Thesis Proposal:
Practical Refinement-type Checking

Rowan Davies

November 13, 1997

Abstract

One of the major benefits of statically-typed programming languages is that they significantly improve programmer productivity. An obvious reason for this is that they dramatically reduce the amount of time spent debugging by catching most common errors at compile time. A perhaps more important reason is that programmers can use the types to guide understanding of the structure of a piece of code, both during the development of the code, and during code maintenance. One proposal for increasing these benefits is to extend an existing statically-typed language so that each ordinary type is refined by a number of *refinement-types*, which allow many common program invariants to be expressed and checked. In the resulting system a part of a program which is assigned a particular type may also be assigned multiple refinements of that type. Previous work indicates that automatically inferring refinement-types is algorithmically impractical in general. However, if a programmer annotates their program with enough of the intended refinement-types the problem of checking the annotated program has been found to be much easier in some preliminary experiments.

The goal of this work is to demonstrate that refinement-type checking can be a practical and useful addition to a real programming language. To achieve this I intend to design an extension of Standard ML which is suitable for refinement-type checking, extend a real compiler with an efficient refinement-type checker, and demonstrate that this allows many common program invariants to be captured in a practical way.

1 Introduction

The advantages of statically-typed programming languages are well known, and have been described many times (e.g. see [Car97]). The type systems of many real programming languages allow much of the structure of programs to be captured. However, programs generally involve many invariants which can not be captured with these type systems. For example, when programming with lists, it is common to have the invariant that a particular result is not an empty list, which generally can not be expressed by the type system.

The standard approach to capturing such program invariants is to use program analyses, particularly when these invariants are needed to allow code optimization. Some recent work has also used automatic program analyses to obtain some of the benefits of static typing, namely catching errors and supporting code maintenance and modularity, particularly when the language is dynamically typed [FFK^{+}96]. Unfortunately, these analyses must make approximations, since the underlying problems are undecidable, and often it is tedious 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. This is particularly tedious when a piece of code is modified many times.

The work proposed here aims to demonstrate that another approach to capturing program properties can be used to build practical tools. This approach 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 extends all the advantages of static typing to a wide class of properties of programs, i.e. more errors are caught at compile time, understanding of programs is aided, and there is increased support for modularity, code maintenance, and code optimization.

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. For example, we may have a sort for functions which map empty lists to empty lists, and also map non-empty lists to non-empty lists. The operator & is taken directly from the numerous forms of intersection types (also called conjunctive types) which 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 or object subtyping.

Previous work on sorts has demonstrated that they would be a useful addition to languages as diverse as the functional programming language ML [FP91, Fre94] and the logical framework LF [Pfe93]. Work on sorts in ML has focused on refinements of datatypes which are defined using recursive definitions, similar to the definitions of the datatypes themselves. 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-type systems have thus been limited to small prototypes.

The starting point for the proposed work is the observation that *sort checking* a program which has been annotated with some of the intended sorts should be much easier than sort inference. *The goal of this work is to demonstrate that sort checking can be a practical and useful addition to a real programming language.*

2 An Introductory Example

To illustrate the practical use of sorts in a language like ML, consider the following example. Suppose we are writing an application which manipulates simple arithmetic expressions. In some parts of this application we maintain the invariant that the expressions manipulated are sums of products, i.e., they do not contain additions as sub-expressions of multiplications. Additionally, in some parts of the application we also maintain the invariant that only ground expressions are manipulated, i.e., expressions which contain no variables. Also, in some parts of the program we maintain both of these invariants.

If we attempt to enforce these invariants using the type system of ML, we will need to create separate types for arbitrary expressions, expressions which are sum of products, ground expressions, and ground expressions which are sum of products. We will then need to duplicate functions which manipulate expressions so that they can be applied to each of the required types, which leads to very awkward code maintenance problems. This is usually too awkward to be feasible, so instead only one type is defined, and the intended invariants remain implicit, and are only documented in comments. To manually check that the invariants actually hold can be a very tedious and error prone process, particularly when the code is modified many times.

The following small code sample shows how these invariants can be captured with sorts:

```
datatype exp = Num of int | Var of string | Plus of exp * exp | Times of exp * exp
datasort prod = Num of int | Var of string | Times of prod * prod
datasort sum_prods = <prod> | Plus of sum_prods * sum_prods
datasort ground = Num of int | Plus of ground * ground | Times of ground * ground

datasort gnd_sp = datasort (ground & sum_prods)

fun sp_times(Plus(st1a, st1b), st2) = Plus(sp_times(st1a, st2), sp_times(st1b, st2))
  | sp_times(st1, Plus(st2a, st2b)) = Plus(sp_times(st1, st2a), sp_times(st1, st2b))
  | sp_times(st1, st2) = Times (st1, st2)
withsort sp_times :> (sum_prods * sum_prods -> sum_prods) & (gnd_sp * gnd_sp -> gnd_sp)
```

In this example code we define a type `exp` for simple arithmetic expressions. Then we define some refinements of this type using `datasort` declarations, which are part of the proposed extension to ML. These refinements correspond to the restricted forms required by the invariants of our application. The form of `datasort` declarations is similar to `datatype` declarations, except that value constructors may appear many times in the same declaration, and previously declared sorts may be extended, as in the definition of `sum_prods` which extends `prod`. Also, the replicating form of `datasort` declarations allows the body to be an intersection, as illustrated in the definition of `gnd_sp` above.

We then define an example function from our application, `sp_times`, which takes two expressions which are sums of products, and returns a sum of products expression which is equivalent to their product. Additionally, we have the invariant that if both input expressions are also ground, it returns a ground expression. We express these invariants of the `sp_times` function by attaching a `withsort` clause to it, which is another one of our proposed extensions to ML. This clause assigns a sort to the function which is an intersection, which intuitively means that the function must have both of the intersected sorts, each of which corresponds to one of the invariants. The assigned sort will prevent the programmer from applying this function to an argument that is not determined to represent a sum of products (or a ground sum of products), even though the function will not raise an error in this case. This correctly reflects the intention that this function only be applied to sums of products.

Part of the proposed work is to build a sort checker into an existing SML compiler. In fact, an initial implementation of a sort checker has already been built which will successfully check the example code above, thus verifying that the invariants hold. If instead there was an error in our code, such that these invariants did not hold, then sort checking would fail with an informative error message. In fact, because we are only doing sort checking, we can generally give better error messages than those generated when ordinary type *inference* fails, since the latter may propagate incorrect information before detecting the error. However, static debugging based on sort checking is more involved than ordinary static debugging, since `datasort` definitions and sort annotations need to be debugged in addition to the actual code being checked. This debugging process is often beneficial, since it forces the programmer to be aware of the exact invariants of their code, thus increasing confidence that the code is correct.

3 Previous work on sorts

Sorts were first introduced in [FP91], which showed how to extend a small subset of ML with recursively defined refinements of datatypes. This paper presented some small examples to show the expressiveness of these refinements, and proposed an algorithm for performing sort inference. This algorithm first constructed a subsorting lattice for each datatype containing the defined refinements and their intersections. Then the sort of an expression could be inferred using a form of abstract interpretation, using the subsorting lattices to decide subsorting problems via a procedure for determining subsorting between refinements of function types. The language in this paper included polymorphism, with the restriction that the only refinement of a type variable is a single corresponding sort variable. It also included a union operator on sorts as well as an intersection operator, although in later work unions have been omitted.

Work on sorts in ML was continued by Freeman in his PhD thesis work [Fre94]. This thesis concentrated mostly on algorithms for inferring the sort of an expression, using the basic ideas in [FP91]. Efficiency turned out to be a major problem, since inferring the principal sort of a function essentially required enumerating all refinements of the argument type. Additionally, inferring the sorts of recursive functions required finding a fixed point by iteration, as is done in abstract interpretation. Using sophisticated representations for sorts and memoization allowed an implementation to perform sort inference for some examples, though the largest such example was only 50 lines of SML code. Further, the time taken for sort inference increased dramatically when more refinements were declared, and when higher-order functions were used.

Freeman's implementation only accepted a subset of ML, which did not include modules, pattern matching, exceptions, references, or local datatype definitions. Together with the efficiency problems, this made realistic experiments somewhat difficult.

Freeman also presented an algorithm for calculating lattices of refinements of constructed types, but this algorithm was somewhat unsatisfactory, since it rejects some subsortings which should intuitively hold. For example:

```
datatype nat = z | s of nat
datasort nat1 = z | s of nat
datasort even = z | s of odd and odd = s of even
datasort nat2 = z | s of odd | s of even
```

In this example, `nat1` and `nat2` are equivalent when we think of them as regular tree grammars. However, Freeman's algorithm will not determine that `nat1` is a subsort of `nat2`. In fact, there does not seem to be any declarative way to specify the results of this algorithm, so the programmer must be familiar with the algorithm itself in order to predict which subsortings will be determined to hold.

Freeman's thesis also treated refinements of parameterized datatypes, in particular treating covariant and contravariant parameters. It also points out a problem with the interaction between polymorphism and sorts, namely that instantiating the principal sort of an expression with respect to a polymorphic type yields a less accurate sort than the principal sort of the expression with respect to the monomorphic type.

Other work on sorts includes that by Pfenning [Pfe93] on an extension of LF with sorts. This work is very similar in spirit to work on sorts in ML, but the technical and practical issues are very different. The LF sorts are open ended, in the sense that they could be extended with further declarations. As a result subsorting can not be determined by analyzing the declarations as is done with sort declarations in ML. Instead, the programmer directly declares the subsortings which are assumed to hold. Also, in LF there is no sort inference problem to be solved, but only a sort checking problem, thus avoiding some of the efficiency problems with ML sort inference mentioned above. Furthermore, LF has no recursion or polymorphism, but does have dependent types, so sort checking requires very different techniques to ML sort inference.

One more use of sorts is that of Hayashi [Hay94], who uses them to express specifications of programs, in a similar way to work on extracting programs from constructive logic proofs. Here, the ordinary type system is simply that of second-order λ-calculus, while the refinements include dependent sorts, intersections and unions. Thus, program extraction simply involves ignoring the level of refinements, and the underlying program is readily apparent during proof construction.

4 Overview of the proposed work

The proposed work departs from previous work on sorts in ML by focusing on sort checking rather than sort inference. The main motivation for this is to avoid the efficiency problems encountered in [Fre94]. Sort checking requires the programmer to annotate their program at certain places with sorts, which are then used to check the program in a top-down fashion. If enough sorts are declared, then whenever a function needs to be checked the intended sort will be known. This avoids enumerating all refinements of the argument type, and also avoids the iterative fixed-point calculation required for recursive functions.

Some work has already been completed which investigated the practicality of sort checking in ML. A prototype sort checker for the core language of Standard ML has been built, which is described briefly in [Dav97]. Some more details are contained in [DP97], which concentrated on a formal investigation of sort checking. A major contribution of the work to date is a method for sort checking the pattern matching construct of ML which correctly reflects the sequential nature of patterns, which was not addressed in previous work on sorts in ML. An important property of the sort checker is that the language it accepts is a conservative extension of the core of SML. This allows sort checking of existing SML programs with sort annotations added to only some parts.

Unfortunately, the sort checker is unsound for programs which use references or exceptions. A significant part of the work still to be done will be to find an approach which avoids this problem, and some possibilities will be discussed later. Despite this unsoundness, the results of some experiments with this prototype on medium sized programs are still encouraging. For each of these experiments it is not too hard to see that references and exceptions are not used in ways which lead to unsoundness. Further, the results indicate that sort checking can be done efficiently enough to be practical, and that the sort annotations required are not a great burden on the programmer. In fact, these annotations are more likely to be a benefit than a burden, since they document invariants which the programmer should already be aware of when writing the code, but which may not be obvious when examining the code later.

Based on this experience, it seems that it should be practical to add sort checking to a real compiler for full Standard ML. Achieving this will be the main practical focus of the proposed work. Some additional issues still need to be resolved to make this possible. Firstly, sort checking needs to be extended to include SML modules. This will involve adding sort specifications to signatures, including some way to specify the lattice of refinements of an abstract type. It may also prove necessary to provide some way to specify the variance of a parameterized sort with respect to each of its parameters.

Another significant issue is the design of a satisfactory algorithm for determining subsorting by analyzing `datasort` declarations. The algorithm in the current prototype improves upon previous work on ML sorts in that it is complete in the case that all sorts are defined by regular tree grammars. However, it is still incomplete when parameterized sort definitions are included. Some recent work by Skalka [Ska97] has designed algorithms for comparing parameterized regular tree grammars, but the approach depends on global knowledge of the definitions of all parameterized sorts, so it is not clear how to extend it the case where some definitions are hidden by the ML module system. Aside from parameterization, another problem is how to handle functions and references in `datasort` definitions. In this case, it is not even clear what subsortings should hold, since the analogy with grammars breaks down. The current implementation includes an algorithm which handles functions and references, but which has not even been shown to be sound, and thus needs to be studied further. In Freeman's thesis work [Fre94] argument sorts of functions in sort definitions were restricted to previously defined sorts, which simplifies the problem, though may prove to be too restrictive in practice, particularly if the same restriction were applied to references.

5 Related Work

5.1 Intersection types

The inclusion of an intersection operator allows sorts to express very precise information about programs by combining many pieces of information into a single sort. Such intersection operators have been extensively studied in the context of typed λ-calculi [CDCV81]. An important property in that context is that in a simply-typed λ-calclus with intersections the well-typed terms are exactly those that have normal forms. This of course means that it is undecidable whether an untyped term has a type.

The use of intersection types in programming languages was first proposed by Reynolds who used them in the language Forsythe [Rey81, Rey96]. In Forsythe intersections are used to represent overloading of arithmetic functions, to represent polymorphism (in the absence of parameteric polymorphism), and to represent records. However, they are generally not used to represent program invariants as is possible with ML sorts, which is not surprising since Forsythe does not have any equivalent of ML datatypes. Forsythe is explicitly typed, and type checking is decidable. The algorithm for Forsythe type checking presented in [Rey96] is somewhat different to that used for ML sort checking in [DP97]. In fact, the Forsythe algorithm relies on the assumption there do not exist base types a, b, c such that $a \mathbin{\&} b \leq c$ but not $a \leq c$ nor $b \leq c$, which does not generally hold in the case of sorts.

An important result proved in [Rey96] is that Forsythe type checking is PSPACE-hard. The proof involves translating quantified boolean expressions into Forsythe programs which type check exactly when the boolean expression is typable. Exactly the same technique can be used to show that ML sort checking is PSPACE-hard. In practical terms this means that it is possible for sort checking not to terminate within a reasonable amount of time. However, our experience so far with the current implementation indicates that our sort checking algorithm is likely to be efficient enough for programs encountered in practice.

5.2 Regular tree types and set constraints

Part of the original inspiration for adding sorts to ML was the use of regular tree types in logic programming. These types represent subsets of the Herbrand universe of a logic programming languge. Thus, it is very natural for regular tree types be have a subtype ordering corresponding to inclusion of the sets they represent. Regular tree types are defined by regular term grammars, which allow this ordering to be computed relatively efficiently, while also being expressive enough to specify appropriate sets in most cases. This also allows intersections, unions and complements of regular tree types to be computed.

Much of the work on regular tree types, such as [Mis84], has restricted the grammars further so that they are tuple distributive, which means that if $f(A_1, A_2)$ and $f(B_1, B_2)$ are in the set generated by the grammar, then so must $f(A_1, B_2)$ and $f(A_2, B_1)$. This allows the regular tree grammars to be treated as ordinary regular grammars, for which algorithms for are well known, but it also reduces the expressiveness considerably. For general regular tree grammars checking inclusion is EXPTIME-hard, which can easily be shown by reduction from the problem of inequivalence of finite tree automata, which was shown to be EXPTIME-complete by Seidl [Sei90]. Despite this, algorithms for comparing and calculating intersections and unions of general regular tree grammars have been proposed by Dart and Zobel [DZ92], and appear to be somewhat practical, though have not been shown to be correct. Aiken and Murphy [AM91] have proposed somewhat different algorithms for a generalization of regular tree grammars which include free variables and negation, though found it necessary to make some approximations for efficiency. An algorithm for the even more general problem of solving set constraints over terms was presented by Aiken and Wimmers [AW92], and shown to have non-deterministic exponential time complexity.

In the context of ML, we would like to be able define sorts by general regular tree grammars, though we also need to handle parameterized definitions and refinements of function types and reference types in definitions. The algorithm for comparing sorts in the current implementation is complete when sorts are defined using only regular tree grammars, and it uses an apparently more efficient method for handling (non-distributive) tuples than Dart and Zobel. Skalka [Ska97] has formally proved the correctness of an algorithm related to this one, including an extension to parameterized sort definitions. Parameterized sort definitions might also be handled using the free variables allowed in [AM91]. Some hints on how to handle function types in sort definitions can be obtained from an algorithm due to Aiken and Wimmers [AW93] for the more general problem of solving type inclusion constraints, which are essentially set constraints with function types added. Unfortunately their algorithm requires restrictions on occurrences of intersections and unions, and these restrictions do not seem appropriate for sort definitons.

5.3 Soft typing and conditional types

Soft typing aims to bring some of the benefits of statically-typed languages to dynamically typed ones, and was first considered by Reynolds [Rey69], although the term "soft typing" is due to Cartwright and Fagan [CF91]. This is done using analyses or type systems which do not require any type declarations by the programmer, and then automatically inserting dynamic type checks based on the results to guarantee that the resulting program cannot raise a runtime error. Programmers can use the dynamic type checks to help locate possible errors, and some work has been done by Flanagan et al. [FFK+96] to design a sophisticated interface to help in this process. However, even with such an interface this approach makes it somewhat tedious for a programmer to determine which dynamic type checks are the result of errors in a program, particularly when a large program is modified many times.

Two systems used for soft typing are set-based analysis, due to Heintze [Hei94], and conditional types which are originally due to Reynolds [Rey69] and have been used more recently by Aiken*et al.* [AWL94]. Both capture quite accurate invariants of programs in some cases, so it is very interesting to compare them to sorts in ML. There do not seem to be major differences in accuracy between these two systems, and so we concentrate on conditional types here to simplify the comparison.

Conditional types allow the control flow information resulting from a conditional (or "case" construct) to be directly represented in a type. Perhaps the most accurate soft-typing systems are those that make use of *constrained types* with conditional types. Constrained types are polymorphic types which include inclusion constraints on their variables, of which many varieties have been studied, starting with Mitchell [Mit84]. These were combined by Aiken *et al.* [AWL94] to obtain a very expressive type inference system. In fact, this system is expressive enough that at first it appears that it might be able to capture many of the invariants that sorts were designed to express. Alas, the following example illustrates that this is not the case:

```
datatype bitstring = Nil | Zero of bitstring | One of bitstring
datasort even_parity = Nil | Zero of even_parity | One of odd_parity
     and odd_parity = Zero of odd_parity | One of even_parity

fun append_one Nil = One Nil
  | append_one (Zero bs) = Zero (append_one bs)
  | append_one (One bs) = One (append_one bs)
withsort append_one :> even_parity -> odd_parity & odd_parity -> even_parity
```

This program is well sorted, thus verifying two interesting invariants of the function `append_one`. However, using constrained conditional types for an equivalent function, we obtain the type

$$
\begin{array}{ll}
\multicolumn{2}{l}{\forall\alpha.\alpha \to \beta} \\
\quad\text{where} & \alpha \le \mathtt{Nil} \cup \mathtt{Zero}(\alpha) \cup \mathtt{One}(\alpha) \\
 & \beta = (\mathtt{One(Nil)}?(\alpha \cap \mathtt{Nil})) \cup (\mathtt{Zero}(\beta)?(\alpha \cap \mathtt{Zero}(1))) \cup (\mathtt{One}(\beta)?(\alpha \cap \mathtt{One}(1)))
\end{array}
$$

Roughly, the first constraint means that the argument type α is a particular form of subtype of `bitstring` above, and the second constraint means that if the α has a non-nil intersection with the type containing only `Nil`, then the result type β contains `One(Nil)`, and if α contains any value begining with `Zero` (1 is an all-inclusive type) then β contains `Zero`(x) for each x in β, and if α

contains any value begining with `One` then β contains `One`(x) for each x in β. This type contains a lot of information, but it certainly does not capture the desired invariants above. In particular, `even_parity` and `odd_parity` do not correspond to valid instantiations of α, since they would violate the first constraint. In fact, the only supertype of either of these types that would be a valid instantiation of α is the one which corresponds to the whole of the type `bitstring`.

A closer analysis of this result indicates that the problem is the lack of polymorphic recursion in the system of [AWL94]. This suggests that polymorphic recursion would be a very useful addition to this system, though it seems very unlikely that a type inference would be decidable for such a system.

5.4 Dependent types in ML

Xi and Pfenning [XP97b] have proposed a general schema for adding dependent types to ML, and have applied this schema to the problem of array bounds checking [XP97a]. Their motivations are somewhat similar to those for sorts, namely capturing invariants of programs. However, the invariants that can easily be captured using their form of dependent types are quite different to those that can be captured using sorts. Also, their schema for type checking ultimately relies on a constraint solver for whatever domain is used to index types, and constraint solving can involve arbitrary theorem proving, which is undecidable in general.

6 Simple sort checking

This section formally describes a simple checking algorithm which is a fundamental part of the proposed work, and which is the basis for the current implementation of sort checking for core SML. As mentioned earlier, this sort checker is unsound for programs which contain references or exceptions, and some approaches to solving this problem will be discussed later in Section 7.5

Most of this section is based on [DP97], and it includes a basic presentation of sorts based on previous work [FP91]. It also presents a simply-typed λ-calculus with **let**, and gives declarative sorting rules for it so that the correctness of the sort checking algorithm can be proved. **let**-terms are included 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. The basic algorithm presented here could also form the basis for sort checking languages other than ML, such as LF extended with sorts [Pfe93].

6.1 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.

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

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

$\Gamma \vdash M : A$ Term M has type A in context Γ.

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 7.4.

6.2 Refinement

The refinement judgement has the form

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

$$\frac{}{r^a :: a}\,\textsf{rf\_base} \qquad \frac{R :: A \qquad S :: B}{R \to S :: A \to B}\,\textsf{rf\_arrow} \qquad \frac{R :: A \qquad S :: A}{R \,\&\, S :: A}\,\textsf{rf\_conj}$$

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 $R = R_1 \,\&\, R_2$ 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.

6.3 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: it allows refinements of datatypes to be defined using regular tree grammars in **datasort** declarations (as illustrated in Section 2), with the relation $\rho^a \preceq r^a$ holding exactly when the 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 some appropriate consistency conditions.

The subsorting judgement has the form:

$R \leq S$ Sort R is a subsort 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].

$$\frac{cnjs\ R \preceq s^a}{R \leq s^a}\,\textsf{sub\_base} \qquad \frac{}{R \leq R}\,\textsf{sub\_reflex} \qquad \frac{R_1 \leq R_2 \quad R_2 \leq R_3}{R_1 \leq R_3}\,\textsf{sub\_trans}$$

$$\frac{}{R \,\&\, S \leq R}\,\textsf{sub\_conjL1} \qquad \frac{}{R \,\&\, S \leq S}\,\textsf{sub\_conjL2} \qquad \frac{R \leq S \qquad R \leq S'}{R \leq S \,\&\, S'}\,\textsf{sub\_conjR}$$

$$\frac{R' \leq R \qquad S \leq S'}{R \to S \leq R' \to S'}\,\textsf{sub\_arrow} \qquad \frac{}{(R \to S) \,\&\, (R \to S') \leq R \to (S \,\&\, S')}\,\textsf{sub\_dist}$$

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.

6.4 Sorting

The sort 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.

$$\frac{x{\triangleright}R \text{ in } \Delta}{\Delta \vdash x \triangleright R}\,\textsf{srt\_var} \qquad \frac{\Delta, x{\triangleright}R \vdash M \triangleright S \quad R :: A}{\Delta \vdash \lambda x{:}A.\ M \triangleright R \to S}\,\textsf{srt\_lam} \qquad \frac{\Delta \vdash M \triangleright R \to S \quad \Delta \vdash N \triangleright R}{\Delta \vdash M\,N \triangleright S}\,\textsf{srt\_app}$$

$$\frac{\Delta \vdash M \triangleright R \qquad \Delta, x{\triangleright}R \vdash N \triangleright S}{\Delta \vdash \textbf{let}\ x = M\ \textbf{in}\ N \triangleright S}\,\textsf{srt\_let} \qquad \frac{\Delta, x{\triangleright}R \vdash M \triangleright R \qquad R :: A}{\Delta \vdash \textbf{fix}\ x{:}A.\ M \triangleright R}\,\textsf{srt\_fix}$$

$$\frac{\Delta \vdash M \triangleright R \qquad \Delta \vdash M \triangleright S}{\Delta \vdash M \triangleright R \,\&\, S}\,\textsf{srt\_conj} \qquad \frac{\Delta \vdash M \triangleright R \qquad R \leq S}{\Delta \vdash M \triangleright S}\,\textsf{srt\_subs}$$

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 for brevity.

6.5 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. We omit the details of algorithmic subsorting judgment $R \trianglelefteq S$, since variations of it have been considered previously [Fre94] and we do not consider it a major contribution of this work.

6.6 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:

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

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 $(r^a \to r^a)\ \&\ (s^a \to s^a)$ 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:

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

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. See [DP97] for a formal description of these expansions. A compiler which hoists constant arguments of locally defined functions would undo these expansions, so that they would not result in slower programs. Some compilers do not perform this optimization, so effect of these expansions in practice needs to be considered further, and perhaps addressed by introducing type annotations like Reynolds and Pierce.

For the example above, expanding and adding sorts yields:

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

After expansion, we have have a well-sorted term which contains only occurrences of $\mathbf{fix}\ x{:}A.\ M$ 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. We can then construct a checkable term by adding sorts to **let** and **fix** terms as required, based on the sorting derivation. 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.

6.7 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 $\Delta :: \Gamma$, $R :: A_1 \to A_2$, $\Gamma \vdash C : A_1$ and $S :: A_2$.
$\Delta \vdash R \not@ C$	Sort R can not be projected on term C, where $\Delta :: \Gamma$, $R :: A_1 \to A_2$ and $\Gamma \vdash C : A_1$.

$$\frac{\Delta \vdash C \downarrow R \qquad \Delta \vdash C \downarrow S}{\Delta \vdash C \downarrow R \,\&\, S}\,\textsf{dn\_conj} \qquad \frac{\Delta, x{\triangleright}R \vdash C \downarrow S \qquad R :: A}{\Delta \vdash \lambda x{:}A.\, C \downarrow R \to S}\,\textsf{dn\_lam}$$

$$\frac{\Delta \vdash I \uparrow R \qquad \Delta, x{\triangleright}R \vdash C \downarrow S}{\Delta \vdash \mathbf{let}\ x = I\ \mathbf{in}\ C \downarrow S}\,\textsf{dn\_letA} \qquad \frac{\Delta \vdash C_1 \downarrow R \qquad \Delta, x{\triangleright}R \vdash C_2 \downarrow S}{\Delta \vdash \mathbf{let}\ x{\triangleright}R = C_1\ \mathbf{in}\ C_2 \downarrow S}\,\textsf{dn\_letR}$$

$$\frac{\Delta \vdash I \uparrow R \qquad R \trianglelefteq S}{\Delta \vdash I \downarrow S}\,\textsf{dn\_atom}$$

....

$$\frac{x{\triangleright}R \text{ in } \Delta}{\Delta \vdash x \uparrow R}\,\textsf{up\_var} \qquad \frac{\Delta \vdash I \uparrow R \qquad \Delta \vdash R @ C \gg S}{\Delta \vdash I\, C \uparrow S}\,\textsf{up\_app} \qquad \frac{\Delta, x{\triangleright}R \vdash C \downarrow R}{\Delta \vdash \mathbf{fix}\ x{\triangleright}R.\, C \uparrow R}\,\textsf{up\_fix}$$

....

$$\frac{\Delta \vdash C \downarrow R_1}{\Delta \vdash R_1 \to R_2 @ C \gg R_2}\,\textsf{aptm\_arrow} \qquad \frac{\Delta \vdash R @ C \gg S_2 \quad \Delta \vdash R' @ C \gg S_2'}{\Delta \vdash R \,\&\, R' @ C \gg S_2 \,\&\, S_2'}\,\textsf{aptm\_conj}$$

$$\frac{\Delta \vdash R @ C \gg S_2 \quad \Delta \vdash R' \not@ C}{\Delta \vdash R \,\&\, R' @ C \gg S_2}\,\textsf{aptm\_conj1} \qquad \frac{\Delta \vdash R \not@ C \quad \Delta \vdash R' @ C \gg S_2'}{\Delta \vdash R \,\&\, R' @ C \gg S_2'}\,\textsf{aptm\_conj2}$$

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 **fix**). This erasure process defines a function $\mathit{erase}(\cdot)$, which is a left inverse of the function defined by the annotation process (excluding expansion).

Theorem 3 (Algorithmic sort-checking soundness)

1. *if* $\Delta \vdash C \downarrow R$ *then* $\Delta \vdash \mathit{erase}(C) \triangleright R$.

2. *if* $\Delta \vdash I \uparrow R$ *then* $\Delta \vdash \mathit{erase}(I) \triangleright R$.

3. *if* $\Delta \vdash R \mathbin{@} C \gg S_2$ *then there exists* S_1 *such that* $\Delta \vdash \mathit{erase}(C) \triangleright S_1$ *and* $R \leq S_1 \to S_2$.

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

Theorem 4 (Algorithmic sort-checking completeness)

1. *if* $\mathcal{D} :: \Delta \vdash M \triangleright R$ *and* $\mathit{ann}(M, \mathcal{D}) = C$ *then* $\Delta \vdash C \downarrow R$.

2. *if* $\mathcal{D} :: \Delta \vdash M \triangleright S$ *and* $\mathit{ann}(M, \mathcal{D}) = I$ *then there exists* R *such that* $\Delta \vdash I \uparrow R$ *and* $R \trianglelefteq S$.

3. *if* $\mathcal{D} :: \Delta \vdash M \triangleright S_1$ *and* $R \leq S_1 \to S_2$ *and* $\mathit{ann}(M, \mathcal{D}) = C$ *then there exists* S_2' *such that* $\Delta \vdash R \mathbin{@} C \gg S_2'$ *and* $S_2' \trianglelefteq S_2$.

Proof: By induction on the corresponding derivations, making use of the appropriate replacement lemma, and a lemma validating subsumption in the algorithmic system. □

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.

7 Sort Checking Core Standard ML

We have built a sort checker based on the algorithm in the previous section for an extension of the core language of Standard ML excluding references and exceptions. Some of the issues which had to be resolved in order to achieve this are described in this section, as are issues which still need to be resolved, including the extension to references or exceptions.

7.1 Generating lattices of refinements

During sort checking the datasort definitions in a program are analyzed to generate lattices of refinements for each datatype. In order to do this we need to compare the defined sorts and their intersections for inclusion. As mentioned earlier, the current implementation includes an algorithm for comparing sort definitions which is complete in the case that the definitions are regular tree grammars. This algorithm is based on a recursive function for checking the slightly generalized problem of whether one sort is contained in the union of a number of sorts, which we write as $R \leq S_1 \cup \ldots \cup S_n$. This generalization fixes the incompleteness problem encountered by Freeman [Fre94]. The algorithm is somewhat similar to Freeman's otherwise, in particular it succeeds if a subproblem arises which we are already in the process of solving.

The generalization is needed to handle the case that R and the S_j are refinements of a datatype. In this case we check that each argument-free constructor c in the definition of R appears in one the definitions of the S_j. We also check that for each clause $c_i R_i$ in the definition of R that R_i is contained in the union of S_{jk} such that $c_i S_{jk}$ appears in one of the definitions of the S_j.

In the case that R and the S_j are refinements of a record type, we need to generate a number of subproblems. For example, to check $R * R' \leq \bigcup_{i \in I}(S_i * S'_i)$ where $I = \{1, \ldots, n\}$ we use the fact that this is equivalent to:

$$\forall J \subset I.\ (R \leq \bigcup_{j \in J} S_j \ \text{ or } \ R' \leq \bigcup_{j' \in (I-J)} S'_{j'})$$

Exactly the same equivalence was considered in [AM91], though rejected in favor of an approximation for efficiency reasons. Our experience has been that the time taken is very reasonable for the examples we have tried so far, although we did find it necessary to memoize the results for subproblems and to optimize some important common cases. As an example of these common cases, we avoid generating all the subproblems indicated by the equivalence above when the sort on the left is a subsort of one part of the union on the right, or when one of the components on the left is empty or is not contained in the union of the corresponding components on the left.

The soundness and completeness of this algorithm needs be formally shown for the case of regular tree grammars, which will probably be based on the proofs by Skalka [Ska97] for a related algorithm. While the current approach is incomplete in the case of parameterized sort definitions, it is still easy to specify its behavior in this case provided we adopt a similar restriction to Freeman [Fre94], namely that occurrences of mutually defined sort constructors in sort definitions must have the same arguments as the sort constructor being defined. Disallowing function and reference sorts in sort definitions is likely to be very restrictive in practice. At a minimum, functions and references should be allowed when only the default refinement of a type appears on the left of an arrow or as an argument to the `ref` constructor. In this case, it is possible to use the above algorithm with little modification to obtain a sound and predictable system.

Based on experiments with a full implementation it may prove necessary to design a more powerful system for comparing sort definitions. For example completeness in the case of parameterized definitions may be useful, as may allowing arbitrary function and reference sorts, and it may even be useful to determine the variance of each parameterized sort constructor in terms of each of its parameters. These extensions are considered further in Section 10.

7.2 Determining principal sorts for value constructors

Given a set of refinement definitions for a type, at first it seems trivial to generate the sorts of the value constructors. However, the following example shows that this is not the case:

```
datatype nat = z | s of nat
datasort even = z | s of odd and odd = s of even
datasort nonzero = s of odd | s of even
```

At first glance, it may seem that we have the principal sort:

```
s :> nat -> nat & odd -> even & even -> odd & odd -> nonzero & even -> nonzero
```

However, this sort does not reflect that `s :> nat -> nonzero`.

The current implementation uses a somewhat inefficient method for finding the principal sorts of value constructors, namely applying each value constructor to each refinement of the argument type, and then using the algorithms for comparing sort definitions to compare the result with every refinement of the result type. It seems that there should be a better way to do this.

7.3 Sort constraints

In Section 6 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, as illustrated in Section 2. We cannot always give individual sort constraints for the arguments and the result of a function, as is sometimes done with types, since the desired sort for the function 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 `exp -> exp` and it appears without an annotation in a context where an annotation is required, then we add the sort annotation `exp -> exp`. Here `exp` is a sort which is implicitly defined when the type `exp` is defined, essentially by reading the type definition as though it were a sort definition as well. For example, if we have:

```
datatype exp = Num of int | Var of string | Plus of exp * exp | Times of exp * exp
```

then this implicitly defines the same sort as the definition:

```
datasort exp = Num of int | Var of string | Plus of exp * exp | Times of exp * exp
```

The automatic addition of these default sort annotations avoids the need to annotate sections of code for which no interesting sort checking is required. Our intention is that sort checking should be a conservative extension of Standard ML, i.e. sort checking always succeeds for well-typed Standard ML programs with no sort annotations. This needs be shown formally for at least a substantial fragment of core SML.

This approach to obtaining conservativity is quite different to that used by Freeman [Fre94], who chose to require a "top" element in every lattice. We do not adopt his solution because it requires the top element to have somewhat odd behavior in the presence of function and reference types in sort definitions.

7.4 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 variable has the same name as the type variable.

During sort checking our algorithm needs to determine the principal sort of an occurrence of a variable which has a polymorphic type, which is done by enumerating all combinations of refinements of the instantiations of the type variables chosen during type inference. This has not proved to be a bottle neck in the experiments done so far, but in general it is infeasible if the type instantiations involve functions which themselves involve types for which refinements have been defined. Freeman solved this problem using a somewhat complicated method which may or may not be correct, and which depends on his method for doing sort inference. A much simpler solution is to force the programmer to declare the required instance of a polymorphic sort in the case that the enumeration is not feasible, which may serve as good program documentation as well. This could be improved by avoiding the need for such an annotation in the common case of a polymorphic function which is applied to an argument whose sort can be inferred (which can be determined syntactically).

7.5 References and exceptions

As mentioned earlier, the current approach to sort checking is unsound when directly extended to references to exceptions. The following example illustrates the problem:

```
datasort tt = true and ff = false
let
  fun f x =
    let
      val y = ref x
    in
      (fn () => !y, fn z => y:=z)
    end
  withsort f :> 'a -> (unit -> 'a) * ('a -> unit)

  val f' = f :> (tt -> (unit -> tt) * (tt -> unit))
              & (bool -> (unit -> bool) * (bool -> unit))
  val x :> ((unit -> tt) * (tt -> unit)) & ((unit -> bool) * (bool -> unit))
         = f' true
  val (r:>(unit -> tt) & (unit -> bool), w:>(tt -> unit) & (bool -> unit)) = x
in
  (w false; r()) :> tt
end
```

Ignoring the sort annotations, executing this code would yield `false`, thus a sound sort checker should reject it. The problem is that when we instantiate `f` to define `f'` we instantiate the sort variable to both `tt` and `bool`. In the absence of references and exceptions this is fine, but in this case it means that a during evaluation a reference will be created which is implicitly assigned both the sorts `tt ref` and `bool ref`, which should not be allowed.

Now, one way to correct this is to force each occurrence of `f` to only have a single sort instantiation. Restricting all polymorphic functions in this way would greatly reduce the accuracy of sorts in the presence of polymorphic functions which do not use references. For example:

```
let
  fun compose (f, g) x = f (g x)
  withsort compose ('b -> 'c) * ('a -> 'b) -> 'a -> 'c

  fun not true = false
    | not false = true
  withsort not :> (tt -> ff) & (ff -> tt) & (bool -> bool)
  val h = compose (not, not)
in
  (h true, h false)
end
```

Without the restriction, this expression can be assigned the sort `tt * ff`, though with it it must be assigned the sort `bool * bool`. This suggests that it might be useful to distinguish those sort variables that appear inside the sort of a reference (or exception) from those that do not. Thus, we are lead to the idea of "imperative sort variables", which are just like the imperative type variables used in the original definition of SML to allow generalization of expansive (i.e. non-value) definitions. This is somewhat surprising, since it has been shown that distinguishing imperative type variables is unnecessary and awkward in ML, and that a simple value restriction on the introduction of parametric polymorphism suffices [WF94].

An alternative to the above approach is to note that it is quite natural to extend the value restriction on parametric polymorphism to a value restriction the finite polymorphism expressed by intersection sorts. Then, in different branches of the intersection rule, different instantiations of polymorphic sorts could be chosen. However, to make this work it seems that we can no longer have the subsorting rule:

$$\frac{}{(R \to S) \mathbin{\&} (R \to S') \leq R \to (S \mathbin{\&} S')} \textsf{sub\_dist}$$

since then from the following sorting:

```
  (fn x => ref x) :> (tt -> tt ref) & (tt -> nat ref)
```

we could obtain:

```
(fn x => ref x) :> tt -> ((tt ref) & (nat ref))
```

which is obviously wrong. In the absence of the rule **sub_dist** it is not clear whether a system of sorts would be intuitive and practical, nor even whether an efficient sort checking algorithm can be constructed. One way to increase the expressiveness of sorts would be to introduce a refinements like `A => B` for functions of type `A -> B` which cannot involve effects. Then, the rule **sub_dist** would still be valid with `->` replaced by `=>`.

This issue certainly needs to be investigated as soon as possible to find a practical solution, and the soundness of such a solution needs to be proven for at least a simple language involving effects (beyond non-termination).

7.6 Pattern matching

Since our extension of ML is based on refinements of datatypes, the approach we adopt for sort checking the ML pattern matching construct is somewhat critical. 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. However, actually doing this expansion is not very practical, and it is not clear that it would lead to a predictable system.

The current implementation uses 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 grammar describing the possible 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. The approach is strictly more accurate than any method for expanding general pattern matching to Freeman's elimination forms, which gives a rough method for predicting its behavior which is sometimes easier than reasoning in terms of descriptions of residuals.

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.

A formal specification of this approach to sort checking pattern matching has been made, though it seems to be more complicated than necessary, so some work needs to be done to allow a more elegant specification. The correctness of algorithm in the implementation should then be proved correct with respect to this specification, which should not be too hard.

8 Modules

The current implementation does not handle the module system of SML. Part of the proposed work will be to design and implement an appropriate extension of SML which includes sort information in modules. Here we outline a basic design and highlight some issues. In Section 10 we describe possible extensions of this design that may prove desirable based on experiments with an implementation.

Our basic design allows the sorts of values to be specified in signatures, and includes two kinds of sort specifications. The first is `datasort` specifications of refinements of datatypes. The second is `sort` specifications of refinements of opaque types. We do not include specifications of transparent refinements of transparent types other than datatypes at this stage even though it seems that these would be quite natural, since some issues need to be resolved on exactly how to do this.

8.1 Refinements of datatypes in signatures

We allow refinements of datatypes to be specified in signatures using `datasort` specifications, which are similar to `datatype` specifications. For example, the following signature could be assigned to a structure containing the definitions for arithmetic expressions in Section 2.

```
sig
 datatype exp = Num of int | Var of string | Plus of exp * exp | Times of exp * exp
 datasort prod = Num of int | Var of string | Times of prod * prod
 datasort sum_prods = <prod> | Plus of sum_prods * sum_prods
 datasort ground = Num of int | Plus of ground * ground | Times of ground * ground

 datasort gnd_sp = datasort (ground & sum_prods)

 val sp_times :> (sum_prods * sum_prods -> sum_prods) & (gnd_sp * gnd_sp -> gnd_sp)
```

```
end
```

Based on this signature, a lattice of refinements of the datatype `exp` can be constructed without referring to a particular structure implementing the signature. Note that the declaration of `sp_times` does not need a type, since this can be reconstructed from the sort.

8.2 Refinements of opaque types in signatures

We also allow refinements of opaque types to be specified in signatures using `sort` specifications. For example, if we want to hide the implementation type in the arithmetic expressions example we might use the signature:

```
sig
  type exp
  sort prod and sum_prods and ground and gnd_sp
    refining exp

  val sp_times :> (sum_prods * sum_prods -> sum_prods) & (gnd_sp * gnd_sp -> gnd_sp)
end
```

Based on this signature, a lattice of refinements of the opaque type exp can be constructed. This lattice contains the default refinement `exp`, as well as the named sorts, and all possible intersections of these. The lattice is not the same as that obtained above, since here the signature does not specify any subsortings between refinements. This signature doesn't even expose the fact that `exp` is implemented using a datatype.

Now it seems useful to be able to extend this signature to specify some of the subsortings which hold between refinements, for example that `gnd_sp` is a subsort of `sum_prods`. We do this using `subsort` specifications, as follows:

```
sig
  type exp
  sort prod and sum_prods and ground and gnd_sp
    refining exp
    subsort gnd_sp < sum_prods
        and gnd_sp < ground
        and ground & sum_prods < gnd_sp

  val sp_times :> (sum_prods * sum_prods -> sum_prods) & (gnd_sp * gnd_sp -> gnd_sp)
end
```

To be able to specify any lattice, need to allow descriptions such as the last one with an intersection on the left. It may also be convenient to allow intersections on the right, and to allow equalities to be specified as well as subsortings. To check a structure against such a signature, we simply check that the sorts implementing `gnd_sp`, `sum_prods` and `ground` satisfy the specified subsortings. Of course, they may also satisfy other subsortings not specified by signature.

To create a lattice from this signature, we generate an element for each sort, and all possible intersections of sorts, up to the equivalences that are a consequence of the subsorting descriptions in the signature for the parameter.

9 Experiments

In addition to many small functions, we have conducted three medium-size experiments with the current sort checker. While this sort checker is unsound in the for programs that use references or exceptions, we consider these results to still be very relevant, since in each case it is not too hard to see that the particular way references and exceptions are used can not lead to incorrect results. The absence of functors made it difficult to annotate larger code samples with sorts, but since sort checking time should increase 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 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 the Twelf interpreter (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 an 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, although this solution would not work if generic list functions were needed. This suggests that it would be useful to allow datasort declarations which define refinements of specific types constructed with type constructors: in this case we would like a refinement of the type of "operator lists".

Finding practical solutions to the remaining language design issues will certainly be aided by the experience gained from these experiments. We intend to carry out larger experiments as soon as we have built a basic sort checker for full Standard ML (including modules), and to use the results of these experiments to determine what extensions to the basic sort checker are desirable. Some possible extensions are described in Section 10. The experiments will be based mostly on existing SML code, including the TIL2 SML compiler, the FoxNet implementation [BHLM94], the Twelf interpreter, and the implementation of the SML basis library included with the SML/NJ compiler.

10 Possible Extensions

In this section we outline some possible extensions of the work outlined so far. Each of these extensions will only be considered further if experiments with a basic implementation indicate a practical need for the extension.

10.1 Defining refinements of instances of type constructors

As mentioned earlier, we could allow the definition of refinements of specific types formed using type constructors. For example, we might want to allow the following:

```
datasort tf_list = :: of tt * ft_list | nil
     and ft_list = :: of ff * ft_list
```

It seems possible to allow such refinements, though it would require the basic idea of lattices of refinements of type constructors to be changed.

10.2 Enriching lattices

Since we are only doing sort checking of expressions, efficiency is not as dependent on keeping lattices small as it is in the work of Freeman. Thus, we can consider methods for defining large lattices of refinements to easily allow more accurate information to be expressed using sorts. For example, we may allow one sort to be defined as the union or difference of two others. We may even automatically add all unions and/or differences to a lattice, though this might lead to very large lattices in some cases. We might also consider adding one refinement for each value constructor of a datatype. To make very large lattices manageable, it may be possible to construct them lazily as each refinement is needed. How useful enriching lattices like this is in practice still needs to be determined.

10.3 Splitting in lattices

Freeman used a notion called splitting to improve the efficiency of sort inference. Splitting involves expressing each element of a lattice as a union of some basic elements, which themselves cannot be expressed as unions of other elements. This allows the principal sort for a function to be inferred by trying each of the basic elements in the lattice as the argument sort, rather than trying all the elements in the lattice.

Since we are only doing sort checking, splitting does not seem so essential, though it may still be useful, since it also allows function sorts to be kept in a canonical form which allows them to be compared for subsorting more quickly. It also allows function sorts to be written more compactly in some cases, for example the negation function has the principal sort `(tt -> ff) & (ff -> tt) & (bool -> bool)` which is equivalent to `(tt -> ff) & (ff -> tt)` in the presence of splitting, but not otherwise. In the presence of splitting it is useful to have way to specify splitting information in signatures, so that the programmer has control over the splitting information propagated between modules for opaque type constructors. More experience is needed to determine whether splitting is desirable in practice.

10.4 Parameterized regular tree grammars

The current approach to generating lattices of refinements is incomplete for parameterized regular tree grammars, since when we encounter a goal like $(R\ c) \leq (R'\ c')$ where c and c' are previously defined, we simply check $R \leq R'$ and check whether $\alpha\ c \leq \alpha\ c'$ according the appropriate lattice. The following example illustrates the incompleteness:

```
datasort tt = true and ff = false  (* Refinements of bool *)
sortdef eb = tt & ff               (* empty refinement of bool *)

datatype ('a, 'b) option_sum = None | Inl of 'a | Inr of 'b
datasort ('a, 'b) none = None

datatype t = T of (bool, bool) option_sum
datasort tnone = T of (bool, bool) none
datasort tnone' = T of (eb, eb) option_sum
```

Here, our algorithm fails to determine that `tnone'` is a subsort of `tnone`, since `('a, 'b) option` is not a subsort of `('a, 'b) none`. Skalka [Ska97] has studied algorithms designed to solve this problem. Unfortunately, it is not clear how to combine this solution with the idea of a lattice of refinements for type constructors to be used during sort checking of expressions. An alternative approach would be to make use of the algorithms for comparing regular tree grammars with free variables in [AM91]. Yet another alternative is to determine the strictness of each parameterized refinement in terms of its parameters, which has the advantage that this strictness can be propagated via signatures without exposing the implementation of the refinement.

10.5 Functions and references in sort definitions

The restriction on the occurrences of function and reference types in sort definitions in Section 7.1 might rule out many interesting uses of sorts. The current implementation already includes an algorithm which appears to be sound when this restriction is removed. However, this needs to be proved, and perhaps an appropriate notion of completeness should be formulated and proved so that a programmer has an easy way to predict which subsortings will be determined to hold. This algorithm is based on the intuition of treating function sorts as representing sets of partial functions which additionally take a store argument and return a store result. This is not quite an accurate model for ML functions, though the subsortings which result seem to be quite natural. One method that could be used to prove soundness of this algorithm is to define a consistency condition for the subsorting results generated, which is sufficient to guarantee that sort checking of expressions based on the results will not incorrectly accept programs.

In presence of function and reference sorts in sort definitions, we need some method for determining the variance of sort constructors in terms of their parameters. Freeman used a scheme in which each parameter of a type constructor was determined to be either covariant, contravariant, ignored, or mixed variance. All refinements of a type constructor then had the same variance, however it seems that it would be more accurate to allow refinements to have different variances. One way to represent this would be to automatically add to the lattice of a type the maximal refinements which are covariant and contravariant in each parameter (similar to the ignoring refinements above), so that the variance of any refinement could be determined by comparing it with these maximal refinements. This works because any subsort of a covariant refinement is also covariant, and similarly for contravariance.

10.6 Multiple sort variables refining a single type variable

An important issue regarding sorts in ML is that we have principal sorts only with respect to a particular ML typing derivation. The following example (due to Freeman) demonstrates this:

```
datasort tt = true and ff = false

fun twice f x = f (f x)
withsort twice :> ('a -> 'a) -> ('a -> 'a)  (* principal sort *)

fun twice' f (x:bool) = f (f x)
withsort twice' :> (tt -> ff & ff -> tt) -> (tt -> tt & ff -> ff)
                 & (tt -> tt & ff -> ff) -> (tt -> tt & ff -> ff)
                 & (bool -> bool) -> (bool -> bool)
fun not true = false
  | not false = true
withsort not :> (tt -> ff) & (ff -> tt) & (bool -> bool)

val f = twice not
 (* f :> (bool -> bool) & ((tt & ff) -> (tt & ff)) (principal sort) *)
val f' = twice not'
 (* f' :> (bool -> bool) & (tt -> tt) & (ff -> ff) (principal sort) *)
```

Here, the sort for `f` is much less useful than that for `f'`, even though we used the principal refinement for `twice`. In fact, there is only one refinement of the type for `twice`, so we cannot expect this refinement to carry useful information. Thus, the programmer may be forced to copy polymorphic functions to obtain accurate sorts. An alternative might be to allow general intersection types, though this would be quite a drastic change. Another solution which we discuss in detail later is to use the module system to ascribe different monomorphic signatures to the same structure, thus sort checking the structure many times. A solution at the core level would be to allow multiple sort variables to refine a single type variable. Thus we could have:

```
fun twice f x = f (f x)
withsort twice :> ('a1 -> 'a2 & 'a2 -> 'a3) -> ('a1 -> 'a3)
```

This is in fact the principal sort for twice, although in this system principal sorts do not always exist, as shown by Freeman. For Freeman the absence of principal sorts made sort inference impossible for this system, however in the case of sort checking it may be a reasonable option.

10.7 Using sorts in signatures for sort checking expressions

We can reduce the number of sort annotations required for expressions in a structure by delaying sort checking until the structure is matched against signature. Of course, the sorts of local value bindings will still need to be provided, as will sorts for which a more accurate sort is required within the structure than is specified in the signature. Aside from reducing the amount of annotations required, this also has the advantage of allowing a structure to be matched against several signatures. This allows the programmer to avoid the problem of refinements of polymorphic types being less expressive than refinements of monomorphic ones, since different signatures may use different monomorphic types. This also allows each client of a structure to define some extra refinements to be used when sort checking the structure which express invariants of particular interest to the client, rather than forcing all such refinements to be defined prior to a single phase of sort checking for the structure.

One problem with matching against several signatures is that sorts cannot be ascribed to locally defined functions in a structure, and so it seems necessary to add some sort inference to a sort checker in order to make it work. While it is not clear whether this can be done in a practical way, if it could it would likely be a useful addition in its own right. Another problem is that we do not have any control over what is externally visible about a structure which is not matched against a signature, and a client can rely on incidental properties of the implementation. However, for some commonly used structures such as lists there does not seem to be any need to hide any details, and this scheme would avoid the need to put all the required invariants for the basic list functions into a single signature.

10.8 Refining other types

It seems possible to extend the idea of refinements to types other than datatypes. For example, positive and negative integers could be defined as refinements of the type of integers. Also, refinements of exceptions could be defined, and then the sort of every expression and every function sort could be annotated with the possible exceptions that could be raised, somewhat similar to [FA97] except that locally defined exceptions could be handled more cleanly by locally defined refinements. This idea could also be extended to annotations restricting the references that could be touched by an expression. As mentioned earlier, it might also be useful to define a refinement of functions for functions which cannot produce effects. It also seems likely that refinements could be defined corresponding to various other program analyses.

10.9 Compiler optimizations based on sorts

A quite different extension is to use the information contained in sorts to optimize the code generated for a program. 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 heavy code explosion if not done carefully.

11 Work Plan

The following is a rough work plan listing the major tasks which need to be done, along with rough estimates of the amount of time for each:

Design a sound system including effects:	1 month
Build a basic core implementation:	4 months
Add basic module level constructs:	1 month
Experiments:	1 month
Proving soundness of a sort checking algorithm:	2 months
Proving soundness (and completeness?) of lattice creation:	1 month
Extensions:	1-3 months
Writing:	4 months

Unfortunately the current implementation is based on a version of the ML-Kit compiler which implements the original definition of SML. We would like to extend the module system of the revised definition of SML, which is different enough justify reimplementing sort checking based on another compiler. This explains the time above for implementing a basic core implementation. I intend to base the new version of the sort checker on the TIL2 compiler, which has the added advantage of being a more practical compiler than the Kit, which should increase the number of likely users of the sort checker.

The proof of the sort checking algorithm above will include effects (beyond non-termination), polymorphism, and a fixed set of datatypes and refinements.

References

[AM91] Alexander Aiken and Brian R. Murphy. Implementing regular tree expressions. In *Proceedings of the 1991 Conference on Functional Programming Languages and Computer Architecture*, pages 427–447, August 1991.

[AW92] Alexander Aiken and Edward L. Wimmers. Solving systems of set constraints. In *Symposium on Logic in Computer Science*, pages 329–340, June 1992.

[AW93] Alexander Aiken and Edward L. Wimmers. Type inclusion constraints and type inference. In *Proceedings of the 1993 Conference on Functional Programming Languages and Computer Architecture*, pages 31–41, Copenhagen, June 1993.

[AWL94] Alexander Aiken, Edward L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *Proceedings of the ACM SIGPLAN Conference on Principles of Programming Languages*, pages 163–173, 1994.

[BHLM94] Edoardo Biagioni, Robert Harper, Peter Lee, and Brian Milnes. Signatures for a network protocol stack - a systems application of Standard ML. In *1994 ACM Conference on Lisp and Functional Programming*, pages 55–64, Orlando, Florida, June 1994.

[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.

[CF91] Robert Cartwright and Mike Fagan. Soft typing. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 278–292, June 1991.

[Dav97] Rowan Davies. A refinement-type checker for Standard ML. In *Proceedings of the Sixth International Conference on Algebraic Methodology and Software Technology*, December 1997. To appear.

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

[DP97] Rowan Davies and Frank Pfenning. Practical refinement-type checking. Unpublished draft available from `http://www.cs.cmu.edu/~rowan`, July 1997.

[DZ92] Philip W. Dart and Justin Zobel. A regular type language for logic programs. In Frank Pfenning, editor, *Types in Logic Programming*, chapter 5, pages 157–187. MIT Press, Cambridge, Massachusetts, 1992.

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

[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.

[Hay94] Susumu Hayashi. Singleton, union and intersection types. *Information and Computation*, 109:174–210, 1994.

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

[Mis84] Prakeesh Mishra. Towards a theory of types in prolog. In *Proceedings of the 1984 Symposium on Logic Programming*, pages 289–298, Atlantic City, New Jersey, 1984.

[Mit84] John Mitchell. Coercion and type inference (summary). In *Eleventh Annual ACM Symposium on Principles of Programming Languages*, pages 175–185, January 1984.

[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.

[Rey69] John C. Reynolds. Automatic computation of data set definitions. *Information Processing*, 68:456–461, 1969.

[Rey81] John C. Reynolds. The essence of algol. In J. W. de Bakker and J. C. van Vliet, editors, *Proceedings of the International Symposium on Algorithmic Languages*, pages 345–372, Amsterdam, 1981. North-Holland.

[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.

[Rey96] John C. Reynolds. Design of the programming language Forsythe. Technical Report CMU-CS-96-146, Carnegie Mellon University, Pittsburgh, Pennsylvania, June 1996.

[Sei90] Helmut Seidl. Deciding equivalence of finite tree automata. *SIAM Journal of Computing*, 19:424–437, June 1990.

[Ska97] Chris Skalka. Some decision problems for ML refinement types. Master's thesis, Carnegie-Mellon University, July 1997. To appear.

[WF94] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115:38–94, 1994. Preliminary version is Rice Technical Report TR91-160.

[XP97a] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. Unpublished draft available from `http://www.cs.cmu.edu/~fp`, November 1997.

[XP97b] Hongwei Xi and Frank Pfenning. A schema for adding dependent types to ML. Unpublished draft available from `http://www.cs.cmu.edu/~fp`, July 1997.

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 9 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
```