WORKING DRAFT: Refinement-Type Checking via Quantified Boolean Formulae

Rowan Davies and Frank Pfenning
Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213, U.S.A.

rowan@cs.cmu.edu and fp@cs.cmu.edu
Telephone: +1 412 268-6343 *

March 8, 2001

Abstract

Refinement types allow many more properties of programs to be expressed and statically checked than conventional type systems. We present a practical algorithm for refinement-type checking for an ML-like language which reduces the problem to checking Quantified Boolean Formulae.

1 Introduction

```
- This section is mostly taken from another (unpublished) paper, and is
  only here as a starting point.
- We should mention the Forsythe algorithm, and the algorithms
  used by Aiken et al.
```

The advantages of statically-typed programming languages are well known, and have been described many times (e.g. see [Car97]). However, conventional type systems for realistic programming languages do not capture all of the interesting properties of a program. For example, when programming with lists, it is common to have the invariant that a particular result is not an empty list, but generally there is no easy way to express this in the type system. Further, conventional type systems generally only allow a single property to be expressed for any particular part of a program, when sometimes there are many important properties. For example, a particular function may have two important invariants: that the result is a non-empty list if the input is a non-empty list, and that the result is an empty list if the input is an empty list.

When we want to capture more properties of a program to allow better code optimization, the standard solution to this problem is to use some form of program analysis, of which there are many different varieties. Some recent work has used automatic program analyses to obtain

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

some of the benefits of static typing, namely catching errors and supporting code maintenance and modularity, when the language is dynamically typed [FFK+96], or when the static type system of the language cannot express the properties of interest [FA97]. Unfortunately, these analyses must make approximations, since the underlying problems are undecidable, and often it is difficult for the programmer to determine whether an apparent error found by the analysis is due to these approximations or due to an actual error in their code.

In this paper we present results which indicate that another approach to this problem can be used to build practical tools. This solution is to add a more detailed level of types called *refinement types* to a statically typed language. Each ordinary type may then be refined by many different refinement types, which we also call *sorts* in accordance with the use of this term in order-sorted algebras [DG94]. Refinement types have been studied previously [FP91, Fre94, Pfe93], and allow many more properties of programs to be expressed and checked than conventional type systems. For example, empty and non-empty lists could be defined as refinements of the type list, in which case we also have a sort for functions which map non-empty lists to non-empty lists, which is a refinement of the type of functions from lists to lists. This allows more programming errors to be caught at compile time, and extends all the advantages of static typing, including support for modularity, code maintenance, and code optimization, to a much wider class of properties of programs.

To allow more than one property to be expressed for a particular part of a program, sorts include an intersection operator $\wedge$, which allows a sort $R \wedge S$ to be formed from two sorts R and S which refine the same type. Numerous variants of intersection types (also called conjunctive types) have been studied in the literature [CDCV81]. In our setting, the presence of intersection allows the construction of a principal sort for each expression, given only its type. Base sorts refining the same base type are naturally ordered by inclusion, which is extended in a standard way to the full type hierarchy. The character of the resulting system is quite different from record subtyping.

Previous work on sorts has demonstrated that they are a very useful addition to languages as diverse as the functional programming language Standard ML [FP91, Fre94] and the logical framework LF [Pfe93]. Part of that work considered algorithms for sort inference, but this turns out to be problematic because common programs often satisfy many accidental properties which must be reflected in the inferred sort. Further, there is a huge combinatorial explosion in the number of refinements and the size of principal sorts as we move to more complicated (especially higher-order) types. Experiments with refinement types systems have thus been limited to small prototypes.

The main contribution of this paper is a practical solution to this problem, based on an algorithm for *sort checking* instead of inference which requires some explicit sort annotations and which reduces the problem to checking Quantified Boolean Formulae. We prove that this algorithm is sound and complete with respect to a simple declarative system.

We expect that the basic approach could also be extended to allow sort checking of other languages, for example the logic programming language Elf [Pfe91], which is based on LF.

For a motivating example of the use of sorts in ML, see Appendix A, which includes part of one of our experiments. The full code for these experiments and various other examples of sorts in ML are available electronically via `http://www.cs.cmu.edu/~rowan/sorts.html`, and other examples have appeared in [FP91]. Unfortunately, space limitations do not allow us to provide enough motivating examples in this summary, particularly since the need for sorts is not very convincing in small examples.

2 An Inefficient Bidirectional Sort Checker

```
- This section is based on another paper, and some is only here
  as a starting point.
- I haven't yet decided whether to have types and sorts, or just sorts,
  so the following is a bit muddled in places.
  A compromise might be to have only types, but with a ``compatibility''
  restriction on intersection.
- Adding a sort for ``zero'' makes the practical need for unversal constraints
  more evident.  So would adding products and/or adding pattern matching.
```

In the previous paper [**?**] 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. We then presented the core of a programming language which allows the programmer to specify types, along with an algorithm for type checking which includes excessive backtracking and often would require solving of many similar subproblems. Here we avoid both these problems by generating constraints as Quantified Boolean Formulae.

Our language 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 sort ascription $C \triangleright R$ which checks C against R and then yields R as the synthesized sort 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.

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 [**?**], we can also easily admit a "by name" **let** expression.

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 $\epsilon\mathsf{110}$.

$$
\begin{array}{rl}
\text{Types} & A ::= A_1 \to A_2 \mid \mathsf{bits} \\
\text{Sorts} & R ::= R_1 \to R_2 \mid R_1 \wedge R_2 \mid \mathsf{bits} \mid \mathsf{nat} \mid \mathsf{pos} \mid \mathsf{zero} \mid \mathsf{bot} \\
\text{Inferable} & I ::= x \mid u \mid I\,C \mid \epsilon \mid I\,\mathsf{0} \mid I\,\mathsf{1} \mid C \triangleright A \\
\text{Checkable} & C ::= I \mid \lambda x.\, C \mid \mathbf{let}\ x = I\ \mathbf{in}\ C \mid \mathbf{fix}\ u.\, C \\
& \quad \mid \mathbf{case}\ I\ \mathbf{of}\ \epsilon \Rightarrow C_1 \ \mathsf{I}\ x\,\mathsf{0} \Rightarrow C_2 \ \mathsf{I}\ y\,\mathsf{1} \Rightarrow C_3
\end{array}
$$

```
- Fix the below paragraph
```

We use A, B for types, a, b for base types, R, S for sorts and M, N for terms. We assume that a variable x can be declared at most once in a context. Bound variables may be renamed tacitly. We omit leading $\cdot$'s from contexts, and write Γ, Γ' for the result of appending two contexts. 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 use the notation $\mathcal{D} :: J$ to indicate that $\mathcal{D}$ is a derivation of judgement J.

We distinguish the following terms as values:

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

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

2.2 Refinement

```
- Do we want refinements or general intersection types in this paper?
- This subsection currently doesn't fit with the rest of the section.
```

The refinement judgement has the form $R :: A$ Sort R refines type A.

$$\frac{}{r^a :: a}\ \mathsf{rf\_base} \qquad \frac{R :: A \qquad S :: B}{R \rightarrow S :: A \rightarrow B}\ \mathsf{rf\_arrow} \qquad \frac{R :: A \qquad S :: A}{R \wedge S :: A}\ \mathsf{rf\_conj}$$

A direct consequence of this definition is that every sort refines at most one type. We extend the refinement notation pointwise to $\Delta :: \Gamma$ for contexts Δ and Γ which bind the same variables. Note that the refinement relation is *not* a subtyping (or subsorting) relation and is therefore neither co- nor contra-variant in the usual sense. Subsorting is introduced in the next section.

We say that a sort R is *well-formed* if it refines some type A. In what follows, we will only interested in well-formed sorts, and so when we use the term "sort" we implicitly mean "well-formed sort". We say that two sorts are *compatible* if they refine the same type. We say that a sort S is a *conjunct* of sort R if $S = R$, or (inductively) if $R = R_1 \wedge R_2$ and S is a conjunct of either R_1 or R_2. We write *cnjs* R for the set of conjuncts of R which are not intersection sorts.

2.3 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. Eliminating this non-determinism was the original motivation for work that lead to this paper.

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 below. 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. [1]

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

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

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

$$\frac{\Gamma \vdash I \uparrow A \qquad A\, \mathsf{0} \Uparrow B}{\Gamma \vdash I\, \mathsf{0} \uparrow B}\,\mathsf{ti\_z} \qquad \frac{\Gamma \vdash I \uparrow A \qquad A\, \mathsf{1} \Uparrow B}{\Gamma \vdash I\, \mathsf{1} \uparrow B}\,\mathsf{ti\_o}$$

$$\frac{\Gamma \vdash I_v \uparrow A \qquad A \trianglelefteq B^o}{\Gamma \vdash I_v \downarrow B^o}\,\mathsf{tc\_subsv} \qquad \frac{\Gamma \vdash I_{nv} \uparrow A \qquad A \trianglelefteq B}{\Gamma \vdash I_{nv} \downarrow B}\,\mathsf{tc\_subsnv}$$

$$\frac{\Gamma \vdash C_v \downarrow A \qquad \Gamma \vdash C_v \downarrow B}{\Gamma \vdash C_v \downarrow A \wedge B}\,\mathsf{tc\_conj}$$

[1] *I changed these slightly from our ICFP paper: we didn't handle the case where the argument to a constructor is inferred to have an intersection type. This isn't so critical until we have something like* zero $\wedge$ pos.

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

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

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

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

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

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

$$\frac{}{\textsf{bits 0} \Uparrow \textsf{bits}} \qquad \frac{}{\textsf{nat 0} \Uparrow \textsf{bits}} \qquad \frac{}{\textsf{pos 0} \Uparrow \textsf{pos}} \qquad \frac{}{\textsf{zero 0} \Uparrow \textsf{bits}} \qquad \frac{}{\textsf{bot 0} \Uparrow \textsf{bot}}$$

$$\frac{}{\textsf{bits 1} \Uparrow \textsf{bits}} \qquad \frac{}{\textsf{nat 1} \Uparrow \textsf{pos}} \qquad \frac{}{\textsf{pos 1} \Uparrow \textsf{pos}} \qquad \frac{}{\textsf{zero 1} \Uparrow \textsf{pos}} \qquad \frac{}{\textsf{bot 1} \Uparrow \textsf{bot}}$$

$$\frac{S_1\, \textsf{0} \Uparrow S_1' \qquad S_2\, \textsf{0} \Uparrow S_2'}{(S_1 \wedge S_2)\, \textsf{0} \Uparrow (S_1' \wedge S_2')} \qquad \frac{S_1\, \textsf{1} \Uparrow S_1' \qquad S_2\, \textsf{1} \Uparrow S_2'}{(S_1 \wedge S_2)\, \textsf{1} \Uparrow (S_1' \wedge S_2')}$$

3 An Algorithm for Sort Checking via QBF

In this section we present an algorithm based on that in the previous section but which avoids checking a subexpression many times. It achieves this by generating constraints which are Quantified Boolean Formula.

3.1 Syntax

Sets of Boolean Variables	$\overline{\chi}$	$::=$	$\cdot \mid \overline{\chi}, \chi$
Sorts	R	$::=$	$r \mid R_1 \to R_2 \mid R_1 \wedge R_2$
Variable Sorts	S	$::=$	$R \mid \chi\, S_1\, S_2$
Lattice Sorts	ρ	$::=$	$r \mid \rho_1 \wedge \rho_2$
Formulae	ϕ	$::=$	$\top \mid \bot \mid \chi\, \phi_1\, \phi_2 \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \phi_1 \subset \phi_2 \mid \forall \chi.\, \phi \mid \exists \chi.\, \phi$
Contexts	Γ	$::=$	$\cdot \mid \Gamma, x \triangleright S$

Here $\chi\, \phi_1\, \phi_2$ represents "if χ then ϕ_1 else ϕ_2", and similarly for $\chi\, S_1\, S_2$.

3.2 Subsorting

The constraints version of the subsorting algorithm takes two variable sorts and returns a constraint ϕ which is equivalent to the truth of the subsorting. We make use of three auxillary judgments.

$S_1 \trianglelefteq S_2 \setminus \phi$	Variable sort S_1 is a sub-sort of variable sort S_2 under constraint ϕ.
$\rho \sim r$	Lattice sort ρ is equivalent to base sort r.
$r_1 \sqsubseteq r_2$	Base sort r_1 is below r_2.
$R \uparrow \overline{\chi}.\, S_1 \Rightarrow S_2$	Sort R has a conjunct with argument sort S_1 and result sort S_2 for each instantiation of $\overline{\chi}$.

$$\frac{R_1 \uparrow \overline{\chi}_1.\, S_{11} \Rightarrow S_{12} \qquad R_2 \uparrow \overline{\chi}_2.\, S_{21} \Rightarrow S_{22} \qquad S_{21} \trianglelefteq S_{11} \setminus \phi_1 \qquad S_{12} \trianglelefteq S_{22} \setminus \phi_2}{R_1 \trianglelefteq R_2 \setminus \forall \overline{\chi}_2.\, \exists \overline{\chi}_1.\, \phi_1 \wedge \phi_2}$$

$$\frac{\rho_1 \sim r_1 \qquad \rho_2 \sim r_2 \qquad r_1 \sqsubseteq r_2}{\rho_1 \trianglelefteq \rho_2 \setminus \top} \qquad \frac{\rho_1 \sim r_1 \qquad \rho_2 \sim r_2 \qquad r_1 \not\sqsubseteq r_2}{\rho_1 \trianglelefteq \rho_2 \setminus \bot}$$

$$\frac{S_1 \trianglelefteq S_1' \setminus \phi_1 \qquad S_2 \trianglelefteq S_2' \setminus \phi_2}{\chi\, S_1\, S_2 \trianglelefteq \chi\, S_1'\, S_2' \setminus \chi\, \phi_1\, \phi_2}$$

$$\frac{\nexists S_1' S_2'.\, S' \equiv \chi\, S_1'\, S_2' \qquad S_1 \trianglelefteq S' \setminus \phi_1 \qquad S_2 \trianglelefteq S' \setminus \phi_2}{\chi\, S_1\, S_2 \trianglelefteq S' \setminus \chi\, \phi_1\, \phi_2}$$

$$\frac{R \trianglelefteq S_1' \setminus \phi_1 \qquad R \trianglelefteq S_2' \setminus \phi_2}{R \trianglelefteq \chi\, S_1'\, S_2' \setminus \chi\, \phi_1\, \phi_2}$$

$$\frac{}{r \sim r} \qquad \frac{\rho_1 \sim r_1 \qquad \rho_2 \sim r_2}{\rho_1 \wedge \rho_2 \sim r_1 \sqcap r_2}$$

$$\frac{}{R_1 \to R_2 \uparrow (\cdot).\, R_1 \Rightarrow R_2}$$

$$\frac{R_1 \uparrow \overline{\chi}_1.\, S_{11} \Rightarrow S_{12} \qquad R_2 \uparrow \overline{\chi}_2.\, S_{21} \Rightarrow S_{22}}{R_1 \wedge R_2 \uparrow (\chi, \overline{\chi}_1, \overline{\chi}_2).\, (\chi S_{11} S_{21}) \Rightarrow (\chi S_{12} S_{22})}\ (\chi \text{ not in } \overline{\chi}_1, \overline{\chi}_2)$$

Now, for correctness with respect to the less algorithmic system we would like to prove the following:

Lemma 1 *Suppose $\rho_1 \sim r_1$ and $\rho_2 \sim r_2$. Then $\rho_1 \leq \rho_2$ iff $r_1 \sqsubseteq r_2$.*

Boolean Substitutions $\Psi \;::=\; \cdot \mid \Psi, [\top/\chi] \mid \Psi, [\bot/\chi]$

We write $\Psi{:}\overline{\chi}$ to indicate that Ψ is a ground boolean substitution for the variables in $\overline{\chi}$. We write $\Psi\, S$ to indicate the result of instantiating S with Ψ defined as follows:

$$
\begin{array}{rcl}
\Psi\,(\chi\, S_1\, S_2) & = & \begin{cases} S_1 & \text{if } \Psi \text{ contains } [\top/\chi] \\ S_2 & \text{otherwise} \end{cases} \\
\Psi R & = & R
\end{array}
$$

Lemma 2 *Suppose $R \uparrow \overline{\chi}.\, S_1 \Rightarrow S_2$. Then*

1. *If $\Psi{:}\overline{\chi}$ then $R \uparrow (\Psi S_1) \rightarrow (\Psi S_2)$.*
2. *If $R \uparrow R_1 \rightarrow R_2$ then for some $\Psi{:}\overline{\chi}$ we have $R_1 \equiv (\Psi S_1)$ and $R_2 \equiv (\Psi S_2)$.*

Theorem 3 *If $S_1 \trianglelefteq S_2 \setminus \phi$ and Ψ contains all boolean variables in S_1 and S_2 then $\Psi\phi$ is true iff $\Psi S_1 \leq \Psi S_2$.*

3.3 Sorting

The algorithmic sorting judgments with constraints have the form:

$\Gamma \vdash I \uparrow \overline{\chi}.\, S \setminus \phi$	Term I is inferred to have sort S for each $\overline{\chi}$ under constraint ϕ.
$\Gamma \vdash C \downarrow S \setminus \phi$	Term C checks against sort S under constraint ϕ.

We make use of the following additional judgments:

$S\,\mathsf{0} \uparrow S'$	Constructor $\mathsf{0}$ maps variable sort S to S'
$S\,\mathsf{1} \uparrow S'$	Constructor $\mathsf{1}$ maps variable sort S to S'
$S \bowtie P \uparrow \overline{\chi}.\, \Gamma \setminus \phi$	Variable sort S when matched with pattern P yields the contexts obtained by instantiating $\overline{\chi}$ in Γ under constraint ϕ.
$\chi\, \Gamma_1\, \Gamma_2 \uparrow \Gamma$	Distributing χ into Γ_1 and Γ_2 yields Γ.

Here the third judgment is more general than we need since we only have a fixed, single level pattern match construct in our language. Thus the only patterns we have are:

$$\text{Patterns} \quad P \quad ::= \quad \epsilon \mid x\,\mathsf{0} \mid x\,\mathsf{1}$$

$$\frac{x{\triangleright}S \text{ in } \Gamma}{\Gamma \vdash x \uparrow (\cdot).\, S \setminus \top}\,\textsf{ti\_var} \qquad \frac{u{:}S \text{ in } \Gamma}{\Gamma \vdash u \uparrow (\cdot).\, S \setminus \top}\,\textsf{ti\_var}'$$

$$\frac{\Gamma \vdash I \uparrow \overline{\chi}_1.\, S \setminus \phi_1 \qquad S \uparrow \overline{\chi}_2.\, S_1 \Rightarrow S_2 \qquad \Gamma \vdash C \downarrow S_1 \setminus \phi_2}{\Gamma \vdash I\, C \uparrow (\overline{\chi}_1, \overline{\chi}_2).\, S_2 \setminus (\phi_1 \wedge \phi_2)}\,\textsf{ti\_app}$$

$$\frac{\Gamma \vdash C \downarrow R \setminus \phi}{\Gamma \vdash (C \triangleright R) \uparrow (\cdot).\, R \setminus \phi}\,\textsf{ti\_sort}$$

$$\frac{}{\Gamma \vdash \epsilon \uparrow (\cdot).\ \mathsf{zero} \setminus \top}\ \mathsf{ti\_e}$$

$$\frac{\Gamma \vdash I \uparrow \overline{\chi}.\ S \setminus \phi \qquad S\,\mathsf{0} \uparrow S'}{\Gamma \vdash I\,\mathsf{0} \uparrow \overline{\chi}.\ S' \setminus \phi}\ \mathsf{ti\_z} \qquad \frac{\Gamma \vdash I \uparrow \overline{\chi}.\ S \setminus \phi \qquad S\,\mathsf{1} \uparrow S'}{\Gamma \vdash I\,\mathsf{1} \uparrow \overline{\chi}.\ S' \setminus \phi}\ \mathsf{ti\_o}$$

$$\frac{\Gamma \vdash I \uparrow \overline{\chi}.\ S_1 \setminus \phi_1 \qquad S_1 \trianglelefteq S_2 \setminus \phi_2}{\Gamma \vdash I \downarrow S_2 \setminus \exists\overline{\chi}.\ \phi_1 \wedge \phi_2}\ \mathsf{tc\_subs} \qquad \frac{S \uparrow \overline{\chi}.\ S_1 \Rightarrow S_2 \qquad \Gamma, x{\triangleright}S_1 \vdash C \downarrow S_2 \setminus \phi}{\Gamma \vdash \lambda x.\ C \downarrow S \setminus \forall\overline{\chi}.\ \phi}\ \mathsf{tc\_lam}$$

$$\frac{\Gamma \vdash I \uparrow \overline{\chi}.\ S_1 \setminus \phi_1 \qquad \Gamma, x{\triangleright}S_1 \vdash C \downarrow S_2 \setminus \phi_2}{\Gamma \vdash \mathbf{let}\ x = I\ \mathbf{in}\ C \downarrow S_2 \setminus \exists\overline{\chi}.\ \phi_1 \wedge \phi_2}\ \mathsf{tc\_let} \qquad \frac{\Gamma, u{\triangleright}S \vdash C \downarrow S \setminus \phi}{\Gamma \vdash \mathbf{fix}\ u.\ C \downarrow S \setminus \phi}\ \mathsf{tc\_fix}$$

$$\frac{\begin{array}{c}\Gamma \vdash I \uparrow \overline{\chi}_1.\ S_1 \setminus \phi_1 \qquad S_1 \bowtie \epsilon \uparrow \cdot \setminus \phi_2 \qquad S_1 \bowtie x\,\mathsf{0} \uparrow x{\triangleright}S_3 \setminus \phi_3 \qquad S_1 \bowtie y\,\mathsf{1} \uparrow y{\triangleright}S_4 \setminus \phi_4 \\ \Gamma \vdash C_1 \downarrow S \setminus \phi_2' \qquad \Gamma, x{:}S_3 \vdash C_2 \downarrow S \setminus \phi_3' \qquad \Gamma, y{:}S_4 \vdash C_3 \downarrow S \setminus \phi_4'\end{array}}{\Gamma \vdash (\mathbf{case}\ I\ \mathbf{of}\ \epsilon \Rightarrow C_1 \mid x\,\mathsf{0} \Rightarrow C_2 \mid y\,\mathsf{1} \Rightarrow C_3) \downarrow S \setminus \exists\overline{\chi}_1.\ \phi_1 \wedge (\phi_2 \subset \phi_2') \wedge (\phi_3 \subset \phi_3') \wedge (\phi_4 \subset \phi_4')}\ \mathsf{tc\_case}$$

$$\frac{\rho \sim \mathsf{bits}}{\rho\,\mathsf{0} \uparrow \mathsf{bits}} \qquad \frac{\rho \sim \mathsf{nat}}{\rho\,\mathsf{0} \uparrow \mathsf{bits}} \qquad \frac{\rho \sim \mathsf{pos}}{\rho\,\mathsf{0} \uparrow \mathsf{pos}} \qquad \frac{\rho \sim \mathsf{zero}}{\rho\,\mathsf{0} \uparrow \mathsf{bits}} \qquad \frac{\rho \sim \mathsf{bot}}{\rho\,\mathsf{0} \uparrow \mathsf{bot}}$$

$$\frac{\rho \sim \mathsf{bits}}{\rho\,\mathsf{1} \uparrow \mathsf{bits}} \qquad \frac{\rho \sim \mathsf{nat}}{\rho\,\mathsf{1} \uparrow \mathsf{pos}} \qquad \frac{\rho \sim \mathsf{pos}}{\rho\,\mathsf{1} \uparrow \mathsf{pos}} \qquad \frac{\rho \sim \mathsf{zero}}{\rho\,\mathsf{1} \uparrow \mathsf{pos}} \qquad \frac{\rho \sim \mathsf{bot}}{\rho\,\mathsf{1} \uparrow \mathsf{bot}}$$

$$\frac{S_1\,\mathsf{0} \uparrow S_1' \qquad S_2\,\mathsf{0} \uparrow S_2'}{(\chi\, S_1\, S_2)\,\mathsf{0} \uparrow (\chi\, S_1'\, S_2')} \qquad \frac{S_1\,\mathsf{1} \uparrow S_1' \qquad S_2\,\mathsf{1} \uparrow S_2'}{(\chi\, S_1\, S_2)\,\mathsf{1} \uparrow (\chi\, S_1'\, S_2')}$$

$$\frac{\rho \sim \mathsf{bits}}{\rho \bowtie \epsilon \uparrow \cdot \setminus \top} \qquad \frac{\rho \sim \mathsf{nat}}{\rho \bowtie \epsilon \uparrow \cdot \setminus \top} \qquad \frac{\rho \sim \mathsf{pos}}{\rho \bowtie \epsilon \uparrow \cdot \setminus \bot}$$

$$\frac{\rho \sim \mathsf{zero}}{\rho \bowtie \epsilon \uparrow \cdot \setminus \top} \qquad \frac{\rho \sim \mathsf{bot}}{\rho \bowtie \epsilon \uparrow \cdot \setminus \bot}$$

$$\frac{\rho \sim \mathsf{bits}}{\rho \bowtie x\, \mathsf{0} \uparrow x \triangleright \mathsf{bits} \setminus \top} \qquad \frac{\rho \sim \mathsf{nat}}{\rho \bowtie x\, \mathsf{0} \uparrow x \triangleright \mathsf{pos} \setminus \top} \qquad \frac{\rho \sim \mathsf{pos}}{\rho \bowtie x\, \mathsf{0} \uparrow x \triangleright \mathsf{pos} \setminus \top}$$

$$\frac{\rho \sim \mathsf{zero}}{\rho \bowtie x\, \mathsf{0} \uparrow x \triangleright \mathsf{bot} \setminus \bot} \qquad \frac{\rho \sim \mathsf{bot}}{\rho \bowtie x\, \mathsf{0} \uparrow x \triangleright \mathsf{bot} \setminus \bot}$$

$$\frac{\rho \sim \mathsf{bits}}{\rho \bowtie x\, \mathsf{1} \uparrow x \triangleright \mathsf{bits} \setminus \top} \qquad \frac{\rho \sim \mathsf{nat}}{\rho \bowtie x\, \mathsf{1} \uparrow x \triangleright \mathsf{nat} \setminus \top} \qquad \frac{\rho \sim \mathsf{pos}}{\rho \bowtie x\, \mathsf{1} \uparrow x \triangleright \mathsf{nat} \setminus \top}$$

$$\frac{\rho \sim \mathsf{zero}}{\rho \bowtie x\, \mathsf{1} \uparrow x \triangleright \mathsf{bot} \setminus \bot} \qquad \frac{\rho \sim \mathsf{bot}}{\rho \bowtie x\, \mathsf{1} \uparrow x \triangleright \mathsf{bot} \setminus \bot}$$

$$\frac{S_1 \bowtie P \uparrow \Gamma_1 \setminus \phi_1 \qquad S_2 \bowtie P \uparrow \Gamma_2 \setminus \phi_2 \qquad \chi\, \Gamma_1\, \Gamma_2 \uparrow \Gamma}{\chi\, S_1\, S_2 \bowtie P \uparrow \Gamma \setminus \chi\, \phi_1 \phi_2}$$

$$\frac{}{\chi\ \cdot\ \cdot \uparrow \cdot} \qquad \frac{\chi\, \Gamma_1\, \Gamma_2 \uparrow \Gamma}{\chi\, (\Gamma_1, x \triangleright S_1)\ (\Gamma_2, x \triangleright S_2) \uparrow \Gamma, x \triangleright \chi\, S_1\, S_2}$$

Now, we would like to prove the following:

Lemma 4 *If $S\,\mathsf{0} \uparrow S'$ and Ψ contains all boolean variables in S and S' and $R \sim \Psi S$ and $R\,\mathsf{0} \Uparrow R'$ then $R' \sim \Psi S'$ (and similarly for $\mathsf{1}$).*

Lemma 5

4 Conclusion and Future Work

```
- We should mention the Forsythe algorithm, and the algorithms
  used by Aiken, Fahndrich, et al. (particularly for conditional types).
  Also, Heintze, and Flanagan (Rice).
- Work on algorithms for QBF.  Compare the likely size of formula
  with our approach.
- Also work on verifcation via model checking and theorem proving, particularly
  for software.  There's a survey article by Ed Clarke and Jeanette Wing
  called ``Formal Methods: State of the Art and Future Directions''
  (Tech Report CMU-CS-96-178).
```

```
- Also, perhaps mention the recent work with Bob on modal refinements
  for effects, and relate checking for this to model checking for
  modal and temporal logics (QBF with modalities?).
- Conclude by saying that our approach leverages from the advantages
  of both type systems and model checking.  Presenting the system
  as types allows the programmer to understand it easily, and
  provide annotations that are not much more than would be needed in
  ML anyway.  Adding more ``logical'' constructs to types such as
  conjunction allows more accurate properties to be expressed.
- Union types?  Expose the boolean quantified types and
  \chi conditional types to the programmer?
  Add types like ``c R'' (termination problems if not careful).    This
  starts to look a bit like Aiken et al.'s ``Soft-typing with Conditional Types'',
  particularly if we try to do type inference (which has the usual
  problems with fixed points).
- Vague idea: we could try to represent base lattices and their
  subtyping by replacing base sorts with unions of the
  ``unsplittable'' elements s_1, ..., s_n.  Then, we could
  quantify over base sorts using boolean variables X_1, ..., X_n
  corresponding to (X_1 s_1 bot) union ... union (X_n s_n bot).
  More abstractly, we could represent base sorts as tuples of
  booleans.  E.g. bitstr = (T, T)  nat = (T, F)  pos = (F, F)
  Then subtyping is implication, intersection is logical
  conjunction.
```

5 Acknowledgements

We gratefully acknowledge discussions with Ed Clarke regarding the subject of this paper. [2]

References

[Car97] Luca Cardelli. Type systems. In Allen B. Tucker Jr., editor, *The Handbook of Computer Science and Engineering*, chapter 103, pages 2208–2236. CRC Press, 1997.

[CDCV81] Mario Coppo, Maria Dezani-Ciancaglini, and B. Venneri. Functional character of solvable terms. *Zeitschrift für mathematische Logic und Grundlagen der Mathematik*, 27:45–58, 1981.

[DG94] Razvan Diaconescu and Joseph Goguen. An Oxford survey of order sorted algebra. *Mathematical Structures in Computer Science*, 4:363–392, 1994.

[FA97] Manuel Fähndrich and Alexander Aiken. Program analysis using mixed term and set constraints. In *International Static Analysis Symposium*, September 1997. To appear.

[2] *Also briefly discussions with John Reynolds, Robert Harper.*

[FF97] Cormac Flanagan and Matthias Felleisen. Componential set-based analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1997.

[FFK+96] Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Stephanie Weirich, and Matthias Felleisen. Catching bugs in the web of program invariants. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 23–32, May 1996.

[FP91] Tim Freeman and Frank 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.

[Fre94] Tim Freeman. *Refinement Types for ML*. PhD thesis, Carnegie-Mellon University, March 1994. Available as technical report CMU-CS-94-110.

[Hei94] Nevin Heintze. Set based analysis of ML programs. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 306–317, 1994.

[LG88] John M. Lucassen and David K. Gifford. Polymorphic effect systems. In *Proceedings of the Fifteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, San Diego, 1988*, San Diego, California, 1988. ACM.

[Pfe91] Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

[Pfe93] Frank Pfenning. Refinement types for logical frameworks. In Herman Geuvers, editor, *Informal Proceedings of the 1993 Workshop on Types for Proofs and Programs*, pages 285–299, Nijmegen, The Netherlands, May 1993. University of Nijmegen.

[Pie93] Benjamin C. Pierce. Intersection types and bounded polymorphism. In M. Bezem and J.F. Groote, editors, *Proceedings of the International Conference on Typed Lambda Calculi and Applications, TLCA'93*, pages 346–360, Utrecht, The Netherlands, March 1993. Springer-Verlag LNCS 664. A version will also appear in the journal MSCS.

[Rey88] John C. Reynolds. Preliminary design of the programming language Forsythe. Technical Report CMU-CS-88-159, Carnegie Mellon University, Pittsburgh, Pennsylvania, June 1988.

[Rey91] John C. Reynolds. The coherence of languages with intersection types. In T. Ito and A. R. Meyer, editors, *International Conference on Theoretical Aspects of Computer Software*, pages 675–700, Sendai, Japan, September 1991. Springer-Verlag LNCS 526.

A Extended Example: Red/Black Trees

The following is part of the code for the red/black trees experiment, including all of the `datasort` declarations, and the code for the function to insert an element in a tree. We check the critical invariant that no red-node has a red child. See Section ?? for more details.

```
type key = int
type 'a entry = key * 'a

datatype 'a dict =                                  (* general dictionaries *)
  Empty | Black of 'a entry * 'a dict * 'a dict     (* Empty is considered black *)
| Red of 'a entry * 'a dict * 'a dict

datasort 'a rbt =  Empty | Black of 'a entry * 'a rbt * 'a rbt   (* red/black trees *)
                 | Red of 'a entry * 'a bt * 'a bt
     and 'a bt = Empty | Black of 'a entry * 'a rbt * 'a rbt     (* black root node *)
datasort 'a red = Red of 'a entry * 'a bt * 'a bt                (* red root node *)

datasort 'a badRoot                  (* invariant possibly violated at the root *)
   = Empty | Black of 'a entry * 'a rbt * 'a rbt | Red of 'a entry * 'a bt * 'a bt
   | Red of 'a entry * 'a red * 'a bt | Red of 'a entry * 'a bt * 'a red

datasort 'a badLeft             (* invariant possibly violated at the left child *)
   = Empty | Black of 'a entry * 'a rbt * 'a rbt | Red of 'a entry * 'a bt * 'a bt
   | Black of 'a entry * 'a badRoot * 'a rbt

datasort 'a badRight            (* invariant possibly violated at the right child *)
   = Empty | Black of 'a entry * 'a rbt * 'a rbt | Red of 'a entry * 'a bt * 'a bt
   | Black of 'a entry * 'a rbt * 'a badRoot

fun restore_right ... = ...  (* code omitted for brevity *)
withsort restore_right :> 'a badRight -> 'a rbt

fun restore_left ... = ...   (* code omitted for brevity *)
withsort restore_left :> 'a badLeft -> 'a rbt

fun insert (dict, entry as (key,datum:'a)) =
  let
    fun ins (Empty) = Red(entry, Empty, Empty)
      | ins (Red(entry1 as (key1, datum1), left, right)) =
        (case compare(key,key1)
           of EQUAL => Red(entry, left, right)
            | LESS => Red(entry1, ins left, right)
            | GREATER => Red(entry1, left, ins right))
      | ins (Black(entry1 as (key1, datum1), left, right)) =
        (case compare(key,key1)
           of EQUAL => Black(entry, left, right)
            | LESS => restore_left (Black(entry1, ins left, right))
            | GREATER => restore_right (Black(entry1, left, ins right)))
    withsort ins :> 'a rbt -> 'a badRoot & 'a bt -> 'a rbt
  in              (* the second conjuct is needed for the recursive cases *)
    case ins dict
      of Red (t as (_, Red _, _)) => Black t (* re-color *)
       | Red (t as (_, _, Red _)) => Black t (* re-color *)
       | dict => dict                        (* depend on sequential matching *)
  end
withsort insert :> 'a rbt * 'a entry -> 'a rbt
```