CS410/510
Compiling Advanced Languages
Spr2000 Lecture Notes
Core Language (2)

Another Record Example

This is not expressible in ML:

```
let f x = x.a
in f a = 1;
   f b = 2; a = 3
```

but we can encode it:

```
Let(102,
    Abs(120,
        Select(97,Var 120)),
    Let(999,
        App(Var 102,Record[(97,Int 1)]),
        App(Var 102,Record[(98,Int 2);
                           (97,Int 1)])))
```


Supporting Sums

Although existing FLs typically require that all constructors of a sum be declared simultaneously, we will be more flexible in our core language, for two reasons:

• It can be handy to allow the user to extend the set of constructors for a sum type "on the fly" during program linking or execution. In fact, **exception** values are often implemented this way (under the user's feet).

• We may want to allow a single constructor to participate in multiple sums (provided it always has the same type).

The key operations are creating labeled constructions and performing dispatch based on the label.

With an eye on typing considerations (to be considered later), we make the following additions to our language:

```
type lexp =
...
| Labeled of label * lexp
| Switch of lexp *
       (label * var * lexp) list * lexp option
```

`Labeled`(l,e) evaluates e to a value v and then produced a labeled construction (l, v).

`Switch`(e,$arms$,d) evalutes e to a labeled construction (l, v). If there is a list member $(l, x, e') \in arms$, the overall `Switch` evaluates to $e'[v/x]$. If not, it evalutes to default expression d.

Example

```
type yyy = M of int | N of int | P of int

(fn M x -> 0
  | N x -> x
  | P x -> x+1)(P 88)
```

might become:

```
App(Abs(42,Switch(Var 42,
                  [(0,101,Int 0);
                   (1,102,Var 102);
                   (2,103,App(Prim Iadd,
                              Record[(0,Var 103);
                                     (1,Int 1)]))],
                  None)),
    Labeled(2,Int 88))
```

Representation of Sum Types

The obvious representation of a summand (the result of applying a constructor) is as a pair containing the label followed by the value.

It is desirable, but not essential, for label values to be (or map to) a compact set of small integers, so that case dispatch can be implemented using a jump table.

As usual, we have a choice about whether to box the value and whether to box the summand itself.

There are several competing considerations:

• Boxing the value may be a good idea if it is large, or if source language semantics permit it to be manipulated separately.

– In SML, each constructor has arity 0 or 1; constructors that appear to take more than one argument actually take a (single) tuple. Thus we can legally write either:

```
fn Check(r,s) => floor(r * 50)
```

or

```
fn Check p => floor((#1 p) * 50)
```

– In CAML, `Check` is treated as a binary constructor, and only the first form of match is allowed.

• Boxing the entire summand in the heap gives an easy way to cope with the fact that different summands of the same type have different sizes.

• Summands of recursive types ultimately **must** be boxed, since they can have arbitrarily large sizes.

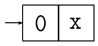
• As with records, avoiding heap boxed values can be a big win. But to support stack or register allocation of summands, must allocate for the worst case (maximum size) summand (just as for C unions).

• If we know the entire set of possible summands, we can often find a more efficient representation.

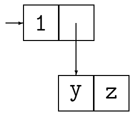
Example: Trees

```
type tree = Leaf of int
          | Tree of tree * tree
```

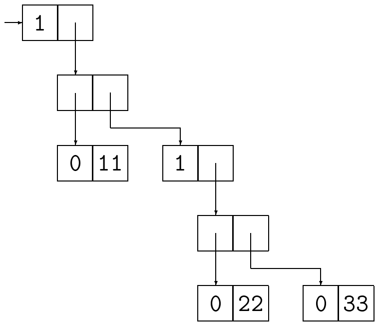
`Leaf(x)`



`Tree(y,z)`



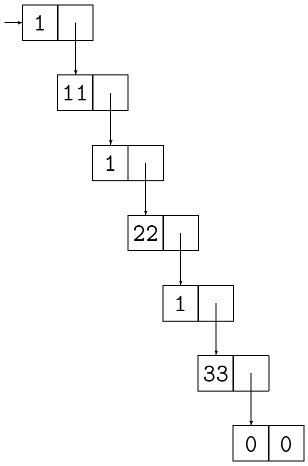
`Tree(Leaf(11),Tree(Leaf(22),Leaf(33)))`



Look how badly this representation does for lists!

```
type 'a list = [] | ::  of 'a * 'a list
```

```
11::22::33::[]
```



It uses twice as many records as in the "classic" representation. The latter relies on the fact that `[]` has a special representation—the integer `0`— that can't be confused with a pointer. Thus, given a value of type `list`, it is either a `0` (meaning `[]`) or a pointer to a `cons` pair.

Compact Representations

We can generalize this reasoning to get more compact representations for lots of concrete constructors. Eventually we'll recover the "classic" list representation as a special case.

All these compaction techniques rely on being able to distinguish a class of non-pointer values easily at runtime.

Assumption 1. We can distinguish **small integers** (e.g. < 64K) from pointers, because it's easy to arrange that all legitimate pointers have fairly large addresses.

Consequence 1. All "constant" (i.e., non-value-carrying) constructors can be represented as small integers.

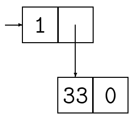
This is great for **enumerated types** like

`type color = Red | Blue | Green | Yellow`

whose values can be represented by the integers `0,1,2,3`, with no need for boxing.

It also works for `[]`:

`33::[]`

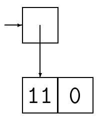


Consequence 2. If a sum type has only one value-carrying constructor, its record doesn't need a tag.

`type xxx = M | N | P of int list`

We represent `M` by `0`, `N` by `1`, and `P` by a pointer to a singleton record containing a list value. E.g.,

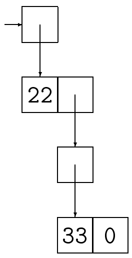
`P(11::[])`



Note that we can't point to the list directly, because then we couldn't distinguish `M` from `P([])` (both would be represented by `0`).

This trick also improves lists a little bit:

`22::33::[]`

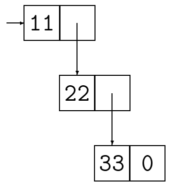


But we can do better...

Consequence 3. If a sum type has only one value-carrying constructor, and that constructor's value is **always-boxed**, we can omit the extra indirection and point directly to the boxed value.

This is true for `list`: the `cons` (`::`) constructor always carries a tuple value. So we finally get the standard LISP-style list representation:

`[11,22,33]`



There's a problem with this representation in SML: types may be **abstract**, so that the compiler cannot always know whether the arguments to a constructor are always-boxed or not. Worse yet, in legal separately compiled programs, the compiler may reach conflicting decisions about the always-boxed property! But getting a good representation for lists is so important that SML/NJ historically hasn't worried about this much.

We can make no end of stronger assumptions. For example:

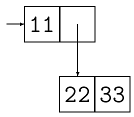
Assumption 2. We can distinguish **all** integers from pointers. (This is often arranged for the benefit of the garbage collector, though it has unpleasant consequences.)

Consequence. If a sum type has two value-carrying constructors, one of which carries integers and the other an always-boxed value, we don't need tags to distinguish the values.

This can be great:

`sum type Tree = Leaf of int | Tree of Tree * Tree`

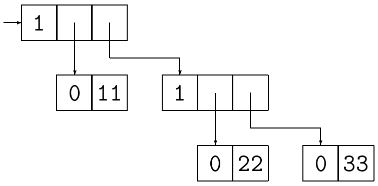
`Tree(Leaf(11),Tree(Leaf(22),Leaf(33)))`



Records may have additional information for garbage-collection purposes,e.g. length, that could enable further short-cuts. But in practice, we try to avoid undue dependence on the details of the runtime system.

Alternative Semantics

If (as in CAML) we think of constructors like `Tree` as taking two arguments rather than one tuple argument, we can achieve the following relatively compact representation (without Assumption 2):



Note that the constructor tag can always be extracted from the first word; other fields mustn't be fetched until the tag has been inspected and the total tuple size is known.

This won't work for SML, unless we permit values to be pointers into the middle of records, which is awkward for the garbage collector, or we're prepared to do a lot of copying.

Identity and Mutability

FL's commonly support two different notions of equality on structured values like products and sums.

• **Pointer equality** (Scheme's `Eq?`) means that the values are occupy the same memory location.

• **Structural equality** (Scheme's `Equal?`) means that the values have equal contents, where numeric fields are compared using the standard numeric comparisons, and pointer fields are compared recursively using structural equality.

`(Eq? a b)` implies `(Equal? a b)`, but not vice-versa.

Obviously, `Equal?` is more expensive to compute than `Eq?`. In some cases, such as comparing functions, it is impossibly expensive, and is not permitted, either statically (as in SML) or dynamically (as in CAML).

In SML, there is one identity operator (`=`) which behaves like `Equal?` on immutable records and like `Eq?` on mutable ones. In CAML, there are two operators: `=` behaves like `Equal?`; `==` behaves like `Eq?` on mutable records, like `Equal?` on integers and characters, and randomly on immmutable records. Thus, in both systems it is impossible to determine (reliably) whether two immutable records occupy the same address. This permits an implementation to introduce sharing under the user's feet.

Since our core language so far makes no guarantees about boxing, except in the case of mutable records, `Eq?` makes no sense in general. On the other hand, `Equal?` is painful to implement. We'll need to fix this fairly soon.

Exceptions

In imperative programming, it is often handy to perform a "non-local exit" from a function, passing control, and perhaps a value, to some dynamically enclosing function activation without performing the normal return sequence.

ML, Ada, C++, Java, etc. support this mechanism in the form of **exceptions**, which are **raised** at the point of escape (either by the user program or by an underlying hardware condition such as division by zero), and can be **handled** by user-specified code in a (dynamically) enclosing activation. An unhandled exception terminates execution.

Example: short-circuit multiplication

```
exception Zero
let prodlist l =
  let rec p l =
    match l with
      [] -> 1
    | h::t -> if h = 0 then raise Zero

    | (h::t) -> h * (p t) in
  try p l
  with Zero -> 0
```

(In SML, we could make the exception definition local to `prodlist`, but not in CAML.)

Exceptions (cont.)

Exception values all belong to type `exn`, which is like a sum type whose constructors are specified piece-meal by `exception` declarations. `try...with` performs pattern-matching just like a `fun`.

There's no easy way to express exception handling in terms of function calls, so we add

```
type lexp =
... Raise of lexp | Handle of lexp * var * lexp ...
```

to the core language. `Raise`(v) raises the exception with value v. `Handle`(e,x,h attempts to evalute e; if an exception with value v is raised during the evaluation, `Handle` returns the value of $h[v/x]$; otherwise the ordinary value of e is returned.

Scheme (and SML of New Jersey) support a more general mechanism known as **call with current continuation (callcc)**. A continuation is a description of the future of a computation; concretely, it consists of the program counter and the stack contents at a particular moment in execution. We'll say more later.