The Design of the TILT Compiler for SML

P. Cheng R. Harper G. Morrisett L. Petersen C. Stone

June 3, 2021

1 The Structure of TILT

The TILT compiler consists of 3 main phases: the front end, the middle end, and the back end. The front end, also called the elaborator, is responsible for type-checking the source ML program and converting the program into our high-level intermediate language (HIL); the HIL can be considered an explicitly typed, stripped-down version of SML. Code is then converted via a transformation called phase-splitting which removes modules and signatures; this yields a program in the mid-level intermediate language (MIL). Optimization and closure conversion are implemented as transformations mapping MIL code to MIL code; we are able to typecheck the program throughout the process, even after closure conversion. Finally, we convert the program to a RTL pseudo-assembly program, which the back end immediately translates to register-allocated machine-specific assembly code. We currently use the system asembler to produce object files linkable with the runtime.

2 Front End

The specification of the HIL and the elaborator have been discussed elsewhere [**?**, **?**]. We here only give a high-level overview.

2.1 High-Level Intermediate Language

The HIL is an explicitly typed language containing both core and module-level parts. The syntactic classes include expressions, constructors (types, record of types, and functions from types to types), kinds, modules, and signatures. Expressions are classified by types, and constructors are classified by kinds. Similarly, modules are classified by signatures.

Like SML, the HIL has first-class functions, exceptions, references, and a module system. Unlike SML, there is no polymorphism or pattern-matching. Rather than built-in datatypes, the HIL has generic recursive type constructors and sum types.

2.2 Elaboration

2.2.1 Highlights

The major jobs of the elaborator include:

- **Type reconstruction.** A substantial part of the elaborator (though not the specification, which is written nondeterministically) is simply determining the types of variables and functions arguments; if these types are consistent, they are used to annotate the translated code.
- **Identifier resolution.** Variables with the same name in the external program may have different names in the translated code; the translation must resolve the scopes of variables. This includes dealing with the SML's `open` construct.

- **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.

- **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.

- **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.

- **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.

- **Equality Compilation.** Equality polymorphism and equality types are eliminated in terms of passing equality functions at run-time. Alternatively, we could postulate a primitive operation implementing polymorphic equality. However, the elaborator would still have to do all the same bookkeeping in order to typecheck uses of SML equality types and equality polymorphism.

- **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.** There are SML expressions whose type cannot be expressed in SML. 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.

2.2.2 Implementation Issues

- **Type inference.** [This section is fairly standard and should probably be omitted.] Throughout elaboration, types are inferred using a standard unification algorithm. To achieve type inference for ML, three type constructs (type meta-variable, flexible record types, and overloaded types) and one term constrct (overloaded expression) were added to HIL.

  Whenever we encounter a variable binding without an explicit type, we create a new type meta-variable to stand for the unknown type. For a well-typed program, the meta-variable will eventually be unified with a type. (Note that type variables and type meta-variable are different. Type variables are part of the HIL language whereas meta-variable exist solely for the purpose of type inference.)

  Flexible record types are needed to typecheck the ML projection construct "`#`" and "`...`". When unified with record types, they become inflexible. Two flexible record types can also be unified to produce a consistent union flexible record type.

  Overloaded types are needed to typecheck overloaded identifiers like "+". An overloaded type is a disjunction of several types. During unification with other types, the choices of overloaded types are pruned until only one choice remains. Corresponding to each overloaded type is an overloaded expression which consists of the same number of choices. When an overloaded type is narrowed, the corresponding expression is narrowed.

If there is still an unresolved meta-variable, flexible record type, or overloaded type when the compilation unit has been elaborated, an appropriate default is taken. Meta-variables are resolved to the unit type, flexible record types are made inflexible without additional widening, and the default choice of the overloaded type is used.

Meta-variables include two flags indicating whether it is an equality type and whether it may be generalized. Meta-variables that have been unified with overloaded types may no longer be generalized. At a valuable binding, type meta-variables created during elaboration of the bound expression that have not been unified or marked ungeneralizable may be generalized to yield a polymorphic value binding. If an equality type is required the generalization must parameterize over the unknown type and an equality function at that type. The translation of an ML polymorphic value is a HIL module which takes a structure containing types and equality functions.

- **Representation of Contexts.** When a signature is added to the translation context, the signature is "selfifed" with respect to the path that the signature describes. This permits the classifier of the translations of paths to be determined quickly since there will be no dependencies in the selfified signature. Transparent type components are selfified to replace occurrences of previous components with self paths while opaque type components are made transparent with a self path.

  During type inference, types are normalized before being unified. Normalization involves beta reduction and looking up type definitions in the context. Since it is apparent which projections are true type definitions and which arose from selfified opaque components, normalization will not enter a loop.

  When a structure path is locally bound (just inside a let or local), the bound variable becomes an alias. Occurrences of that variable will be replaced by the path. [This may be irrelevant given the pre-projection performed in phase-splitting.]

  Given a label, the context normally returns a path and a classifier. Sometimes, an expression rather than a path is returned. This permits primitives and datatype constructors to be inlined. This may be extended to inline expressions, types, and modules.

- **Beta Reduction.** Whenever there is an application in the ML code, the translation is reduced if the function is a lambda. If the argument is a variable, the varible is substituted into body; otherwise, the argument is let-bound around the body. In conjunction with the inlining, this will get rid of many small functions and their calls.

- **Primitives.** The HIL datatype also contains eta-expanded primitives (primitives that are applied only to their type arguments) in addition to fully-applied primitives. This theoretically permits only one copy of each primitive at each type for the whole program. In contrast, eta-expanding the primitives with a lambda can cause many small functions to be generated if the primitives are not applied and thus reducible. ML primitives are translated to eta-expanded HIL primitives. Since most primitives in ML are imediately applied, their translation becomes applications of eta-expanded primitives which are immediately reduced by beta-reduction.

- **Differences from the Specification.** There are other minor differences from the "Tech Report." For efficiency reasons, "let" is not a derived form. Cases analysis on exception tags are n-ary. Pattern compilation is different. (No exceptions are used and decision tree generated is not ridiculous.) Datatype definitions are compiled to reduce structure nesting. Polymorphic arguments are compiled to be a flat structure taking types and equality functions rather than a list of strucutres each containing one type and possibly one equality function.

3 Middle End

3.1 Mid-Level Intermediate Language

The *Middle Intermediate Language* (MIL) is an explicitly-typed polymorphic lambda calculus. In contrast to the HIL, polymorphism is directly supported but there are no module constructs.

The kind structure is rather complex, and has a subkinding relation. All types classifying values have kind `T`, but all types classifying values which fit in one machine word (for TIL, everything except unboxed floating-point values) have the more-specific kind `W`. Furthermore, we allow still more-precise *singleton kinds* [**?**], the kind of types equivalent to a particular type. For technical reasons, we restrict singleton kinds to types of kind `W` or `T`, but singletons at higher kinds can be encoded in terms of these.

Furthermore, we have dependent record kinds (the kind of records whose components are constructors) and dependent function kinds (the kind of functions mapping constructors to constructors).

3.2 Design Considerations

3.2.1 Functions

MIL functions and arrow types are multi-argument. Further, the distinction in kinds permit functions to take two set of arguments: word arguments and float-point arguments. These refinements in functions reflects the realities of modern architectures: registers are fast and seprated into general-purpose and floating-point. By making these separations in MIL, we permit MIL optimizations to take advantage of the archtecture.

Term functions in MIL take potentially take type and term arguments simultaneously. Similarly, all applications can pass both type and term arguments. This was designed so that applications of polymorphic functions can be made with one call. By currying, it is still possible to take only type arguments and perform type specialization.

Though not yet implemented, we plan to perform a systematic argument flattening transformation as in TIL. As in TIL, MIL has `vararg` and `Vararg` as matched term-level and constructor-level primitives. These [do stuff]. Details can be found in [**?**].

Importantly, these wrappers are added statically, rather than being added dynamically at run-time. Therefore, there is no chance of functions being repeatedly wrapped at run-time, as can happen with coercion-based approaches to argument flattening.

If only bounded record flattening is desired, `vararg` and `Vararg` are definable in terms of `typecase` and `Typecase`. However having these as primitives reduces code bloat and speeds reduction steps in the optimizer. This approach also faciliates experimenting with versions of `vararg` and `Vararg` that flatten arguments in a way that is inexpressible with `typecase` and `Typecase` alone. [Such as?]

Since closure conversion occurs as a MIL transformation and we want to be able to typecheck the closure converted result, we distinguish between open functions and code functions and give them different types. In order to typecheck, code functions must be closed with respect to non top-level term and constructor variables. However, it may be open with respect to any top-level variables as well as type variables within scope. Finally, closures are created from aggregating code functions with their environment. Each of these three constructs are given a different arrow type. Similarly, applications may be open, closed, or code. After closure-conversion, there will be no open lambdas, open arrow types, or open application calls.

3.2.2 Boxing

To support efficient floating-point arithmetic, we make a distinction between the types `float` and `boxedfloat`; only the latter has kind `W`. Consequently, we also make `box` and `unbox` primitive operations are explicit so that the optimizer can precisely specify when boxing and unboxing should occur.

3.2.3 Representation of sums

It is very important to optimize the representation of datatypes. [Explain what would happen to `list` if we were very naive?]

For example, the datatype

```
datatype bool = true | false
```

is represented by the type `unit + unit`, whose values can be represented as the integers 0 and 1. Similarly, the datatype

```
datatype intoption = NONE | SOME of int
```

which encodes the type `unit + int` would be represented as either the null pointer (the integer 0), or a pointer to an integer.

Note that the representation of the second case of the sum varies with the type. Naively certain datatypes such as

```
datatype 'a option = NONE | SOME of 'a
```

would therefore require the `SOME` datatype constructor to test whether the type of its argument is `unit` or not, in order to choose the correct representation.

In our scheme, we decide at compile-time that we will never specialize the result of applying `SOME` whether `'a` is instantiated at type `unit` or not; the result of `SOME ()` will always be a pointer to the unit value. This is represented by recording within our sum types a constant integer which represents the minimal number of `unit` arms which we are sure are present, and whose representations should be optimized to small integers as in the `bool` case above.

Furthermore, since datatype constructors carrying records are a common ML idiom (e.g., cons cells in lists), we would like a particularly efficient data representation. That is, rather than a tagged pointer to a record, we would like the tag and the record to be laid out in consecutive memory locations. In general, this means that the `inject` primitive (injecting a value into a sum type) must dispatch on the type of the value being tagged. However, the head normal-form is often known to be a record. In this case, we can optimize the inject primitive into the `inject_record` primitive which does not use its type argument. Similarly, at deconstruction, we convert as many occurrences of `project_sum` into uses of `project_sum_record` as possible.

3.2.4 Intensional Type Analysis

We include a `typecase` term-level operation and a corresponding `Typecase` constructor-level operation for performing intensional type analysis at runtime. [?]

We briefly considered an alternative design without these primitives; type dispatch could be contained within certain polymorphic primatives (such as array subscripting and update). However, it is important to pull type analysis out of loops. Though we could design the language such that the applications of types to primitives can occur once outside a loop with the result being used each time around the loop, there is no good way to compile this directly; making these primitives into function calls is likely to be slower than doing the type analysis.

The arms of a typecase may be optimized based on the extra type information present with the typecase.

Unlike TIL, TILT only supports typecase rather than typerec. [What do we lose without typerec?]

3.3 Phase-Splitting

The translation from HIL to MIL is accomplished via a "phase-splitting" transformation based on the work of Harper, Mitchell, and Moggi [?]. The primary result is that every HIL module is separated into a constructor-level piece and an expression-level piece. Structures turn into a record of types and a record of values. Functors turn into a function mapping types to types and a polymorphic function mapping values to values. (The function is polymorphic since the values returned by the functor may have types depending on the type inputs of the functor.)

The primary difference from the HMM calculus is that the MIL has no primitive structure construct. Instead, the association between the type-level and expression-level parts of a module is maintained at the metalevel.

One difficulty in using the phase-splitting transformation is that it doesn't match the "type generativity" of SML functors. Due to datatypes and opaque signature ascription, applying the same functor twice to exactly the same input should yield "different" abstract types for the two applications. In contrast, all functions on types at the MIL constructor level are total (or "functional") in the sense that they return exactly the same type every time they are given the same input types. Fortunately, by this point in the compiler we have verified that the program respects abstraction, and hence it is safe to drop the generativity constraint.

Were the source language strengthened to include a conditional at the module level, it would no longer be possible to do this phase-splitting; the types returned by a functor would depend on input *values*.

For efficiency we have added several minor optimizations to this transformation. Since MIL supports polymorphic recursion but HIL doesn't, the two levels have different forms for collections of mutually-recursive polymorphic functions: a collection of polymorphic functions compared to polymorphic function returning a collection of functions. Fortunately, such cases are easy to recognize so we handle these specially.

We also eliminate "trivial" constructors, such as empty records of constructors or functions returning such empty records. Sometimes they cannot be eliminated entirely, but can be simplified.

Early projection? [Perry]

One optimization we would like to add is to preserve the sharing of types generated by the elaborator's unification algorithm. That is, the phase-splitting translates a HIL type metavariable by translating the HIL type with which it is unified. Several unified metavariables thus generate several translations of the type. In contrast, we could do some on-the-fly CSE guided by this sharing information we get for free.

"avoiding inlining (hence duplication) of constructors"

4 Reflections

To some extent, the two steps from SML through HIL to MIL might be unnecessary. Changing the HIL elaboration to go directly to MIL (including the phase splitting) would be an option except that we could not enforce SML type generativity. Therefore, we would have to do a pre-typecheck of the SML code (perhaps simply following the stamp-based approach of The Definition) to ensure that abstraction is being respected and then ignore generativity in the MIL as we do now. It might be difficult to do this efficiently in a non-ad-hoc manner, however, since SML cannot be written in a fully-type-explicit manner (explicit in ML) and we wouldn't want to do type inference twice.

A different alternative would be to discard the MIL entirely, and extend the HIL with enough constructs (closures, etc.) to allow the remaining phases to transform HIL code. We considered this, but decided against it because (1) the HIL was insufficiently close to the old TIL1 IL to reuse code, and (2) working with the HIL would have required more duplication of code for the expression and module levels (e.g., closure converting functions and functors). It turns out we didn't reuse much code anyway.

A solution to (2) would be to work with a first-class module system. However, Mark Lillibridge has shown the typechecking problem to be undecidable in general, which makes it less suitable for our purposes.

The translation of SML polymorphism to HIL functors is elegant. However, given that we immediately translate these back to MIL polymorphism again, we might have just added explicit polymorphism as a special "derived form" (like let) to the HIL datatype and translated this specially.

Even once the decision to use phase-splitting was made, there was still a fair amount of flexability left in deciding exactly what the MIL would be like. The main question was whether we wanted to use singleton kinds and dependent types or whether these should be eliminated entirely. (The FLINT compiler took the latter route.)