}\,\textsf{tc\_fix}$$

$$\frac{\Gamma \vdash C \downarrow A}{\Gamma \vdash \mathbf{ref}\ C \downarrow A\,\mathbf{ref}}\,\textsf{tc\_ref}$$

$$\frac{\Gamma \vdash I \uparrow B \quad B \uparrow \textsf{bits} \quad \Gamma \vdash C_1 \downarrow A \quad \Gamma, x{:}\textsf{bits} \vdash C_2 \downarrow A \quad \Gamma, y{:}\textsf{bits} \vdash C_3 \downarrow A}{\mathbf{case}\ I\ \mathbf{of}\ \epsilon \Rightarrow C_1 \mid x\ 0 \Rightarrow C_2 \mid y\ 1 \Rightarrow C_3 \downarrow A}\,\textsf{tc\_case1}$$

$$\frac{\Gamma \vdash I \uparrow B \quad B \uparrow \textsf{nat} \quad \Gamma \vdash C_1 \downarrow A \quad \Gamma, x{:}\textsf{pos} \vdash C_2 \downarrow A \quad \Gamma, y{:}\textsf{nat} \vdash C_3 \downarrow A}{\mathbf{case}\ I\ \mathbf{of}\ \epsilon \Rightarrow C_1 \mid x\ 0 \Rightarrow C_2 \mid y\ 1 \Rightarrow C_3 \downarrow A}\,\textsf{tc\_case2}$$

$$\frac{\Gamma \vdash I \uparrow B \quad B \uparrow \textsf{pos} \quad \Gamma, x{:}\textsf{pos} \vdash C_2 \downarrow A \quad \Gamma, y{:}\textsf{nat} \vdash C_3 \downarrow A}{\mathbf{case}\ I\ \mathbf{of}\ \epsilon \Rightarrow C_1 \mid x\ 0 \Rightarrow C_2 \mid y\ 1 \Rightarrow C_3 \downarrow A}\,\textsf{tc\_case3}$$

Figure 3: Type Checking Rules

allows us to typecheck the term with this type. This could also be solved, at some cost in elegance, by introducing a type enumeration construct as in [12]. We will not pursue this further; more experiments with our implementation are needed to decide whether this is warranted and intuitive to the programmer.

In order to relate the terms of this section with those of the previous one, we define the *erasure* function $|\cdot|$ in the obvious way, namely compositionally except that we remove type ascriptions:

$$|C : A| = |C|$$

LEMMA 6. *If $A \uparrow B$ then $A \leq B$.*

PROOF. By induction on the definition of $A \uparrow B$. $\square$

THEOREM 7 (SOUNDNESS OF TYPE CHECKING).

1. *If $\Gamma \vdash I \uparrow A$ then $\cdot; \Gamma \vdash |I| : A$.*
2. *If $\Gamma \vdash C \downarrow A$ then $\cdot; \Gamma \vdash |C| : A$.*

PROOF. By induction on the type checking derivations, using the above lemma. $\square$

4. PARAMETRIC POLYMORPHISM

We now show how to add parametric polymorphism to our language. Our approach is to consider parametric polymorphism as the infinite analog of intersection polymorphism, hence our typing and subtyping rules are infinite analogs of those in Section 2. In particular we include a value restriction on the introduction of polymorphism, and omit a rule for distributivity with the function type constructor. We do not consider type checking here, however we hope that some of the ideas from local-type inference [13] for parametric polymorphism may be incorporated into the type-checking algorithm of Section 3 to obtain a practical programming language.

4.1 Syntax

We add type variables and universal quantification to the types of Section 2. The terms and values of the language are as before.

$$\begin{array}{lll} \text{Types} & A & ::= A_1 \to A_2 \mid A\ \mathbf{ref} \mid \mathbf{unit} \\ & & \quad \mid \mathsf{bits} \mid \mathsf{nat} \mid \mathsf{pos} \mid A_1 \wedge A_2 \mid \alpha \mid \forall \alpha.\ A \end{array}$$

As usual, we allow tacit renaming of bound type variables. We write $\{B/\alpha\}A$ for the capture-avoiding substitution of B for α in A.

4.2 Subtyping

We add the following subtyping rules for parametric polymorphism, which are infinite analogs of the rules for intersection polymorphism.

$$\frac{}{\forall \alpha.\ A \leq \{B/\alpha\}A} \qquad \frac{A \leq B}{A \leq \forall \alpha.\ B}\ (\alpha \text{ not free in } A)$$

Analogously with intersection polymorphism, we omit the following distributivity rule since it leads to unsoundness when functions involve effects:

$$\frac{}{\forall \alpha.\ (A \to B) \leq A \to \forall \alpha.\ B}\ (\alpha \text{ not free in } A)$$

If we were to include this rule, our subtyping relation for the fragment containing functions and parametric polymorphism would be equivalent to that proposed by Mitchell [11]. Mitchell's subtyping relation has been shown to be undecidable [18, 16], but the techniques used in these proofs do not seem to apply directly in the absence of distributivity. We therefore do not know at present whether our subtyping relation is decidable.

As before, we present an more directed version of the subtyping relation. These rules are analogous to those for intersection polymorphism, and allow the proof of progress to be extended appropriately. Alas, they do not actually describe an algorithm, since they do not describe a method for choosing the type used to instantiate a polymorphic parameter when this is required. We therefore refer to our relation as *structural subtyping*.

We use the notation A° as before for an ordinary type, that is, that is neither an intersection nor a quantified type.

$A \trianglelefteq B$ Type A is structurally a subtype of type B.

The rules are given in Figure 4 We include a general reflexivity rule here even though it is only strictly needed for type variables and base types. This allows us to treat a subtyping derivation of a conclusion involving an unbound type variable as a judgment parametric in that variable, that is, we may instantiate the variable with any type and obtain a valid derivation.

As in the Section 2, we now prove some simple lemmas needed to show that structural and declarative subtyping coincide.

LEMMA 8 (STRUCTURAL SUBTYPING). *The structural subtyping judgment satisfies:*

1. *If $A \trianglelefteq B$ then $A \wedge A' \trianglelefteq B$ and $A' \wedge A \trianglelefteq B$.*
2. *If $\{A'/\alpha\}A \trianglelefteq B$ then $\forall \alpha.\ A \trianglelefteq B$.*
3. *If $A_1 \trianglelefteq A_2$ and $A_2 \trianglelefteq A_3$ then $A_1 \trianglelefteq A_3$.*

$$\frac{}{A \trianglelefteq A}$$

$$\frac{}{\mathsf{pos} \trianglelefteq \mathsf{nat}} \qquad \frac{}{\mathsf{pos} \trianglelefteq \mathsf{bits}} \qquad \frac{}{\mathsf{nat} \trianglelefteq \mathsf{bits}}$$

$$\frac{B_1 \trianglelefteq A_1 \qquad A_2 \trianglelefteq B_2}{A_1 \to A_2 \trianglelefteq B_1 \to B_2} \qquad \frac{A \trianglelefteq B \qquad B \trianglelefteq A}{A\ \mathbf{ref} \trianglelefteq B\ \mathbf{ref}}$$

$$\frac{A_1 \trianglelefteq B^o}{A_1 \wedge A_2 \trianglelefteq B^o} \qquad \frac{A_2 \trianglelefteq B^o}{A_1 \wedge A_2 \trianglelefteq B^o}$$

$$\frac{A \trianglelefteq B_1 \qquad A \trianglelefteq B_1}{A \trianglelefteq B_1 \wedge B_2}$$

$$\frac{\{A'/\alpha\}A \trianglelefteq B^o}{\forall\alpha.\ A \trianglelefteq B^o} \qquad \frac{A \trianglelefteq B}{A \trianglelefteq \forall\alpha.\ B}(\alpha \text{ not free in } A)$$

Figure 4: Structural Subtyping Rules

Proof. The first two are by simple inductions on the supertype B. For transitivity we proceed by structural induction and inversion on the two derivations. All cases are easy, although the following one is interesting:

$$\frac{\begin{array}{c}\mathcal{D}_1\\ A_1 \trianglelefteq A_2'\end{array}}{A_1 \trianglelefteq \forall\alpha.\ A_2'}(\alpha \text{ not in } A_1) \quad \text{and} \quad \frac{\begin{array}{c}\mathcal{D}_2\\ \{A'/\alpha\}A_2' \trianglelefteq A_3^o\end{array}}{\forall\alpha.\ A_2' \trianglelefteq A_3^o}$$

Here, we substitute A' for α in the derivation $\mathcal{D}_1$ to obtain an instance with conclusion $A_1 \trianglelefteq \{A'/\alpha\}A_2$ and then apply the induction hypothesis to this derivation (which has the same structure as $\mathcal{D}_1$) and $\mathcal{D}_2$ to obtain the required result. □

Theorem 9. *$A \trianglelefteq B$ if and only if $A \leq B$.*

Proof. In each direction, by induction on the given derivation, using the properties in the preceding lemma. □

4.3 Typing

We add the following typing rule for introducing parametric polymorphism with a value restriction. Instantiation of polymorphic types is done via the existing subsumption rule.

$$\frac{\Delta; \Gamma \vdash V : A}{\Delta; \Gamma \vdash V : \forall\alpha.\ A}\,\mathsf{tp\_para} \quad (\alpha \text{ not free in } \Delta, \Gamma)$$

The substitution lemma extends naturally to include this rule. We treat typing derivations with free type variables as parametric derivations, just as for subtyping.

The proof of progress is structured exactly as before. The main change is that we have additional cases in each part of the Typing Inversion Lemma for the new typing rule, each of which makes use of parametricity in a similar manner to the proof of transitivity of structural subtyping.

As before, all of our restrictions are needed in this proof:

- The value restriction on parametric polymorphism is needed because otherwise the induction hypothesis is applied to the premise of the introduction rule

  $$\frac{\Delta; \cdot \vdash M : A}{\Delta; \cdot \vdash M : \forall\alpha.\ A}\,\alpha \text{ not free in } \Delta$$

  which yields that for some C', M' and Δ'

  $$\begin{array}{l}(C \triangleright M) \longmapsto (C' \triangleright M')\\ \text{and } \vdash (C' \triangleright M') : (\Delta, \Delta' \triangleright A)\end{array}$$

  But α may appear in Δ', so we cannot apply the introduction rule for parametric polymorphism.

- The absence of distributivity is critical in the inversion property for values $V : A$ for $A \trianglelefteq B_1 \to B_2$ which relies on the property that if $\forall\alpha.\ A \trianglelefteq B_1 \to B_2$ then there exists A' such that $[A'/\alpha]A \trianglelefteq B_1 \to B_2$.

5. EXAMPLES

We now show some examples, primarily ones manipulating natural numbers represented as bit-strings without leading zeroes. We have already given some counterexamples to soundness in the absence of appropriate restrictions above, so the code below mostly illustrates bi-directional type-checking and the use of subtyping and intersection types in a source language to enforce data representation invariants.

We present examples as top-level definitions of the form **val** $x : A = M$, which should be interpreted as syntactic sugar for **let** $x = (M : A)$ **in** ... with open-ended scope. Here, $M : A$ is inferable, which means M only needs to be checkable. First, a function for incrementing a natural number.

$$\begin{array}{lcl}\mathbf{val}\ \mathit{inc} & : & (\mathsf{bits} \to \mathsf{bits}) \wedge (\mathsf{nat} \to \mathsf{pos})\\ & = & \mathbf{fix}\ \mathit{inc}.\ \lambda n.\ \mathbf{case}\ n\\ & & \mathbf{of}\ \epsilon \Rightarrow \epsilon\,\mathsf{1}\\ & & \mid\ x\,\mathsf{0} \Rightarrow x\,\mathsf{1}\\ & & \mid\ x\,\mathsf{1} \Rightarrow (\mathit{inc}\ x)\,\mathsf{0}\end{array}$$

Note that types for variables bound by λ or in the branches of the **case** expression do not have type labels. In fact, it would be difficult to allow for this, since the body of the **fix** expression will be checked twice: once against $\mathsf{bits} \to \mathsf{bits}$ and once against $\mathsf{nat} \to \mathsf{nat}$. In the first situation we analyze the body of the function with $n : \mathsf{bits}$, in the second with $n : \mathsf{nat}$. Further notice that subsumption allows us to derive additional types for inc, for example, using

$$\begin{array}{lcl}(\mathsf{bits} \to \mathsf{bits}) \wedge (\mathsf{nat} \to \mathsf{pos}) & \leq & \mathsf{nat} \to \mathsf{nat}\\ (\mathsf{bits} \to \mathsf{bits}) \wedge (\mathsf{nat} \to \mathsf{pos}) & \leq & \mathsf{pos} \to \mathsf{pos}\end{array}$$

Finally we remark that ascribing simply $\mathit{inc} : \mathsf{nat} \to \mathsf{nat}$ instead of $\mathsf{nat} \to \mathsf{pos}$ would have been insufficient: in order to see that the result of the last case of the function definition is a valid natural number, we need to know that the result of the recursive call $(\mathit{inc}\ x)$ is positive.

Our second example is binary addition.

$$\begin{array}{lcl} \textbf{val}\ \mathit{plus} & : & (\mathsf{nat} \to \mathsf{nat} \to \mathsf{nat}) \\ & & \wedge\ (\mathsf{pos} \to \mathsf{nat} \to \mathsf{pos}) \\ & & \wedge\ (\mathsf{nat} \to \mathsf{pos} \to \mathsf{pos}) \\ & = & \textbf{fix}\ \mathit{plus}.\ \lambda n.\ \lambda m.\ \textbf{case}\ n \\ & & \textbf{of}\ \epsilon \Rightarrow m \\ & & \mid\ x\,0 \Rightarrow \textbf{case}\ m \\ & & \qquad\quad \textbf{of}\ \epsilon \Rightarrow x\,0 \\ & & \qquad\quad \mid\ y\,0 \Rightarrow (\mathit{plus}\ x\ y)\,0 \\ & & \qquad\quad \mid\ y\,1 \Rightarrow (\mathit{plus}\ x\ y)\,1 \\ & & \mid\ x\,1 \Rightarrow \textbf{case}\ m \\ & & \qquad\quad \textbf{of}\ \epsilon \Rightarrow x\,1 \\ & & \qquad\quad \mid\ y\,0 \Rightarrow (\mathit{plus}\ x\ y)\,1 \\ & & \qquad\quad \mid\ y\,1 \Rightarrow (\mathit{inc}\ (\mathit{plus}\ x\ y))\,0 \end{array}$$

Again, for this definition to type-check we need to know that $\mathit{inc} : \mathsf{nat} \to \mathsf{pos}$. Application of *plus* to arguments of type bits which can not be shown to have type nat or pos is prohibited by type-checking. In this example, a weaker type such as $(\mathsf{nat} \to \mathsf{nat} \to \mathsf{nat}) \wedge (\mathsf{pos} \to \mathsf{pos} \to \mathsf{pos})$ would have sufficed, but given subsequent calls to *plus* less information about its behavior.

Next, we show how type-checking can catch errors which could not be caught without subtyping:

$$\begin{array}{lcl} \textbf{val}\ \mathit{double} & : & (\mathsf{nat} \to \mathsf{nat}) \wedge (\mathsf{pos} \to \mathsf{pos}) \\ & = & \lambda x.\ x\,0 \end{array}$$

will fail to type-check for $x : \mathsf{nat}$. Indeed, $\mathit{double}\,\epsilon$ evaluates to $\epsilon\,0$ which is not a valid representation of zero. We leave the simple fix to the reader.

We can also write and type-check a function to standardize arbitrary bit-strings by erasing leading zeroes.

$$\begin{array}{lcl} \textbf{val}\ \mathit{stdize} & : & \mathsf{bits} \to \mathsf{nat} \\ & = & \textbf{fix}\ \mathit{stdize}.\ \lambda b.\ \textbf{case}\ b \\ & & \textbf{of}\ \epsilon \Rightarrow \epsilon \\ & & \mid\ x\,0 \Rightarrow \textbf{case}\ \mathit{stdize}\ x \\ & & \qquad\quad \textbf{of}\ \epsilon \Rightarrow \epsilon \\ & & \qquad\quad \mid\ y\,0 \Rightarrow y\,0\,0 \\ & & \qquad\quad \mid\ y\,1 \Rightarrow y\,1\,0 \\ & & \mid\ x\,1 \Rightarrow (\mathit{stdize}\ x)\,1 \end{array}$$

The following example shows how references and intersections may be used together in a useful and sound way, and also demonstrates the syntactic structuring that is sometimes needed for typechecking. Each call to the function *count* with some initial value n generates a counter of type $\textbf{unit} \to \mathsf{nat}$ or $\textbf{unit} \to \mathsf{pos}$. This counter can be called repeatedly, successively returning n, $n+1$, ... The type of *count* reflects that the counter is always positive if it is initialized with a positive number. count' is an auxiliary function that allows the type of the internal reference to be specified explicitly for the purpose of typechecking.

$$\begin{array}{lcl} \textbf{val}\ \mathit{count}' & : & (\mathsf{nat}\ \textbf{ref} \to (\textbf{unit} \to \mathsf{nat}))\ \wedge \\ & & (\mathsf{pos}\ \textbf{ref} \to (\textbf{unit} \to \mathsf{pos})) \\ & = & \lambda c.\ \lambda x. \\ & & \quad \textbf{let}\ r = !c\ \textbf{in} \\ & & \quad \textbf{let}\ y = (c{:=}\mathit{inc}\ r)\ \textbf{in}\ r \end{array}$$

$$\begin{array}{lcl} \textbf{val}\ \mathit{count} & : & (\mathsf{nat} \to (\textbf{unit} \to \mathsf{nat}))\ \wedge \\ & & (\mathsf{pos} \to (\textbf{unit} \to \mathsf{pos})) \\ & = & \lambda n.\ \mathit{count}'\ (\textbf{ref}\ n) \end{array}$$

Finally, an example not connected to refinements, but more general uses of intersection types and parametric polymorphism which are accommodated by our system:

$$\begin{array}{lcl} \textbf{val}\ \omega & : & \forall\alpha.\ \forall\beta.\ ((\alpha \to \beta) \wedge \alpha) \to \beta \\ & = & \lambda x.\ x\ x \end{array}$$

6. CONCLUSION AND FUTURE WORK

We are in the process of implementing a refinement-type checker for a conservative extension of Standard ML based on the algorithm in Section 3. To avoid much of the non-determinism in this algorithm, we synthesize all types for an inferable term, and generate boolean constraints to represent the choice between them. We can then detect type errors by checking the satisfiability of the accumulated constraints, using an efficient representation such as Binary Decision Diagrams.

The combination of parametric polymorphism with intersection types should be investigated further. In particular it seems that the local type inference of [13] could be integrated with our bi-directional checking for intersection types. In our work on refinement types for ML we place the restriction that the only refinement of an ML type variable is a single corresponding refinement type variable. This simplifies the situation greatly and allows the refinement type checker to be guided by the type derivation constructed using ordinary ML type inference.

Our present examples do not indicate this, but it is conceivable that the value restriction rejects too many natural programs whose evaluation can be seen not to have effects and could therefore be typed more loosely without compromising safety. In that case we could generalize our approach from a value restriction to a *valuability* restriction, as proposed by Harper and Stone [8]. This would classify some functions as being total and effect-free so that they can essentially be treated as values for the purpose of type-checking. Interestingly, if we make such function types *subtypes* of the ordinary ML function types, then intersections once again may arise naturally during type-checking. We intend to investigate this further.

Acknowledgments. We would like to thank the anonymous referees for helpful suggestions and gratefully acknowledge discussions with Bob Harper regarding the subject of this paper.

7. REFERENCES

[1] A. Aiken, E. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *Twenty-First ACM Symposium on Principles of Programming Languages (POPL'94)*, pages 163–173, Portland, Oregon, Jan. 1994.

[2] V. Breazu-Tannen, T. Coquand, C. Gunter, and A. Scedrov. Inheritance as implicit coercion. *Information and Computation*, 93:172–221, 1991.

[3] L. Cardelli. Type systems. In A. B. Tucker, Jr., editor, *The Handbook of Computer Science and Engineering*, chapter 103, pages 2208–2236. CRC Press, 1997.

[4] R. Davies. A practical refinement-type checker for Standard ML. In M. Johnson, editor, *Algebraic Methodology and Software Technology Sixth International Conference (AMAST'97)*, pages 565–566, Sydney, Australia, Dec. 1997. Springer-Verlag LNCS 1349.

[5] R. Davies. Practical refinement-type checking. Thesis Proposal, Carnegie Mellon University Computer Science Department, Nov. 1997. `http://www.cs.cmu.edu/~rowan/papers/proposal.ps`

[6] T. Freeman. *Refinement Types for ML*. PhD thesis, Carnegie-Mellon University, Mar. 1994. Available as technical report CMU-CS-94-110.

[7] T. Freeman and F. Pfenning. Refinement types for ML. In *Proceedings of the SIGPLAN '91 Symposium on Language Design and Implementation, Toronto, Ontario*, pages 268–277. ACM Press, June 1991.

[8] R. Harper and C. Stone. A type-theoretic interpretation of Standard ML. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language, and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000.

[9] X. Leroy. Polymorphism by name. In *Twentieth ACM Symposium on Principles of Programming Languages(POPL'93)*, pages 220–231, Charleston, South Carolina, January 1993.

[10] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[11] J. C. Mitchell. Polymorphic type inference and containment. *Information and Computation*, 76(2/3):211–249, 1988.

[12] B. C. Pierce. Intersection types and bounded polymorphism. *Mathematical Structures in Computer Science*, 7(2):129–193, Apr. 1997. Summary version appeared in Conference on Typed Lambda Calculi and Applications, March 1993, pp. 346–360. Preliminary version available as University of Edinburgh technical report ECS-LFCS-92-200.

[13] B. C. Pierce and D. N. Turner. Local type inference. In *The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'98)*, pages 252–265, San Diego, California, 1998. Full version to appear in *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2000.

[14] J. C. Reynolds. Preliminary design of the programming language Forsythe. Technical Report CMU-CS-88-159, Carnegie Mellon University, Pittsburgh, Pennsylvania, June 1988.

[15] J. C. Reynolds. Design of the programming language Forsythe. Technical Report CMU-CS-96-146, Carnegie Mellon University, Pittsburgh, Pennsylvania, June 1996.

[16] J. Tiuryn and P. Urzyczyn. The subtyping problem for second-order types is undecidable. In *Proceedings, 11^{th} Annual IEEE Symposium on Logic in Computer Science*, pages 74–85, New Brunswick, New Jersey, 27–30 July 1996. IEEE Computer Society Press.

[17] M. Tofte. Type inference for polymorphic references. *Information and Computation*, 89:1–34, November 1990.

[18] J. B. Wells. The undecidability of Mitchell's subtyping relation. Technical Report 95-019, Boston University, Boston, Massachusetts, December 1995.

[19] J. B. Wells, A. Dimock, R. Muller, and F. Turbak. A typed intermediate language for flow-directed compilation. In *TAPSOFT'97: Theory and Practice of Software Development, Proc. 7th International Joint Conference CAAP/FASE*, pages 757–771, Lille, France, Apr. 1997. Springer-Verlag LNCS 1214.

[20] A. K. Wright. Simple imperative polymorphism. *Information and Computation*, 8:343–55, 1995.

[21] A. K. Wright and R. Cartwright. A practical soft type system for Scheme. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, pages 250–262, Orlando, Florida, June 1994.

[22] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115:38–94, 1994. Preliminary version is available as Technical Report TR91-160, Rice University.