06-02552 *Princ. of Progr. Languages (and "Extended")*
Spring Semester 2019-20

The University of Birmingham
School of Computer Science

Handout 5: Functional languages as Typed Lambda Calculi

Haskell has adopted many of the convenient syntactic structures that have become popular in functional programming. In this Report, the meaning of such syntactic sugar is given by translation into simpler constructs. If these translations are applied exhaustively, the result is a program written in a small subset of Haskell that we call the Haskell kernel.

Although the kernel is not formally specified, it is essentially a slightly sugared variant of the lambda calculus with a straightforward denotational semantics. The translation of each syntactic structure into the kernel is given as the syntax is introduced. This modular design facilitates reasoning about Haskell programs and provides useful guidelines for implementors of the language.

Haskell Report

Functional languages as "syntactic sugar" for typed lambda calculus

1. The syntax of the lambda calculus is rather terse, because it was designed as a mathematical system. It also lacks various features that we find important in programming, such as the ability to define and name functions. One might consider extending the lambda calculus in various ways to address these problems. There is another way. Because already has the power to express functions but just lacks convenient notation, we can define new notations as syntactic forms for concepts that are expressible in the basic calculus. Christopher Strachey introduced the term "syntactic sugar" to describe such notations. The idea is that, if our core calculus has all the expressive power but lacks convenient notation, we can sweeten it up by adding new notation and specify how the new notation corresponds to the notations in our core calculus. The Haskell report followed this approach in defining the meaning of its constructs in terms of a "Haskell kernel," which is close to the typed lambda calculus.

2. Omitting types. First of all, we decide that we will omit type declarations of variables whenever convenient. Thus we write $\lambda x{:}T.\ M$ as simply as $\lambda x.\ M$, assuming that the reader or the compiler can "guess" what type T we intend. It may not always be easy to guess what type is intended. So, until you become an expert in types, it is better to declare the types.
Haskell does not actually provide the syntax for us to declare the types of parameters. However, it allows us to declare the types of defined functions separately (as what we call *type signatures*). In ML and Java, on the other hand, we can declare the types of parameters just as in typed lambda claculus.

3. Notation for primitive functions. In typed lambda calculus all functions are unary and written using prefix notation. We can treat infix and mixfix notations as syntactic sugar for the basic notation:

- All binary arithmetic operations ("+", "-", "*", "/") can be used in infix notation and they mean the corresponding prefix equivalents. For example, $x + 1$ is sugared notation for the term $+\ x\ 1$ of the base calculus.
- All comparison operations ("$=$", "$\neq$", "$<$", "$\leq$", "$>$", "$\geq$") can be used in infix notation.
- The applications of "if" can be written in the mixfix notation **if** p **then** x **else** y. This is sugared notation for $(\text{if}\ p\ x\ y)$.

4. Local declarations. In Haskell and other functional languages, one can define a local name for the value of an expression using **let** or **where** declaration:

$$\begin{array}{ll} \textbf{let } x = M \qquad\qquad & N \\ \textbf{in } N & \quad \textbf{where } x = M \end{array}$$

They are equivalent notations.[1] So, we can just discuss one of them, say **let**. The scope of the local name x is the term N (which is called the "body" of **let**).

These forms can be understood as syntactic sugar for the the basic typed lambda calculus expression $(\lambda x.\, N)\, M$.

$$\textbf{let } x = M \textbf{ in } N \quad \rightsquigarrow \quad (\lambda x.\, N)\, M$$

For example:

$$\begin{array}{lll} \textbf{let } x = \text{square } z & \rightsquigarrow & (\lambda x.\, x + y)\ (\text{square } z) \\ \textbf{in } x + y & & \end{array}$$

Thinking about why this translation works can give you insight into both **let** as well as λ. Note that in **let** $x = M$ **in** N, the name x occurs only N. So, we can think of $N = N(x)$ as a "function" of x. Setting x to M then means that we are instantiating $N(x)$ for the case $x = M$. In lambda calculus notation the "function" $N(x)$ is expressed as $\lambda x.\, N$ and instantiating it for $x = M$ is formalised as applying the function to M.

Any kind of value can be given a local names using **let**. That includes data structures such as pairs, lists, and trees and functions. Even larger program units like modules, objects, classes etc. can be named using **let**. So, **let** is a one-size-fits-all naming mechanism. For example, here is a declaration of a local function:

```
let max = λx. λy. if x > y then x else y
in max(a, max(b, c))
```

5. Local variables and scope. Note that x is a *local variable* (also called a *bound variable*) in the term **let** $x = M$ **in** N. Its *scope* is the term N. Whenever we use a symbol as a local variable, we have to designate its scope. If there occurrences of the same symbol outside the scope, there are *nonlocal occurrences* (also called *free occurrences*) of that symbol.

In the term **let** $x = M$ **in** N, if there are any occurrences of x in the term M, they would be *nonlocal* occurrences. For example, in the term

$$\begin{array}{l} \textbf{let } x = \text{square } x \\ \textbf{in } x + y \end{array}$$

the occurrence of x in square x is nonlocal. On the other hand, the occurrence of x in $x + y$ is local by the **let**. So, the term means the same as $(\text{square } x) + y$.

6. Multiple declarations. In the same way, we can devise syntactic sugar to allow the local declaration of multiple names:

$$\begin{array}{lll} \textbf{let } x_1 = M_1; & & \\ \quad x_2 = M_2; & & \\ \quad \ldots; & \rightsquigarrow & (\lambda(x_1, \ldots, x_n).\, N)\ (M_1, \ldots, M_n) \\ \quad x_n = M_n & & \\ \textbf{in } N & & \end{array}$$

We see that multiple-declaration let's are understood as applications of multiple-argument functions. Equivalently, they are applications of curried multiple-argument functions:

$$(\lambda x_1.\, \ldots \lambda x_n.\, N)\ M_1\ \ldots\ M_n$$

Once again, note that we are regarding $N(x_1, \ldots, x_n)$ as a function of $x_1, \ldots, x_n$ and treating $M_1, \ldots, M_n$ as the instantiations of $x_1, \ldots, x_n$, i.e., as the arguments to the function $N(x_1, \ldots, x_n)$.

[1] In real Haskell, however, the two notations are used with different scope rules.

7. Function declarations. We can extend **let** declarations with function application notation for convenience in defining functions:

$$\begin{array}{lcl} \textbf{let}\ f\ x_1\ \ldots\ x_n = M & \rightsquigarrow & \textbf{let}\ f = \lambda x_1.\ \ldots \lambda x_n.\, M \\ \textbf{in}\ N & & \textbf{in}\ N \end{array}$$

Note that the right hand side in turn means

$$(\lambda f.\, N)\ (\lambda x_1.\ \ldots \lambda x_n.\, M)$$

An example of the function declaration let is:

```
let max x y = if x > y then x else y
in max(a, max(b, c))
```

This can be viewed as meaning the same as:

```
let max = λx. λy. if x > y then x else y
in max(a, max(b, c))
```

8. Local recursive declarations. In a basic let-declaration **let** $x = M$ **in** N introduced above, the expression M cannot have free occurrences of the very name x that it is defining. In other words, our let-declarations do not allow recursive definitions. To facilitate recursive declarations, we introduce another level of syntactic sugar:

$$\textbf{letrec}\ x = M\ \textbf{in}\ N \quad \rightsquigarrow \quad \textbf{let}\ x = \textbf{fix}\ (\lambda x.\, M)\ \textbf{in}\ N$$

In other words, when we recursively define x via an expression M, what we intend is that x should be the (least defined) fixed point of the function $\lambda x.\, M$. Recall that **fix** $(\lambda x.\, M)$ has the infinite expansion:

$$\textbf{fix}\ (\lambda x.\, M) \longrightarrow (\lambda x.\, M)\ (\textbf{fix}\ (\lambda x.\, M)) \longrightarrow M[x \mapsto (\textbf{fix}\ (\lambda x.\, M))] \longrightarrow \cdots$$

In other words, if we ask for the value of **fix** $(\lambda x.\, M)$ the computation will evaluate M where it assumes that x in turn has the value **fix** $(\lambda x.\, M)$.[2]
For example, the expression

```
letrec fibs = 1:1:(zipWith (+) fibs (tail fibs))
in fibs !! 3
```

becomes:

```
let fibs = fix (λfibs. 1:1:(zipWith (+) fibs (tail fibs)))
in fibs !! 3
```

which evaluates, in two steps, to:

```
((1:1:(zipWith(+) fibs (tail fibs))) !! 3
  [fibs ↦ (fix (λfibs. (1:1:(zipWith (+) fibs (tail fibs)))))])
```

Notice that the occurrence of `fibs` in the body of **`let`** expands out to the right hand side of the definition of `fibs`, where it is assumed in turn that `fibs` denotes the very same expression again. This corresponds to our intuitive understanding of recursive definitions.
(In real Haskell, our **`letrec`** is actually denoted "**`let`**". Haskell does not have a non-recursive let. On other hand, call-by-value functional languages, such as Lisp and ML, have distinct forms of **`let`** and **`letrec`**.)

[2]The notation $M[x \mapsto N]$ means *substitution*, i.e., the term M with all (free) occurrences of x substituted by N.