Compiling FL Programs

Main concepts of this unit:

- Source language
  - data
  - functions
- Target language
  - data
  - functions
- Order of evaluation
  - call-by-need
  - call-by-value
- Compilation
  - abstract
  - low level

The problem

Consider the program:

```
loop = loop
snd (_,y) = y
```

and evaluate the expression

```
snd (loop,0)
```

Applying the first rule, "makes no progress."
If only the first rule is applied, no result is ever found.

Applying the second rule, gives the result.

A *compiler* must generate code that applies the right rule to the right redex so that if an expression has a value, that value is eventually produced.

The generated code represents expressions/graphs as linked structures. It encodes procedures that traverse these graphs and replace subgraphs in a graph until no more replacements are possible.

Source language

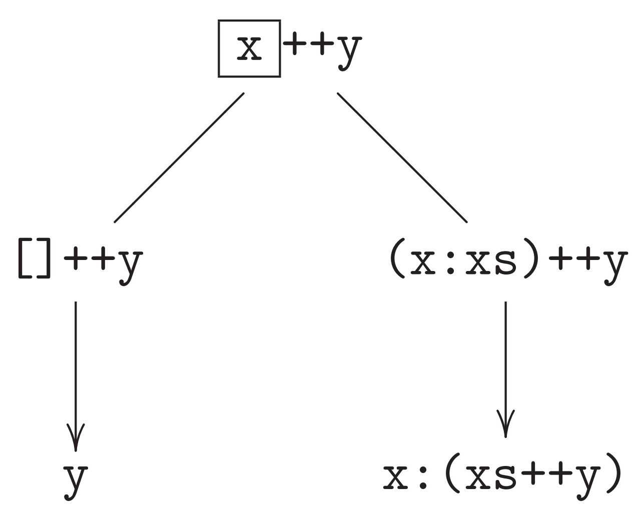
The source language being compiled consists of a definitional tree of each operation and the arity of each symbol, in particular the constructors

Example in Curry:

```
[]++y = y
(x:xs)++y = x:(xs++y)
```

Corresponding source language:

arity of `[]` $=0$
arity of `:` $=2$
tree of `++` $=$



Target language

- The target language consists of two functions: **H** and **N**.
- Each function takes and returns an expression.
- These expressions are made up by *all* the symbols of the source language.
- Each function performs case analysis of its argument and selection of subarguments. Hence, they can be conveniently defined by rules with pattern matching. Hence, the target language is a rewrite system!
- The evaluation in the target system is eager/by-value.
- The rules of the target system are tried in textual order, the first one that is applicable is the only one being applied.
- Hence, the control (execution) is *simple.*

Function **H** (1)

Let S and T denote the source and target systems.

Function **H** takes an expression e of S and returns a head constructor form of e (a constructor application), or aborts if this form doesn't exist.

The rules of **H** are generated piecemeal for each operation f of S by a post-order traversal, let's call it *compile*, of a definitional tree of f. We define *compile* by examples.

Let N be a *branch* node with pattern π, some inductive variable, and a few children. First *compile* each child (post-order traversal). Then produce the rule:

$$\mathbf{H}(\pi) = \mathbf{H}(\pi')$$

where π' is like π with the inductive variable wrapped by **H**.

Example using the root of the tree of `++`:

$$\mathbf{H}(\texttt{x++y}) = \mathbf{H}(\mathbf{H}(\texttt{x})\texttt{++y})$$

Rationale: x is needed and matches a function application or a textually preceeding rule would have been fired.

Function **H** (2)

Let N be a *leaf* node with rule $l \to r$. We distinguish 3 exhaustive and mutually exclusive cases for r.

1. r is a constructor application. Produce the rule:

$$\mathbf{H}(l) = r$$

2. r is a function application. Produce the rule:

$$\mathbf{H}(l) = \mathbf{H}(r)$$

3. r is a variable, say x. Produce the rules:

$$\mathbf{H}(l') = c_i(x_1, \ldots x_k)$$

where l' is like l with x replaced by $c_i(x_1, \ldots x_k)$ for every constructor symbol c_i of arity k. Finally, produce:

$$\mathbf{H}(l) = \mathbf{H}(x)$$

Example, compile the left leaf of the tree of `++`:

$$\mathbf{H}(\texttt{[]++[]}) = \texttt{[]}$$
$$\mathbf{H}(\texttt{[]++(y:ys)}) = \texttt{y:ys}$$
$$\mathbf{H}(\texttt{[]++y}) = \mathbf{H}(\texttt{y})$$

Note, the right-hand side of the rule of `++` is a variable. In the 3rd rule of **H**, y matches a function application.

Exercise 6.A Compile the right leaft of `++`.

Exercise 6.B Compile operation `take` defined at page 5 of the "Strategies" unit.

Function **H** (3)

Let N be an *exempt* node with pattern π. *compile* produces:

$$\mathbf{H}(\pi) = \text{abort}$$

where "abort" is a directive to abort the computation since the expression being evaluated has no value.

Optimization.

An effective *optimization* is often available. Consider the previously discussed rule:

$$\mathbf{H}(\texttt{x++y}) = \mathbf{H}(\mathbf{H}(\texttt{x})\texttt{++y})$$

We know that the recursive outermost call to **H** will always match ++ at the root. We can specialize this call and avoid first constructing and later matching the root:

$$\mathbf{H}(\texttt{x++y}) = \mathbf{H}_{\text{++}}(\mathbf{H}(\texttt{x}), \texttt{y})$$

Non-determinism.

This compilation scheme is for deterministic functions. Various approaches to non-determinism, e.g., *backtracking* could be integrated

Higher order.

Not discussed at this time. Maybe later.

Function **N**

Function **N** takes an expression e of S and returns the value of e (in S) or it "aborts" if e has no value.

It invokes function **H** that takes an expression e of S and evaluates it to a head constructor form, or aborts if it doesn't exist.

Operation **N** is defined by one rule for each symbol of S. In the following *metarules*, c stands for a constructor of S of arity m, f stands for an operation of S of arity n, and x_i is a fresh variable for every i.

$$\mathbf{N}(c(x_1, \ldots x_m)) = c(\mathbf{N}(x_1), \ldots \mathbf{N}(x_m))$$
$$\mathbf{N}(f(x_1, \ldots x_n)) = \mathbf{N}(\mathbf{H}(f(x_1, \ldots x_n)))$$

Examples:

$$\mathbf{N}(\texttt{[]}) = \texttt{[]}$$
$$\mathbf{N}(\texttt{x:xs}) = \mathbf{N}(\texttt{x})\texttt{:}\mathbf{N}(\texttt{xs})$$
$$\mathbf{N}(\texttt{x++y}) = \mathbf{N}(\mathbf{H}(\texttt{x++y}))$$

The 3rd rule can be optimized as discussed earlier.

After execution of the 3rd rule, the recursive call executes either the 1st or the 2nd.

Low level implementation

The target system can be implemented relatively easily in a low-level language such as *C*.

The graphs abstracting the expressions have nodes and arcs. A node is a *struct* containing a label/symbol and pointers to the node successors.

Functions **H** and **N** are ordinary *C* functions.

Pattern matching is implemented by case analysis through a traversal of (the top portion of) the argument.

A working system must accommodate built-in types, like the integers, and provide some library functions that cannot be coded in Curry.

Exercise 9. Sketch the case analysis required to dispatch the rule of **H** in the target system for an argument rooted by ++. Hint: start with writing all the rules of **H**.