Functional Compilation from the Standard ML Core Language to Lambda Calculus

Nick Rothwell

September 10, 1992

1 Introduction

This paper describes a compiler from the SML core language to a simple intermediate code based on the lambda calculus. It forms an optional part of the SML Kit, and the interface presented to the rest of the Kit is the same as that presented by the "pure" interpreter. The lambda compiler is not intended to be an example of how to generate highly efficient code for SML; instead, it serves to illustrate in a clear, modular fashion some of the techniques which are needed to compile core SML into a simple intermediate language.

The Kit is, inasmuch as it is possible, a direct implementation in SML of the SML Definition [7], such that the internal structure of the compiler reflects the objects and inference rules of the formal semantics. It contains a typechecker which is a fairly direct coding of the static semantics of SML and a "pure" interpreter which is a corresponding coding of the dynamic semantics. The purpose of this paper is to describe a modular extension to the Kit: a replacement interpreting phase which actually compiles SML to a simple intermediate code and executes this. This extension, which we shall refer to as "the compiler", has the same external interface as the interpreter of the dynamic semantics (henceforth "the interpreter"), all differences being internal. (In fact, there is one difference: the compiler will report inexhaustive and redundant matches, since it has the machinery to do so; the interpreter does not.)

Our motivation in designing and implementing the compiler was not efficiency; instead, it was to produce a system to illustrate the techniques needed to compile SML into a simple intermediate language. If these techniques are implemented in a clear and modular fashion, gaining efficiency is largely a matter of refinement. However, a compiler which has been constructed with efficiency as a driving force in its design is unlikely to demonstrate its internal structures and algorithms in a clear and modular fashion.

A motivation for the SML Kit was to have a system which was complete in its implementation of the SML language and yet open enough in terms of its internal structure and interfaces to allow aspects of its operation to be investigated, altered and experimented with. The desire was for a tool which would allow changes or additions to the formal semantics to be implemented with the minimum of effort, so that they could actually be seen in operation and played with (and this aim has been achieved in a number of projects, including Extended ML [8]). While there are a number of SML compilers in existence or under development at present, none can be considered simple enough to allow this kind of experimentation. In addition, large, commercial-quality compilers tend to contain extra machinery required for efficient code generation, and this tends to obscure modularity. Also, there is no motivation on the part of compiler writers to produce a piece of software which reflects the Definition in its internal structure—all that is important for such an implementation is adherence to the language semantics "on the outside" (and even here, compilers can vary).

Is this then perhaps a good reason *not* to implement any kind of compiler in the SML Kit? Possibly; but it was considered a useful exercise for the reasons outlined below, and in addition, the highly modular nature of the Kit allowed the compiler to be integrated in isolation with no compromises to the structure or operation of the rest of the system, apart from some minimal information required from the typechecker. The Kit can be built as an interpreter or as a compiler from a large body of shared code—30000 lines of the system (including the generated parser and all the static semantics machinery) is common between the interpreter version (an additional 1500 lines) and the compiler version (an additional 5500 lines).

The Kit provides a good testbed for experimentation at the level of the formal semantics. New ideas for typechecking, for example, would be formulated in the static semantics and could them be implemented in a fairly direct manner in the Kit. Similarly, alterations in the semantics of program execution could be described formally and then implemented. However, there are experiments and projects which would benefit from a simple, modular, implementation of the complete SML language and yet might not fall into an area which can be addressed by the formal semantics. This is probably unlikely in the case of the static semantics—type systems are, by their very nature, formal, and can only be dealt with in a satisfactory manner using semantic techniques. In the case of the dynamic semantics, the situation is rather different. Pragmatics still plays a major part in compiler research, and program execution can be considered independently of the dynamic semantics. In particular, the provision of a functional intermediate language (in this case, one related to the lambda calculus) means that experiments which are often performed on the source language (such as code instrumentation) might be done at the level of the cleaner, simpler, intermediate language instead.

For such pragmatic experiments, it was felt that the dynamic semantics of the Definition fell at too high a level to be of direct use. While the dynamic semantics of SML is fairly simple, it is simple at a level which might fail to convince a compiler writer. The dynamic semantic objects are at quite a high-level, are built from high-level abstractions like finite maps, and sometimes have fairly complex behaviour. By contrast, research in compilation techniques is often concerned with representations of data objects and the ways in which such objects can be efficiently manipulated. The notion of representation is different to a compiler writer with his concept of machine words and memory, to that of a theorist who formulates rules in terms of environments and maps, and concepts of efficiency (whether at the level of algorithm complexity or numbers of operations) do not arise at all at the semantic level.

A greater impetus is given by the fact that a large portion of the SML Kit can be shared between the interpreter and the compiler, and the compiler can benefit from the parser and typechecker for the entire SML language, plus the overall modular skeleton of the interpreted system.

It should be stated that some of the main ideas in the compiler (such as the adoption of an intermediate lambda language with unique variables, and the use of higher-order functions to compile declarations) were first seen in Standard ML of New Jersey [3], and the reader is referred to [2] for a more extensive coverage of SML code generation, together with some of the techniques employed in the Kit. Suffice to say that the Kit Compiler is a lot simpler in overall structure and function, but operates in a similar way to the back-end and runtime system of SML/NJ.

2 Structural Overview

The SML Kit's evaluation stage revolves around the concept of *dynamic basis* presented in the Definition. The evaluator takes a dynamic basis representing the current top-level environment, and a top-level declaration, and attempts to evaluate it. The evaluation is either successful, yielding the dynamic basis corresponding to the declaration, or else it fails due to an exception propagating to the top level. The `EVALTOPDEC` signature is given in Figure 1[1].

The interpreter and the compiler each form a subsystem which conforms to this interface. The implementation of the dynamic basis is different in each case, although both implementations provide the operations of the dynamic semantics (`E_in_B`, `E_of_B`, `B_plus_B`). The rest of the SML kit accesses the evaluation modules through the interfaces provided by the signatures `EVALTOPDEC` and `DYNAMIC_BASIS` (Figure 2), and has no knowledge of any of their internals.

The compiler contains a number of modules associated with analysing, compiling and executing declarations. These are brought together in a single linkage module which compiles and evaluates top-level declarations. This top-level module of the compiler translates (core-level) declarations to lambda code which is fed through a simple optimiser and then executed. If successful, the execution results in a new set of dynamic bindings to add to the top-level environment. As far as

[1]The code examples have been beautified slightly: "`*`" (for type tuples) is written "$\times$", `->` (for the function type) is "$\rightarrow$", and type variables are written "α", "β" and so on (or "$\alpha_=$", "$\beta_=$" if they have the equality attribute).

```
signature EVALTOPDEC =
  sig
    type DynamicBasis
    type topdec

    datatype result = SUCCESS of DynamicBasis
                    | FAILURE of string

    val eval: DynamicBasis × topdec → result
  end
```

Figure 1: The signature EVALTOPDEC

```
signature DYNAMIC_BASIS =
  sig
    type Basis

    val emptyBasis: Basis
    val initialBasis: Basis
    val B_plus_B: Basis × Basis → Basis
  end
```

Figure 2: The signature DYNAMIC_BASIS

the rest of the Kit is concerned, the interface is just an evaluator for declarations; the compiler is totally hidden. (At present there is some temporary clutter in this module since we do not yet have support for the modules language, and yet require support for top-level `local` declarations which can only be parsed as module-level constructs.)

The compilation phase is fairly straightforward, and we describe it in Section 6. The part of the compiler responsible for compiling pattern matches, however, is sophisticated and quite complex; an overview is given in Section 7. The interface between the two is a data structure called a *decision tree*. The compiler works at the level of dynamic objects which bear some similarity to the objects of the Definition's dynamic semantics, but are at a lower level and much simpler. The intermediate lambda language is fairly simple, as is its interpretation. The runtime objects are discussed in Section 4, the lambda language in Section 5, and the lambda interpreter in Section 10.

3 Variables and Environments

Where the dynamic semantics retains the notion of a *variable* in the source language, and builds environments as mappings from such variables, we introduce a new type of identifier, the *lvar*, which is known only to the compiler and lambda interpreter. Lvars are represented numerically and are generated uniquely during compilation. Therefore, even though the same SML variable can occur in several places in the source text, with different meanings, an lvar is bound exactly once. This property simplifies much of the compiler since a lot of environment analysis is no longer necessary; many operations on the intermediate code can be performed with respect to an lvar without worrying about the same lvar being rebound in a nested scope[2].

Corresponding to the various environments of the dynamic semantics, the compiler has two kinds, whose interfaces are shown in Figures 3 and 4. The *dynamic environment*, or DEnv, maps lvars to runtime objects. The *compiler environment*, or CEnv, maps SML variables to lvars. CEnvs are used during compilation to deal with SML identifiers occurring in the source text being compiled; DEnvs are used during execution to deal with the evaluation of lvars.

When a declaration is compiled, each variable bound by the declaration is associated with a new, unique, lvar. This mapping of SML variables to lvars constitutes the CEnv for the declaration. The code is then executed to yield a set of objects, one for each variable. These objects are bound to the lvars to form the DEnv for this declaration. Finally, the CEnv and DEnv are incorporated into the top-level environment. In subsequent declarations, the compiler will translate occurrences of the newly-bound SML variables into their corresponding lvars via the CEnv; during execution, the lvars will be mapped to their runtime values by

[2] and some cannot; beta-reduction of functions, for example, would require variable renaming.

the DEnv.

Recall from Section 2 that the compiler must match the signature `EVALTOPDEC` in order to link with the rest of the system. This it does by implementing the type `DynamicBasis` as a pair (`CEnv` $\times$ `DEnv`). This allows SML identifiers to be mapped to runtime objects by looking them up in the CEnv and then looking up the resulting lvars in the DEnv.

4 Runtime Data Objects and Representations

The Definition has a large selection of dynamic, or "runtime", data objects, including constructors, records, exception packets, and various kinds of environment. These semantic objects are defined abstractly at a fairly high-level, and some (records and environments) are defined using other high-level constructs (finite maps). The compiler has a small number of simple data objects and a single notion of environment, in which the semantic objects are implemented. This implementation requires the compiler to make choices of representation; some of the choices require information from the typechecker.

The compiler takes SML declarations and generates lambda code which manipulates these objects; the compiler therefore needs to be aware of the mapping between semantic objects and their representations. Execution of lambda code takes place in (a representation of) the top-level environment, and produces new top-level bindings in terms of compiler objects; the module responsible for the top-level environment must manipulate the objects and dynamic environments consistently, without invalidating the abstract view of program execution which the rest of the SML Kit relies upon.

The compiler's `OBJECTS` signature is shown in Figure 5. This is essentially an interface to a (hidden) datatype, but there are advantages to making the interface functional—in particular, closures are built in a clean way. The types `lvar` and `DEnv`, as described above, are needed for building closures, as is the type `LambdaExp` of lambda expressions (Section 5).

The simplest runtime objects correspond to the *special values* of the Definition: integers, reals and strings. In addition, there is a *void* object used in the representation of `unit` and of nullary exception and data constructors (Section 6). `vector` and `select` allow vectors (non-assignable arrays) of objects to be built and decomposed. `Ref` and `deRef` build and decompose references. `equal` provides equality on objects; it respects the identity of references, and is undefined on closures.

The representation of closures corresponds very closely to that in the Definition, but has an extra component. `bodyEnv` and `recEnv` correspond to the elements E and VE in the closures of the semantics. In place of the *match*, we have an lvar and a lambda expression. Where the Definition applies the match to an argument, we instead evaluate the lambda code in an environment which binds the lvar to the argument. The dynamic environment signature (Figure 3) provides

```
signature DYNAMIC_ENV =
  sig
    type DEnv
    type lvar
    type object

    val emptyDEnv: DEnv
    val declare: (lvar × object × DEnv) → DEnv
    val plus: DEnv × DEnv → DEnv
    val REC: DEnv → DEnv       (* Semantics V4, p48 *)
    val lookup: DEnv → lvar → object
  end
```

Figure 3: The signature `DYNAMIC_ENV`

the *Rec* operation described in the Definition. As an aside, the types of objects and dynamic environments are mutually recursive: objects include closures which contain dynamic environments, and dynamic environments map lvars to objects.

There are a number of invariants concerning the operations on objects: the unpacking operations must be applied to objects of the correct type, `select` must use an index within an appropriate range for the length of the vector, and so on.

5 The Lambda Language

We now describe the lambda language generated by the compiler. The signature specifying the lambda language datatype is shown in Figure 6. The language is at a level usually associated with compiler intermediate codes, but the Kit interprets it directly rather than compiling it down into lower-level code. The lambda language is essentially lambda calculus with one or two simple extensions. It contains constants of the base types associated with SML's special values, as well as *switches* (simple case expressions) over them. We consider the lambda language to be *untyped*, in that there is no provision for defining or expressing types in the language. The integers, strings and reals of the lambda language can be considered to have SML types `int`, `string` and `real` respectively, but there is no notion of type abstraction, datatype and so on.

The variables of the language are the *lvars* introduced in Section 3. Lvars are bound in function (`FN`) expressions and recursive declarations (`FIX`) and referenced by `VAR` leaf nodes. The Definition's *special constants* map to lambda-expressions of the kind `INTEGER`, `STRING` or `REAL`, which have corresponding kinds of *object* (Section 4). Similarly, there is a `VOID` lambda constructor which yields a `void`

```
signature COMPILER_ENV =
  sig
    type CEnv
    type var                          (* Unqualified variables *)
    type longvar                      (* Qualified vars (for lookup) *)
    type excon                        (* Unqualified exception constructors *)
    type longexcon                    (* Qualified excons (for lookup) *)
    type lvar                         (* Unique lambda identifiers *)

    val emptyCEnv: CEnv
    val initialCEnv: CEnv

    val declareVar: (var × lvar × CEnv) → CEnv
    val declareExcon: (excon × lvar × CEnv) → CEnv

    val plus: CEnv × CEnv → CEnv

    datatype result = LVAR of lvar
                    | PRIM    (* Looking up a value identifier will *)
                              (* either get you a real lvar, *)
                              (* or PRIM (which means you have to *)
                              (* extract the integer argument and *)
                              (* use it in a PRIM_APP lambda-exp) *)

    val lookupLongvar: CEnv → longvar → result
                              (* Only prim should give you back *)
                              (* PRIM (and even then it can be *)
                              (* overwritten) *)

    val lookupLongexcon: CEnv → longexcon → lvar
                              (* Looking up an excon must give *)
                              (* you an lvar *)

    val lvarsOfCEnv: CEnv → lvar list
  end
```

Figure 4: The signature COMPILER_ENV

```
signature OBJECTS =
  sig
    type lvar
    type LambdaExp               (* For building closures. *)
    type DEnv                    (* lvar→object environment. *)
    type object

    val void: object

    val integer: int → object
    val deInteger: object → int

    val real: real → object
    val deReal: object → real

    val string: string → object
    val deString: object → string

    val closure:
      {arg: lvar, body: LambdaExp, bodyEnv: DEnv} → object
                                 (* The recEnv part is created empty. *)

    val deClosure:
      object → {arg: lvar, body: LambdaExp, bodyEnv: DEnv, recEnv: DEnv}

    val vector: object list → object
    val select: int × object → object

    val Ref: object → object
    val deRef: object → object

    val equal: object × object → bool
  end
```

Figure 5: The signature OBJECTS

object. (The name `VOID` is used rather than `UNIT` since there are many circumstances in which a `VOID` lambda term might be used, and the word `UNIT` might misleadingly suggest some relation to SML's `unit` data object.)

Function (`fn`) expressions are compiled into `FN` expressions in the lambda language. These only take a single lvar as argument, so a fair amount of work is involved in transforming SML function expressions containing matches into lambda functions; most of the work is done by the *match compiler* (Section 7).

Mutually recursive function definitions compile into `FIX` expressions which bind a set of lvars in parallel. (Recall that the compiler associates a new lvar with each variable in an SML declaration.) The right hand sides of the bindings are expected to be function (`FN`) expressions. The `APP` constructor represents application of one lambda-expression to another in the obvious way. `PRIM_APP` is used for pervasive functions (the *basic values* of the Definition) in a manner described in Section 8.

Corresponding to the many data objects of the Definition (records, data constructors, exception constructors, references), the lambda language has two distinct data manipulating constructors. `VECTOR` takes a list of lambda expressions and returns a fixed-length list of objects from which elements can be extracted using the `SELECT` constructor. (In an implementation, one would expect this to be an array with constant access time, but this is not a requirement.) `SELECT` takes an index i between 0 and $n-1$ where n is the length of the vector and returns the i'th element (and is undefined for values of $i < 0$ or $\geq n$). `REF` is a constructor for creating a reference to an object using a new address, as in the Definition.

The lambda language has conditional expressions over the three basic types (integers, strings, reals); comparison of reals is assumed to make sense because it is required by the Definition. There are three "branch" operations (`SWITCH_I`, `SWITCH_S` and `SWITCH_R`), one for each corresponding object type, although the switches have identical structure (and in fact the switch datatype is polymorphic). A switch can be considered a degenerate `case` expression: it has an *argument*, which is a lambda expression which must evaluate to a basic value (constant) of the correct type, and contains a finite map from constants to lambda expressions. If the argument is in the domain of the map the corresponding lambda expression from the range is evaluated as the value of the switch. If the map does not contain the argument, the switch evaluates and returns its optional *wildcard*. The value of an unmatched switch with no wildcard is undefined. The wildcard is optional for integer switches, since the compiler often knows that the switch is exhaustive for the possible range of values of the argument. The wildcard is mandatory, however, for string and real switches because they are generated only as a result of matches in the source SML program and can never be exhaustive. In these case the compiler always generates a wildcard which is appropriate.

Finally, the lambda language provides for exception raising and handling. `RAISE(`e`)` raises the exception packet denoted by e (and whose format is described in Section 6). `HANDLE(`e_1`, `e_2`)` evaluates e_1; if an exception is propagated, e_2 is evaluated to yield a `FN` expression, and the result of the `HANDLE` expression is e_2

applied to the exception packet. Any exception resulting from this application is propagated outward.

It can be seen that the interpreter for the lambda language carries an implicit dynamic exception context during evaluation; the interpreter is described in detail in Section 10.

The `LAMBDA_EXP` signature contains some convenient shorthand functions for building the common forms of lambda construct. Of particular note is the form

`Let(`v`, `e_1`, `e_2`)`

which evaluates to

`APP(FN(`v`, `e_2`), `e_1`)`

This is a translation which can be performed safely in the untyped lambda language; the same is not true in SML itself, due to the way in which `let`-bound variables have their types generalised.

6 The Declaration Compiler

6.1 Conventions and Representations

The declaration compiler has two purposes:

- to translate declarations in the SML core language into expressions in the lambda language;
- to generate a compiler environment (CEnv) for the variables bound in each declaration.

However, since (SML) declarations yield environments and (lambda) expressions yield values, there is no obvious translation from declarations to lambda expressions. The following, rather elegant, convention is due to Andrew Appel [3].

The compiler is a function `compileDec` with type

`CEnv` $\rightarrow$ `dec` $\rightarrow$ `(CEnv` $\times$ `(LambdaExp` $\rightarrow$ `LambdaExp))`

Every SML declaration is treated as if it had a *scope*. For top-level declarations, the scope does not actually exist, although each such declaration *dec* is treated as if it occured as part of an expression of the form

`let` *dec* `in` *scope* `end`

```
signature LAMBDA_EXP =
  sig
    type lvar
    type (α₌, β) map

    datatype α option = NONE | SOME of α

    datatype LambdaExp =
        VAR of lvar             (* Lambda variables *)
      | INTEGER of int          (* Constants... *)
      | STRING of string
      | REAL of real
      | FN of lvar × LambdaExp (* Function-terms *)
      | FIX of lvar list × LambdaExp list × LambdaExp
                                (* Mutually recursive fns *)
      | APP of LambdaExp × LambdaExp (* Function application *)
      | PRIM_APP of int × LambdaExp (* Primitive function application *)
      | VECTOR of LambdaExp list (* Records/tuples *)
      | SELECT of int × LambdaExp (* Con/record indexing *)
      | SWITCH_I of int Switch (* Switch on integers *)
      | SWITCH_S of string Switch (* Switch on strings *)
      | SWITCH_R of real Switch (* Switch on reals *)
      | RAISE of LambdaExp      (* Raise exception *)
      | HANDLE of LambdaExp × LambdaExp (* Exception handling *)
      | REF of LambdaExp        (* ref(expr) *)
      | VOID                    (* nil, () etc *)

    and α Switch = SWITCH of {arg: LambdaExp,
                              selections: (α, LambdaExp) map,
                              wildcard: LambdaExp option
                              (* Wildcard mandatory for *)
                              (* REAL or STRING switches *)
                             }

      (* Some convenient lambda-building utilities: *)
    val pair: LambdaExp × LambdaExp → LambdaExp
    val first: LambdaExp → LambdaExp
    val second: LambdaExp → LambdaExp
    val Let: ((lvar × LambdaExp) × LambdaExp) → LambdaExp
  end
```

Figure 6: The signature LAMBDA_EXP

The result of compiling a declaration is a CEnv for the declaration, plus a function which, when applied to a compiled *scope* for the declaration, returns a lambda expression corresponding to the declaration compiled locally for that scope.

There are two reasons for this particular convention. The first is abstraction—an alternative approach would have been to have the compiler translate declarations directly into lambda expressions with some convention for the scope of the declaration (such as a vector of lvars corresponding to the identifiers ordered lexically), but it is more elegant to choose the convention elsewhere, separately from the compiler itself. The second reason for the higher-order nature of `compileDec` is that it makes sequential and local declarations easy to deal with: the functions resulting from compilation of the sub-phrases can just be composed together (as shown in subsection 6).

This convention carries down into the functions private to the compiler. Functions which compile expression-like phrases (expressions, atomic expressions and so on) return a `LambdaExp`. Functions which compile declaration-like phrases (declarations, value bindings, and so on) return a function of type `LambdaExp` $\rightarrow$ `LambdaExp`.

The compiler has a simple set of runtime data structures, onto which it must map the runtime data structures (the dynamic semantic objects) of the Definition. The representations are as follows:

- Records become vectors whose elements represent the record fields in lexical order.

- Constructed datatypes become pairs (vectors of two elements), the first of which (element 0) is the argument of the constructor (or `void` for nullary constructors) and the second of which is an integer tag (≥ 0) denoting the lexical index of the constructor in the datatype.

- Exception constructors in the DEnv are held as string references; the string contains the exception's name (the SML identifier) for printing, and the reference models the exception's generative behaviour.

- Exception values and packets have the same representation—a pair whose first element is the value carried by the exception and second element is a *string ref* which embodies the dynamic name of the exception (as the address of the reference) as well as a string for top-level printing.

- Value-carrying value and exception constructors compile into functions which build a data structure of the appropriate form from their argument.

There are two problems with these representations. Firstly, the representation for exceptions means that the expression

```
let
   exception A
   exception B = A
in
   raise B
end
```

results in the uncaught exception `A` being reported at top level, rather than `B`. Secondly, the obvious definition of lists (using a `datatype` declaration) results in two vectors being allocated for each list element. Both problems can be circumvented with a little effort, but this is beyond the scope of this report. The issues involved in the efficient representation of datatypes is discussed further in [1].

We now consider the declaration compiler in more detail. The input to the compiler is a data structure bearing a close resemblance to the abstract syntax of the SML core presented in the Definition (pages 8–9), and is therefore not described here. Note that the abstract syntax of the Definition has been *resolved* such that identifiers are distinguished as variables, data constructors or exception constructors. In addition to this resolution, the compiler needs some type information in order to compile matches and choose data representations in some circumstances; this is discussed in Section 7.

6.2 Compiling Atomic Expressions

Atomic expressions are compiled as follows. Special constants (integers, reals and strings) have direct counterparts in the lambda language, and are trivially compiled into these without reference to the CEnv. Identifiers are more complicated, since by this stage they have been resolved by the elaborator into one of three classes: variable, data constructor or exception constructor. Variables are looked up in the CEnv yielding lvars (although the pervasives are treated slightly differently, as described in Section 8). Nullary data constructors compile into pairs whose first element is `VOID` and second is an integer tag denoting the constructor. When a unary (value-carrying) constructor is applied to a value, a pair is built containing the value and an integer tag for the constructor. The value itself might be a pair (such as arguments to the cons operator, '`::`'); there are more efficient representations for such datatypes, but none has been implemented yet. Each unary constructor is compiled into a function expression which builds a pair when applied to its argument (Figure 7). (The code generated for something like `2 :: nil` is, therefore, initially rather inefficient, but the optimiser described in Section 9 transforms it into a more sensible form, as shown in the first example of Figure 7.)

The compilation of constructors requires information from the elaborator for two reasons. Firstly, the value of the integer tag depends on where the constructor identifier falls alphabetically compared to the other constructors of the datatype. Secondly, value-carrying constructors are detected by their type; if the constructor

Source: `let datatype` α `t = T of` α `in T 1 end`

Code: `VECTOR[VECTOR[1, 0]]`

Result: `val it = T 1 : int t`

Source: `let datatype` α `t = T of` α `in T end`

Code: `VECTOR[FN v192. VECTOR[v192, 0]]`

Result: `val it = fn :` $\alpha \rightarrow \alpha$ `t`

Figure 7: Code for a unary data constructor

has a type which is an instance of $\alpha \rightarrow \beta$ then it is value-carrying. The compilation of constructors does not require the CEnv, just as the dynamic semantics doesn't require an environment for evaluation of constructors (rule 105).

Exception constructors are treated similarly to data constructors, but with the important difference that exceptions are generated dynamically during execution, which complicates matters. Each time an exception declaration is encountered a new exception is generated, distinct from any exception generated from a previous evaluation of the declaration. Each exception is a first-class value, and can be regarded as a function (if the exception is value-carrying, with type $\alpha \rightarrow$ `exn` for some type α) or as an exception packet (if the exception is nullary, with type `exn`). Each exception must, however, be available for pattern matching (regardless of whether it is nullary or unary), and exceptions only match if they are identical at runtime.

The implementation scheme is as follows. A declaration of a new exception will bind an lvar to a new *string reference*, regardless of whether the exception carries a value. The string holds the exception's name for printing. This representation is most convenient for pattern matching, since exception matching then becomes a matter of determining identity of references. When an exception identifier is encountered in an expression, it must be converted into a form consistent with a value of type `exn` or $\alpha \rightarrow$ `exn`. For nullary exceptions, the compiled lambda code pairs the string reference with `VOID`. For unary exceptions, the code builds a closure which takes a value x and returns the pair (x, n) where n is the string reference. Note once again that the aim is to have a unified scheme for application, regardless of the kind of object being applied; this is in contrast to the dynamic semantics of the Definition where there are a number of rules for application.

Records (and tuples) compile to vectors, using a canonical representation where the fields are stored according to the lexical ordering of the field labels. No type in-

Source: `{C=f(), B=f(), A=f()};`

Code: LET(v197 = APP($lvar_f$, VOID):
v198 = APP($lvar_f$, VOID):
v199 = APP($lvar_f$, VOID):
VECTOR[v199, v198, v197]
)

Figure 8: Code generation for records

formation is needed since record expressions are exhaustive in the labels. However, some subtlety is required, since record fields must be stored in canonical order but evaluated in the order in which they occur in the program, in order to maintain determinacy where side effects might be present. This is done by means of a set of local declarations which evaluate the fields in the required order and then build the record in canonical order. Figure 8 gives an example where record fields have to be reordered.

Atomic expressions of the form `let` *dec* `in` *exp* `end` cause the declaration to be compiled, yielding a CEnv ce_1 for the declaration and a function f to be applied to the scope, which is the result of compiling *exp* in the original CEnv augmented with ce_1.

6.3 Compiling Expressions

Expressions are compiled in the context of a CEnv, yielding a lambda expression, as follows. Function application is fairly simple, because the type of the functional part is not discriminated at application time; instead, the compiler ensures that it will always be a closure. However, it is here that we deal with pervasive functions, using a mechanism described in Section 8.

Function (`fn`) expressions are dealt with directly by the match compiler (Section 7). Exception handling is done at runtime by a meta-circular interpreter (one which implements exceptions using exceptions): `raise` expressions are compiled into `RAISE` lambda terms and `handle` expressions are compiled into `HANDLE` lambda terms, which are implemented in the lambda interpreter with exceptions. In the case of `HANDLE`, the lambda term contains a function `FN` sub-term built by the match compiler.

6.4 Compiling Declarations

The function `compileDec` forms the top-level of the compiler; since we don't currently support the modules system, phrases passed to the compiler correspond to core declarations. `compileDec` has type

```
CEnv → dec → (CEnv × (LambdaExp → LambdaExp))
```

as explained in Section 6.1. The subsidiary functions for dealing with valbinds, exbinds and so on have similar types.

We shall dismiss the simple cases first. Type declarations play almost no part in SML's dynamic semantics, and so the compiler can ignore their effect[3]. Therefore, `compileDec` compiles `type` and `datatype` declarations into an empty CEnv together with the identity function. Empty declarations and infix declarations[4] are treated similarly. For phrases of the form

`abstype` *datbind* `with` *dec* `end`

the *datbind* is discarded. `local` and sequential declarations are similar to each other; they differ only in the CEnv produced as a result. In both cases, the first declaration is compiled yielding a CEnv ce_1 and result function f_1. The second declaration is compiled in an environment augmented with ce_1, yielding ce_2 and f_2. The resulting CEnv is $ce_1 + ce_2$ (for sequential declarations) or ce_2 (for `local` declarations); in both cases, the result function (of type `LambdaExp` → `LambdaExp`) is $f_1 o f_2$.

Some complexity arises when dealing with value bindings and exception bindings. We deal with exception bindings first.

Exceptions require rather sophisticated compiler support, for a variety of reasons. Exceptions are generative: if an exception declaration serves to introduce a new exception (as opposed to a new identifier binding for an existing exception), then each execution of that declaration creates a new, distinct, exception. (By contrast, datatype declarations compile to no code.) Exceptions may be nullary or they may carry values, in the same way as constructors. And, exceptions may be pattern-matched, in which case they must be matched sequentially according to their dynamic identity, rather than identifier or textual occurrence. By contrast, data constructors are not generative, and may not be renamed without losing their constructor status; hence, constructors in patterns may be distinguished merely by their identifiers.

The representation of exception values and packets is described in Section 6.1. However, the dynamic environment (DEnv) holds exceptions simply as string references. This makes pattern matching over exceptions straightforward; if they were held in the same representation as values of type `exn`, then extra dereferencing would result, and value-carrying exceptions would have to decomposed from their closure representations.

[3]We say *almost* no part because there is an obscure situation, highlighted in the SML Commentary [6], which requires `datatype` declarations to have an effect at evaluation time. However, this situation occurs in connection with the modules system which the Kit currently does not implement.

[4]Infix declarations find their way into the abstract syntax, although they are omitted from the semantic rules of the Definition.

When an exception is encountered in the context of an expression, therefore, it is converted into a value of type `exn` (which is a vector of length two) or a value of type $\tau \rightarrow$ `exn` for some τ (which is a closure).

We can now consider exception declarations in detail. A declaration of the form

```
exception A
```

causes the identifier `A` to be bound to a new lvar in the CEnv when the declaration is compiled; the lvar is bound to `ref("A")` in the DEnv each time the declaration is executed. A value-carrying exception is treated identically. A declaration of the form

```
exception B = A
```

introduces a binding for `B` to `A`'s lvar in the CEnv at compile-time, and leaves the DEnv unchanged at runtime (and in fact, no code is generated). Compound declarations using `and` are straightforward.

An actual exception value is built when an exception identifier is encountered in the context of an expression. It is here that value-carrying exceptions are distinguished from nullary exceptions. Consider the declaration

```
exception A1
      and A2 of int
```

The declaration produces the bindings

$$\texttt{A1} \rightarrow lv_1$$
$$\texttt{A2} \rightarrow lv_2$$

in the CEnv and

$$lv_1 \rightarrow \texttt{ref "A1"}$$
$$lv_2 \rightarrow \texttt{ref "A2"}$$

in the DEnv. If identifier `A1` is encountered in an expression, the code generated retrieves the string reference and builds a vector. If the identifier `A2` occurs, the code returns a closure for a function which pairs its argument with the string reference.

We now describe the other kinds of declaration. Recursive function declarations are straightforward; SML disallows any other kind of object to be declared recursively, and even places syntactic restrictions on the functions (recursive declarations may not, for instance, bind variables to expressions which just happen to have functional type).

Within a `rec` declaration (of which `fun` is just a syntactic derived form), there may only be function bindings and further, nested, occurrences of `rec` (which serve no purpose). The entire topmost `rec` declaration is flattened out to yield a set of mutually recursive function bindings, which are then translated into a `FIX` declaration in the lambda language.

The recursive declaration is compiled in two passes. The first pass assembles the function identifiers to be bound and creates new lvars for them. The second pass compiles the declaration's right hand sides (which must all be lambda expressions) in the outer CEnv plus the CEnv for the function identifiers. The resulting lvars and compiled functions are then assembled into a `FIX`.

Finally we consider plain value bindings. These are actually the most complicated in terms of the code generated, because value bindings may contain patterns and must therefore use the match compiler.

An expression of the form

`let` *pat* `=` *exp* `in` *scope* `end`

is regarded as the equivalent

`(fn` *pat* `=>` *scope*`)` *exp*

where the lambda expression contains a *match*. Value bindings are therefore dealt with by the machinery responsible for function expressions and applications.

7 The Match Compiler

The whole area of efficient pattern matching is quite a complex one. The match compiler implemented for the Kit is fairly sophisticated, and beyond the scope of this paper, but it is at least necessary to describe what it does, and how it interfaces with the rest of the compiler. We begin with an overview of the semantics of pattern matching in SML.

The Definition describes a syntactic construct called a *match*, which is a list of (pattern, expression) pairs. Matches occur in many of the contexts where identifiers are bound (specifically, case expressions, function arguments and exception handlers), and it is convenient to interpret non-recursive value bindings in a similar manner.

In fact, case expressions can be ignored since they are nothing more than convenient sugaring for function application. Similarly, we can ignore "formal" function declarations using `fun`, including curried functions, since they are just shorthand for (recursive) value bindings to lambda expressions.

Consider the expression

```
fn (0, _) => e1
 | (_, 0) => e2
 | (_, _) => e3
```

This is an anonymous function which takes a pair of integers as argument. The SML semantics requires that it attempt to match its actual parameter against the patterns in sequence, so that calling it with argument `(0, 0)` would return the result e_1, even though all three patterns match the actual argument. This particular function employs a match which is *exhaustive*, in that every possible actual parameter is matched by some pattern (in fact, `(_, _)` will match any argument which typechecks). The match is also *irredundant*, since each pattern in the match will be chosen for some argument. Inexhaustive and redundant matches are legal, but the Definition requires that each must result in a warning message from the compiler.

We choose to treat all non-recursive `val` bindings as instances of pattern matching. Each is an instance of the phrase class

```
val pat = exp
```

A simple declaration such as

```
val x = 3
```

performs pattern matching on the pattern consisting of the variable `x`. The compiler, parameterised on a scope for the declaration, transforms it into the form

```
(fn x => scope) (3)
```

at which point it can use the match compiler for the `fn` expression.

The Definition gives a formal, operational, definition of pattern matching. A value can be matched against a pattern in the context of an environment (which is needed only to give meaning to exception constructors in the pattern: see the Definition, rules 146 and 147). The match either succeeds, yielding a value environment for the variables in the pattern, or fails. When evaluating a match, failure causes execution to pass to the next rule so that the next pattern can be matched against the value. When evaluating a value binding, failure to match causes an exception (`Bind`) to be raised.

Although perfectly acceptable for a semantic definition, this operational, backtracking, view is inefficient from a practical point of view, for a number of reasons which are best illustrated by examples (in which we assume the declaration of a datatype providing constructors `RED`, `ORANGE`, `YELLOW` and so on in the obvious way).

Consider the following:

```
fn (1, 2, (3, [RED, GREEN, RED])) => e1
 | (1, 2, (3, [RED, GREEN, GREEN])) => e2
 | (1, 2, (3, [RED, GREEN, BLUE])) => e3
```

Executing this match in the sequential, backtracking manner suggested by the semantics would be inefficient. The patterns are largely identical, differing only in the third element of a list expression which is nested quite deeply in the patterns, and sequential matching might well cause the argument value to be decomposed several times, with largely identical tests being performed each time.

Even in examples with minimal backtracking, the sequential nature of the Definition's matching process precludes any efficient case analysis over data constructors. In the example

```
fn RED => e1
 | ORANGE => e2
 | YELLOW => e3
 | GREEN => e4
 | BLUE => e5
 | INDIGO => e6
 | VIOLET => e7
```

there is little backtracking involved, yet sequential matching is still inefficient compared to a single linear switch.

The Definition decomposes record patterns from left to right, but this is essentially a notational convenience. Since SML is a strict language, there is no obligation to do left-to-right pattern decomposition, and it may be advantageous to choose a different order. In terms of the number of comparisons performed for an arbitrary data value, there is no overall optimal order in which to decompose the tuples of this match—different orders will be optimal for different data values. A more practical aim might be to minimise the number of decision nodes in the code generated for a match; in this case, the problem can be shown to be NP-complete [4]. In practice, we eliminate backtracking (example 1) and identify linear switches (example 2), and adopt a simple scheme to choose the most appropriate first match in a record. The details are beyond the scope of this paper.

The scheme adopted for compilation of matches and bindings is based on the concepts behind an earlier effort by Marianne Baudinet [4], although that code was written in Lisp and not referred to for the current implementation. The scheme involves the generation of *decision trees*. A decision tree is an intermediate data structure which encompasses the decompositions and selections necessary to determine, for any data value, the topmost pattern which matches that value. (We must, of course, provide the same effect as the top-to-bottom, backtracking matching of the semantics.) For any data value, a decision tree performs an analysis which either delivers the first matching rule, or indicates a failure (if the value matches no rule). If a rule is matched, the decision tree also delivers the *value environment* for the pattern. This is a CEnv mapping the pattern's variables to lvars corresponding to internal nodes of the argument.

The generation of decision trees is beyond the scope of this paper; however, we shall describe the trees themselves and show some examples. The `DecisionTree` data structure is shown in Figure 9. It should be noted that decision trees are *typed*, in that they contain entities such as constructors and record labels whose interpretation (as far as the code generator is concerned) is dependent on their type. Decision trees carry lvars which are relevant to the lambda code generator. In addition, decision trees are *symbolic* in that they refer to data decompositions in an abstract sense (label decomposition, constructor decomposition, and so on) rather than in terms of the concrete data operations (specifically, `SELECT`) of the lambda language.

The rôle of a decision tree is to decompose and examine an argument value to determine whether it matches a specific pattern, and if so to return the CEnv binding the variables of the pattern to the appropriate sub-components of the argument. A decision tree contains nodes which perform decompositions of records and constructors, as follows:

`LAB_DECOMPOSE{bind, parent, lab, child}` extracts a field `lab` from a record `parent`, binding it to a new lvar `bind`;

`CON_DECOMPOSE{bind, parent, child}` decomposes a constructed value `parent`, binding the value to which the constructor was applied to the lvar `bind`;

`EXCON_DECOMPOSE{bind, parent, child}` performs a similar operation for exception constructors;

`CON_SWITCH{arg, selections, wildcard}` performs a case selection over a set of constructors, depending on the constructor of the constructed value bound to `arg`. If the set is exhaustive then the wildcard is omitted, otherwise the wildcard is the decision tree to be traversed if the value matches none of the constructors present;

`SCON_SWITCH{arg, selections, wildcard}` performs a case selection over *special constants* (integers, reals or strings). The wildcard must always be present since a selection over constants can never be verified as exhaustive at compile-time. (Recall from Section 5 that the wildcard is optional in the lambda language's `SWITCH_I` instruction, since this can be generated by the compiler from `CON_SWITCH` decision nodes. An `SCON_SWITCH` corresponds to constant patterns in the source code, and these can never be exhaustive.)

`EXCON_SWITCH{arg, selections, wildcard}` performs a case selection over exception constructors. Note that the selection is *sequential* (a list rather than a map). This is because it is impossible to determine the identity of exceptions at compile-time (cf. value constructors, whose values are, semantically, simply the identifiers). Consider the following example:

```
let
   exception E1 of int
   exception E2 = E1
```

```
signature DECTREE_DT =
  sig
    type lab and lvar and var and con
    type longexcon and scon and pat
    type type_info
    type RuleNum sharing type RuleNum = int
    type (α=, β) map
    type CEnv

    datatype α option = NONE | SOME of α

    datatype DecisionTree =
      LAB_DECOMPOSE of {bind: lvar, parent: lvar, lab: lab,
                        child: DecisionTree, info: type_info
                       }

    | CON_DECOMPOSE of {bind: lvar, parent: lvar, child: DecisionTree}
    | EXCON_DECOMPOSE of {bind: lvar, parent: lvar, child: DecisionTree}

    | CON_SWITCH of {arg: lvar,
                     selections: (con, (type_info × DecisionTree)) map,
                     wildcard: DecisionTree option
                               (* An option because we may notice that all *)
                               (* the constructors are present.   *)
                    }
    | SCON_SWITCH of {arg: lvar,
                      selections: (scon, DecisionTree) map,
                      wildcard: DecisionTree
                     }
    | EXCON_SWITCH of {arg: lvar,
                       selections: (longexcon × DecisionTree) list,
                       wildcard: DecisionTree
                      }

    | END of {ruleNum: RuleNum, environment: CEnv}
    | FAIL
  end
```

Figure 9: The decision tree data structure

```
in ... handle E1(0) => e1
            | E2(n) => e2
            | E1(n) => e3
end
```

It is incorrect to merge the two occurrences of `E1` in the match since `E2` is dynamically identical to `E1` and so a packet `E1(1)` matches `E2(n)`.

(It might be argued that the compiler could maintain an environment to track the identity of exceptions in cases such as this. However, examples using the modules language illustrate that exception identity cannot always be determined at compile-time, and we wish to encompass the modules language at some stage.)

`END{ruleNum, environment}` indicates a successful match and provides the index `ruleNum` of the matching rule together with a CEnv `environment` for the variables of the appropriate pattern. An `END` node does not contain any vestige of the right-hand-side of the match rule, since the right-hand-sides may not exist, as explained below.

`FAIL` indicates the failure of all patterns to match the value.

The decision tree contains some embedded type information provided by the typechecker. The compiler must convert the labels and constructors of the decision tree into integer selections in the lambda language, and therefore needs to know the ordinal index of each constructor and record label. In addition, the type information is needed to determine the exhaustiveness of a value constructor switch; it allows the compiler to determine whether or not all the constructors are present.

The compile-time warnings (irredundancy, exhaustiveness) are performed in a separate post-pass over the decision tree. The unreachable rules of a match are those which don't occur in any `END` node. A match is inexhaustive if the decision tree contains any `FAIL` nodes.[5]

The final stage in compiling a match is to generate lambda code from the decision tree. This is done by a function with the following specification:

```
fun compileDecisionTree (env: CEnv)
                        (tree: DecisionTree,
                         compiler: (int × CEnv) → LambdaExp,
                         failure: LambdaExp
                        ): LambdaExp
```

[5] The assumption here is that unreachable `FAIL` nodes are never generated in our decision tree building algorithm. Experimentation suggests that this is the case, but it is not yet proven.

`compileDecisionTree` takes a CEnv as (curried) argument. This is only needed to generate code for case selection over exception constructors: the identity of an exception constructor depends on the dynamic environment. The `failure` argument is the lambda expression to plant for each `FAIL` node: an expression is chosen to raise the pervasive exception `Match` in lambda expressions, `Bind` in value bindings, and will re-raise the root of the match in exception handlers. Of most interest is the higher-order argument `compiler`. This is an encapsulation of the compiling function to be applied to the right hand sides of the match. Recall that decision trees are generated from a root lvar and a list of patterns only—the right hand side expressions are not provided. This is for the very good reason that they might not exist: consider the case of a top-level binding such as

```
val (x, y) = (1, 2)
```

This is passed to the match compiler as

(fn (x, y) => *scope*) (1, 2)

where *scope* is lambda-abstracted in the compiler. So, `compileDecTree` is called with an encapsulation of the expression compiler which delivers the lambda code for the appropriate right hand side. The compiler is passed the rule number and the CEnv for the variables in the pattern.

It should be noted that a rule might occur in several leaves of a decision tree. In such cases, it is possible to lambda-abstract the right hand side in order to avoid the duplication of code. The abstraction would be performed over the variables of the pattern. This optimisation is not performed at present.

Compilation from decision tree to lambda code is fairly straightforward. All the decomposition operations compile to the appropriate kind of `SELECT`. `CON_SWITCH` compiles to an integer switch over the tag field of the argument. `SCON_SWITCH` compiles directly to the appropriate kind of `SWITCH` in the lambda language. `EXCON_SWITCH` is slightly more complicated, since the matches must be performed in sequential order. The code generated consists of a nested sequence of conditional expressions: the argument value's exception reference is compared against each exception constructor in turn, using the pervasive equality predicate over the string references.

8 The Primitive Functions

Any Standard ML system must provide the built-in functions described in appendices C and D of the Definition. Some of these (such as `map` and `rev`) can be defined directly in SML. Others (such as the arithmetic operators and the equality predicate) cannot. The Definition defines the latter in two ways: the initial static basis contains the types of the pervasives, and the initial dynamic basis contains the basic values which are differentiated at application time (rule 116).

The special status of the basic values is inelegant in a compiler, partly because the contents of several environments have to be "hard-wired" into the compiler in several places, and also because we have decided that the different kinds of function application should be differentiated at compile-time rather than runtime. Therefore, we choose to have a single built-in function called `prim`, and we define the other pervasives in terms of it.

The initial static environment contains a single entry: the identifier `prim`, with type `(int` $\times$ α`)` $\rightarrow$ β. In order to describe the dynamic binding of `prim`, we must revise our description of compiler environments (CEnv's). The interface `COMPILER_ENV` contains the specifications

```
datatype result = LVAR of lvar
                | PRIM

val lookupLongvar: CEnv → longvar → result
```

and the initial compiler environment just contains a mapping from `prim` to `PRIM`. Whenever identifiers are declared in a CEnv, they map to `LVAR(`*lv*`)` for some *lv*.

The compiler looks out for the special case where an identifier mapping to `PRIM` is applied directly to an argument. (Looking for textual occurrences of `prim` is inelegant, and would require redeclaration of `prim` in any way to be prohibited.) In such an application, the argument is required to be a record with two fields, the first of which must be an integer constant. This number must coincide with the implementation of the semantic function APPLY which in the compiler takes an integer index. Occurrences of `PRIM` are converted into `PRIM_APP` in the lambda language. Because the application of `PRIM` is detected and converted in the compiler, the DEnv has no knowledge of it, and no lvars are involved.

`PRIM_APP` always takes two arguments. The first is an integer index indicating the primitive function required, and the second is a either a single argument or a tuple of arguments for the primitive function. Thus, the type of `prim` can be restrained to `(int` $\times$ α`)` $\rightarrow$ β rather than the more general (and less safe) $\alpha \rightarrow \beta$. If a primitive function is capable of raising an exception, the exception packet or function is passed as one of the arguments; hence, there are no reserved exceptions in the implementation of APPLY.

The pervasive functions are defined in a *prelude*, part of which is shown in Figure 10. This prelude is processed by the Kit when it is built. Each pervasive identifier is bound to an application of `prim` to the correct arguments, type constraints being added to constrain the unsafe polymorphic type of `prim` to the correct type for the primitive in question. By convention, equality is primitive 0, and is defined in the prelude in the same way. (This perhaps violates the assertion in the Definition that the equality symbol may not be re-bound; however, we would argue that this is its *initial* binding.) Parts of the compiler which require the equality predicate generate lambda code containing `PRIM_APP(0, ...)` directly. This implementation scheme means that the primitives are implemented

```
val (op =) = fn (x: α₌, y: α₌) => prim(0, (x, y)): bool
val floor = fn (x: real, y: real) => prim(1, (x, y, Floor)): int
val real = fn (x: int) => prim(2, x): real
val sqrt = fn (x: real) => prim(3, (x, Sqrt)): real
val sin = fn (x: real) => prim(4, x): real
val cos = fn (x: real) => prim(5, x): real
...
```

Figure 10: Part of the initial prelude

in a rather inefficient manner (there is an extra level of function application for each primitive application), but we would rather defer that to later optimising passes, and keep the basic scheme simple.

9 Lambda Code Optimisation

It might seem strange to implement an optimiser in a compiler which makes no pretence at being efficient. The reasons for implementing an optimiser for the lambda language are two-fold. Firstly, the earlier stages of the compiler (the decision tree generator in particular) generate a lot of garbage, such as bindings of variables which are never used. In addition, simple constructor applications such as `CON(3)` are compiled into function applications with a closure for `CON`, so we should a least implement an optimiser to remove these. It is therefore desirable to do simple optimisations in order to clarify the final code for the purposes of inspection. Secondly, the optimiser was an afternoon's work, and an interesting exercise.

The interface to the optimiser is shown in Figure 11, and its function is fairly self-evident. Internally, it is implemented in two parts. There is a generic tree-walker which walks over a lambda expression, applying a `LambdaExp` $\rightarrow$ `LambdaExp` function to each node, and in addition there is a set of simple optimising functions, each of which has type `LambdaExp` $\rightarrow$ `LambdaExp` and operates just on the node which is passed to it. The optimiser operates by building the composition of all the optimising functions, and calling the tree-walker with this composition. The optimiser strives for a fixpoint at which none of the individual optimisations can be performed anywhere over the lambda expression. Each optimising function touches a reference variable whenever it succeeds in applying its optimisation; the optimiser makes repeated passes over the input until a pass is made which leaves the reference untouched.

We describe the optimisations below, but must first define some terms. A lambda sub-expression is said to be *small* if it can be substituted for a variable occurrence without a large increase in code size. A sub-expression is said to be *safe* if it performs no side-effects. The leaf nodes denoting variables and constants are considered small. The conditions for safeness are, in theory, more complex, but the algorithm used is a simple-minded one: lambda terms are safe if they contain no applications or pervasive applications, and don't raise exceptions. Safe terms are those which can be removed or relocated (within the same or a nested context) without changing the meaning of the program.

It is here that the uniqueness of lvars is especially useful. Lambda terms can be examined for specific lvars without consideration of scope, since there is only one binding of each lvar. And, terms can be moved from outer to inner contexts without variable renaming being necessary, since no inner context can rebind a variable referred to by an enclosing term.

The optimisations are as follows:

UNUSED: An lvar binding can be removed if the lvar is not used in the scope of the binding, *and* the lvar is bound to a safe term.

BETA1: If an lvar is bound to a small, safe term, each occurrence of the lvar in the scope is replaced by the term. When the small term is another variable, this implements variable renaming.

BETA2: If an lvar is bound to a safe term (of any size) and the lvar is only used once in the scope, the binding is removed and the lvar replaced with its term.

HOISTFIX: By default, "`fun`" declarations compile into `FIX` lambda code constructions. This optimisation lifts non-recursive function definitions out of the `FIX`, making them available for beta optimisation.

FIX0: The HOISTFIX optimisation might leave behind empty `FIX` constructors; FIX0 removes them.

SWITCH: The match compiler can, in some circumstances, produce switches with a wildcard but no discriminants. This optimisation replaces each such switch directly with its wildcard.

Beta reduction is not performed over functions. If it were, variable renaming would be necessary to avoid duplication of binding occurrences of lvars, and steps would have to be taken to deal with the Y combinator.

10 Execution

Recall that `compileDec` has type

```
CEnv → dec → (CEnv × (LambdaExp → LambdaExp))
```

```
signature OPT_LAMBDA =
  sig
    type LambdaExp

    val optimise: LambdaExp → LambdaExp
  end
```

Figure 11: The signature OPT_LAMBDA

which allows the compiler to deal with declarations rather than expressions. At top-level, each core declaration is passed to `compileDec`, to return a CEnv for the declaration plus a function f of type `LambdaExp → LambdaExp`. From the CEnv is built a `VECTOR` lambda expression containing all the lvars in some canonical order, and f is applied to this. The result is a lambda expression for the top-level declaration over a scope containing a tuple of the variables bound by the declaration. This lambda expression is then executed, to return (if successful) a vector of object values, one for each variable. The vector is then decomposed to form a DEnv for the declaration. Finally, the top-level basis is augmented with the CEnv and the DEnv.

The top-level context of the compiler is a `DynamicBasis`, comprising a CEnv (which maps all the visible top-level identifiers to lvars) and a DEnv (which maps all the lvars for the visible top-level identifiers to their runtime representations[6]). Each top-level declaration is compiled in the top-level CEnv and executed in the top-level DEnv, and the new bindings are added to the top-level if it terminates successfully.

The interface to the lambda interpreter is shown in Figure 12, and is fairly self-explanatory. In practice, the implementation is fairly simple as well. Lambda expressions are passed to a function

```
run: DEnv → LambdaExp → object
```

which executes them in a DEnv representing the current top-level environment. Variables in the lambda expression are retrieved directly from the DEnv. Constants evaluate directly to the corresponding objects. Function (`FN`) expressions evaluate to closures encapsulating the current environment. The treatment of `FIX` is similar to the treatment of `val rec` in the Definition: firstly, the function expressions which form the right hand sides of the fix are evaluated in the current environment, yielding closures which do not incorporate the identifiers in the recursive bindings. Then, an environment is built which maps the identifiers to the

[6] It is assumed that when SML variables go out of scope at top level, their lvars are marked for garbage collection.

```
signature RUN_LAMBDA =
  sig
    type DEnv
    type LambdaExp
    type object

    exception UNCAUGHT of string

    val run: DEnv → LambdaExp → object
  end
```

Figure 12: The signature RUN_LAMBDA

closures. Finally, the Rec operation is applied to the environment—this traverses the closures, replacing the recursive environment component of each closure with the entire environment, including the `FIX`-bound variables. This provides for a single level of recursion; indefinite recursion is provided for by subsequent applications of Rec within function applications. The operation of Rec is described more fully in the SML Definition and the SML Commentary.

Application (`APP`) of one lambda expression to another results in the function and argument being evaluated, and then the body of the function being executed in the closure environment of the function extended by an lvar binding for the argument. Calls to the pervasive functions (`PRIM_APP`) result in calls to the `Apply` module on the evaluated argument. `VECTOR` and `SELECT` produce calls to `Objects.vector` and `Objects.select`.

Exception handling is done in a meta-circular manner; the lambda interpreter contains a single exception called `PACKET`. When a lambda expression raises an exception, the interpreter raises `PACKET` with the exception packet. Expressions of the form `HANDLE(`e_1`, `e_2`)` cause e_1 to be interpreted in a context which handles `PACKET`, in which case e_2 (which must be a `FN` lambda term) is applied to the packet's value. If `PACKET` propagates to the top level it is transformed into an exception called `UNCAUGHT` so that it can be reported at top level (or signalled into an enclosing `use` context).

The implementation of SML references and assignment (i.e. the constructor `ref` and the function `:=`) is currently incomplete, and so we defer a discussion of references to a future paper.

11 Conclusions

We have attempted to show that it is possible to build an SML to intermediate lambda language compiler which is modular and which uses simple data structures. Our thesis is that modularity and abstraction are initially more important than raw efficiency—a well-structured compiler can be made more efficient by optimising the modules and refining the interfaces between them, but an efficient compiler can rarely be improved in modularity and structure.

The lambda language is small and simple, with nice properties (such as the unique identity of lvars) and yet it is a full functional language in its own right. Currently, many projects depend on making alterations and enhancements to the SML source text or treating it in some special way; we hope that the lambda language can be used in this way instead, thus avoiding the complexity and ambiguities of the SML language itself.

The match compiler is largely complete but only contains trivial optimisation heuristics. We would like to develop it further and make comparisons of heuristics. Since we are claiming modularity as a virtue of the Kit, it should be easy to replace the current match compiler with others; an implementation of Wadler's scheme [5] would be a useful exercise, as well as a "trivial" sequential implementation (in the style of the semantic rules), in order to compare code size and performance directly.

References

[1] Andrew W. Appel. A runtime system. Technical Report CS-TR-220-89, Princeton University, February 1989.

[2] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[3] Andrew W. Appel and David B. MacQueen. A Standard ML compiler. In *Functional Programming Languages and Computer Architecture*, pages 301–324. Springer-Verlag, 1987.

[4] Marianne Baudinet and David MacQueen. Tree pattern matching for ML. Extended abstract, unpublished, December 1985.

[5] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall, 1987.

[6] Robin Milner and Mads Tofte. *Commentary on Standard ML.* MIT Press, 1991.

[7] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1990.

[8] Donald Sannella and Andrzej Tarlecki. Extended ML: Past, present and future. Technical Report ECS–LFCS–91–138, Laboratory for Foundations of Computer Science, University of Edinburgh, 1991.