CS410/510
Compiling Advanced Languages
Spr2000 Lecture Notes
Core Language

Core λ-language

We'll develop our core intermediate language as a set of CAML datatypes. Initially, the core language will be similar to the calculus we've used so far:

```
type var (* = int *)
```

```
type primop = Iadd | Isub | Imul | ...
            | Ieql | Ige | Igt | ...
            | ...
```

```
type lexp
= Var of var
| Abs of var * lexp
| App of lexp * lexp
| Int of int
| Prim of primop
| ...
```

We assume that variables have been converted from names to unique numbers (but not to deBruijn).

We need to fix an **evaluation order** for this language. Source language may demand particular order. E.g., if translating from Standard ML, we would fix call-by-value, left-to-right evaluation. CAML and Scheme demand call-by-value, but no particular argument ordering; in practice, CAML uses right-to-left ordering.

let bindings

It gets boring to keep writing out long λ-expressions. (We can use the $\equiv$ notation to define abbreviations for expressions, but this is merely a meta-notation for a form of macro-expansion.)

We need a way to give an expression a **name** by which it can be referenced. To be most useful, each name **binding** should have a syntactically limited **scope**. A standard syntax is:

```
let <var> = <exp> in <exp>
```

(Some languages allow

```
<exp> where <var> = <exp>.)
```

This construction is very easy to translate to pure λ-calculus (if we want to):

```
let v = B in E
```

becomes `((λv.E) B)`

which has exactly the same operational semantics.

Example:

```
let a = + b 8 in * a a
```

becomes `((λa.(* a a))(+ b 8))`

We'll see later that these forms are often **not** identical for type-checking purposes, however.

letrec bindings

Recursive bindings need special treatment, since the simple translation for `let` won't work (why not?).

We introduce

$$
\begin{array}{ll}
\texttt{letrec} & \texttt{v}_1 \texttt{ = } \texttt{E}_1 \\
 & \texttt{v}_2 \texttt{ = } \texttt{E}_2 \\
 & \ldots \\
 & \texttt{v}_n \texttt{ = } \texttt{E}_n \\
\texttt{in} & \texttt{E}
\end{array}
$$

Example

```
letrec f = λx.g(x+1)
       g = λy.f(y-1)
in     f 0
```

Here the $\texttt{v}_i$ can be referenced within the $\texttt{E}_j$ as well as within `E`. The $\texttt{E}_j$ must all be λ-abstractions. Note that provision for multiple simultaneous definitions is essential to allow mutual recursion.

We could translate `letrec` to pure λ-calculus using the Y-combinator, but since there are more efficient ways to implemente recursion, we'll add it explicitly as a new operator in our core calculus:

```
type lexp =
... Fix of (var * lexp) list * lexp ...
```

If we allow the `lexp` list to contain arbitrary expressions, we can use FIX to represent ordinary `let`-bindings too.

Aside: Real Source Isn't Abstract

By assuming that we're given "source" λ-language expresssions, we've already elided a number of details that a real front-end must take care of. We give some examples from CAML here.

• The concrete syntax of CAML allows for **infix** constants. (In SML, fixity of operators is programmer-controlled, a parsing nightmare).

• CAML allows functions to be declared with the formal parameter on the l.h.s. This is just shorthand for a `fun` expression on the right.E.g.,

```
let rec f x = x + 1 ≡ let rec f = fun x -> x + 1
```

• ML allows several identifiers to be declared simultaneously, e.g.:

```
let a = 2 in let a = 3 and b = a in a + b
```

This can be translated to a nested pair of simple `let`s (maybe after renaming):

```
let a = 2 in
    let a' = 3 in let b = a in a' + b
end
```

Every different source language will introduce issues of this kind, though most FL's are strikingly similar. As we proceed to more sophisticated constructs, we'll worry even less about such problems.

Records and Tuples

A **record** is a cartesian product of labeled values called **fields**. Examples: `record` in Pascal, `struct` in C.

Each record has a fixed number of fields, determined when the record is created.

In source languages, field labels are typically alphanumeric identifiers; for now, we'll assume that they are just numbers (as with variables). Within a given record, labels must be unique.

```
type label (* = int *)
```

Note that label numbers are **not**, in general, positions or offsets within the record. Thus, accessing a field by label requires some kind of mapping operation to determine the field's actual offset. We may or may not know enough about the record to perform this mapping at compile-time.

We also permit the possibility of using the **same** label in **multiple** record types, coexisting with different labels in each. But the associated value must have the **same** type in each case.

FL's commonly provide extra support for **tuples**, which are a special kind of record whose fields are accessed by position rather than by label. By convention, positions are numbered from 0.

Scheme has only two-element tuples (**pairs**); many FL's have tuples of arbitrary **fixed** size, though pairs tend to be most commonly used. We will find uses for tuples of size 0 and 1, as well as sizes ≥ 2.

Records (continued)

In CAML, record types must be declared before corresponding values can be created. Labels are uniquely associated with particular record types, so cannot be reused. Tuples are a separate built-in type, not requiring a declaration before use.

```
type r = {a:int; b:int}
let x = {a=3;b=5}
let y = x.a
```

In SML, records can be constructed without prior declaration of a record type. Labels can be reused in different record types. An n-tuple is actually **defined** to be a record whose labels are the integers $\{0, 1, \ldots, n-1\}$. (We will use this convention when we want internal tuples in examples, although it violates the type rules.)

We'll support records by extending the core language with

```
type lexp =
... Record of (label * lexp) list
| Select of label * lexp ...
```

Example: `{a = 2; b = 3}.a`

might translate to

```
Select(97, Record[(97,Int 2);(98,Int 3)])
```

Now that we have tuples, we'll assume that all the primops are **uncurried**, i.e., when they require multiple arguments they take them as a tuple.

Mutable Records

Record fields may or may not be **mutable**, i.e., capable of being updated, depending on the language. This is an important question for implementation!

In Scheme all fields are potentially mutable. CAML allows individual fields in record types to be declared **mutable**. Standard ML restricts mutability to special one-tuples called **refs**, which are essentially pointers; `ref`s have a distinct type and special user-level operators. (The CAML library implements refs as a particular record type.)

For now, we'll use the standard record creation and selection expressions to handle mutable records, though we might want special versions later on.

To support mutating record contents we can add a primop parameterized by which field to update:

```
type primop = ... Update of label ...
```

which expects as argument a pair (2-tuple) of expressions:
• the record to be updated
• the value to write.

Example

```
let a = ref 11 in a := 17 end
```

might translate to

```
  App(Abs(97,App(Prim(Update 0),Record[(0,Var 97);
                                        (1,Int 17)])),
      Record[(0,Int 11)])
```


Representing Records

For now, we will be deliberately vague about how records are actually **represented**.

Ultimately, we want to map everything to fixed length records where each label corresponds to a known offset. If possible, we'd like to know offsets at compile time.

This mapping depends on both field **order** and field **size**.

Note that we don't support record **extension** as a primitive, since this would require variable length records or lots of copying.

The size issue can be simplified by adopting a **uniform** data representation model, in which field occupies the same number of bytes – typically enough to hold a single pointer (e.g., one word).

– Values which naturally fit within that size are stored there directly.

– If the value too large to fit within that size, it is stored **in memory** and its **address** is placed in the record. Such values are said to be **boxed**.

Polymorphism

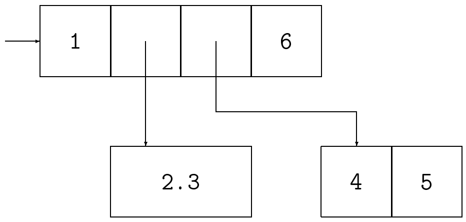
This approach is particularly good for implementing **polymorphic** code that programs that can process tuples without "knowing" the types of the values within them, e.g.,

```
let swap p = (snd p,fst p)
```

We might say that large values (larger than one word) are manipulated **by reference** rather than **by value**.

Thus, every n-field record occupies n contiguous words (plus the storage for any values pointed to), so fetching fields is easy.

Example: `(1, 2.3, (4, 5), 6)`



Boxing is expensive

Note that tuples of size ≥ 2 are themselves always too big to fit in a word and so must be stored boxed in memory under the uniform representation model.

This seems very reasonable for really large values (e.g., 10-tuples); after all, we would normally manipulate such a beast by value in a language like C.

It's not so nice for a pair of integers, though, or for a solitary double-precision float, both of which must also be stored boxed.

This approach leads to **big** costs in loading and storing boxed data. Also, since the lifetime of values is often hard to predict, most boxed data must be stored in the **heap**, leading to further overheads for allocation and garbage collection.

So we'll do our best to avoid boxing values, whether for polymorphism or any other reason. In particular, when possible we'll try to represent records in collections of machine **registers**.

N.B. Since all 0-tuples are essentially the same, they can be represented by a fixed constant pointer value (e.g., 0) (when boxed) or by nothing at all (when not boxed).

A 1-tuple does not need a box of its own (though its contents might), but if it's mutable (i.e., like an ML `ref` cell) we box it anyway since copying it would be a mistake! (This is the same reason that one manipulates updateable parameters by reference in imperative languages.)

Arrays

Arrays are much like tuples, but have some key differences:

• Programs can access arbitrary elements using an index expression provided at runtime.

• In most modern languages, the **size** of an array is fixed dynamically at array creation time (not at compile time), and hence it must be kept at runtime, typically in a **header**.

Arrays might be mutable or immutable. (SML calls immutable arrays **vectors**.

Safety-oriented languages like Java and ML encourage or require **bounds checking** on all vector accesses. Bounds checking is normally performed at runtime, although checking at compile time is sometimes possible.

If polymorphic access to vectors is required, they are usually stored one word per element, boxing as necessary, just like tuples. This is very unpleasant for, e.g,. vectors of double-precision reals!

If the vector contains objects of uniform size, a better alternative is to store elements packed consecutively (not boxed). Polymorphic access is still possible if the element size is stored in the vector header.

Note that a **string** is essentially an array of bytes (or unicode integers) with a special constructor for literal constants.

Sum Types

ML, Hope, Haskell, Miranda, etc. all support user-defined data types in form of **disjoint unions** or **sums** of existing types, a.k.a., sums of products, structured types, concrete types, algebraic types, etc.

Examples (CAML):

```
type money    = Coin of int | Bill of int
              | Check of float * string
type tree     = Leaf of int
              | Tree of tree * tree
type color    = Red | Blue | Green | Yellow
type register = Reg of int
```

These declarations create new types **and** introduce a set of **constructors** for each type (e.g., `Coin, Bill, Check`, etc.) A value of the type is created by applying a constructor to the required arguments (if any), e.g,

```
Tree(Leaf 1,Tree(Leaf 2,Leaf 3))
```

Sum types are roughly like Pascal variant record types or C unions, each variant being a constant or product. But unlike these languages, we can **always** determine which variant a particular value belongs to.

Pattern Matching

The contents of a sum-type value are accessed via **pattern matching**, e.g. in CAML:

```
let evaluate = function
Coin c -> c
 | Bill b -> b * 100
 | Check(r,"Joe Deadbeat") -> truncate(r *.  50.0)
 | Check(r,_) -> truncate(r *.  100.0)
```

ML also uses pattern matching as the fundamental mechanism for extracting members of tuples. E.g., we could define

```
let fst = fun (a,b) -> a
```

We can also write pattern matches in `match` constructions:

```
match e
with Coin c -> c
| Bill b -> b * 100
| ...
```


Booleans

We can also view truth values as a (built-in) special case of datatypes,

```
type bool = false | true
```

(Although case conventions prevent us from writing this in CAML source)

A conditional statement is just syntactic sugar for a `match` statement:

```
if e then e1 else e2
```

becomes

```
match e
of true => e1
| false => e2
```

Booleans are "built-in" in the sense that primops returning a truth value must know the (internal) representation of the `true` and `false` constructors; naturally, we hope to arrange things so that these are just `1` and `0` in the usual way.

Lists

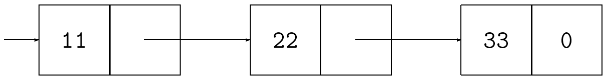
This fundamental datatype can be described as a **recursive** sum type, e.g.,

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

Here `'a` is a **type variable** indicating that a list can be made up of elements of any (one) type. The `cons` operator is written using an infix `::` in ML.

Alternatively, we could construct lists as in Scheme, using 2-tuples and a special constant (say, the integer 0) for the empty list. This leads to the classic representation, which is very flexible, though not very compact. E.g.,

```
[11 22 33]       (11, (22, (33, 0)))
```



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 datatype "on the fly" during program 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'), the overall `Switch` evaluates to $e'[v/x]$. If not, it evalutes to the 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))
```