5

IMPLEMENTATION OF λ-EXPRESSIONS

5.1 THE HASKELL PROGRAMMING LANGUAGE

Haskell is the purely functional programming language of our choice to implement Mikrokosmos, our λ-calculus interpreter. Its own design is heavily influenced by the λ-calculus and is a general-purpose language with a rich ecosystem and plenty of consolidated libraries[1] in areas such as parsing, testing or system interaction; matching the requisites of our project. In the following sections, we describe this ecosystem in more detail.

In the 1980s, many lazy programming languages were independently being written by researchers such as *Miranda*, *Lazy ML*, *Orwell*, *Clean* or *Daisy*. All of them were similar in expressive power, but their differences were holding back the efforts to communicate ideas on functional programming. A comitee was created in 1987 with the mission of designing a common lazy functional language. Several versions of the language were developed, and the first standarized reference of the language was published in the **Haskell 98 Report**, whose revised version can be read in [P+03]. Its more popular implementation is the **Glasgow Haskell Compiler (GHC)**; an open source compiler written in Haskell and C. The complete history of Haskell and its design decisions is detailed on [HHJW07]. Haskell is

1. **strongly and statically typed**, meaning that it only compiles well-typed programs and it does not allow implicit type casting; the compiler will generate an error if a term is non-typeable;
2. **lazy**, with *non-strict semantics*, meaning that it will not evaluate a term or the argument of a function until it is needed; in [Hug89], John Hughes, codesigner of the language, argues for the benefits of a lazy functional language, which could solve the traditional efficiency problems on functional programming;
3. **purely functional**; as the evaluation order is demand-driven and not explicitly known, it is not possible in practice to perform ordered input/output actions or

1 : In the central package archive of the Haskell community, Hackage, a categorized list of libraries can be found: https://hackage.haskell.org/packages/

any other side-effects by relying on the evaluation order; this helps modularity of the code, testing, and verfication;

4. **referentially transparent**; as a consequence of its purity, every term on the code could be replaced by its definition without changing the global meaning of the program; this allows equational reasoning with rules that are directly derived from λ-calculus;
5. based on **System F**ω with some restrictions; crucially, it implements **System F** adding quantification over type operators even if it does not allow abstraction on type operators; the GHC Haskell compiler, however, allows the user to activate extensions that implement dependent types.

Example 4 (A first example in Haskell). This example shows the basic syntax and how its type system and its implicit laziness can be used.

```haskell
-- The type of the term can be declared.
id :: a -> a  -- Polymorphic type variables are allowed,
id x = x      -- and the function is defined equationally.
-- This definition performs short circuit evaluation thanks
-- to laziness. The unused argument can be omitted.
(&&) :: Bool -> Bool -> Bool
True  && x = x                -- (true and x) is always x
False && _ = False            -- (false and y) is always false
-- Laziness also allows infinite data structures.
nats :: [Integer]         -- List of all natural numbers,
nats = 1 : map (+1) nats  -- defined recursively.
```

Where most imperative languages use semicolons to separate sequential commands, Haskell has no notion of sequencing, and programs are written in a purely declarative way. A Haskell program essentially consist on a series of definitions (of both types and terms) and type declarations. The following example shows the definition of a binary tree and its preorder as

```haskell
-- A tree is either empty or a node with two subtrees.
data Tree a = Empty | Node a (Tree a) (Tree a)
-- The preorder function takes a tree and returns a list
preorder :: Tree a -> [a]
preorder Empty            = []
preorder (Node x lft rgt) = preorder lft ++ [x] ++ preorder rgt
```

We can see on the previous example that function definitions allow *pattern matching*, that is, data constructors can be used in definitions to decompose values of the type. This increases readability when working with algebraic data types.

While infix operators are allowed, function application is left-associative in general. Definitions using partial application are allowed, meaning that functions on multiple

arguments can use currying and can be passed only one of its arguments to define a new function. For example, a function that squares every number on a list could be written in two ways as

```haskell
squareList :: [Int] -> [Int]
squareList list = map square list
squareList' :: [Int] -> [Int]
squareList' = map square
```

where the second one, because of its simplicity, is usually preferred. A characteristic piece of Haskell are **type classes**, which allow defining common interfaces for different types. In the following example, we define `Monad` as the type class of types with suitably typed `return` and `bind` operators.

```haskell
class Monad m where
  return :: a   -> m a
  (>>=)  :: m a -> (a -> m b) -> m b
```

And lists, for example, are monads in this sense.

```haskell
instance Monad [] where
  return x = [x]               -- returns a one-element list
  xs >>= f = concat (map f xs) -- map and concatenation
```

Haskell uses monads in varied forms. They are used in I/O, error propagation and stateful computations. Another characteristical syntax bit of Haskell is the `do` notation, which provides a nicer, cleaner way to work with types that happen to be monads. The following example uses the list monad to compute the list of Pythagorean triples.

```haskell
pythagorean = do
  a <- [1..]                -- let a be any natural
  b <- [1..a]               -- let b be a natural between 1 and a
  c <- [1..b]               -- let c be a natural between 1 and b
  guard (a^2 == b^2 + c^2) -- filter the list
  return (a,b,c)            -- return matching tuples
```

Note that this list is infinite. As the language is lazy, this does not represent a problem: the list will be evaluated only on demand.

Another common example of an instance of the `Monad` typeclass is the *Maybe monad* used to deal with error propagation. A `Maybe a` type can consist of a term of type `a`, written as `Just a`; or as a `Nothing` constant, signalling an error. The monad is then defined as

```haskell
instance Monad Maybe where
  return x = Just x
  xs >>= f = case xs of Nothing -> Nothing | Just a -> Just (f a)
```

and can be used as in the following example to use *exception-like* error treatment in a pure declarative language

```haskell
roots :: (Float,Float,Float) -> Maybe Int
roots (a,b,c) = do
  -- Some errors can occur during this computation
  discriminant <- sqroot (b*b - 4*c*a)         -- roots of negative numbers?
  root1 <- safeDiv ((-b) + discriminant) (2*a) -- division by zero?
  root2 <- safeDiv ((-b) - discriminant) (2*a)
  -- The monad ensures that we return a number only if no error has been raised
  return (root1,root2)
```

For a more detailed treatment of monads, and their relation to categorical monads, see the chapter on Category Theory and the chapter on Type Theory, where we will program with monads in Agda.

5.2 DE BRUIJN INDEXES

Nicolaas Govert **De Bruijn** proposed in [dB72] a way of defining λ-terms modulo α-conversion based on indices. The main goal of De Bruijn indices is to remove all variables from binders and replace every variable on the body of an expression with a number, called *index*, representing the number of λ-abstractions in scope between the ocurrence and its binder.

Consider the following example: the λ-term

$$\lambda x.(\lambda y.\ y(\lambda z.\ yz))(\lambda t.\lambda z.\ tx)$$

can be written with de Bruijn indices as

$$\lambda\ (\lambda(1\lambda(21))\ \lambda\lambda(23)\ ).$$

De Bruijn also proposed a notation for the λ-calculus changing the order of binders and λ-applications. A review on the syntax of this notation, its advantages and De Bruijn indexes, can be found in [Kam01]. In this section, we are going to describe De Bruijn indexes while preserving the usual notation of λ-terms; that is, the *De Bruijn indexes* and the *De Bruijn notation* are different concepts and we are going to use only the former.

Definition 28 (De Bruijn indexed terms)**.** We define recursively the set of λ-terms using de Bruijn notation following this BNF

$$\texttt{Exp} ::= \underbrace{\mathbb{N}}_{\textit{variable}} \mid \underbrace{(\lambda\ \texttt{Exp})}_{\textit{abstraction}} \mid \underbrace{(\texttt{Exp}\ \texttt{Exp})}_{\textit{application}}$$
$$\mathbb{N} ::= 0 \mid 1 \mid 2 \mid \dots$$

Our internal definition closely matches the formal one. The names of the constructors here are `Var`, `Lambda` and `App`:

```haskell
-- | A lambda expression using DeBruijn indexes.
data Exp = Var Integer -- ^ integer indexing the variable.
         | Lambda Exp  -- ^ lambda abstraction
         | App Exp Exp -- ^ function application
         deriving (Eq, Ord)
```

This notation avoids the need for the Barendregt's variable convention and the α-reductions. It will be useful to implement λ-calculus without having to worry about the specific names of variables.

5.3 SUBSTITUTION

We define the substitution operation needed for the β-reduction on de Bruijn indices. In order to define the substitution of the n-th variable by a λ-term P on a given term, we must

- find all the ocurrences of the variable. At each level of scope we are looking for the successor of the number we were looking for before;
- decrease the higher variables to reflect the disappearance of a lambda;
- replace the ocurrences of the variables by the new term, taking into account that free variables must be increased to avoid them getting captured by the outermost lambda terms.

In our code, we apply `subs` to any expression. When it is applied to a λ-abstraction, the index and the free variables of the replaced term are increased with `incrementFreeVars`; whenever it is applied to a variable, the previous cases are taken into consideration.

```haskell
-- | Substitutes an index for a lambda expression
subs :: Integer -> Exp -> Exp -> Exp
subs n p (Lambda e) = Lambda (subs (n+1) (incrementFreeVars 0 p) e)
subs n p (App f g)  = App (subs n p f) (subs n p g)
subs n p (Var m)
  | n == m    = p         -- The lambda is replaced directly
```

```
  | n <  m    = Var (m-1) -- A more exterior lambda decreases a number
  | otherwise = Var m     -- An unrelated variable remains untouched
```

Then β-reduction can be defined using this `subs` function.

```
betared :: Exp -> Exp
betared (App (Lambda e) x) = substitute 1 x e
betared e = e
```

5.4 DE BRUIJN-TERMS AND λ-TERMS

The internal language of the interpreter uses de Bruijn expressions, while the user interacts with it using lambda expressions with alphanumeric variables. Our definition of a λ-expression with variables will be used in parsing and output formatting.

```
data NamedLambda = LambdaVariable String
                 | LambdaAbstraction String NamedLambda
                 | LambdaApplication NamedLambda NamedLambda
```

The translation from a natural λ-expression to de Bruijn notation is done using a dictionary which keeps track of the bounded variables

```
tobruijn :: Map.Map String Integer -- ^ names of the variables used
         -> Context                -- ^ names already binded on the scope
         -> NamedLambda            -- ^ initial expression
         -> Exp
-- Every lambda abstraction is inserted in the variable dictionary,
-- and every number in the dictionary increases to reflect we are entering
-- a deeper context.
tobruijn d context (LambdaAbstraction c e) =
     Lambda $ tobruijn newdict context e
        where newdict = Map.insert c 1 (Map.map succ d)

-- Translation distributes over applications.
tobruijn d context (LambdaApplication f g) =
     App (tobruijn d context f) (tobruijn d context g)

-- We look for every variable on the local dictionary and the current scope.
tobruijn d context (LambdaVariable c) =
  case Map.lookup c d of
    Just n  -> Var n
    Nothing -> fromMaybe (Var 0) (MultiBimap.lookupR c context)
```

while the translation from a de Bruijn expression to a natural one is done considering an infinite list of possible variable names and keeping a list of currently-on-scope variables to name the indices.

```haskell
-- | An infinite list of all possible variable names
-- in lexicographical order.
variableNames :: [String]
variableNames = concatMap (`replicateM` ['a'..'z']) [1..]

-- | A function translating a deBruijn expression into a
-- natural lambda expression.
nameIndexes :: [String] -> [String] -> Exp -> NamedLambda
nameIndexes _    _    (Var 0) = LambdaVariable "undefined"
nameIndexes used _    (Var n) =
  LambdaVariable (used !! pred (fromInteger n))
nameIndexes used new (Lambda e) =
  LambdaAbstraction (head new) (nameIndexes (head new:used) (tail new) e)
nameIndexes used new (App f g) =
  LambdaApplication (nameIndexes used new f) (nameIndexes used new g)
```

5.5 EVALUATION

As we proved on Corollary 1, the leftmost reduction strategy will find the normal form of any given term provided that it exists. Consequently, we will implement reduction in our interpreter using a function that simply applies the leftmost possible reductions at each step. As a side benefit, this will allow us to show how the interpreter performs step-by-step evaluations to the final user, as discussed in the verbose mode section.

```haskell
-- | Simplifies the expression recursively.
-- Applies only one parallel beta reduction at each step.
simplify :: Exp -> Exp
simplify (Lambda e)           = Lambda (simplify e)
simplify (App (Lambda f) x)   = betared (App (Lambda f) x)
simplify (App (Var e) x)      = App (Var e) (simplify x)
simplify (App a b)            = App (simplify a) (simplify b)
simplify (Var e)              = Var e

-- | Applies repeated simplification to the expression until it stabilizes and
-- returns all the intermediate results.
simplifySteps :: Exp -> [Exp]
simplifySteps e
  | e == s    = [e]
```

```
  | otherwise = e : simplifySteps s
  where s = simplify e
```

From the code we can see that the evaluation finishes whenever the expression stabilizes. This can happen in two different cases

- there are no more possible β-reductions, and the algorithm stops; or
- β-reductions do not change the expression. The computation would lead to an infinite loop, so it is immediately stopped. An common example of this is the λ-term $(\lambda x.xx)(\lambda x.xx)$.

5.6 PRINCIPAL TYPE INFERENCE

The interpreter implements the unification and type inference algorithms described in Lemma 4 and Theorem 5. Their recursive nature makes them very easy to implement directly on Haskell. We implement a simply-typed lambda calculus with Curry-style typing and type templates. Our type system has

- an unit type;
- a void type;
- product types;
- union types;
- and function types.

```
-- | A type template is a free type variable or an arrow between two
-- types; that is, the function type.
data Type = Tvar Variable
          | Arrow Type Type
          | Times Type Type
          | Union Type Type
          | Unitty
          | Bottom
         deriving (Eq)
```

We will work with substitutions on type templates. They can be directly defined as functions from types to types. A basic substitution that inserts a given type on the place of a variable will be our building block for more complex ones.

```
type Substitution = Type -> Type

-- | A basic substution. It changes a variable for a type
subs :: Variable -> Type -> Substitution
subs x typ (Tvar y)
  | x == y    = typ
```

```
  | otherwise = Tvar y
subs x typ (Arrow a b) = Arrow (subs x typ a) (subs x typ b)
subs x typ (Times a b) = Times (subs x typ a) (subs x typ b)
subs x typ (Union a b) = Union (subs x typ a) (subs x typ b)
subs _ _ Unitty = Unitty
subs _ _ Bottom = Bottom
```

Unification will be implemented making extensive use of the `Maybe` monad. If the unification fails, it will return an error value, and the error will be propagated to the whole computation. The algorithm is exactly the same that was defined in Lemma 4.

```
-- | Unifies two types with their most general unifier. Returns the substitution
-- that transforms any of the types into the unifier.
unify :: Type -> Type -> Maybe Substitution
unify (Tvar x) (Tvar y)
  | x == y    = Just id
  | otherwise = Just (subs x (Tvar y))
unify (Tvar x) b
  | occurs x b = Nothing
  | otherwise  = Just (subs x b)
unify a (Tvar y)
  | occurs y a = Nothing
  | otherwise  = Just (subs y a)
unify (Arrow a b) (Arrow c d) = unifypair (a,b) (c,d)
unify (Times a b) (Times c d) = unifypair (a,b) (c,d)
unify (Union a b) (Union c d) = unifypair (a,b) (c,d)
unify Unitty Unitty = Just id
unify Bottom Bottom = Just id
unify _ _ = Nothing

-- | Unifies a pair of types
unifypair :: (Type,Type) -> (Type,Type) -> Maybe Substitution
unifypair (a,b) (c,d) = do
  p <- unify b d
  q <- unify (p a) (p c)
  return (q . p)
```

The type inference algorithm is more involved. It takes a list of fresh variables, a type context, a lambda expression and a constraint on the type, expressed as a type template. It outputs a substitution. As an example, the following code shows the type inference algorithm for function types.

```
-- | Type inference algorithm. Infers a type from a given context and expression
-- with a set of constraints represented by a unifier type. The result type must
-- be unifiable with this given type.
typeinfer :: [Variable] -- ^ List of fresh variables
```

```
          -> Context    -- ^ Type context
          -> Exp        -- ^ Lambda expression whose type has to be inferred
          -> Type       -- ^ Constraint
          -> Maybe Substitution

typeinfer (x:vars) ctx (App p q) b = do -- Writing inside the Maybe monad.
  sigma <- typeinfer (evens vars) ctx                 p (Arrow (Tvar x) b)
  tau   <- typeinfer (odds  vars) (applyctx sigma ctx) q (sigma (Tvar x))
  return (tau . sigma)
  where
    -- The list of fresh variables has to be split into two
    odds [] = []
    odds [_] = []
    odds (_:e:xs) = e : odds xs
    evens [] = []
    evens [e] = [e]
    evens (e:_:xs) = e : evens xs
```

The final form of the type inference algorithm will use a normalization algorithm shortening the type names and will apply the type inference to the empty type context. The complete code can be found on the [BROKEN LINK: *Mikrokosmos complete code]. A generalized version of the type inference algorithm is used to generate derivation trees from terms, as it was described in Propositions as types.

In order to draw these diagrams in Unicode characters, a data type for character blocks has been defined. A monoidal structure is defined over them; blocks can be joined vertically and horizontally; and every deduction step can be drawn independently.

```
newtype Block = Block { getBlock :: [String] }
  deriving (Eq, Ord)

instance Monoid Block where
  mappend = joinBlocks -- monoid operation, joins blocks vertically
  mempty  = Block [[]] -- neutral element

-- Type signatures
joinBlocks :: Block -> Block -> Block
stackBlocks :: String -> Block -> Block -> Block
textBlock :: String -> Block
deductionBlock :: Block -> String -> [Block] -> Block
box :: Block -> Block
```