Practical Refinement-Type Checking: **Thesis Summary**

Rowan Davies
rowan@cs.cmu.edu

(Thesis oral: 1pm Friday February 11, 2005 in Wean 4623.)

Abstract

Software development is a complex and error prone task. Programming languages with strong static type systems assist programmers by capturing and checking the fundamental structure of programs in a very intuitive way. Given this success, it is natural to ask: can we capture and check more of the structure of programs?

In this work I consider a new approach called *refinement-type checking* that allows many common program properties to be captured and checked. This approach builds on the strength of the type system of a language by adding the ability to specify refinements of each type. Such *refinement types* have been considered previously, and following previous work I focus on refinements that include subtyping and a form of intersection types.

Central to my approach is the use of a bidirectional checking algorithm. This does not attempt to infer refinements for some expressions, such as functions, but only checks them against refinements. This avoids some difficulties encountered in previous work, and requires that the programmer annotate their program with some of the intended refinements, but the required annotations appear to be very reasonable. Further, they document properties in a way that is natural, precise, easy to read, and reliable.

I demonstrate the practicality of my approach by showing that it can be used to design a refinement-type checker for a widely-used language with a strong type system: Standard ML. This requires two main technical developments. Firstly, I present a new variant of intersection types that achieve soundness in the presence of call-by-value effects by incorporating a value restriction. Secondly, I present a practical approach to incorporating recursive refinements of ML datatypes, including a pragmatic method for checking the sequential pattern matching construct of ML.

I also report the results of experiments with my implementation of refinement-type checking for SML. These indicate that refinement-type checking is a practical method for capturing and checking properties of real code.

Thesis Committee: Frank Pfenning (Chair), Robert Harper, Peter Lee, John Reynolds, Alex Aiken (Stanford University).

1 Thesis

A new technique called refinement-type checking provides a practical mechanism for expressing and verifying many properties of programs written in fully featured languages.

2 Motivation

Static type systems are a central feature of many programming languages. They provide a natural and intuitive mechanism for expressing and checking the fundamental structure of programs. They thus allow many errors in programs to be automatically detected at an early stage, and they significantly aid the process of understanding unfamiliar code. This is particularly true for large, modular programs, since types can be used to describe module interfaces.

While strong static type systems are very effective at capturing the basic structure of a program, generally programs involve many important properties that are not captured by types. For example, a particular function may always return a non-zero number, or may require that its argument be non-zero, but programming languages generally do not provide a specific type for non-zero numbers.

Such invariants are often critical to the understanding and correctness of a program, but usually they are only informally documented via comments. While such comments are certainly useful, it takes considerable discipline to ensure that properties are described accurately and precisely, particularly when the code may be modified frequently. Further, the lack of any convenient mechanism for checking whether the code actually satisfies the stated properties means that such comments cannot be relied upon.

We might consider attempting to construct formal proofs that such properties are satisfied. However, constructing such proofs is generally difficult or infeasible. Further, as a program evolves, the proof needs to be evolved, which is likely to be awkward. Additionally, when the intended properties do not hold due to an error in the code, it is unlikely that this method will guide the programmer to the source of the error as quickly as the error messages produced by a type checker.

This work demonstrates that a new approach to capturing and checking some of these properties can be used to build practical tools. This approach builds on the strength of the static type system of a language by adding the ability to specify *refinements* of each type. These *refinement types* include

constructs which follow the structure of the type that they refine, and additionally include features that are particularly appropriate for specifying program properties.

3 Background: refinement types

Refinement types were introduced by Freeman and Pfenning [FP91]. They do not require altering the type system of a language: instead we add a new kind of checking which follows the structure of the type system, but additionally includes features that are appropriate for expressing and checking properties of programs. This means that we are conservatively extending the language: all programs in the original language are accepted as valid programs in our extension.

We also refer to refinement types as *sorts* in accordance with the use of this term in order-sorted algebras [DG94]. This allows us to use convenient terminology that mirrors that for types: we can use terms such as subsorting and sort checking and thus make a clear link with the corresponding notions for types.

We illustrate the features of sorts with a running example. We first illustrate the constructs that follow the structure of types, focusing on the situation for functions. Suppose we have a sort pos for positive integers which refines a type num for numbers. Then, we can form a sort for functions mapping positive integers to positive integers: $\mathsf{pos} \to \mathsf{pos}$. This uses the construct $\to$ which mirrors the corresponding construct for types. If we have an additional refinement nat for natural numbers, then we can form the following refinements of the type $\mathsf{num} \to \mathsf{num}$.

$$\mathsf{pos} \to \mathsf{pos} \qquad \mathsf{pos} \to \mathsf{nat} \qquad \mathsf{nat} \to \mathsf{pos} \qquad \mathsf{nat} \to \mathsf{nat}$$

Sorts include similar constructs mirroring each type construct. We now consider the other features of sorts, which are included specifically because they are appropriate for capturing program properties.

Sorts express properties of programs, and generally there are natural inclusion relationships between these properties. For example, every positive number is a natural number, so we should allow a positive number whenever a natural number is required. Thus, we have a natural partial order on the refinements of each type, and we write $\mathsf{pos} \leq \mathsf{nat}$ to indicate this order. This is essentially a form of subtyping, although we refer to it as *subsorting* since the order is on the refinements of a particular type rather than on the types themselves. This partial order is extended to refinements of function

types following the standard contravariant-covariant subtyping rule. Thus, the following inclusion holds.

$$\mathsf{nat} \to \mathsf{pos} \leq \mathsf{pos} \to \mathsf{nat}$$

In practice it is sometimes necessary to assign more than one property to a particular part of a program. For example, if we have a function double with type $\mathsf{num} \to \mathsf{num}$ that doubles a number, we may need two properties of this function: that it maps positive numbers to positive numbers, and that it maps natural numbers to natural numbers. To allow multiple properties to be specified in such situations, sorts include an *intersection* operator & which allows two refinements of the same type to be combined. Thus, we can specify the desired property of double with the following sort.

$$(\mathsf{pos} \to \mathsf{pos}) \ \& \ (\mathsf{nat} \to \mathsf{nat})$$

The operator & is based on work on intersection types, such as that of Coppo, Dezani-Ciancaglini and Venneri [CDV81] and Reynolds [Rey96].

One might notice that refinements are essentially another level of types, and wonder whether it is really necessary to have both ordinary types and refinements as two separate levels for the same language. In fact, it is possible to design a language which instead includes intersections and subtyping in the ordinary type system. We consider such a language in Chapter 4 of the dissertation, and a real language with these features has been described by Reynolds [Rey96]. However, we have both philosophical and practical reasons for considering types and refinements as two separate levels.

The philosophical reason is that we consider type correctness to be necessary in order for the semantics of a program to be defined, while refinements only express properties of programs that have already been determined to be valid. This is essentially the distinction between typed languages in the style of Church [Chu40] and type assignment systems in the style of Curry [Cur34]. Reynolds [Rey02] has considered a similar distinction between intrinsic and extrinsic semantics. In our case we consider that we have both, with one system refining the other, and we would argue that this is a natural design since the two levels serve different purposes.

The practical reason for considering types and refinements as two separate levels is that it allows us to extend an existing widely-used typed language without modifying it in any way. This allows us to easily experiment with refinements in real code. It also allows others to use refinements without committing to writing code in an experimental language. Thus, this approach allows significant experience to be gained in programming

with advanced features such as intersection types and subtyping without introducing a new language.

Sorts are of little use to a programmer without a practical tool for checking the sorts associated with a program. Previous work on sorts focused on algorithms for sort inference, but this seems to be problematic. One reason for this that code generally satisfies many accidental properties which must be reflected in the inferred sort. Such accidental properties often prevent errors from being reported appropriately, such as when a function is applied to an inappropriate argument that nevertheless matches part of the inferred sort for the function. Further, as we move to more complicated types there is a combinatorial explosion in the number of refinements and the potential size of principal sorts. Experiments with refinements have thus been limited to relatively small and simple code fragments in previous work.

4 Our approach

4.1 Bidirectional checking

Central to this thesis work is a new approach called *refinement-type checking*, which we also call *sort checking*. This approach uses a bidirectional algorithm that does not attempt to infer sorts for some forms of expressions, such as functions, but instead only checks them against sorts. We still infer sorts for other forms of expressions, such as variables and applications. The technique is called bidirectional because it works top-down through a program when checking against sorts, and bottom-up when inferring sorts.

Bidirectional algorithms have been considered previously by Reynolds [Rey96] and Pierce [Pie97] for languages with general intersection types, and by Pierce and Turner [PT98] for a language with impredicative polymorphism and subtyping.

Generally bidirectional algorithms require that the programmer annotate some parts of their program with the intended sorts. In our case, these annotations are only required for function definitions, and only those for which the programmer has in mind a property beyond what is checked by the ordinary type system. Our experience suggests that this requirement is very reasonable in practice. Further, these annotations usually appear at locations where it is natural to describe the properties using a comment anyway. They thus document properties in a way that is natural, easy to read, and precise. Additionally, these annotations can be relied upon to a greater extent than informal comments, since they are mechanically verified by sort checking.

To demonstrate the practicality and utility of sort checking for real programs, we have designed and implemented a sort checker for Standard ML, which is a widely used programming language with a strong static type system and advanced support for modular programming [MTHM97]. We now briefly outline the two main technical developments required to extend our approach to this fully featured language. These form the technical core of the dissertation, along with our bidirectional approach to sort checking.

4.2 Intersection types with call-by-value effects

The first main technical development is a new form of intersection types that achieves soundness in the presence of call-by-value effects. The standard form of intersection types is unsound in the presence of such effects, as illustrated by the following SML code, which includes sort annotations in stylized comments (as used by our implementation).

```
(*[ cell <: (pos ref) & (nat ref) ]*)
val cell = ref one
val () = (cell := zero)

(*[ result <: pos ]*)
val result = !cell
```

Here we create a reference `cell` that initially contains `one`. (We assume that `one` and `zero` have the expected refinements `pos` and `nat`.) Since `one` has sort `pos` we can assign `ref one` the sort `pos ref`, and since `one` has sort `nat` we can assign `ref one` the sort `nat ref`. The standard rule for intersection introduction then allows us to assign `ref one` the intersection of these two sorts `(pos ref) & (nat ref)`.

This leads to unsoundness, because the first part of the intersection allows us to update `cell` with `zero`, while the second part of the intersection allows us to conclude that reading the contents of `cell` will only return values with sort `pos`. Hence, standard intersection types allow us to assign `result` the sort `pos` when it will actually be bound to the value `zero`, which is clearly incorrect.

Our solution to this problem is to restrict the introduction of intersections to values. Our restricted form of intersection introduction states that if we can assign a value V the sort R and also the sort S then we can assign the intersection of those sorts $R \& S$. (The general form of the rule allows any expression, not just values.)

This follows the value restriction on parametric polymorphism in the revised definition of SML [MTHM97], which was first proposed by Wright and Felleisen [WF94]. In our case we find that we additionally need to remove one of the standard subtyping rules for intersection types. The resulting system has some pleasing properties, and seems even better suited to bidirectional checking than standard intersection types.

4.3 Datasorts and pattern matching

The second main technical development is a practical approach to refinements of ML datatypes, including the sort checking of sequential pattern matching. Following previous work on sorts, we focus on refinements which are introduced using a mechanism for refining datatypes using recursive definitions. These refinements are particularly appropriate for ML because datatypes play an important role in the language: e.g. conditional control-flow is generally achieved by pattern matching with datatypes.

We illustrate the expressiveness of these refinements with an example. Suppose we have a program that includes the following ML datatype for strings of bits.

```
datatype bits = bnil | b0 of bits | b1 of bits
```

Further, suppose that in part of the program this datatype is used to represent natural numbers with the least significant digits at the beginning of the string. To ensure that there is a unique representation of each number, the program uses the following representation invariant: a natural number should have no zeros in the most significant positions (i.e. at the end). We can capture this invariant with the following *datasort* declarations, which define refinements of the datatype `bits`.

```
(*[ datasort nat = bnil | b0 of pos | b1 of nat
         and pos =        b0 of pos | b1 of nat ]*)
```

The syntax for datasort declarations mirrors that that of the datatype declarations which are being refined, except that some value constructors may be omitted and some may appear more than once with different sorts for the constructor argument. The datasort `nat` represents valid natural numbers, while `pos` represents valid positive natural numbers. In this declaration, the datasort `pos` is necessary in order to define `nat`, since `b0 bnil` is not a valid representation of a natural number. The inclusion `pos` $\leq$ `nat` clearly holds for these two datasorts, just like the refinements of the type `num` that we considered in Section 3.

In general, we would like to determine which inclusions hold for a particular set of datasort declarations. We differ from previous work on refinements in that we formulate an algorithm for determining which inclusions hold that is complete with respect to an inductive semantics in the case when the recursive definitions correspond to a regular-tree grammar. This makes it easier for a programmer to determine which inclusions should hold. We also show how to extend our approach to refinements of datatypes which include functions and references, including datatypes with recursion in contravariant positions, unlike previous work on refinements. We can no longer formulate an inductive semantics in this case, but our experience suggests that this extension validates those inclusions that are intuitively expected, and that forbidding such refinements would be limiting in practice.

For convenience, each datatype has a *default refinement* which has the same name as the datatype, and a datasort declaration that mirrors the datatype declaration. Thus, we can think of the above datatype declaration as also including the following declaration.

```
(*[ datatype bits = bnil | b0 of bits | b1 of bits ]*)
```

These datasorts make it easy to provide sorts that do no more checking than done by the type system. Further, our sort checker uses these default refinements when annotations are missing in positions required by our bidirectional algorithm, to ensure that we have a conservative extension of SML.

In the presence of datasort declarations, sort checking the pattern matching construct of ML presents a number of challenges. For example, consider the following code for a function which standardizes an arbitrary bit string by removing zeros at the end to satisfy the sort `nat`.

```
(*[ stdize <: bits -> nat ]*)
fun stdize bnil = bnil
  | stdize (b0 x) = (case stdize x
                       of bnil => bnil
                        | y => b0 y
                     )
  | stdize (b1 x) = b1 (stdize x)
```

The inner pattern match is the most interesting here. To check the branch "`y => b0 y`" it is critical that we take account of the sequential nature of ML pattern matching to determine that `y` can not be `bnil`. We achieve this

by using a generalized form of sorts for patterns that accurately capture the values matched by previous patterns.

The example above is relatively simple; in the presence of nested patterns and products the situation is considerably more complicated, and generally requires "reasoning by case analysis" to check the body of each branch. When we perform such an analysis, we avoid the "splitting" into unions of basic components that was used in previous work on refinements. This is because it leads to a potential explosion in the case analysis that needs to be performed. We instead focus on *inversion principles* that are determined relatively directly from datasort declarations.

4.4 Extending to a sort checker for Standard ML

We tackled a number of other smaller challenges while extending sort checking to the full SML language. The following are some of the more notable related to the design of sort checking (as opposed to its implementation). When a required annotation is missing during sort checking of expressions, we use a default refinement that results in similar checking to that performed during type checking. We allow parameterized datasort declarations with variance annotations for each parameter. When we have a type sharing specification for types which have refinements, we also share all refinements according to their names. We allow the specification of refinements of opaque types in signatures, including specifications of the inclusions that should hold between them.

We also briefly mention some of the more notable challenges tackled while implementing our sort checker for SML. The implementation of the datasort inclusion algorithm uses a sophisticated form of memoization and other techniques in order to obtain acceptable performance. The implementation of the bidirectional checking algorithm uses a library of combinators for computations with backtracking and error messages which is designed to be efficient and allow a natural style of programming. The implementation of checking for pattern matching uses some important optimizations, such as avoiding redundancy during case analysis.

Our experiments with our implementation indicate that refinement-type checking is a practical and useful method for capturing and checking properties of real SML code. We found that the annotations required were very reasonable in most cases, and that the time taken to check was at most a few seconds. In general, the error messages produced were even more informative than those produced by an SML compiler for similar errors: this is because bidirectional checking localizes the effect of errors better than type

inference based on unification (as generally used for SML).

5 Introductory examples

We now show some additional examples. These use the type for bit strings and the sorts for the associated representation invariant for natural numbers that were introduced in the previous section.

```
datatype bits = bnil | b0 of bits | b1 of bits

(*[ datasort nat = bnil | b0 of pos | b1 of nat
         and pos =        b0 of pos | b1 of nat ]*)
```

We start with a very simple example: the constant four. As expected it has the sort `pos`. This sort could be inferred, so the annotation is not required, but serves as handy, mechanically checked documentation.

```
(*[ four <: pos ]*)
val four = b0 (b0 (b1 bnil))
```

In contrast, the following bit string consisting of three zeros is not even a natural number. The best sort it can be assigned is `bits`, i.e. the default refinement of the type bits. Again, this sort could be inferred.

```
(*[ zzz <: bits ]*)
val zzz = b0 (b0 (b0 bnil))
```

The next example is a function that increments the binary representation of a number.

```
(*[ inc <:  nat -> pos ]*)
fun inc bnil = b1 bnil
  | inc (b0 x) = b1 x
  | inc (b1 x) = b0 (inc x)
```

We remark that ascribing `inc <: nat -> nat` instead of `nat -> pos` would be insufficient: in order to determine that the last clause returns a valid natural number, we need to know that the result of the recursive call `inc x` is positive. This is reminiscent of the technique of strengthening an induction hypothesis which is commonly used in inductive proofs. Our experience

indicates that it is not too hard for a programmer to determine when such stronger sorts are required for recursive calls, and in fact they must be aware of the corresponding properties in order to write the code correctly. In fact, using a sort checker actually helps a programmer to understand the properties their code, and thus allows correct code to be written more easily, particularly when the properties are complicated.

We further remark that subsorting allows us to derive additional sorts for `inc`, including `nat -> nat`, via inclusions such as the following.

```
nat -> pos  ≤  nat -> nat
nat -> pos  ≤  pos -> pos
```

We also remark that the sort we have ascribed will result in each call to `inc` having its argument checked against the sort `nat`, with an error message being produced if this fails. If some parts of the program were designed manipulate natural numbers that temporarily violate the invariant (and perhaps later restore it using `stdize`), it might be appropriate to instead ascribe the following sort.

```
(*[ inc <: (nat -> pos) & (bits -> bits) ]*)
```

It seems reasonable to ascribe this sort, since the result returned is appropriate even when the invariant is violated.

Our next example is a function which adds together two binary natural numbers.

```
(*[ plus <:  (nat -> nat -> nat)
           & (nat -> pos -> pos) & (pos -> nat -> pos) ]*)
fun plus bnil n = n
  | plus m bnil = m
  | plus (b0 m) (b0 n) = b0 (plus m n)
  | plus (b0 m) (b1 n) = b1 (plus m n)
  | plus (b1 m) (b0 n) = b1 (plus m n)
  | plus (b1 m) (b1 n) = b0 (inc (plus m n))
```

Again, for this definition to sort-check we need to know that `inc <: nat -> pos`. We also need a stronger sort than `plus <: nat -> nat -> nat` for the recursive calls. For these, the following sort would have sufficed, but subsequent calls to `plus` would have less information about its behavior.

```
(*[ plus <: (nat -> nat -> nat) & (pop -> pos -> pos) ]*)
```

Next, we show an example of an error that is caught by sort checking.

```
(*[ double <: (nat -> nat) & (pos -> pos) ]*)
fun double n = b0 n
```

Our sort checker prints the following error message for this code.

```
  fun double n = b0 n
                 ^^^^
  Sort mismatch: bits
      expecting: nat
```

This tells us that the expression `b0 n` has only the sort `bits`, but should have sort `nat`. To figure out why it does not have sort `nat`, we can look at the datasort declaration for `nat`: it specifies that `b0` must only be applied to arguments with sort `pos`. Indeed, `double bnil` evaluates to `b0 bnil` which is not a valid representation of zero. We are thus led towards an appropriate fix for this problem: we add an additional case for `bnil`, as follows.

```
(*[ double <: (nat -> nat) & (pos -> pos) ]*)
fun double bnil = bnil
  | double n = b0 n
```

We conclude this section with an example which shows some of the expressive power of sort checking with recursive datasort declarations. This example uses datasort declarations to capture the parity of a bit string, and verifies that a function that appends a one bit to the end of a bit string appropriately alters the parity.

```
(*[ datasort evPar = bnil | b0 of evPar | b1 of odPar
         and odPar =        b0 of odPar | b1 of evPar ]*)

(*[ append1 <: (evPar -> odPar) & (odPar -> evPar)    ]*)
fun append1 bnil = b1 bnil
  | append1 (b0 bs) = b0 (append1 bs)
  | append1 (b1 bs) = b1 (append1 bs)
```

The full power of sorts is not demonstrated by this small example. It it best demonstrated in the context of real code with all its complexities. See the experiments in Chapter 9 of the dissertation for examples of real code with sort annotations.

6 Organization of the dissertation

The dissertation is organized as follows.

Chapter 1 is the introduction, and is similar to this thesis summary, although additionally contains a substantial section on related work.

Chapter 2 focuses on sorts and sort checking for a λ-calculus using the standard intersection type rules.

Chapter 3 focuses on sorts and sort checking for a λ-calculus using a value restriction on intersections, and modified subtyping for intersections so that the language is suitable for extension with effects.

Chapter 4 demonstrates that our value restriction and modified subtyping result in a system that is sound in presence of effects by adding mutable references and an example datatype.

Chapter 5 presents our approach to datasort declarations, including our algorithm for comparing them for inclusion.

Chapter 6 considers sort checking in the presence of datasort declarations, including pattern matching.

Chapter 7 describes the design of an extension of our approach to the full set of features in Standard ML.

Chapter 8 describes our implementation of a sort checker for SML, based on the design and algorithms in previous chapters.

Chapter 9 reports some experiments with sort checking of real SML code using our implementation.

Chapter 10 concludes.

References

[CDV81] Mario Coppo, Mariangiola Dezani-Ciancaglini, and Betti Venneri. Functional character of solvable terms. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 27:45–58, 1981.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[Cur34] H. B. Curry. Functionality in combinatory logic. *Proceedings of the National Academy of Sciences, U.S.A.*, 20:584–590, 1934.

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

[FP91] Tim Freeman and Frank Pfenning. Refinement types for ML. In *Proceedings of the SIGPLAN '91 Symposium on Language Design and Implementation*, pages 268–277, Toronto, Ontario, June 1991. ACM Press.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[Pie97] Benjamin C. Pierce. Intersection types and bounded polymorphism. *Mathematical Structures in Computer Science*, 7(2):129–193, April 1997. Summary in *Typed Lambda Calculi and Applications*, March 1993, pp. 346–360.

[PT98] Benjamin C. Pierce and David N. Turner. Local type inference. In *Conference Record of POPL '98: the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1998. Full version to appear in *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2000.

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

[Rey02] John C. Reynolds. What do types mean? — From intrinsic to extrinsic semantics. In Annabelle McIver and Carroll Morgan, editors, *Essays on Programming Methodology*. Springer-Verlag, New York, 2002.

[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.