Compiling with Continuations

Andrew W. Appel

Book DOI: http://dx.doi.org/10.1017/CBO9780511609619

Online ISBN: 9780511609619

Hardback ISBN: 9780521416955

Paperback ISBN: 9780521033114

Chapter

4 - ML-specific optimizations pp. 37-54

Chapter DOI: http://dx.doi.org/10.1017/CBO9780511609619.004

Cambridge University Press

CHAPTER FOUR

ML-SPECIFIC OPTIMIZATIONS

In a compiler using continuation-passing style, most optimizations (partial evaluation, dataflow, register allocation, etc.) should be done in the CPS representation. However, some representation decisions are best done at a level closer to the source language. Here we describe several optimizations, specific to ML, that are done before conversion into continuation-passing style. Most of them are related to static types, which is why they are most naturally done before the types are stripped off during the conversion into CPS.

4.1 Data representation

Standard ML has *record* types that are essentially Cartesian products of other types (like records in Pascal or structs in C, except that their fields may not be modified after the record is created), and *datatypes* that are disjoint sums (like variant records in Pascal or unions in C).

A record type is a set of named fields, each of a given (possibly polymorphic) type. For example, the type

```
type t = {name: string, number: int}
```

contains values such as `{name="Sam",number=5}`; the order in which the fields are written down is immaterial (this value is the same as `{number=5,name="Sam"}`).

Since types in ML can be polymorphic, type t is also an instantiation of the type constructor

```
type 'a r = {name: string, number: 'a}
```

so $t = int\ r$. Similarly, `type s = real r` is a record type of which each element contains a string a real number. (ML novices should note that the parameter of a type constructor is put *before* the constructor, not after!)

Since an ML program can only access a record field in a context where the names of all the fields are known, the implementation can represent records as simple n-tuples. We choose in our implementation to number the fields in alphabetical order starting at zero, so (in this case) `name` is field 0, and `number` is field 1. The alphabetization is necessary because the field names might occur in any

order in expressions where the record type is used. Presumably, the fields will all be contiguous in storage.

However, because ML is a polymorphic language, the (ground) types of all the fields are not necessarily known. For example, the function `printname`

```
fun printname {name=s, number=n} = output(std_out, s)
```

has type $\alpha\ r \rightarrow unit$, that is, a function from $\alpha\ r$ (for any α) to *unit*, which is just a placeholder for functions that don't return any interesting result (like `void` in the C language). But this means that the type of n is not known at compile time! The solution to this problem is to make all record fields the same size—for example, one word each. Every ML object will be represented in exactly one word; of course, that word may be a pointer to some data structure in memory.

Polymorphism is not unique to ML. Lisp, Scheme, Prolog, and other languages also have variables whose complete types are not known until runtime. When it is necessary to manipulate these values (e.g. to make lists of them), they must all have the same size—in most implementations, one word.

So, a record of n fields will be represented using n contiguous words in memory. ML also has a *tuple* type, which is like a record type with unlabeled fields; the type `int*bool*int` contains values such as `(4,true,7)`, and is represented in an implementation just like a three-element record.

A disjoint union type, called a "datatype" in ML, is represented in the source language by a set of constructors that may be applied to values, for example,

```
type posint = int (* positive integers *)
datatype money = COIN of posint | BILL of posint
               | CHECK of {amount:real, from: string}
datatype color = RED | BLUE | GREEN | YELLOW
datatype 'a list = nil | :: of 'a * 'a list
datatype register = REG of int
datatype tree = LEAF of int | TREE of tree * tree
datatype xxx = M | N | P of int list
datatype yyy = W of int * int | X of real * real * real
datatype gen = A | B | C | D of int | E of real
             | F of gen * gen | G of int * int * gen
```

A value of type `money` is either a "coin" with an integer value, a "bill" with an integer value, or a "check" with a value that consists of a real number and a string. A program can examine a piece of `money` to see which constructor has been applied, and can extract the associated value carried by the constructor. This is done using a "pattern match" in ML:

```
val evaluate = (* calculate value of a piece of money *)
     fn COIN c => c
      | BILL b => b * 100
      | CHECK{amount=r, from="Joe Deadbeat"} = floor(r*50)
      | CHECK{amount=r, from=f} = floor(r*100)
```

Some constructors (such as `RED`, `BLUE`, `nil`, `A`, etc.) do not carry values. Thus, the datatype `color` is like an enumeration type in Pascal.

How are constructors to be represented in memory? The most straightforward way is to say that each value of a datatype is represented as a two-word value, with the constructor (represented as a small integer) in one word, and the carried value (if any) in the other word. However, there are several improvements that can be made. First, datatypes with only one constructor (such as the `register` type above) can be represented completely transparently; the representation can be exactly the same as the representation of the carried value. Such datatypes are used in ML mainly to assist in catching type errors (in this case, for example, the mistake of using an integer as a register number).

It is possible to make more specialized representations for datatypes [26]. We might make the assumption:

Assumption 1: At runtime, pointers can be distinguished from small integers. This might be true if, for example, no pointer pointed into the first 256 bytes of memory, or if the low-order bit of each word is used as a tag to distinguish pointers from integers.

Using this assumption, we can say that all "constant" constructors (those that don't carry values) are represented as small integers, whereas value-carrying constructors are still represented as two-word records with the (small-integer) tag in one of the words and the value in the other. This makes the representation of constant constructors much more efficient, as they won't require allocation of memory on the heap.

Also, datatypes with only one value-carrying constructor don't need the tag in their record. For example, the datatype `xxx` has three constructors, but only one is not a constant. Therefore, if a value of type `xxx` is a pointer, it must be an application of the `P` constructor. The pointer will point to a single-word record containg the value of type `int list`. Note that the value of type `P` needs the extra indirection; it can't be an `int list` directly, since lists are not always pointers—sometimes they are `nil`. And the value `nil` has the same representation as the constructor `M` of the `xxx` datatype. Thus, we need the indirection so values with the `P` constructor will always be boxed, and distinguishable from `M` and `N`.

Datatypes with only one value-carrying constructor applied to an *always-boxed* value don't even need the extra indirection. A *boxed* value is simply one represented as a pointer; an *unboxed* value is represented as an integer (or other nonpointer). Consider the type `'a list`: Since the `::` (pronounced *cons*) constructor is always applied to a record, the value it carries is always a pointer. Since the `nil` value is represented as a small integer, we know that `::` values can always be distinguished from `nil` values. Thus, we can represent `5::nil` as a record containing 5 and `nil`, without any extra indirection.

More specialization is possible. Consider:

Assumption 2: All pointers can be distinguished from all integers. This might be true if one bit of each word is used as a tag to distinguish pointers from integers. The `boxed` predicate of the CPS language performs exactly this test. Of course, this assumption might be unpalatable for several reasons. It makes the represen-

tation of "big" (arbitrary precision) integers more difficult, since those are usually represented by pointers to some complicated representation. Also, it uses up one bit of precision in the representation of integers, which can be a great inconvenience. Usually, Assumption 2 is required by the garbage collector anyway. But it is possible to make a collector for ML that does not need any runtime tags on data, even to distinguish pointers from integers [5]. Therefore, we might prefer to have full 32-bit integers, and no overhead to tag and untag integers when performing arithmetic; in this case, Assumption 2 would not be valid.

Using Assumption 2, we can specialize the representation of datatypes such as `tree` above. Any `LEAF` value can be represented by the integer that the constructor is "carrying," and any `TREE` value can be represented by the two-word record. Then no "extra" indirections are required in either case.

If we are willing to presume the existence of lots of runtime tags, we can make more assumptions:

Assumption 3: Two-element records can be distinguished from three-element records, and so forth. This is easy enough with a record descriptor at the beginning of each record (or built into the record pointer itself). But the use of Assumption 3 may unduly constrain the implementation of the runtime system. In particular, it makes the use of a BIBOP ("BIg Bag Of Pages") scheme more difficult. With such an arrangement, objects of the same type are grouped together on a page, and there is just one descriptor for each page—this saves the space required by descriptors on each object. But then, if the compiler generates pattern-matching code that must distinguish between different types of records, the compiled code must use the BIBOP descriptor table. This is inefficient and clumsy.

Using this assumption, we can specialize the implementation of type `yyy`; since the value carried by constructor `X` can be distinguished from the value carried by `Y`, we don't need any extra bits or words for the constructors themselves.

There are many variants on Assumption 3, for example,

Assumption 4: Secords of different sizes, strings, integers, reals, and arrays are all distinguishable at runtime. This has the same sorts of disadvantages as assumption 3.

Then the type `gen` can have the following representation:

- Constructors `A`, `B`, and `C` will be represented as the integers 0, 1, and 2.
- The value `D(i)` will be represented as a one-word record containing i (to distinguish it from the constant constructors).
- The constructors `E`, `F`, and `G` will be represented transparently, since the values they carry can all be distinguished.

For languages like Lisp that have *runtime type checking*, Assumption 4 (or something even stronger) is required. Because ML has compile-time type checking, it is not necessary to distinguish all the different types at runtime—these assumptions merely allow more efficient data constructor representations.

Assumption 5: The type `posint` *will be enforced by the compiler to contain only positive integers.*

Then the datatype `money` can be represented without any extra indirection: Coins will be represented as negative integers, bills as positive integers, and checks as records.

Clearly, we are proceeding *ad absurdem* with these assumptions. We can use arbitrarily complicated encodings of constructors, with diminishing returns in the representations of datatypes. But there is a problem with many of these representations in Standard ML: Types can be abstract, so the details of their representation are not known at compile time. Consider:

```
datatype ('a,'b) t = A of 'a | B of 'b
type u = (int, real*real) t
```

In this case, the type constructor t can be applied to any pair of types, so the representation of A and B cannot be specialized. On the other hand, the representation of u might be specialized, since its values are all of known type. However, any function applicable to values of type t can also be applied to values of type u, so the representation must be the same, or conversion must take place.

Things get worse. Consider the functor:

```
functor F(S: sig  type 'a t
                  datatype 'a list = nil | :: of 'a t
             end
         ) = struct . . . end
```

applied to the structure

```
structure A =
struct  datatype 'a list = nil | :: of 'a * 'a list
        type 'a t = 'a * 'a list
end
```

Now, the functor F must assume nothing about the structure of t, and must therefore use an extra indirection in the representation of `::` to ensure that it is boxed. On the other hand, structure A makes use of Assumption 1 to avoid the extra indirection. Since the modules F and A can be compiled separately, there is no perfect solution to this problem.

If functors in ML behaved like the "generic" modules of Ada, this problem would not exist: It is expected of an Ada implementation that each application of a "generic" will generate new code specialized to the particular argument. But in ML the intent of the designers was that machine-code generation (and type checking, etc.) needs to be done only once for each functor, and is independent of the actual parameter to the functor.

In the implementation of Standard ML of New Jersey, we wished to avoid the functor problem, and we wanted to avoid too many constraints on the runtime system. On the other hand, we felt it was unacceptable to have extra indirections in the `list` datatype. Therefore, we rely only on Assumption 1 in the representation of datatype constructors (though we use Assumption 2 elsewhere in the compiler),

and we assume types are boxed *only* when they are records explicitly written down in the `datatype` declaration, for example,

```
datatype a = A | B of int * int
type pair = int * int
datatype c = D | E of pair
```

The datatype a will be represented without extra indirections, but c will use extra indirections. This makes functor mismatches very rare, though it does not completely eliminate them; we detect this problem at functor application time, however, and print an error message for the user.

There is one last twist to constructor representation. Standard ML has an *exn* datatype to represent exceptions that can be raised (by the `raise` operator) and handled (by `handle`). This datatype has an unbounded number of "exception constructors," some of which carry values and some of which are "constant." Clearly, we cannot associate *small* integer tags with each one, since there are so many of them. We have chosen to represent the *exn* type as a pair of words, where one word is the carried value (if any) and the other word is the tag—just as for ordinary value-carrying constructors. This will be true even for "constant" constructors; for them the "value word" will just be zero.

To represent the tag, any type with an unbounded number of values and that admits equality (to test the identity of a constructor) will do. We have chosen to use "string ref" instead of integer type for this purpose, since the string is useful to hold the name of the exception for error reporting by the runtime system (as in "Uncaught exception Match," when the `Match` exception is propagated to top level).

In summary, we have discussed several different kinds of constructor representation:

Tagged: A two-word record; one word is the value and one word is a small-integer tag. The value-carrying constructors in any one datatype are numbered contiguously starting at zero.

Constant: An unboxed integer representing a constant data constructor. The constant constructors for any one datatype are numbered contiguously starting at zero.

Transparent: A value-carrying constructor in a datatype with only one constructor need not be represented at all; $c(v)$ will have the same representation as v.

TransB: Transparent boxed: A value-carrying constructor which is known to be applied to an always-boxed type, in a datatype with no other value-carrying constructors, can be transparent (if Assumption 1 is used).

TransU: Transparent unboxed: A value-carrying constructor which is known to be applied to an always-unboxed type, if there are no constant constructors in the datatype and all other constructors are boxed, can be transparent (if

Assumption 2 is used). We have chosen not to use this representation at present in Standard ML of New Jersey.

Variable: A value-carrying exception constructor, as described above.

VariableC: An exception constructor without an argument, as described above.

As each datatype is defined, the compiler analyzes the constructors and types to choose a representation for each constructor.

4.2 Pattern matching

One important and nontrivial job of the ML-specific part of the compiler is to select optimal comparison sequences for the compilation of pattern-matching. A *match* in ML is a sequence of pattern–expression pairs, called *rules*. When a match is applied to an argument, the argument is matched against the patterns, and the first rule with a matching pattern is selected and its expression is evaluated. A pattern is either a constant, which must match the argument exactly; a variable, which matches any argument (and is bound to it for the purposes of evaluating the expression); a tuple of patterns, which matches a corresponding tuple argument whose components match the components of the pattern tuple; or a constructor applied to a pattern, which matches an argument built using that constructor if the rest of the pattern matches.

As an example, consider the case statement:

```
case a
 of (false, nil)   =>  nil
  | (true, w)    =>   w
  | (false, x::nil)  =>  x::x::nil
  | (false, y::z)  =>  z
```

The argument `(false, 4::nil)` matches the third pattern, whereas the argument `(true, 4::nil)` matches the second pattern.

One could imagine a naive compilation of matches just by testing the rules in turn as called for by the semantics. Our approach is to transform a sequence of patterns into a decision tree [19]. Each internal node of the decision tree corresponds to a test, and each branch is labeled with one of the possible results of the test and with a list of the patterns that remain potential candidates in that case. It is then straightforward to translate the decision tree into code for pattern matching. During the construction of the decision tree it is also easy to determine whether the pattern set is "exhaustive," meaning that every possible argument value matches at least one pattern, and whether there are any "redundant" patterns that only match arguments covered by previous rules. Nonexhaustive and redundant patterns result in warning messages by the compiler.

Our goal in constructing the decision tree is simply to minimize the total number of test nodes. This minimizes the size of the generated code and also generally

reduces the number of tests performed on value terms. However, finding the decision tree with the minimum number of nodes is an NP-complete problem [19]; so a set of efficient heuristics is used that in practice produces an optimal decision tree in almost all cases.

In the example above, testing the first component of the pair for truth or falsity suffices to distinguish the second rule from the others; then testing the second component to see whether it is `::` or `nil` distinguishes the first rule from the last two; one more test suffices to separate the last two rules. Thus, in just two or three tests, the appropriate rule can be selected; instead of two or three tests *per rule* that the naive algorithm would use.

The result of the decision-tree algorithm for pattern matches is a sequence of multiway branches, each testing which constructor is attached to a given value. For example, the *case* expression

```
case mygen
 of (true, A) => a
  | (false, B) => b
  | (true, E x) => e(x)
  | (false, F(x,y)) => f(x)
  | (true, G(1,_,x)) => f(x)
  | (false, _) => c
  | (_, G(2,_,_)) => c
  | _ => d
```

might be compiled by the "match compiler" in one of several different ways. For example, the Boolean could be tested first, followed by tests of the *gen* constructor (defined on page 38) lower in the tree; or the *gen* constructor could be tested first as follows (written in ML, though the compiler actually uses a much simpler intermediate representation):

```
let val (i,j) = mygen
 in case j
     of A => (case i of true => a | false => c)
      | B => (case i of true => d | false => b)
      | E x => (case i of true => e(x) | false => c)
      | F(x,y) => (case i of true => d | false => f(x))
      | G(z,y,x) => (case i of true => (case z of 1 => f(x)
                                                | 2 => c
                                                | _ => d)
                             | false => c)
      | _ => (case i of true => d | false => c)
end
```

Now, each *case* expression tests only one datatype to see which constructor has been applied (Boolean and integer values are like "constant" constructors).

4.3 Equality

In Standard ML a programmer may compare two values for equality if they are of the same type, and the type is not a function type or a data type that contains function types. The equality is *structural:* two lists containing equal values are considered equal. However, two references (mutable cells) are considered equal only if they are the same cell (i.e., at the same address). This simplifies the testing of structural equality, as cycles need not be considered—every cycle in an ML data structure goes through a `ref`.

Equality may be tested even if the types are not fully known, for example,

```
fun member(x, a::rest) = x=a orelse member(x,rest)
  | member(x, nil) = false
```

This is a polymorphic function (type $\alpha \times \alpha$ list $\rightarrow$ bool), and the type of x cannot be known at compile time. The type checking rules will enforce, however, that α cannot be a function type or a data structure containing function types.

In some cases, however, the type is known:

```
fun f(x:tree, y:tree, z:tree) = x=y orelse y=z
fun g(i) = if i=0 then j else i
```

(using the tree datatype shown on page 38).

When the type of the equality test is known (i.e., is a ground type, not containing any type variables), then the compiler can generate special-purpose functions to implement each instance of equality. For example, we automatically generate a function such as the following to implement the equal sign in `x=y` above:

```
fun eqtree(LEAF i, LEAF j) = Integer.=(i,j)
  | eqtree(TREE(a,b), TREE(c,d)) =
              eqtree(a,c) andalso eqtree(b,d)
  | eqtree _ = false
```

and for the test `i=0` in `g(i)` we just use the integer equality primitive. In general, the automatically generated functions are mutually recursive, to follow the structure of mutually recursive datatypes.

When the type of the equality test is not known, we must rely on runtime tags. In particular, it is necessary to make use of Assumption 4 (page 40), that the size of each object can be determined at runtime. This is the only place in the implementation of ML where this assumption is necessary; even the garbage collector could in principle learn the sizes and types of objects from a static description of the compile-time type system [5]. It is conceivable that the representation of each and every "equality type" could contain within it an equality predicate, but this would be quite expensive [92].

How is the size of an object to be determined at runtime? There is no `PRIMOP` in the CPS language, for example, that tells the size of a record; this is to allow an implementation's runtime system some freedom in representation decisions. A simple runtime system might put a descriptor word before every record; a fancier

system might use a "big bag of pages" (BIBOP) scheme to use just one descriptor for a large collection of similar records.

To implement "polymorphic equality," we assume that the runtime system provides an *externally defined* function that tells the number of fields of a record; this is accessed using the linkage convention described in Chapter 3. Then we can write an ML function that recursively compares the structure of two values, returning false if there is any difference. The pointer-equality test (*ieql*) is used as a shortcut *at each level of recursion*; when pointer equality fails on any object, structural equality is tried. In fact, this shortcut is also used in the case of known types—we generate a test for object identity in the special-case code for each datatype—but for clarity this was not shown in the *eqtree* function above.

Interpreting tags of polymorphic objects is significantly less efficient than using the specially compiled functions that are used for known types.

In summary: Polymorphic equality is no fun at all.

4.4 Unboxed updates

As mentioned in Chapter 3 and explained more fully in Section 16.3, it is helpful for the compiler to identify those `ref` cells and arrays that can hold only unboxed (nonpointer) objects, and to identify those assignment (`:=`) and `update` operations that are guaranteed to store unboxed objects (even if into a `ref` cell that might also hold boxed values). The ML type system helps to identify such `ref` creations and `update`s, and thus it is helpful to mark them prior to CPS conversion, which will strip off the type annotations. Because of type abstractions in ML, there will be some `ref` cells that the compiler can't determine the "boxity" of, and in this case we settle for a conservative approximation, using the general-purpose `ref`-creation or update operators that can handle either boxed or unboxed values. The result of this analysis will be to replace some of the `:=` operators in the program by `unboxedassign`, some of the `makeref` operators (that create `ref` cells) by `makerefunboxed`, and some of the `update` operators by `unboxedupdate`.

4.5 The mini-ML sublanguage

We now describe a "mini-ML" language, into which Standard ML programs can be translated. The translation will simplify the program significantly, and will incorporate all of the optimizations described earlier in this chapter.

Mini-ML is an untyped language; however, any mini-ML program could in principle be embedded in a parametric-polymorphic (second-order) lambda calculus [75]. Mini-ML programs can't be type checked as ML programs for two reasons: Mini-ML has no `let` expressions, and the parametric modules of ML (functors) are encoded as ordinary functions of mini-ML.

The datatypes of mini-ML are a subset of Standard ML's:

- integers, reals, and strings;

- datatypes with constructors, as in Standard ML;
- n-tuples, for $n \geq 0$;
- mutable arrays;
- single-argument, single-result functions.

The other ML datatypes can be translated into these. Records with named fields can be translated into n-tuples, with the loss of the distinction between different record types of arity n. But this translation is done *after* ML type checking, so the loss of some type information is not harmful.

The expressions of mini-ML are a subset of Standard ML's:

- variables;
- integer, real, and string literals;
- application of data constructors;
- removal of data constructors (see below);
- very simple case expressions (see below);
- n-tuple creation;
- selection of fields from n-tuples;
- function application, which is *strict* (as in ML)—the argument of a function is evaluated before substitution for the bound variable;
- function definition using λ (or `fn` in ML syntax), where each function binds a single variable (not a pattern as in ML);
- mutually recursive function definition using `let val rec`;
- primitive arithmetic and comparison operators;
- operators for manipulating references and arrays;
- simple exception handling (see below).

Significant components of Standard ML that are lacking in mini-ML are pattern matching, abstract types, and the module system (structures and functors). All of these are expressed using the simpler primitives of mini-ML.

The mini-ML `case` expression takes one argument—an element of a datatype—and determines which constructor has been applied. Each case rule must be an expression of the form `c` or `c _`, where c is a constant or value-carrying constructor, respectively; the final rule can be a wildcard (`_`). Case expressions can range over integer, real, and string types as well, just as in Standard ML.

Note that case expressions do not bind variables in mini-ML, as they do in ML. Once it has been determined (using a case expression) that a datatype value

has been made using a certain constructor, the carried value (for a nonconstant constructor) may be accessed by stripping off the constructor (by *deconstructing*, or *projecting*). For each value-carrying constructor c there will be a deconstructor decon_c with the semantics

```
decon_c(e) = case e of c x => x | _ => error
```

for any expression e. Clearly, decon_c must only be used in a context where it is guaranteed to work, for example, in the right-hand side of a case rule that has tested for the constructor c.

Section 4.2 shows how ML case expressions can be simplified to test only one datatype value at a time; now, using deconstructors, we show (for the example of Section 4.2) how variable-binding is handled:

```
let val (i,j) = mygen
 in case j
     of A => (case i of true => a | false => c)
      | B => (case i of true => d | false => b)
      | E _ => (fn x => (case i of true => e(x) | false => c))
               (decon_E j)
      | F _ => (fn (x,y) =>
                      (case i of true => d | false => f(x)))
               (decon_F j)
      | G _ => (fn (z,y,x) =>
                  (case i of true => (case z of 1 => f(x)
                                              | 2 => c
                                              | _ => d)
                           | false => c))
               (decon_G j)
      | _ => (case i of true => d | false => c)
end
```

In each case rule that matches a value-carrying constructor c, the carried value is extracted from the constructed object j by a projection operator decon_c. Since mini-ML does not have *let* expressions, we use a λ (`fn`) applied to an argument. Strictly speaking, mini-ML does not have multiargument λs either, so we must use the selection operator to implement them:

```
fn (x,y,z) => M
```

for variables x, y, z and expression M becomes

```
fn xyz => (fn x => (fn y => (fn z => M)
                            (#3 xyz))
                    (#2 xyz))
          (#1 xyz)
```

where $\#i$ is the operator to select the ith field of an n-tuple.

In Standard ML, the `ref` operator that creates a mutable reference to the store is treated as a constructor; a use of `ref` in an expression creates a ref cell, and a use of `ref` in a pattern extracts the contents. Mutation of the contents is done by the assignment operator (`:=`). In mini-ML we abandon the fiction that `ref` is a constructor, and two new primitive functions are introduced: `makeref`(x) to create a ref and initialize it to x, and the fetch operator (written with an exclamation point `!`) to extract the contents. The assignment operator is unchanged.

Exception handling is simpler in mini-ML than in Standard ML. An exception handler in ML is a pattern match on the *exn* (exception) type; in mini-ML a handler is just a function taking an *exn* as an argument. Decision trees are used to simplify exception pattern matches just as for `fn` and `case` expressions.

4.6 Exception declarations

Mini-ML contains no special syntax for declaring exceptions. Each `exception` declaration of Standard ML is turned into a `val` declaration of `mini-ML`. The declarations

```
exception E of int
exception C
exception D = J
```

are translated into

```
val E = ref "E"
val C = ((), ref "C")
val D = J
```

We choose a `string ref` for the representation of exception constructors for three reasons:

- We need some type which can cheaply be compared for equality; `ref` cells can be compared "by reference," which is as cheaply as any type can be tested.
- We need to be able to make new values conveniently; if we used integers there would have to be some central counter to specify which numbers have been used.
- It is convenient to extract the name of an exception, as a diagnostic, when it is raised all the way to the top level.

In each case, the *string ref* behaves like the integer tag of an ordinary value-carrying constructor. But unlike those constructors, the boxity test cannot distinguish constant from value-carrying constructors, since a *string ref* is boxed. So the value-carrying constructors are represented as string refs that will be used as

tags (in two-element records) when applied to values; and the constant constructors are two-element records with the string-ref tag in the second element, and a placeholder in the first element.

4.7 The lambda language

In the Standard ML of New Jersey compiler, mini-ML is encoded into a concrete data structure called the *lambda language* (figure 4.1). The datatype `conrep` is used to specify constructor representations, and is explained in Section 4.1. All of the previous discussion of mini-ML applies to the lambda language. A "lambda expression" *lexp* can be:

- a variable (`VAR`);
- a lambda function (`FN`);
- a "val rec" declaration (`FIX`) that recursively binds several function names (`var list`) to several lambda functions (`lexp list`) in the scope of an expression;
- a `SELECT(i,e)` expression of the lambda language that selects the ith field of an evaluated expression e (fields in the lambda language are numbered starting at zero, instead of starting at one as in ML);
- a `SWITCH` expression that detects which constant or constructor (from the `(con*lexp) list`) was used to build a datatype value, and then evaluates the resulting expression. The `lexp option` is the default case, to be used if none of the constructors on the list matches. For matches where all the constructors of the datatype are used, the `lexp option` may be `NONE`. The `conrep list` field of the `SWITCH` specifies all the legal constructors of the datatype; this is useful in optimizing the code generated for the `SWITCH`. For switches over integer, real, string, and exception types (i.e., all but the "ordinary" datatypes), this list is `nil`;
- a data constructor—constant or value-carrying—applied to an argument (`CON`). In the constant case, the argument is merely pro forma;
- a data value-carrying constructor removed from an argument (`DECON`);
- the `RAISE`-ing of an exception;
- the evaluation of an expression in the scope of an exception `HANDLE`r;
- a primitive operator (`PRIM`).

The lambda language is not really a lambda calculus: It is a call-by-value langauge with an implied state. The side effects are hidden in the `primop`s, which are roughly the same as those of the CPS language and include such things as assignment (`:=`) to the store.

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

eqtype var  (* = int *)

datatype accesspath = OFFp of int | SELp of int * accesspath

datatype conrep = UNDECIDED
                | TAGGED of int
                | CONSTANT of int
                | TRANSPARENT
                | TRANSU
                | TRANSB
                | REF
                | VARIABLE of var * accesspath
                | VARIABLEc of var * accesspath

datatype con = DATAcon of conrep
             | INTcon of int
             | REALcon of string
             | STRINGcon of string

datatype lexp
  = VAR of var
  | FN of var * lexp
  | FIX of var list * lexp list * lexp
  | APP of lexp * lexp
  | INT of int
  | REAL of string
  | STRING of string
  | SWITCH of lexp * conrep list * (con * lexp) list * lexp option
  | CON of conrep * lexp
  | DECON of conrep * lexp
  | RECORD of lexp list
  | SELECT of int * lexp
  | RAISE of lexp
  | HANDLE of lexp * lexp
  | PRIM of primop
```

Figure 4.1. The lambda language.

The primops of the lambda language include some that are not in the CPS primop set, including `callcc` (*call with current continuation*) and `throw` (*throw to a continuation*) [37]. When these are converted into CPS (see Chapter 5), they are expressed using the `FIX` and `APP` operators of the CPS language.

4.8 The module system

Standard ML has a *module* system to facilitate the structuring of large ML programs and to support separate compilation and generic library units. Figure 4.2 shows the syntax of the module system. An ordinary (unparametrized) module is

$$
\begin{array}{rcl}
\mathit{decl} & \rightarrow & \text{ordinary ML } \texttt{val} \text{ or } \texttt{type} \text{ declaration, etc.} \\
\mathit{decl} & \rightarrow & \texttt{structure}\ \mathit{name}\ =\ \mathit{strexp} \\
\mathit{decl} & \rightarrow & \texttt{structure}\ \mathit{name}\ \texttt{:}\ \mathit{signature}\ =\ \mathit{strexp} \\
\mathit{decl} & \rightarrow & \texttt{functor}\ \mathit{name}(\mathit{name}\texttt{:}\mathit{signature})\ =\ \mathit{strexp} \\
\mathit{decl} & \rightarrow & \texttt{functor}\ \mathit{name}(\mathit{name}\texttt{:}\mathit{signature})\ \texttt{:}\ \mathit{signature}\ =\ \mathit{strexp} \\
\mathit{decl} & \rightarrow & \texttt{signature}\ \mathit{name}\ =\ \mathit{signature} \\
 & & \\
\mathit{strexp} & \rightarrow & \texttt{struct}\ \mathit{decl}\ \texttt{end} \\
\mathit{strexp} & \rightarrow & \mathit{name} \\
\mathit{strexp} & \rightarrow & \mathit{name}(\mathit{strexp}) \\
 & & \\
\mathit{signature} & \rightarrow & \texttt{sig}\ \mathit{specifications}\ \texttt{end} \\
\mathit{signature} & \rightarrow & \mathit{name}
\end{array}
$$

Figure 4.2. Syntax of the ML module system (simplified).

called a *structure*. Any set of core ML declarations (such as `val`, `fun`, `type`, or `datatype` declarations) can be bracketed by `struct...end` and made into a structure S. Thereafter, in the scope of the declaration of S, names i, j, k from those declarations can be accessed using *qualified identifiers* $S.i, S.j, S.k$.

If it is desired to export only some of the names from a module, a *signature* can be used in the structure declaration to restrict visibility of names and to constrain the types of exported values.

A parametrized module is called a `functor`, and takes a structure as an argument. The "type" of the formal parameter structure must be specified using a signature, and the "type" of the result structure can optionally be specified using a signature. Functors and signatures cannot be nested inside structures or functors.

Figure 4.3 shows an example of the use of structures and signatures. The signature `STACK` is implemented in two different ways, by structures `Stack1` and `Stack2`. The structure `User` uses some of the primitives of structure `Stack1`. The functor `F` uses stacks, but doesn't care which implementation is used; it can be

applied to any structure that matches the STACK signature. The structure T is an application of F to the Stack2 structure.

The definition of Stack1 doesn't specify—as Stack2 does—that it must match the STACK signature, but it does have all the right fields and can be used anywhere a STACK is required (e.g., as an argument to F). Similarly, the definition of Stack2 would have been just as adequate without the signature constraint.

To translate structures and functors into mini-ML, we will use RECORDs. The representation of Stack1 will be a five-element record containing the values Empty, push, top, pop, and empty in that order. We don't need to represent types (such as stack), because they are compile-time entities that don't have runtime manifestations.

When elements of structures are accessed from outside (e.g., Stack1.push in the User structure), this is translated as a selection from the structure record (e.g., SELECT(1,VAR Stack1)).

Now, if the structures Stack1 and User are compiled one at a time, the interface between them is quite simple. The Stack1 module is a single value (which happens to be a record), the names of interface files have been compiled into record offsets, and the link-loader need not be concerned with the internals of modules.

On the other hand, if the two modules are compiled together, the CPS optimizer (as will be described) can quite easily evaluate the SELECT at compile time, so the User function can apply push just as efficiently as it could have had structures not been used.

When a signature constrains a structure definition, this can have some effect on the representation of the structure. For example, the representation of any structure matching the STACK signature must be a five-element record empty, Empty, push, top, pop, in that order. Therefore, if this signature is applied to the Stack1 structure, the fields must be reordered. Furthermore, when STACK constrains Stack2, the value extra will not be represented in the interface record. It will still be evaluated, but its result will be discarded.

Furthermore, the name push inside Stack2 refers to a data constructor, but in the signature it is an ordinary function. Thus, the record built for Stack2 must have a function push that simply applies the data constructor. In general, the application of a signature to a structure may result in "thinning," meaning that some fields may be discarded, some constructors will turn into ordinary values, and the fields may be rearranged.

Functors in the module system are translated into ordinary functions in mini-ML. Thus F will just be a function that takes a structure (record) as an argument, and returns another structure (record) as a result.

This translation means that the mini-ML language given to the back end of the compiler contains no special notation for the ML module system. All of the complexities of the module system are handled statically or translated into the RECORD, SELECT, and FN operators of mini-ML. This is a great convenience, not only for the optimizer but also for the Standard ML of New Jersey link-loader and runtime system.

```
signature STACK =
   sig   type 'a stack
         exception Empty
         val empty : 'a stack
         val push: 'a * 'a stack -> 'a stack
         val top : 'a stack -> 'a
         val pop : 'a stack -> 'a stack
   end

structure Stack1 =
struct type 'a stack = 'a list
       exception Empty
       fun push(a,s) = a::s
       fun top(a::rest) = a | top(nil) = raise Empty
       fun pop(a::rest) = rest | pop(nil) = raise Empty
       val empty = nil
end

structure Stack2 : STACK =
struct datatype 'a stack = empty | push of 'a * 'a stack
       val extra = print "hello"
       exception Empty = Match
       fun top(push(a,rest)) = a
       fun pop(push(a,rest)) = rest
end

structure User =
struct val j = Stack1.push(7,Stack1.empty)
end

functor F(S : STACK) = struct  . . . S.empty . . . end

structure T = F(Stack2)
```

Figure 4.3. An example of ML modules.