A Refinement-Type Checker for Standard ML

Rowan Davies*

Carnegie Mellon University Department of Computer Science,
5000 Forbes Ave, Pittsburgh PA 15213, USA

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 more fundamental 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 the benefits of static typing is to extend an existing language so that each ordinary type is refined by a number of *refinement types*, which allow many common program properties 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.

We have designed an extension of the programming language Standard ML which includes refinement types, and have built a practical refinement-type checker for it which handles the full language, excluding modules. Our extension allows the programmer to define refinements of ML datatypes. For example, non-empty lists could be defined as a refinement of lists in order to express the invariant that a particular function always returns a non-empty list, thus allowing this invariant to be automatically checked by the compiler.

To better illustrate the use of refinement types (which we also call "sorts") in our extension, 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-terms of multiplications. Additionally, in some parts of the application only ground terms are manipulated, i.e. terms which contain no variables. The following small code sample shows how invariants like this can be captured with refinement types:

```
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_pr = [prod] | Plus of sum_pr * sum_pr
datasort grnd = Num of int | Plus of grnd * grnd | Times of grnd * grnd

fun mult(Plus(st1a, st1b), st2) = Plus(mult(st1a, st2), mult(st1b, st2))
  | mult(st1, Plus(st2a, st2b)) = Plus(mult(st1, st2a), mult(st1, st2b))
  | mult(st1, st2) = Times (st1, st2)
withsort mult :> (sum_pr * sum_pr -> sum_pr) & (grnd * grnd -> grnd)
```

* Partially sponsored by the Advanced Research Projects Agency under the title "The Fox Project: Advanced Languages for Systems Software", ARPA Order No. C533.

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 our extension to ML. These refinements correspond to the restricted forms required by the invariants of our application.

We then define an example function from our application, `mult`, which takes two expressions which are sums of products, and returns a sum of products expression for their product. Additionally, we have the invariant that if both input expressions are ground, it returns a ground expression. We express these invariants of the `mult` function by attaching a `withsort` clause to it, which is another one of our extensions to ML. This clause assigns a sort to the function which includes the symbol `&`, which intuitively means that the function must have both of the sorts on either side of the `&`, each of which corresponds to one of the invariants.

If we now run our refinement-type checker on this piece of code, it will succeed, thus verifying that the invariants hold. If instead there was an error in our code, such that these invariants did not hold, then refinement-type checking would fail with an appropriate error message, just like the error messages that would be generated by the compiler if there was an ordinary type error in the program.

The syntax of sorts exactly follows that of the existing ML types, with the addition that `&` may appear anywhere in a sort to conjoin two sorts which refine the same type. In order to make sorts manageable, we define a subsorting relation on them, similar to a subtyping relation. As an example of this, the sort `prod` is a subsort of `sum_pr`, and so the function `mult` can actually be applied to two variables which both have sort `prod`, to yield a result of sort `sum_pr`.

Our algorithms for refinement-type checking are quite different from ordinary type checking. Refinement-type inference is problematic, because programs often satisfy many accidental properties, all of which must be reflected in the inferred sorts. Thus, our algorithms depend on the programmer stating the important properties of a program using annotations such as `withsort`. These annotations are also very useful as reliable documentation for the code, particularly since they have been mechanically checked for correctness.

Our experience with adding refinement types to existing SML code indicates that our algorithms are efficient enough to be run every time a program is compiled. As an example of these experiments, we were able to check that an implementation of red-black balanced binary trees maintains the critical invariant that no red node appears immediately below another red node.

The refinement-type checker was built as part of the author's PhD thesis work, and he would like to give special thanks to his advisor, Frank Pfenning. For more information about refinement types, including a number of papers and some extended examples, see the web page:

`http://www.cs.cmu.edu/~rowan/sorts.html`