Elaboration and Phase-Splitting in the TIL/ML Compiler

Chris Stone

1 Introduction

Experiments with a prototype of the TIL compiler[1] for Standard ML (SML) showed significant improvements in code speed and memory usage could be achieved by retaining types throughout compilation where they can be used by the generated code at run-time. Uses of types include optimization of data representations, achieving tag-free or mostly-tag-free garbage collection, and guiding compilation; furthermore, being able to type-check the internal representation of a program at all times is very useful for debugging our compiler optimizations.

However, the TIL prototype only compiled "Core SML", restricting us to relatively small examples; larger programs normally use the entire language. The work discussed here outlines a re-implementation of the TIL/ML compiler front-end to handle the entire language.[2]

2 Translating Standard ML

In implementing a compiler for SML, an obvious place from which to start is its specification, the *Definition of Standard ML*. This defines a relation

$$\vdash_{static} \textit{SML-expression} \Rightarrow \textit{SML-type}$$

holding if the *SML-expression* is well-formed and has type *SML-type*. Separately, an evaluation relation is defined so that

$$\vdash_{dynamic} \textit{SML-expression} \Rightarrow \textit{SML-value}$$

holds if the *SML-expression* evaluates to the final value *SML-value*.

However, the separation of typing and evaluation in the *Definition* does not match our goal of having types as run-time values. In particular, the *Definition* explicitly deletes all type annotations from a program in defining how evaluation occurs. It would be nontrivial to show that our type-passing implementation mimics the type-free evaluation formally described in the *Definition*. Therefore, Bob Harper and I have explored an approach to defining the SML language geared specifically toward our purposes: we formally specify "elaboration" of SML expressions, giving a translation of terms from SML into an explicitly-typed extended lambda calculus, called HIL (High-level Intermediate Language). The emphasis here is on *type reconstruction* (discovering the type of each sub-expression, and annotating the program appropriately) rather than type checking (asking only whether the program as a whole has a type). In analogy with the *Definition* our elaboration can be described as:

$$\vdash_{static} \textit{SML-expression} \leadsto \textit{HIL-expression} : \textit{HIL-type}$$

We then define a Standard ML program to be well-formed if it can be successfully translated; the meaning of such a program is defined to be the meaning of its translation. This indirection is useful because the target language of the elaboration can be easily given a direct semantics in which types may be calculated and passed as parameters at run-time; furthermore, the final answer has the same type as the initial program:

$$\vdash_{dynamic} \textit{HIL-expression} : \textit{HIL-type} \leadsto \textit{HIL-value} : \textit{HIL-type}.$$

[1] The thesis work of Greg Morrisett and Dave Tarditi

[2] See the presentations of Perry Cheng, Andrew Bernard, and Leaf Petersen for more details about the "middle-end" and "back-end" of the compiler.

HIL is then the first of the TIL/ML compiler's intermediate languages. It can be viewed as a very simplified version of SML: it has first-class functions, exceptions, references, and a module system based upon the thesis work of Mark Lillibridge. Unlike SML, it requires explicit type annotations for all variables, and has no polymorphism, datatypes, or pattern-matching.

Features of the elaboration include:

- **Type reconstruction.** A substantial part of the elaborator is simply determining the types of variables and functions arguments; if these types are consistent, they are used to annotate the translated code.

```
fun inc n = n + 1   ⇝   fun inc (n:int):int = n + 1
```

- **Eliminating polymorphism.** Polymorphism can be considered a function which takes some type arguments and returns a value. However, SML also has the concept of a "structure" (a package containing types and values) and a "functor" (a parameterized structure, or equivalently a function mapping structures to structures). Therefore, a polymorphic function can be implemented as a functor which takes a structure containing types, and returns (a structure containing) a value. All polymorphism can be eliminated by the translation, turning into a stylized use of functors.

```
fun id (x:'a):'a = x    ⇝   functor id(type t) =
                            struct
val n = id 3                  fun it (x:t):t = x
val s = id "foo"            end

                            val n:int = (id(type t = int).it) 3
                            val s:string = (id(type t = string).it) "foo"
```

- **Eliminating Datatypes.** One can create abstract types in SML by the use of "datatypes," which combine the functionality of enumerated types, disjoint unions, and recursive types. However, a datatype can be logically decomposed into: an abstract type, ways to create values of this type, and ways to deconstruct values of this type. The SML module system also allows creation of such ADTs, and so we can replace all datatypes with a stylized use of structures.

```
datatype color = RED | BLUE   ⇝   signature color =
                                  sig
                                    type color
                                    val RED : color
                                    val BLUE : color
                                    val eq : color*color -> bool
                                    val expose: color -> unit+unit
                                  end
```

- **Pattern Compilation.** Variable bindings involving patterns are decomposed into simple variable bindings. Similarly, case analyzes involving complex patterns are decomposed into decision trees involving simpler equality and tag checking operations, nested appropriately.

```
val (n,s) = (1+2,"foo")   ⇝   val tmp : int*string = (1+2,"foo")
                              val n : int = tmp # 1
                              val s : string = tmp # 2
```

- **Structure Patching.** SML allows "after the fact" patching of type specifications: "sharing type" adds a constraint that two types be equal, while "where type" adds a definition of a type. These can both be eliminated in favor of type definitions at the point where the types are first declared.

```
sig                          ⇝   sig
  type t                           type t = int
  type u                           type u = t
  sharing type t = u             end
end
  where type t = int
```

- **Equality Compilation.** Equality polymorphism and equality types are eliminated in terms of passing equality functions at run-time.

```
fun f (x:''a):bool = (x=x)   ⇝   functor f(type t; val eq : t*t->bool) =
                                 struct
                                   val it : bool = eq(x,x)
                                 end
```

- **Eliminating Module Subtyping.** In SML, a context expecting a structure with certain components can always be given a structure with extra components. The internal language requires exact matches. Therefore, the elaboration inserts explicit coercions to "trim" structures as necessary.

- **Propagating "Hidden" Types.** In SML, it is possible to have an expression whose type cannot be expressed. In contrast, the type-based internal language insists that every value or expression must have a nameable type. This is reflected in the compiler by the requirement that the type of all values must be determinable at run-time (e.g., to determine how the value is garbage collected, etc.). In order to permit such SML code, the elaborator does program rewriting when necessary to pass values in tandem with their types.

3 Translating HIL

As mentioned above, the HIL still contains constructs directly corresponding to the structures and functors of the SML module system. To remove these, we pass to a second intermediate language, called MIL (Mid-level Intermediate Language), with a simpler syntax but a noticeably more complex type system. This translation forms the second pass of the TIL/ML compiler. This translation is "type preserving" in the sense that all well-typed HIL programs become well-typed MIL programs. Furthermore, HIL types are translated to MIL types, in such a way that the translation of programs matches the translation of their types.

The MIL is essentially a polymorphic lambda-calculus with a fancy type system.[3] The primary non-standard feature is the ability to case-analyze types at runtime in order to calculate values, or even other types.[4] The MIL is general enough that it can be used to represent the code before, during, and after optimizations, and even after closure-conversion when all higher-order functions have been eliminated. Importantly for compiler-debugging purposes, this means that a program can be typechecked after some or all of these complicated program transformations take place to give evidence that the program has not been corrupted. The compiler's "middle-end" repeatedly rewrites the MIL program until essentially the point where assembly code is generated.[5]

The MIL language has no module constructs; perhaps surprisingly, though we turned all SML polymorphism into HIL functors, we now make polymorphic functions primitive and use these to represent HIL functors! Only functors support type abstraction, but polymorphism is simpler. Therefore if you want to simplify the language by providing only one of these, in a language with abstract types (like SML or HIL) it is best to represent polymorphism as functors, and otherwise (MIL) it is best to represent functors as polymorphism.

The main features of the translation into MIL are:

[3] See Leaf Petersen's presentation for more details on MIL typechecking.
[4] See Perry Cheng's presentation for more details on the use of this and other MIL features.
[5] See Andrew Bernard's presentation for more details on the back-end of the compiler.

- **Phase Splitting.** This translation use the ideas of Harper, Mitchell, and Moggi to "phase split" HIL modules into two pieces: their type-level part, and their expression-level part. For example, structures are aggregates containing both types and values; they can therefore be split into a record of types and a record of values.

  Functors map types and values in one structure to types and values in another structure. However, types in the result can only depend on types (not values!) in the argument. This means that we can also split the functors into two parts: their behavior on types, which can be (expressed by a function mapping records of types to records of types) and their behavior on values (expressed as a polymorphic function).

  ```
  functor D(type t; val x : t) =     ⇝    typefun D({t:TYPE}):{u:TYPE} = {u=t*t}
  struct                                  fun D({t:TYPE},{x:t}):{y:t*t} = {y=(x,x)}
    type u = t*t
    val y : u = (x,x)
  end
  ```

  Functor applications then split into an application of a type-level function and an application of a polymorphic function.

  Similarly, signatures (the "type" of a structure) are split are split into a record type and a record kind. (Types and collections of types are classified by kinds in exactly the same way that values are classified by their types.)

- **Eliminating Type Abstraction.** In the HIL, abstract types are achieved by "ignoring" the definitions of certain types during typechecking. For example, if one wanted to have a stack abstract type, this could be implemented with a list and operations which work on lists. One can then force the rest of the program to be typechecked as if stacks and lists were different; this verifies that only stack operations (and not list operations) are being used on stacks.

  This distinction is very good software engineering methodology. However, once the compiler has verified (in the elaborator) that abstraction is being respected, it should be free to take full advantage of its knowledge of how abstract data types are really implemented. Explicit type abstraction is therefore discarded at this point.

4 Discussion

We have re-implemented a substantial portion of the TIL/ML compiler, including the phases discussed here. Though the compiler is not yet complete, we have already been able to compile and run larger and more complex programs than the prototype system could handle.

Formally specifying the elaboration phase of the compiler turned out to be a very useful exercise. It showed that all of Standard ML could translated into a theoretically-based typed intermediate language, all of whose features are relatively well-studied (at least in isolation). The specification was also very useful in guiding the implementation of the TIL/ML elaborator, which follows the specification fairly closely.

Whether this is the best way to define the source language is not completely clear. Since the meanings of programs are given in terms of the translation, and the HIL calculus is fairly clean, it seems plausible that a proof that "well-typed programs won't go wrong" would be much simpler in our framework. On the other hand, to really show that we have given a definition would require a formal argument for *coherence* (i.e., that the same SML program cannot be given two different translations with different meanings). Furthermore, the translation can be fairly involved at points, and requires the reader to understand the HIL calculus. This last point may be mitigated however, if we could show that the HIL (or some minor extension) is a suitable target for translating other languages such as Scheme, Java, or Modula-3. We hope to explore this issue further.

The phase-splitting transformation also seems to work well in practice. Our choice to take this route has also been validated to some extent by the decision of the SML/NJ compiler writers to add a similar phase in their compiler.