COMP3161/COMP9161 Supplementary Lecture Notes

MinHs

Gabriele Keller

August 23, 2016

1 MinHs

We start our discussion of programming languages with MinHs, which is a Haskell-like, stripped down version of a strict functional language and very similar to MinML[**?**].

1.1 Concrete Syntax

We give the concrete syntax of MinHs in EBNF. Note that the definition is ambiguous, but the usual precedences and associativities apply for arithmetic and boolean operations (left associative) and application (left associative). The type constructor `->` is right associative, however.

Variables	id	::=	...
Integer values	n	::=	...
Boolean values	n	::=	`true` | `false`
Types	τ	::=	`Bool` | `Int` | τ_1 `->` τ_2 | `(` τ `)`
Infix Operators	$\otimes$	::=	`+` | `*` | `-` | `=` | $<$ | $>$ | $>=$ | $<=$
Exprs	e	::=	id | n | (e) | $e_1 \otimes e_2$ | $e_1\ e_2$
			| `if` e_1 `then` e_2 `else` e_3
			| `letfun` id_1 :: $(\tau_1$ `->` $\tau_2)$ id_2 `=` e

In MinHs, we have two base types: `Int` and `Bool`, and a binary type constructor `->` (right associative), which denotes function types. We also have three new language constructs: if-expressions, which behave in the usual way, and function definition and application.

Function definitions in MinHs are slightly unusual, and closer to a compiler internal representation (which is good for our purpose), so let us have a more detailed look at them. The expression `letfun` f :: $(\tau_1$ `->` $\tau_2)$ x `=`e defines a function named f, which accepts a single argument x of type τ_1 and returns a value of type τ_2. The type of the function (and therefore, implicitly, also the type of the argument variable) has to be provided by the programmer. The scope of both the function name f and the variable x is e. This means that, if we need the same function at different positions in the program, we have to write down the whole function every time. This is, of course, awkward, and we could improve the situation by including let-expressions into the language. Instead of writing

```
(letfun sqr::Int -> Int x = x * x) 5 +   (letfun sqr::Int -> Int x = x * x) 10
```

we could simply write

```
let
   square = (letfun sqr::Int -> Int x = x * x)
in
   square 5 + square 10
```

and reuse the definition of `sqr`. Extending our language in this way would, however, not add any interesting problems, and the objective of MinHs is to keep the language as simple as possible. Similarly, MinHs admits only a single argument for function definitions, which again is an inconvenience for the programmer, but no restriction of the expressiveness of the language. A function, like integer division, which requires two arguments

```
letfun div:: (Int -> Int -> Int) x y =
  if x < y
    then 0
    else div (x-y) y
```

can be defined in MinHs using nested function definitions as follows:

```
letfun div:: (Int -> Int -> Int) x =
   letfun div2::(Int -> Int) y =
       if x < y
        then 0
        else div (x-y) y
```

Since the type constructor `->` is right associative, the type of `div: Int -> Int -> Int` can be interpreted as the type of a function which requires two integer value to return a value as result, or as `Int -> (Int -> Int)`, a function which, after accepting one integer value as argument, returns a new function of type `Int -> Int` as result. For the same reason, if we have the application `div 14 5`, it is the same as `(div 14) 5` (as application is left associative), and can be interpreted either as the application of the binary function `div` to the two arguments `14` and `5`, or alternatively, as application of the function `div` to the number `14`, which results in a new function which divides `14` by any number it is applied to, and the subsequent application of this function to `5`.

1.2 Higher-order Abstract Syntax

We do not list all the rules for mapping the concrete syntax to the abstract syntax here – for the part of the language which is similar to the arithmetic expression language, the rules would be the same. The translation from concrete to abstract syntax consists of the following steps:

1. The boolean constants are represented by the terms `const(True)`, and `const(False)`.
2. Every application of an infix operation is represented by a suitable term notation. E.g., `x + y` becomes `plus (x, y)`.
3. If-expressions are represented in prefix notation as well. E.g., `if True then 1 else 2` becomes `if (const(True), num(1), num(2))`.
4. Application becomes explicit. E.g., `f 4` becomes `apply (f, num(4))`.
5. Function definitions of the form `letfun` f :: $(\tau_1 \rightarrow \tau_2)$ x = e become `fun(`τ_1, τ_2, $f.x.e$`)`. That is, in the body e of the function f, both the parameter variable x, as well as the variable f, the name of the function, may occur freely. It is important the f can occur freely in e, otherwise we wouldn't be able to express recursion, and the language would not be more powerful that the simple arithmetic expression language we discussed before.

The higher-order abstract syntax representation of the function `div` given above is

```
letfun (Int, Int->Int,
        div.x.letfun (Int, Int,
                      div2.y. (if (less (x,y), num (0), apply (div, sub (x,y))))))
```

1.3 Static Semantics

For the static semantics, we will not only check that each variable is in scope, but also make sure that the program is type correct. This means, that every operator, function or language construct is applied to values of the type that it expects. MinHs is statically typed – that is, each subexpression has a unique type which can be determined at compile time. For example, we do not allow subexpressions like the following:

```
if x < y
  then true
  else 5
```

as it may either evaluate to a value of type `bool` or of type `int`, and we can't know at compile time if the values of `x` and `y` are not statically know. *Dynamically typed languages*, however, do allow such expressions, and type errors may only show up at run time in such a language.

There is only one way for a variable to be introduced into a MinHs expression, namely via `letfun`. The programmer has to provide the types for both the function and the argument variable. In contrast to the arithmetic expression language, where we only needed to keep track of all the variable names when formalising the static semantics, we know have to associate a type with each name. Therefor, an environment now is a sorted sequence of variable names bundled with their type of the form $\Gamma = \{x :: Int, f :: Bool \to Int, \ldots\}$. We assume that each variable name is unique.

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash \texttt{var}(x) : \tau}$$

$$\frac{}{\Gamma \vdash \texttt{num}(n) : \texttt{int}}$$

$$\frac{}{\Gamma \vdash \texttt{const}(b) : \texttt{bool}} \quad b \in \{\texttt{true}, \texttt{false}\}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash \texttt{plus}(e_1, e_2) : \texttt{int}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash \texttt{less}(e_1, e_2) : \texttt{bool}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \texttt{if}(e_1, e_2, e_3) : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash \texttt{apply}(e_1, e_2) : \tau_2}$$

$$\frac{\Gamma \cup \{f : \tau_1 \to \tau_2\} \cup \{x : \tau_1\} \vdash e : \tau_2}{\Gamma \vdash \texttt{letfun}(\tau_1, \tau_2, f.x.e) : \tau_1 \to \tau_2}$$

$$\frac{}{\texttt{apply(letfun}(\tau_1, \tau_2, f.x.e_1), v) \mapsto \{\texttt{letfun}(\tau_1, \tau_2, f.x.e)/f\}\{v/x\}e_1}$$

$$\frac{e_1 \mapsto e_1'}{\texttt{apply}(e_1, e_2) \mapsto \texttt{apply}(e_1', e_2)}$$

$$\frac{e_2 \mapsto e_2'}{\texttt{apply(letfun}(\ldots), e_2) \mapsto \texttt{apply(letfun}(\ldots), e_2')}$$

1.4 Dynamic Semantics