The Definition of the Haskellito Programming Language

Robert W. McGrail
Bard College
Annandale-on-Hudson, NY 12504

September 7, 2013

1 Haskellito Grammar

The following grammar describes the fragment of Haskell that each of you will implement. Notice that this grammar is ambiguous for several reasons. Hence it will be imperative for you to use the features of Happy to set operator and language element precedence.

```
Exp ::=
        Boolean
      | Integer
      | let Var = Exp in Exp
      | if Exp then Exp else Exp
      | (\Var -> Exp)::Type -> Type
      | Exp Exp
      | Exp && Exp
      | Exp || Exp
      | not Exp
      | Exp + Exp
      | Exp - Exp
      | Exp * Exp
      | quot Exp Exp
      | rem Exp Exp
      | - Exp
      | Exp == Exp
      | Exp >= Exp
      | Exp <= Exp
      | Exp /= Exp
      | Exp > Exp
      | Exp < Exp
      | (Exp)
      | Var
```

```
Boolean ::= True | False

Integer ::= [0-9]+

Type ::= PrimType
       | Type -> Type
       | (Type)

PrimType ::= Bool | Int

Var ::= [a-zA-Z][0-9a-zA-Z]*
```

2 Haskellito Token Datatype Definition

```
data Token =
      BOOLVAL Bool
    | INTVAL Int
    | IF
    | THEN
    | ELSE
    | LET
    | BIND
    | IN
    | LAMBDA
    | ARROW
    | COLONS
    | EQUALS
    | AND
    | OR
    | NOT
    | PLUS
    | MINUS
    | TIMES
    | QUOT
    | REM
    | LTEQ
    | GTEQ
    | NOTEQ
    | LT
    | GT
    | BOOL
    | INT
    | LPAREN
    | RPAREN
    | VAR String
```

3 Abstract Syntax

The parser you create will transform source code containing a single Haskellito expression into an *abstract syntax tree.* The Haskell data structure that provides this vehicle is presented below.

```
data AST =
  Boolean Bool
| Integer Int
| Let String AST AST
| If AST AST AST
| Lambda String AST TypeExp TypeExp
| App AST AST
| And AST AST
| Or AST AST
| Plus AST AST
| Minus AST AST
| Times AST AST
| Quot AST AST
| Rem AST AST
| Equals AST AST
| Gt AST AST
| Lt AST AST
| Variable String

data TypeExp =
  BoolType
| IntType
| Arrow TypeExp TypeExp
```

4 Type Rules

4.1 Constants

$$\frac{b \in Bool}{H \vdash b :: Bool}$$

$$\frac{n \in Int}{H \vdash n :: Int}$$

4.2 Variables

$$\frac{lookup\ x\ H == t}{H \vdash x :: t}$$

4.3 Operations

$$\frac{H \vdash E_1 :: Int \quad H \vdash E_2 :: Int}{H \vdash E_1 + E_2 :: Int}$$

$$\frac{H \vdash E_1 :: Int \quad H \vdash E_2 :: Int}{H \vdash E_1 - E_2 :: Int}$$

$$\frac{H \vdash E_1 :: Int \quad H \vdash E_2 :: Int}{H \vdash E_1 * E_2 :: Int}$$

$$\frac{H \vdash E_1 :: Int \quad H \vdash E_2 :: Int}{H \vdash quot\ E_1\ E_2 :: Int}$$

$$\frac{H \vdash E_1 : Int \quad H \vdash E_2 : Int}{H \vdash rem\ E_1\ E_2 :: Int}$$

$$\frac{H \vdash E_1 :: Bool \quad H \vdash E_2 :: Bool}{H \vdash E_1\ \&\&\ E_2 :: Bool}$$

$$\frac{H \vdash E_1 :: Bool \quad H \vdash E_2 :: Bool}{H \vdash E_1\ ||\ E_2 :: Bool}$$

$$\frac{H \vdash E :: Bool}{H \vdash not\ E :: Bool}$$

4.4 Predicates

Here it is assumed that t is any primitive type.

$$\frac{H \vdash E_1 :: t \quad H \vdash E_2 :: t}{H \vdash E_1 == E_2 :: Bool}$$

$$\frac{H \vdash E_1 :: t \quad H \vdash E_2 :: t}{H \vdash E_1 > E_2 :: Bool}$$

$$\frac{H \vdash E_1 :: t \quad H \vdash E_2 :: t}{H \vdash E_1 < E_2 :: Bool}$$

4.5 Let Expressions

$$\frac{H \vdash E_1 :: s \quad (x, s) : H \vdash E_2 :: t}{H \vdash let\ x = E_1\ in\ E_2 :: t}$$

4.6 If Expressions

$$\frac{H \vdash E_1 :: Bool \quad H \vdash E_2 :: t \quad H \vdash E_3 :: t}{H \vdash if\ E_1\ then\ E_2\ else\ E_3 :: t}$$

4.7 Lambda Expressions

$$\frac{(x, s) : H \vdash E :: t}{H \vdash \backslash\ x\ \rightarrow\ E :: s \rightarrow t}$$

4.8 Function Application

$$\frac{H \vdash E_1 :: s \rightarrow t \quad H \vdash E_2 :: s}{H \vdash E_1\ E_2 :: t}$$

5 Operational Semantics

5.1 Constants

$$\frac{}{\sigma(True) \Downarrow True}$$

$$\frac{}{\sigma(False) \Downarrow False}$$

$$\frac{n \in Int}{\sigma(n) \Downarrow n}$$

5.2 Variables

$$\frac{lookup\ x\ \sigma == E \quad \sigma(E) \Downarrow v}{\sigma(x) \Downarrow v}$$

5.3 Operations

$$\frac{\sigma(E_1) \Downarrow n_1 \quad \sigma(E_2) \Downarrow n_2 \quad n == n_1 + n_2}{\sigma(E_1 + E_2) \Downarrow n}$$

$$\frac{\sigma(E_1) \Downarrow n_1 \quad \sigma(E_2) \Downarrow n_2 \quad n == n_1 - n_2}{\sigma(E_1 - E_2) \Downarrow n}$$

$$\frac{\sigma(E_1) \Downarrow n_1 \quad \sigma(E_2) \Downarrow n_2 \quad n == n_1 * n_2}{\sigma(E_1 * E_2) \Downarrow n}$$

$$\frac{\sigma(E_1) \Downarrow n_1 \quad \sigma(E_2) \Downarrow n_2 \quad n == quot\ n_1\ n_2}{\sigma(quot\ E_1\ E_2) \Downarrow n}$$

$$\frac{\sigma(E_1) \Downarrow n_1 \quad \sigma(E_2) \Downarrow n_2 \quad n == rem\ n_1\ n_2}{\sigma(rem\ E_1\ E_2) \Downarrow n}$$

$$\frac{\sigma(E_1) \Downarrow False}{\sigma(E_1\ \&\&\ E_2) \Downarrow False}$$

$$\frac{\sigma(E_1) \Downarrow \mathit{True} \quad \sigma(E_2) \Downarrow b}{\sigma(E_1 \ \&\& \ E_2) \Downarrow b}$$

$$\frac{\sigma(E_1) \Downarrow \mathit{True}}{\sigma(E_1 \ || \ E_2) \Downarrow \mathit{True}}$$

$$\frac{\sigma(E_1) \Downarrow \mathit{False} \quad \sigma(E_2) \Downarrow b}{\sigma(E_1 \ || \ E_2) \Downarrow b}$$

5.4 Predicates

Here it is assumed that all values have primitive type.

$$\frac{\sigma(E_1) \Downarrow v \quad \sigma(E_2) \Downarrow v}{\sigma(E_1 == E_2) \Downarrow \mathit{True}}$$

$$\frac{\sigma(E_1) \Downarrow v_1 \quad \sigma(E_2) \Downarrow v_2 \quad v_1 \neq v_2}{\sigma(E_1 == E_2) \Downarrow \mathit{False}}$$

$$\frac{\sigma(E_1) \Downarrow v_1 \quad \sigma(E_2) \Downarrow v_2 \quad v_1 > v_2}{\sigma(E_1 > E_2) \Downarrow \mathit{True}}$$

$$\frac{\sigma(E_1) \Downarrow v_1 \quad \sigma(E_2) \Downarrow v_2 \quad v_1 \leq v_2}{\sigma(E_1 > E_2) \Downarrow \mathit{False}}$$

$$\frac{\sigma(E_1) \Downarrow v_1 \quad \sigma(E_2) \Downarrow v_2 \quad v_1 < v_2}{\sigma(E_1 < E_2) \Downarrow \mathit{True}}$$

$$\frac{\sigma(E_1) \Downarrow v_1 \quad \sigma(E_2) \Downarrow v_2 \quad v_1 \geq v_2}{\sigma(E_1 < E_2) \Downarrow \mathit{False}}$$

5.5 If Expressions

$$\frac{\sigma(E_1) \Downarrow True \quad \sigma(E_2) \Downarrow v_2}{\sigma(if\ E_1\ then\ E_2\ else\ E_3) \Downarrow v_2}$$

$$\frac{\sigma(E_1) \Downarrow False \quad \sigma(E_3) \Downarrow v_3}{\sigma(if\ E_1\ then\ E_2\ else\ E_3) \Downarrow v_3}$$

5.6 Lambda Expressions

This rule appears to be simple enough, since a lambda expression just reduces to a lambda expression. However, unbound variables in the body of a lambda expression must have those values dictated by the *environment at the moment of the lambda expression's formation.* This approach to determining the value of variable in a function's body is called *static binding* and is the rule for Haskell, Caml, Clean, Scheme, SML, and most other modern functional languages. A system that binds the value of a function's free variable using the environment present during the application of the function is adhering to *dynamic binding.* The original Lisp did this, although that was not McCarthy's original intention!

Hence, given an environment σ, a Haskellito expression E, and a variable x, let σ_x stand for the result of removing all bindings for x from σ. Moreover, let $\sigma_x[E]$ be the result of applying σ_x, considered as a substitution, to E. Then our rule becomes the following.

$$\frac{}{\sigma(\backslash\ x\ \to\ E :: s \to t) \Downarrow \backslash\ x\ \to\ \sigma_x[E] :: s \to t}$$

5.7 Let Expressions (Lazy)

$$\frac{(x, \sigma_x[E_1]) : \sigma(E_2) \Downarrow v_2}{\sigma(let\ x = E_1\ in\ E_2) \Downarrow v_2}$$

5.8 Let Expressions (Eager)

$$\frac{\sigma(E_1) \Downarrow v_1 \quad (x, v_1) :: \sigma(E_2) \Downarrow v_2}{\sigma(let\ x = E_1\ in\ E_2) \Downarrow v_2}$$

5.9 Function Application (Lazy)

$$\frac{\sigma(E_1) \Downarrow \backslash\ x\ \to\ E :: s \to t \quad (x, \sigma_x[E_2]) : \sigma(E) \Downarrow v}{\sigma(E_1\ E_2) \Downarrow v}$$

5.10 Function Application (Eager)

$$\frac{\sigma(E_1) \Downarrow \backslash\ x\ \to\ E :: s \to t \quad \sigma(E_2) \Downarrow v_2 \quad (x, v_2) :: \sigma(E) \Downarrow v}{\sigma(E_1\ E_2) \Downarrow v}$$