

Practical Refinement-Type Checking

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

Technical summary submitted to POPL'98 *

July 19, 1997

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 in a λ -calculus enriched with refinement-type annotations. We prove that our basic algorithm is sound and complete, and show that every term which has a refinement type can be annotated as required by our algorithm.

Our positive experience with an implementation of an extension of this algorithm to the full core language of Standard ML demonstrates that refinement types can be a practical program development tool in a realistic programming language. The required refinement type definitions and annotations are not much of a burden and serve as formal, machine-checked explanations of code invariants which otherwise would remain implicit.

1 Introduction

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 $\&$, which allows a sort $R \& S$ 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. We prove that this algorithm is sound and complete with respect to a simple declarative system, and that every well-sorted program can be annotated appropriately. Especially the latter turned out to be technically more difficult than anticipated, although we cannot give many details in this summary.

To demonstrate that our algorithm is efficient and the annotations practical, we describe experiments with an implementation based on this approach which performs sort checking for an appropriate extension of the core language of Standard ML. The critical issue here turned out to be the treatment of pattern matching, since the correctness of many programs relies on the fact that patterns in a case expression are matched sequentially. Our extension of ML is conservative, so our sort checker could be added to a compiler without causing problems with existing programs. 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 Basic Refinement-Type Checking

In this section we present a simply-typed λ -calculus with **let**, and give declarative sorting rules for it. We include **let**-terms here so that later we can annotate them with the sort of the bound variable, even though operationally they are equivalent to β -redices. We also include the **fix** construct for expressing recursion to demonstrate that this does not essentially complicate sort checking, in contrast to sort inference where it is a major bottleneck. We use explicitly typed (but not explicitly sorted) λ and **fix** forms, because we assume that type inference has been completed before sort checking is attempted. Much of the basic presentation of sorts here is based on previous work [FP91], and so we omit some details from this technical summary for the sake of brevity.

2.1 Syntax and Typing

We assume that there are some base types, denoted by a, b and that each base type is refined by a finite number of base sorts, denoted by r^a, s^b .

Types	$A ::= a \mid A_1 \rightarrow A_2$
Sorts	$R ::= r^a \mid R_1 \rightarrow R_2 \mid R_1 \ \& \ R_2$
Terms	$M ::= x \mid \lambda x:A. M \mid M_1 M_2$ $\mid \mathbf{let} \ x = M_1 \ \mathbf{in} \ M_2 \mid \mathbf{fix} \ x:A. M$
Type Contexts	$\Gamma ::= \cdot \mid \Gamma, x:A$
Sort Contexts	$\Delta ::= \cdot \mid \Delta, x \triangleright R$

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 .

The typing judgement for this calculus has the form

$$\Gamma \vdash M : A \quad \text{Term } M \text{ has type } A \text{ in context } \Gamma.$$

The typing rules are all completely standard. A direct consequence of these rules is that any well-typed term has a unique type and typing derivation. This language does not include polymorphism, which we discuss informally in Section 3.2.

2.2 Refinement

The refinement judgement has the form

$R :: A$ Sort R refines type A .

$$\frac{}{r^a :: a} \text{rf_base} \quad \frac{R :: A \quad S :: B}{R \rightarrow S :: A \rightarrow B} \text{rf_arrow} \quad \frac{R :: A \quad S :: A}{R \& S :: A} \text{rf_conj}$$

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 be 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 $R = R_1 \& R_2$ 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 Subsorting

We assume that subsorting between base sorts is defined by the relation $\rho^a \preceq r^a$, where ρ^a is a set of base sorts, indicating their intersection. In practice this relation will be defined in terms of whatever method is used to introduce base sorts, which here we choose to keep abstract. Our sort checker for SML provides a concrete instance of this: we allow **datasort** refinements of simple SML **datatype** definitions by regular tree grammars, with the relation $\rho^a \preceq r^a$ holding exactly when the elements of ρ^a define sets of values whose intersection is contained in the set defined by r^a . For the moment, we allow any base subsorting relation which satisfies the some appropriate consistency conditions.

The subsorting judgement has the form:

$R \leq S$ Sort R is a sub-sort of S , where R and S must be compatible.

The subsorting rules present a minor variation for other systems with intersection types such as [Rey91].

$$\frac{\text{cnjs } R \preceq s^a}{R \leq s^a} \text{sub_base} \quad \frac{}{R \leq R} \text{sub_reflex} \quad \frac{R_1 \leq R_2 \quad R_2 \leq R_3}{R_1 \leq R_3} \text{sub_trans}$$

$$\frac{}{R \& S \leq R} \text{sub_conjL1} \quad \frac{}{R \& S \leq S} \text{sub_conjL2} \quad \frac{R \leq S \quad R \leq S'}{R \leq S \& S'} \text{sub_conjR}$$

$$\frac{R' \leq R \quad S \leq S'}{R \rightarrow S \leq R' \rightarrow S'} \text{sub_arrow} \quad \frac{}{(R \rightarrow S) \& (R \rightarrow S') \leq R \rightarrow (S \& S')} \text{sub_dist}$$

If $R \leq S$ and $S \leq R$ then we say R and S are *equivalent* sorts. Our main motivation for introducing this definition is that there are only a finite number of refinements of each type modulo sort equivalence, which can be proved by induction on types.

2.4 Sorting

The sorting assignment judgement has the form:

$\Delta \vdash M \triangleright R$ Term M has sort R in context Δ .

The sorting rules are very similar to those for a system with general intersection types (e.g., see [Rey91]). Here each abstraction and fixed point includes the type assigned to the variable during type inference, so the choice of the sort for the variable is restricted to refinements of this type.

$$\begin{array}{c}
\frac{x \triangleright R \text{ in } \Delta}{\Delta \vdash x \triangleright R} \text{srt_var} \quad \frac{\Delta, x \triangleright R \vdash M \triangleright S \quad R :: A}{\Delta \vdash \lambda x:A. M \triangleright R \rightarrow S} \text{srt_lam} \quad \frac{\Delta \vdash M \triangleright R \rightarrow S \quad \Delta \vdash N \triangleright R}{\Delta \vdash M N \triangleright S} \text{srt_app} \\
\\
\frac{\Delta \vdash M \triangleright R \quad \Delta, x \triangleright R \vdash N \triangleright S}{\Delta \vdash \mathbf{let} x = M \mathbf{in} N \triangleright S} \text{srt_let} \quad \frac{\Delta, x \triangleright R \vdash M \triangleright R \quad R :: A}{\Delta \vdash \mathbf{fix} x:A. M \triangleright R} \text{srt_fix} \\
\\
\frac{\Delta \vdash M \triangleright R \quad \Delta \vdash M \triangleright S}{\Delta \vdash M \triangleright R \& S} \text{srt_conj} \quad \frac{\Delta \vdash M \triangleright R \quad R \leq S}{\Delta \vdash M \triangleright S} \text{srt_subs}
\end{array}$$

Theorem 1 (Refinement) *If $\Delta \vdash M \triangleright R$, $R :: A$ and $\Delta :: \Gamma$ then $\Gamma \vdash M : A$.*

Proof: By induction over the structure of the sorting derivation. □

Theorem 2 (Principal sorts) *If $\Delta \vdash M \triangleright R$ then there exists S such that $\Delta \vdash M \triangleright S$ and for all R' such that $\Delta \vdash M \triangleright R'$ we have $S \leq R'$.*

Proof: Choose S to be the intersection of all sorts S' (up to equivalence) such that $\Delta \vdash M \triangleright S'$. Clearly S is unique up to sort equivalence, and we call S the *principal* sort of M with respect to Δ . □

Our calculus satisfies the usual substitution lemmas, subject reduction and sort preservation theorems with respect to β -reduction, δ -reduction (for **let**) and μ -reduction (unrolling of **fix**). Details are omitted here due to space constraints.

2.5 Algorithmic subsorting

The sorting and subsorting rules given above are quite intuitive, but they do not specify a strategy for checking a term. We address the more difficult issue of algorithmic sort checking later. Due to space limitations, we omit the details of algorithmic subsorting judgment $R \leq S$, since variations of it have been considered previously [Fre94] and we do not consider it a major contribution of this paper.

2.6 Annotation

Designing an algorithmic form of the sort assignment judgement is harder than for the subsorting judgement. In particular, there is no obvious way to choose R in the `srt_lam` and `srt_let` rules. Freeman [Fre94] has described and implemented one approach to this problem, based on techniques from abstract interpretation. Unfortunately, the resulting algorithms are not always efficient enough to be practical, particularly in the case of higher-order functions and when features like recursion are included to the language. Here we take an alternative approach, namely letting the programmer provide the required information. Our experience indicates that this approach leads to algorithms which are efficient, simple and predictable, without placing too much burden on the programmer.

To guarantee that enough information has been given by the programmer, we define a language with sort annotations on occurrences of `fix`, no β -redices, and sort annotations on all occurrences of `let` for which the principal sort of the corresponding subterm can not be easily synthesized. In the case of fixed-point expressions the sort annotation must be precise enough for all occurrences of the recursion variable in the body. In particular it is sometimes not enough to simply use the sort intended to express the externally visible properties of a function; one needs to generalize first. This is similar to the case where an induction hypothesis in a correctness proof needs to be strengthened.

The syntax for annotated terms is as follows:

$$\begin{array}{l} \text{Checkable terms } C ::= I \mid \lambda x:A. C \mid \mathbf{let } x = I \mathbf{ in } C \mid \mathbf{let } x \triangleright R = C_1 \mathbf{ in } C_2 \\ \text{Inference terms } I ::= x \mid IC \mid \mathbf{fix } x \triangleright R. C \end{array}$$

Our experience with sort checking existing Standard ML programs indicates that in practice a restricted language like this is very reasonable, and in fact the only modifications we had to make to these programs was to add the required sorts wherever a function was defined. However, to our initial surprise this is not true in general: sometimes well-sorted programs need to be rewritten slightly to generate an equivalent annotated program which can be sort-checked.

The annotation procedure is based on the following observations. Firstly, we notice that we can easily remove β -redices, by introducing a `let` so that the function is bound to a local variable. Then we expect that we can make use of the sorting derivation to generate the required sorts for `let`-definitions. This can not always be done by simply adding a sort to each `let`. Intuitively, this is because the body of a function may need to be sort-checked several times, once for each possible argument sort. If a `let`-binding occurs in the body of a function, it might thus be assigned different sorts each time the function body is checked and we cannot assign it a single sort. Using an intersection for the sort of the `let`-bound variable also does not work in general, since this would require the defined term to have *all* sorts independently of the different assignments of sorts to the term's free variables. For example, consider the following term (for some base type a):

$$\lambda x:a. \mathbf{let } f = (\lambda y:a. x) \mathbf{ in } f x$$

There is an obvious derivation which shows this term has sort $(r^a \rightarrow r^a) \ \& \ (s^a \rightarrow s^a)$ for base sorts r^a and s^a , though there is no single sort annotation that we can add to the `let`, since the corresponding sub-derivations are of the judgements:

$$x \triangleright r^a \vdash \lambda y:a. x \triangleright r^a \rightarrow r^a \quad \text{and} \quad x \triangleright s^a \vdash \lambda y:a. x \triangleright s^a \rightarrow s^a$$

A similar problem was solved by Reynolds [Rey88] in a type system with general intersection types by introducing a type annotation including many alternative types. This was extended by Pierce [Pie93] to a special form binding a type variable to one of a set of alternative types.

Here we take a different approach which we find more intuitive, and avoid this problem without introducing new language features. We do this by using some β -expansions and δ -expansions during annotation, and a simple observational equivalence for functional programs in the case of **fix**. This turns each problematic term into a closed function, thus allowing the required sorts for its originally free variables to be included in the sort for the function. For the sake of simplifying what follows, we actually close over all free variables rather than just those which have different bindings in different branches of an instance of the `srt_conj` rule. For the example above, expanding and adding sorts yields:

$$\lambda x:a. \mathbf{let} f' \triangleright (r^a \rightarrow r^a \rightarrow r^a) \& \mathcal{X}(s^a \rightarrow s^a \rightarrow s^a) = (\lambda x:a. \lambda y:a. x) \mathbf{in} \mathbf{let} f = f' x \mathbf{in} f x$$

The following function on well-typed terms performs the required expansions, including replacement of β -redices. Note that this does not yet introduce sort annotations; it only prepares the term so that annotations are *always* possible.

$$\begin{aligned} |x|_{\Gamma} &= x \\ |\lambda x:A. M|_{\Gamma} &= \lambda x:A. |M|_{\Gamma, x:A} \\ |M N|_{\Gamma} &= |M|_{\Gamma} |N|_{\Gamma} \quad (M \neq \lambda x:A. M') \\ |(\lambda x:A. M) N|_{\Gamma} &= \mathbf{let} f = \mathit{abs}_{(\mathit{free}_{\Gamma} M)}(\lambda x:A. |M|_{\Gamma, x:A}) \\ &\quad \mathbf{in} (\mathit{app}_{(\mathit{free}_{\Gamma} M)} f) |N|_{\Gamma} \quad (f \text{ not free in } N) \\ |\mathbf{let} x = M \mathbf{in} N|_{\Gamma} &= \mathbf{let} f = \mathit{abs}_{(\mathit{free}_{\Gamma} M)}(|M|_{\Gamma}) \mathbf{in} \\ &\quad \mathbf{let} x = \mathit{app}_{(\mathit{free}_{\Gamma} M)} f \\ &\quad \mathbf{in} |N|_{\Gamma, x:A} \quad (\text{where } \Gamma \vdash M : A \text{ and } f \text{ not free in } N) \\ |\mathbf{fix} x:A. M|_{\Gamma} &= \mathbf{let} f = \mathbf{fix} f' : \mathit{arr}_{(\mathit{free}_{\Gamma} M)} A. \\ &\quad \mathit{abs}_{(\mathit{free}_{\Gamma} M)}(\mathbf{let} x = \mathit{app}_{(\mathit{free}_{\Gamma} M)} f' \mathbf{in} M) \\ &\quad \mathbf{in} \mathit{app}_{(\mathit{free}_{\Gamma} M)} f \quad (f, f' \text{ not free in } M) \end{aligned}$$

$$\begin{aligned} \mathit{free}_{(\cdot)} M &= \cdot \\ \mathit{free}_{(\Gamma, x:A)} M &= (\mathit{free}_{\Gamma} M), x:A \quad (\text{if } x \text{ free in } M) \\ \mathit{free}_{(\Gamma, x:A)} M &= \mathit{free}_{\Gamma} M \quad (\text{if } x \text{ not free in } M) \\ \mathit{arr}_{(\cdot)} A &= A \\ \mathit{arr}_{(\Gamma, x:B)} A &= B \rightarrow (\mathit{arr}_{\Gamma} A) \\ \mathit{abs}_{(\cdot)} M &= M \\ \mathit{abs}_{(\Gamma, x:A)} M &= \lambda x:A. (\mathit{abs}_{\Gamma} M) \\ \mathit{app}_{(\cdot)} M &= M \\ \mathit{app}_{(\Gamma, x:A)} M &= (\mathit{app}_{\Gamma} M) x \end{aligned}$$

Theorem 3 (Expansion reduction) *If $\Gamma \vdash M : A$ then $|M|_{\Gamma}$ is observationally equivalent to M .*

Proof: By induction on the definition of $|\cdot|$. □

If M does not contain any occurrences of **fix**, then we have the slightly stronger result that $|M|_{\Gamma}$ reduces to M using only value reductions.

Theorem 4 (Expansion sorting) *If $\Delta \vdash M \triangleright R$ and $\Delta :: \Gamma$ then $\Delta \vdash |M|_{\Gamma} \triangleright R$.*

Proof: By induction on the definition of $|\cdot|$. The proof is constructive, and defines a corresponding function $|\cdot|$ on sorting derivations. \square

We can now use the sorting derivation constructed in the proof of the expansion-sorting theorem to complete the annotation process by adding sorts to **let** and **fix** terms as required. From the definition of $|\cdot|$, we see that any term in its range contains only occurrences of **fix** $x:A.M$ when M is closed, and only occurrences of **let** which bind a variable to a term which is either closed or inferable, i.e., of the form I after annotation. To generate the required sorts for the closed terms, we conjoin the corresponding sorts in each branch of the sorting derivation. This process defines a function $ann(M, \mathcal{D})$ where $\mathcal{D} :: \Delta \vdash M \triangleright R$, and the result of the function is a checkable term C . Due to space limitations, we omit a formal presentation of this function. We have proved that the resulting term has the required sort with respect to the appropriate declarative sorting rules for annotated terms, though this result is not an essential part of the development in this summary.

2.7 Algorithmic sort checking

We now give our algorithm for sort checking annotated terms, presented as a judgement with algorithmic rules, and with two ancillary judgements, one of which synthesizes the principal sort of a term of the form I , and one of which projects a function sort on an argument term. We also need the negation of the projection judgement. The idea of these judgements is that the sort of each variable in the context is always known, and to check whether a term has an intersection sort we simply check that it has each of the conjuncts. There is some small flexibility here as to exactly when intersections are broken down, and in practice we break them down as late as possible. We have omitted the typing rules for annotated terms, though they are completely standard, with the additional requirements that the sort annotations are required to refine the corresponding types.

$\Delta \vdash C \downarrow R$	Term C is determined to have sort R in context Δ , where $\Delta :: \Gamma, R :: A, \Gamma \vdash C : A$.
$\Delta \vdash I \uparrow R$	Term I has principal sort R in context Δ , where $\Delta :: \Gamma, R :: A$ and $\Gamma \vdash C : A$.
$\Delta \vdash R @ C \gg S$	Sort R when projected on term C yields result sort S , where $\Delta :: \Gamma, R :: A_1 \rightarrow A_2, \Gamma \vdash C : A_1$ and $S_2 :: A_2$.
$\Delta \vdash R @\!C$	Sort R can not be projected on term C , where $\Delta :: \Gamma, R :: A_1 \rightarrow A_2$ and $\Gamma \vdash C : A_1$.

$$\begin{array}{c}
\frac{\Delta \vdash C \downarrow R \quad \Delta \vdash C \downarrow S}{\Delta \vdash C \downarrow R \& S} \text{dn_conj} \qquad \frac{\Delta, x \triangleright R \vdash C \downarrow S \quad R :: A}{\Delta \vdash \lambda x:A. C \downarrow R \rightarrow S} \text{dn_lam} \\
\\
\frac{\Delta \vdash I \uparrow R \quad \Delta, x \triangleright R \vdash C \downarrow S}{\Delta \vdash \mathbf{let} \ x = I \ \mathbf{in} \ C \downarrow S} \text{dn_letA} \qquad \frac{\Delta \vdash C_1 \downarrow R \quad \Delta, x \triangleright R \vdash C_2 \downarrow S}{\Delta \vdash \mathbf{let} \ x \triangleright R = C_1 \ \mathbf{in} \ C_2 \downarrow S} \text{dn_letR} \\
\\
\frac{\Delta \vdash I \uparrow R \quad R \trianglelefteq S}{\Delta \vdash I \downarrow S} \text{dn_atom}
\end{array}$$

$$\frac{x \triangleright R \text{ in } \Delta}{\Delta \vdash x \uparrow R} \text{up_var} \quad \frac{\Delta \vdash I \uparrow R \quad \Delta \vdash R @ C \gg S}{\Delta \vdash IC \uparrow S} \text{up_app} \quad \frac{\Delta, x \triangleright R \vdash C \downarrow R}{\Delta \vdash \mathbf{fix} \ x \triangleright R. C \uparrow R} \text{up_fix}$$

$$\frac{\Delta \vdash C \downarrow R_1}{\Delta \vdash R_1 \rightarrow R_2 @ C \gg R_2} \text{aptm_arrow} \quad \frac{\Delta \vdash R @ C \gg S_2 \quad \Delta \vdash R' @ C \gg S'_2}{\Delta \vdash R \& R' @ C \gg S_2 \& S'_2} \text{aptm_conj}$$

$$\frac{\Delta \vdash R @ C \gg S_2 \quad \Delta \vdash R' \not\phi C}{\Delta \vdash R \& R' @ C \gg S_2} \text{aptm_conj1} \quad \frac{\Delta \vdash R \not\phi C \quad \Delta \vdash R' @ C \gg S'_2}{\Delta \vdash R \& R' @ C \gg S'_2} \text{aptm_conj2}$$

The following theorem shows that our algorithm is sound, by relating sort checking for an annotated term with the sorts of the term obtained by erasing the sort annotations (and replacing them with types for occurrences of \mathbf{fix}). This erasure process defines a function $erase(\cdot)$, which is a left inverse of the annotation function.

Theorem 5 (Algorithmic sort-checking soundness)

1. if $\Delta \vdash C \downarrow R$ then $\Delta \vdash erase(C) \triangleright R$.
2. if $\Delta \vdash I \uparrow R$ then $\Delta \vdash erase(I) \triangleright R$.
3. if $\Delta \vdash R @ C \gg S_2$ then $\Delta \vdash erase(C) \triangleright S_1$ with $R \leq S_1 \rightarrow S_2$.

Proof: By induction on the corresponding derivations, making use of inversion on the algorithmic form of subsorting. \square

Theorem 6 (Algorithmic sort-checking completeness)

1. if $\mathcal{D} :: \Delta \vdash M \triangleright R$ and $ann(M, \mathcal{D}) = C$ then $\Delta \vdash C \downarrow R$.
2. if $\mathcal{D} :: \Delta \vdash M \triangleright S$ and $ann(M, \mathcal{D}) = I$ then $\Delta \vdash I \uparrow R$ with $R \leq S$.
3. if $\mathcal{D} :: \Delta \vdash M \triangleright S_1$ and $R \leq S_1 \rightarrow S_2$ and $ann(M, \mathcal{D}) = C$ then $\Delta \vdash R @ C \gg S'_2$ with $S'_2 \leq S_2$.

Proof: By induction on the corresponding derivations, making use of the appropriate replacement lemma, and a lemma validating subsumption. \square

We have also proved a slightly stronger completeness theorem, stating that our sort-checking algorithm is complete with respect to the appropriate declarative sorting rules for annotated terms, though we omit the details here for space reasons.

3 Sort Checking for Standard ML

We now describe an implementation based on the algorithm given in the previous section, which performs sort checking for an appropriate extension of the core language of Standard ML. We then present our experience with this implementation.

Space does not permit us to formally describe all the interesting details of the extension of the sort checking algorithm. A formal presentation of sorts for a fragment of ML was given by Freeman [Fre94], though he did not address pattern matching and did not consider sort checking algorithms. Most of this can be adapted quite easily to build on the sort checking framework given in the previous section. Here we will informally describe our sort checker and the extensions to Standard ML required to support it. We will describe our method for handling pattern matching in some detail, since this is one of the key features that make our sort checker a practical tool, and this aspect has not been considered elsewhere.

3.1 Defining refinements

We allow sorts to be defined using `datasort` declarations, which are similar to ML `datatype` declarations, except that each value constructor may appear many times within a single definition. See the extended example in Appendix A for examples of `datasort` declarations.

We have the restriction that all value constructors in each definition of a sort constructor are associated with the type constructor which is being refined. Further, the arguments of these value constructors must refine the arguments in the definition of the type constructor. Also, for simplicity, in mutually recursive sort definitions the recursive references must have exactly the same parameters as the sort definition in which they appear, which does not appear to be too restrictive in practice.

For convenience, we automatically define a sort constructor for each type constructor in a `datatype` declaration, which has the same name as the type constructor and is obtained by essentially reading the `datatype` declaration as a `datasort` declaration. Similarly, we have built-in refinements of the built-in types which have the same name as the type they refine. We have not yet investigated mechanisms for defining other refinements of built-in types.

Our implementation analyzes the `datasort` declarations, and determines the subsorting relations $\rho^a \preceq r^a$ which hold between the defined sort constructors and their intersections, as well as the principal sorts of the value constructors, captured by the relation $c \triangleright R$. This process is much more complicated than it may appear at first, and the algorithms used are based on algorithms for regular tree grammars. A formal analysis of such algorithms is the subject of a Master's Thesis by Skalka [Ska97].

There are many obvious limitations of `datasort` declarations, in particular they cannot be used to define non-regular refinements, such as lists whose length is prime. They also only allow parametric refinements of parametric types, so for example we cannot define a refinement for ordinary lists of booleans whose head is `true`, although we can define a specialized type for lists of booleans and then refine this type.

3.2 Polymorphism

Following Freeman [Fre94], we allow refinements of ML polymorphic types, with the restriction that there is only a single sort variable refining each type variable. For convenience, this sort vari-

able has the same name as the type variable. We omit the details of sort checking in the presence of polymorphism, since we basically follow the same approach as Freeman. One important issue that we mention here is that we have principal sorts only with respect to a particular ML typing derivation which entails some loss of generality when compared to atomic subtyping [FM90]. This means that sometimes several instances of a generic polymorphic function need to be sort-checked separately. This shortcoming might be addressed in future work at the level of ML modules (ascribing several signatures to a structure) or by combining recent ideas from polymorphic subtyping inference [Reh97] with refinement types.

3.3 Pattern matching

There are few interesting choices to be made when extending the sorting rules to core Standard ML, the most difficult problems being presented by general pattern matching. A simple approach would be to ignore the sequential operational behavior of matching, and simply compare the sort of the subject of the match with the pattern of each case to obtain sorts for the bound variables, which can then be used to check the body of the case. Unfortunately, most real programs depend on the sequential nature of pattern matching, so many of the interesting properties of programs could not be checked using sorts if we adopted this approach. However, it is not clear how to capture the sequential nature of pattern matching using sorts in a way that is accurate, intuitive, and has an efficient implementation. Freeman [Fre94] avoided this issue by only considering a simple elimination form for constructed types, and arguing that pattern matching can be expanded into nested eliminations. Unfortunately, this expansion is not very practical, since it makes it hard for the programmer to understand the sorts associated with their program, and it is not always feasible since it may generate a huge number of cases.

Here we describe a direct approach to sorting pattern matching which accurately and intuitively captures the sequential nature of patterns. The basic idea of our approach is to generate a set of possible variable contexts for each pattern, and a description of the residual values which do not match the pattern. The body of each case must be checked under all possible contexts, and the residual of each case is the input set of the next case, with the residual of the final case empty. This approach may lead to a combinatorial explosion in the number of possible contexts. Even though it is not difficult to create pathological cases, we did not observe this explosion in our experiments. This may be due to a programmer's natural inclination to keep pattern matches as small and manageable as possible in order to remain convinced of their correctness.

According to this approach, sort checking should fail when the patterns do not cover the sort of the subject of a match. In order to obtain a conservative extension of Standard ML, we actually only issue a warning in this case. However, this warning should perhaps be taken more seriously than in ordinary Standard ML, since the use of sorts should allow the possible values for the subject of a match to be determined more accurately.

3.4 Sort constraints

In the Section 2 we used **let**-terms annotated with sorts to allow the programmer to specify the information needed during sort checking. In the implementation we instead allow sort constraints to appear anywhere that a type constraint could. We also include a special form **withsort** so that a sort constraint can be given for a **fun** declaration. We cannot always individually give sort

constraints for the arguments and the result of a function, as is sometimes done with types, since the desired sort may be an intersection.

We also automatically add default sort annotations when the program does not match the grammar for annotated programs. These default sorts correspond to the types assigned during type inference, and are certainly not the principal sorts. For example, if the type assigned to a function is `'a dict -> 'a dict` we add the sort annotation `'a dict -> 'a dict` if we cannot determine the intended sort otherwise. The automatic addition of these default sort annotations avoids the need to annotate sections of code for which no interesting sort checking is required, and makes our implementation of sort checking a conservative extension of Standard ML, i.e. sort checking always succeeds for well-typed (unsorted) Standard ML programs.

3.5 Experiments

In addition to many small functions, we have conducted three medium-size experiments with our sort checker. The absence of functors made it difficult to annotate larger code samples with sorts, but since sort checking time increases linearly with the size of the program, we judged these experiments to be representative of large-scale and pervasive use of refinement types in ML.

In the first experiment we added sorts to an existing implementation of red/black balanced binary trees to check the critical invariant that no red node has a red child. In this code the invariant is temporarily violated and then restored, and so the sort annotations had to use intersection sorts to specify the behavior of some functions both when their arguments satisfy the invariant and in when they slightly violate the invariant in certain ways. Part of the code from this experiment is included in Appendix A.

In the second experiment we added sorts to the parser of a recent re-implementation of the Elf logic programming language [Pfe91]. In this case we were able to check some complicated invariants involving an intermediate stack of unresolved operations which is used to resolve prefix, postfix and infix operations with precedence. Additionally, we were able to remove a lot of non-exhaustive match warnings by defining a sort for infinite streams as a refinement of the type of streams.

In the third example, we added sorts to implementation of the LF logical framework which used a more general datatype than necessary in order to avoid duplication of code for basic operations such as substitution. The sorts properly separate the object, type family and kind levels, and we were able to check that substitution and conversion to weak head normal form obeyed this separation.

During these experiments we were generally happy with the efficiency of our implementation. For the largest file, with around 400 lines of code, the time taken for sort checking was less than five seconds. We expect that our implementation can be made much more efficient, since it is currently rather simplistic in many places. Also the low level operations are based on the ML/Kit compiler, which was built without much consideration for efficiency. Additionally, we expect that the time taken to check a program will grow roughly linearly with its length, since each declaration in a module can be sort checked independently, using only the sort information in the context.

The sort annotations that were required in these experiments seemed to be quite reasonable in most cases, and now serve as mechanically checked documentation for this code. In fact, the existing comments in the code attempted to explain the invariants, but in the second example significant errors were found in these comments, and in the first example the comments were vague enough that they were not much help in constructing the precise invariants, which we needed for

sort checking. No errors were found in the code itself, but this is not surprising, since this code had already been tested extensively and proven correct “by hand” in the case of red/black trees. There were a few situations where it became tedious to add sort annotations that were obviously satisfied, so in future work we intend to consider whether some small amount of sort inference can be added to our sort checker.

With one exception, we did not need to alter our code in any way other than to add sort annotations and remove module level code. The exception is that in the parsing code, the stack of unresolved operations was represented as a list, which prevented the desired refinements from being defined, since any refinement of lists must be parametric on the element sort. This problem was solved by simply defining a specialized type for stacks.

4 Conclusion and Future Work

Our conclusion is that by focusing on sort checking rather than inference, we were able to build a very practical tool around refinement types for a real language. The technical contributions of this paper are the non-trivial soundness, completeness, and annotability results on a rich enough language to exhibit most of the critical issues except pattern matching.

Additionally, we illustrated some of the key elements involved in scaling our approach to the core language of Standard ML and presented the results of some experiments that involved programming with refinement types in ML. These experiments indicate that sort checking is fast enough to be practical, and that many interesting properties of programs can be specified and checked with only a relatively small effort to provide the required sort annotations.

We briefly compare our results with those obtained by others [FFK⁺96] using a different approach, but with similar goals. This work uses set-based analysis [Hei94] as a programming tool, and has demonstrated that this approach aids the construction of reliable programs in the context of the dynamically-typed language Scheme. This work is mostly concerned with obtaining some of the advantages of statically-typed languages, though in some cases the analysis determines more accurate information than could be captured in by the static type system of a language like ML. For example it may be determined that a variable will always be bound to a non-empty list. However, generally the analysis will not produce as accurate results as can be obtained with sort checking when the right `datasort` declarations and sort annotations are used, partially because the analysis can not express many of the properties corresponding to sorts involving intersections. Further, there is some evidence that this kind of analysis does not scale to very large programs [FF97], which has prompted consideration of the possibility of a typed module language for Scheme. By contrast, sort checking is very modular, and scales linearly to very large programs.

The main disadvantage of sort checking compared to program analyses is that the programmer must provide additional information to the sort checker in order to allow interesting properties to be checked. However, in some sense this is also one of the main advantages of our approach, since the sort checker need only check the properties specified by the programmer, and in the case that errors are found good feedback can be given to the programmer. The sort annotations which are required generally correspond to interesting invariants of a program, which the programmer should be aware of anyway, and the annotations also serve as mechanically checked documentation for these invariants. Program analyses also have the disadvantage that they must sometimes fail to prove correct properties, though it is generally hard to identify when this has happened. In contrast, we have an intuitive specification, in the style of a type system, for the behavior of sort

checking.

In future work we intend to extend sort checking to include the modules language of Standard ML. This extension would include the addition of sorts to signatures, and also requires some mechanism for specifying subsorting relations which hold between refinements of abstract types. We also intend to implement some compiler optimizations based on sort information. One simple such optimization is to remove a test resulting from compiling pattern matching when the sort information allows the result of the test to be determined. A more heavyweight optimization is to use optimized representations and functions for different refinements of a type, though this may lead to code explosion if not done carefully. We also intend to investigate whether some existing program analyses can be profitably reformulated using refinement types, for example, analysis of effects [LG88] or the exceptions analysis in [FA97] are good candidates.

5 Acknowledgements

We gratefully acknowledge discussions with John Boyland, Perry Cheng, Denis Dancanet, Manuel Fähndrich, Matthias Felleisen, Tim Freeman, Daniel Jackson, Leaf Petersen, Chris Skalka and Hong-Wei Xi regarding the subject of this paper.

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 Logik 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.
- [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.
- [FM90] You-Chin Fuh and Prateek Mishra. Type inference with subtypes. *Theoretical Computer Science*, 73:155–175, 1990.
- [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.
- [Reh97] Jacob Rehof. Minimal typings in atomic subtyping. In Neil Jones, editor, *Conference Record of the 24th Symposium on Principles of Programming Languages (POPL'97)*, pages 278–291, Paris, France, January 1997. ACM SIGPLAN-SIGACT.
- [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.
- [Ska97] Chris Skalka. Some decision problems for ML refinement types. Master's thesis, Carnegie-Mellon University, July 1997. To appear.

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 3.5 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 conjunct 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

```