Language Implementation: Attribute Grammars

Alejandro Gadea   Emmanuel Gunther

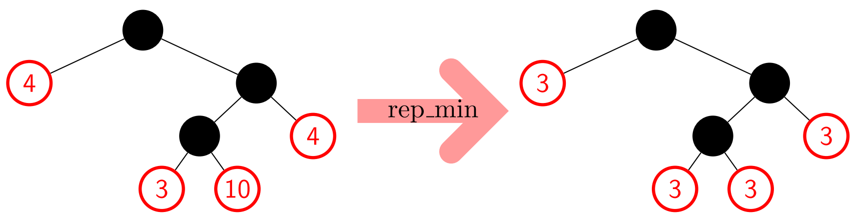
1 de julio de 2014

1. Syntax and Semantics of Languages

1.1. The RepMin problem

The “Rep Min” is a well known example of the expressiveness of Lazy Evaluation and inspires the way to define the semantics that exploits the Attribute Grammar Framework.

Think about the data type of binary trees, with numbers in the leaves. The problem consists in replacing each value at each leaf by the least value in the whole tree:



We can define a data type in Haskell for representing a binary tree in the following way:

```
data Tree =    Leaf Int
             | Bin  (Tree, Tree)
```

A tree can be built from an integer (a leaf) or from others two subtrees, i.e we can think of an element of *Tree* as an element of the disjoint union $Int \uplus (Tree \times Tree)$. This disjoint union can be defined for any type a:

```
type FTree a = Either Int (a,a)
```

and then we can get any element of **Tree** from some element of **FTree Tree**.

AS we will see the type of that function is interesting in itself, so we explicitly define it:

```
type FTreeAlgebra a = FTree a -> a
```

Now, if we want to define a semantics for the binary trees in the semantic set B we can define a **FTreeAlgebra** B. This is, we should give a rule in the case of having an integer and another in the case of having a pair in $B \times B$, which one should think as the semantic value of the subtrees.

The syntactic representation of binary trees with type *Tree* can be seen as the trivial semantic and we can define the next algebra:

```
init_algebra :: FTreeAlgebra Tree
init_algebra = either Leaf Bin
```

Thinking in categories, we can see *FTree* as a functor which transform an object A in an object $Int \uplus (A \times A)$ and then a FTree-algebra will be an arrow $\alpha : FTree\ A \to A$. If α is the initial FTree-algebra, then for any other FTree-algebra $\beta : FTree\ B \to B$ there must exist an unique arrow between α and β, i.e, a single f such that the following diagram commutes:

$$\begin{tikzcd}
FTree\ A \arrow[r, "\alpha"] \arrow[d, "FTree\ f"'] & A \arrow[d, "f"] \\
FTree\ B \arrow[r, "\beta"] & B
\end{tikzcd}$$

Thus, if we want to define a semantics for the binary trees we must find the single arrow between the *init_algebra* (the initial FTree-algebra) and other FTree-algebra. This arrow is called **catamorphism**:

```
cataTree :: FTreeAlgebra b -> Tree -> b
cataTree beta (Leaf i)     = beta (Left i)
cataTree beta (Bin (t1,t2)) = beta (Right (cataTree beta t1,
                                           cataTree beta t2))
```

Let us return to the problem that we want to solve, given a tree we want to transform it in the tree in which the value of the leaves is the least value.

In a first attempt, we get the least value of the tree and then replace it in all leaves. This implies walking the initial tree two times.

Now, to get the least value we define a FTree-algebra and define a function that given an integer n constructs the FTree-algebra which replace the value of the leaves for n, the least value got in the first phase:

```
min_alg :: FTreeAlgebra Int
min_alg = either id (uncurry min)

rep_min_alg :: Int -> FTreeAlgebra Tree
rep_min_alg n = either (const $ Leaf n) Bin
```

Finally, we solve the problem calling twice the function *cataTree*:

```
replace_min :: Tree -> Tree
replace_min t = let n = cataTree min_alg t in
                cataTree (rep_min_alg n) t
```

Notice that in the function *rep_min_alg* we construct a FTree-algebra from an integer. Instead of this, we can define a FTree-algebra which allow us to compute a function that given an integer builds up a tree with all the values of the leaves replaced by that integer:

```
rep_min_alg' :: FTreeAlgebra (Int -> Tree)
rep_min_alg' = either (const Leaf)
                      (\(lfun,rfun) -> \m ->
                           Bin (lfun m,rfun m))
```

so, the solution to our problem can be stated as:

```
replace_min' :: Tree -> Tree
replace_min' t = (cataTree rep_min_alg' t) (cataTree min_alg t)
```

In the last definition we can see that the calls to the function *cataTree* are independent, thus we could compute a single FTree-algebra that get both results at the same time. With *min_alg* making the least value and with *rep_min_alg′* the function for builds up the tree, therefore if we could make the product of those FTree-algebra we would get both results:

```
infix 9 `x`

x :: FTreeAlgebra a -> FTreeAlgebra b -> FTreeAlgebra (a,b)
fa `x` fb = either  (\i -> (fa $ Left i,fb $ Left i))
                    (\((a,b),(a',b')) ->
                       (fa $ Right (a,a'),fb $ Right (b,b')))


rep_min_alg'' :: FTreeAlgebra (Int,Int -> Tree)
rep_min_alg'' = min_alg `x` rep_min_alg'
```

So now we can get the result to the problem making only one call to *cataTree*:

```
replace_min'' :: Tree -> Tree
replace_min'' t = r m
   where (m, r) = cataTree rep_min_alg'' t
```

In final step, we can see that from a value in $(Int, Int \rightarrow Tree)$ we can obtain a value in $Int \rightarrow (Int, Tree)$. The inverse is not always true but is not a matter here. We can consider the following function that given two isomorphic types a and b, and a *FTreeAlgebra a* constructs a *FTreeAlgebra b*:

```
getIsoAlg :: (a -> b) -> (b -> a) -> FTreeAlgebra a -> FTreeAlgebra b
getIsoAlg fab fba fa = either (fab . fa . Left)
                              (fab . fa . Right . (fba *** fba))
```

In our problem we have a FTree-algebra of type

```
(Int,Int -> Tree)
```

and we want to define a FTree-algebra of type:

```
Int -> (Int,Tree)
```

We need define two functions to exchange those types:

```
f1 :: (Int,Int -> Tree) -> (Int -> (Int,Tree))

f2 :: (Int -> (Int,Tree)) -> (Int,Int -> Tree)
```

We can construct the first of these easily, but it's not the case for the second one:

```
f1 :: (Int,Int -> Tree) -> (Int -> (Int,Tree))
f1 (i,f) = const i &&& f

f2 :: (Int -> (Int,Tree)) -> (Int,Int -> Tree)
f2 f = (??? , snd . f)
```

We don't have too many options to define the first component of $f2\ f$. We can return a constant integer or we can evaluate f on a constant value. If we do this, we obtain one side of the isomorphism and we can see that it is sufficient to our problem (we couldn't prove this at the moment)

```
f1 :: (Int,Int -> Tree) -> (Int -> (Int,Tree))
f1 (i,f) = const i &&& f

f2 :: (Int -> (Int,Tree)) -> (Int,Int -> Tree)
f2 f = (fst (f 0) , snd . f)
```

Then, we define a new FTree-algebra:

```
rep_min_alg''' :: FTreeAlgebra (Int -> (Int,Tree))
rep_min_alg''' = getIsoAlg f1 f2 rep_min_alg''
```

And the solution is reached defining the next function that it works because of laziness of Haskell:

```
replace_min''' :: Tree -> Tree
replace_min''' t = r
   where (m, r) = (cataTree rep_min_alg''' t) m
```

1.2. Repmin with Attribute Grammars

In the previous section we saw how to solve the Repmin problem in a more efficient way, performing a single walk thanks to the Lazy Evaluation of Haskell. For this we obtain a tuple in which the first element was the least value of the tree and the second one was a function that given an integer builds up a tree with all the values of leaves replaced for that integer.

Notice that we could take the same steps for any other recursive data type, where the grammar not necessarily has two productions. We could define a functor F and the appropriate catamorphism, and then for calculating any semantics we only need to give one rule for each production of the grammar.

With **Attribute Grammars**, we have a simple notation for specifying a semantics, defining the syntax of a data type and then giving the rules for each production of the grammar. A preprocessor transform the AG notation into Haskell code that is equivalent to that obtained in the previous section.

Let us illustrate this point showing how the Repmin problem can be solved with AG:

```
data Tree
    | Leaf       Int
    | Bin  lt  :: Tree
           rt  :: Tree

deriving Tree : Show

attr Tree
    inh minval :: Int
    syn m      :: Int
    syn res    :: Tree

sem Tree
    | Leaf lhs.m   = @int
              .res = Leaf @lhs.minval
        | Bin  lhs.m   = @lt.m `min` @rt.m
              .res = Bin @lt.res @rt.res

data Root
        | Root Tree
```

```
attr Root
        syn res :: Tree

sem Root
        | Root  tree.minval = @tree.m
```

The definition of the grammar is similar to the one given in Haskell, the only difference is that we just put names in each parameter of each production. To calculate the semantics we define two kinds of attributes, the synthesized (*syn*), which are those that for computing the value in a node we need to know the value of the childrens, and the inherited (*inh*), which are passed top to down in the tree.

In our example we need the least value of the tree to compute the result of Repmin. We define an inherited attribute *minval* and two synthesized attributes, *m* to calculate the least value of the tree and *res* to compute the final tree with the replaced of *minval* that is initialized in *m*.

In the previous piece of code we used many abbreviations that allows us to write shorter code. For example, if a rule of a grammar production has a single argument, the name of this can be omitted and the preprocesor generates one with the name of the data type which has the first letter in lowercase. If we want define a rule for an inherited attribute which doesn't change at child node, it's not necessary to define that trivial rule, we can omit it and the preprocessor generates it for us.

The generated code using **Attribute Grammars** computes the attributes values by doing the procedure that we have show in the previous section. The type of the resulting algebra will be a function where the parameters are the inherited attributes and the result will be a tuple where each element is a synthesized attribute.

2. The Lambda Calulus

In the previous section we saw how we can define a function in a general way, that we called catamorphism, and how this is internalized in the AG framework with a special syntax with the idea that we just need to implement the relating things of the problem.

In the following, we will implement a type inference algorithm for the simply typed lambda calculus, as well as the parser and the pretty printer.

2.1. Syntax: Parser and Pretty Printing

2.1.1. Syntax

Let Var be a countable set. In general, we will use a, b, c, s, x, y, z. To denote elements of Var.

```
Term ::= Var
       | λ Var . Term
       | Term Term
```

2.1.2. Parser

The implementation of a parser for the syntax of lambda calculus is rather simple, even adding the restriction that we only allow to parse closed terms, i.e. terms in which don't occur free variables. However, suppose we have to parse the following string,

$\lambda a.s$

clearly "s" is free, but it would not be wrong to assume that the intended string was

$\lambda a.a$

i.e. a mistake was made by writing the string. Thus, a parser a little bit more interesting would be one that detects that kind of errors, corrects them and gives information about such decision.

For the implementation of the parser we use the library "uu-parsinglib" which provides exactly an error correction mechanism. For example, being "pa" the parser of the letter "a", trying to parse "b" will produce an "a" the correction being the replacement of "b" for "a":

```
>>> run pa  "b"
    Result: "a"
    Correcting steps:
      Deleted   'b' at position LineColPos 0 0 0 expecting 'a'
      Inserted  'a' at position LineColPos 0 1 1 expecting 'a'
```

We would like to highlight that the error correction that we pretend for our parser of terms comes almost for free. In a preliminary summary, for parsing the body of an abstraction we just need to have a list with the introduced variables till that moment and generate parsers for each variable in the list.

Before introducing the parsers concerning each construction of the grammar, we define some general parsers that will be useful. We generate a parser from a default parser and a list of strings.

```
parseTermSym :: Parser String -> [String] -> Parser String
```

Parsers for the symbols λ, . and @. Further we can parser "\\" instead of λ and $->$ or $\rightarrow$ instead of "." .

```
parseXSym :: Parser String
```

with X $\in$ Lam,Dot,App. And parser for variables.

```
parseVar :: Parser Var
```

Now, given the list of variable to parse, say *vars*, identifier we generate parsers for each variable in the list and fail as default parser:

```
parseId :: [Var] -> Parser Term
parseId vars = Id <$> parseTermSym pFail vars
```

Something interesting to highlight is that we include the correcting action in the generated parser for variables.

For the case of the abstraction, leaving aside the parsing of the respective symbols, we are going to parse a variable and we could say that we need two things; (1) the constructor of the abstraction, (2) the list of possibles variables to parse. This brings a problem, since when we parse a variable this is encapsulated inside of the computation "Parser Var", by (2) we can think that the type of the parser that parse the body of the abstraction should be $Var- > ParserTerm$, because it takes the variable, adds it to the list of bound variables and then parse the term.

If we pay attention we need a function which takes a parser for the bound occurrence, the function expecting that variable and parses the body adding the variable to the list of bound variables, i.e. a function with type:

```
Parser Var -> (Var -> Parser Term) -> Parser Term
```

but this is just the type of the bind operator (>>=), $m\ a \to (a \to m\ b) \to m\ b$. This forces us to use monads, for which we need to use the addLength function. We are not sure if there exist a way to solve this without monads, with the purpose of avoiding the use of addLength and of course... monads. Then, the final definition is,

```
parseAbs :: [Var] -> Parser Term
parseAbs vars =
       addLength 1 $
       join $ uncurry (<$>)
           <$>
       (Abs &&& parseTerm . (:vars))
           <$
       parseLamSym <*> parseVar <* parseDotSym
```

For concluding, for the case of the application we try to parse bound variables, abstractions or applications, for which we use the symbol "@", this parser is defined as follows,

```
parseTerm' :: [Var] -> Parser Term
parseTerm' vars =  parseId vars
               <|> parseAbs vars
               <|> pParens (parseTerm vars)

parseApp :: [Var] -> Parser Term
parseApp vars = (App <$ parseAppSym) `pChainl` (parseTerm' vars)
```

then, parsing a term is simply the parsing of an application,

```
parseTerm :: [Var] -> Parser Term
parseTerm vars = parseApp vars
```

In conclusion, we have a parser for the grammar of the lambda calculus which also corrects some mistakes, as we mentioned at the beginning. Something important to comment is that the correction of errors may have some unwanted behaviour due to the "simplistic" implementation using the internal mechanisms of corrections offered by the library. An example of this may be,

```
>>> parserTerm "\\a -> b@a"
(λ a → a
, [-- Deleted   'b' at position LineColPos 0 6 6 expecting Whitespace
  ,-- Deleted   '@' at position LineColPos 0 7 7 expecting "a"
  ]
)
```

where most probably the wanted correction it would be the replacement of "b" for "a", and the production of the term $\lambda\ a \rightarrow a@a$.

2.1.3. Pretty Printing

The implementation of the pretty printing is of interest to us as a first little step to the definition of the semantic function for the data type $Term$ using AG. In summary, the semantics we are thinking transforms a $Term$, i.e. a term of the lambda calculus, in his representation in $String$.

Despite of having few constructors, a term of the lambda calculus can be complex to read and an "organization" of the subterms when it comes to writing can be very helpful for the interpretation. For example, the term $(\lambda x \rightarrow x@x)@(\lambda x \rightarrow x@x)$ does not bring much trouble, however if we have, $(\lambda x \rightarrow \lambda y \rightarrow \lambda f \rightarrow f@(x@(\lambda w \rightarrow f@w))@(y@f@x))\ @\ (\lambda x \rightarrow \lambda y \rightarrow \lambda f \rightarrow \lambda g \rightarrow \lambda h \rightarrow h@(f@x)@(g@y))\ @\ \lambda x \rightarrow \lambda f \rightarrow \lambda g \rightarrow f@g@(x@g)$ it is certainly more difficult to read. Thus, a good presentation of the term can be always helpful. The use of the "uulib" library, in particular "UU.PPrint", gives us a way of implement a pretty printer in a simple way.

As we saw for the case of Repmin, we must define attributes and give rules for each constructor, this time for $Term$, which express how to transform each of its productions in its representation as strings.

We want a synthesised attribute, called "pprint", that is the result of transforming a $Term$ in a $String$ and an inherited attribute that we need to know when to enclose a term between brackets or not.

```
attr Term
    syn pprint  :: {Doc}
    inh paren   :: {ParenInfo}
```

On the other hand the semantic is defined as,

```
sem Term
```

Pretty-printing a variable consists sumply in printing the name

```
    | Id   lhs.pprint   = text @ident
```

In the case of the application, we began explaining that to print an application will be as simple as printing its left side (lt) and its right side (rt) but taking into account some considerations; as for the brackets, because the application is left associative if we have, for example the application of $x@y$ to $z@w$ we will need to place parenthesis only in the right side ($rt.paren = Paren$) for printing the correct application $x@y@(z@w)$. But also, there is a particular case where the left side needs to be enclosed in parentheses, for example applying $\lambda x \rightarrow x$ to $z@w$. That is the reason for putting parenthesis around the left side only when it is an abstraction ($lt.paren = ParenAbs$). Notice that without the parenthesis in the example, applications would be $x@y@z@w$ and $\lambda x \rightarrow x@z@w$ which are not the correct representations of the original terms.

Now, leaving aside parentheses, for printing an application the idea would be to try that both sides, the operator and the operand, stay at the same level and in the case that this is not possible obtain the following,

$$lt$$

@

rt

i.e, to divide the subterms in two levels intercalating the application operator. With the possibility of putting the right side at the same level of the operator.

```
 | App lhs.pprint  = (putParen @lhs.paren)
             (group $ @lt.pprint <> line <>
               (group $ text "@" <> line <> @rt.pprint)
             )
       rt.paren      = Paren
       lt.paren      = ParenAbs
```

To finish, we need to define the semantics for the case of the abstraction constructor. For this case the idea is to write all the abstraction i the same level; but, when it does not fit, we will print the body (*term*) in other level with some indentation (3 characters) to ease the reading of the body, when the abstraction is applied to other term. On the other hand, the body of the abstraction never needs to be between parentheses (*term.paren* = *NoParen*).

```
 | Abs lhs.pprint  = (putAbsParen @lhs.paren)
             (text "λ␣" <> text @var <> text "␣→" <>
                 group (nest 3 $ line <> @term.pprint)
             )
       term.paren    = NoParen
```

Printing the example of the beginning with our pretty-printer, we can note that it is much easier to read. In particular, it contains many of the print cases that the semantics we give implements.

```
>>> App (App t2 t3) t1
(λ x →
   λ y →
      λ f → f @ (x @ (λ w → f @ w)) @ (y @ f @ x))
@
(λ x →
   λ y → λ f → λ g → λ h → h @ (f @ x) @ (g @ y))
@ (λ x → λ f → λ g → f @ g @ (x @ g))
```

2.2. Type Inference

The main purpose of this work is to define type inference for the simply typed lambda calculus. Let us recall the typing rules:

$$\frac{}{\pi, (v : \theta) \vdash v : \theta}\ \texttt{T-VAR}$$

$$\frac{\pi \vdash t_1 : \theta_1 \to \theta_2 \quad \pi \vdash t_2 : \theta_1}{\pi \vdash t_1\ t_2 : \theta_2}\ \texttt{T-APP}$$

$$\frac{\pi, (v : \theta_1) \vdash t : \theta_2}{\pi \vdash \lambda\, v.t : \theta_1 \to \theta_2}\ \texttt{T-ABS}$$

In a context π a term t will have type θ if there exists a derivation of $\pi \vdash t : \theta$ using the previous rules.

To infer the type of a term we can assume that it has type θ and then perform substitutions to this type until obtaining a valid type judgement. Supose we have the next term:

$$\lambda\, x.x$$

We assume that it has type θ, but as this is an abstraction, the only rule to construct a valid judgement is $\mathtt{T-ABS}$ where the type should be of the form $\theta_1 \rightarrow \theta_2$, so we can replace $\theta := \theta_1 \rightarrow \theta_2$ and then we try to construct type derivation for $\pi, (x : \theta_1) \vdash x : \theta_2$. As the term now is a variable, the only possible rule is $\mathtt{T-VAR}$ and we need to replace θ_1 for θ_2. Finally the inferred type is:

$$\theta_2 \rightarrow \theta_2$$

With this idea in mind we'll implement type inference using the UUAG library. First of all we define the grammar, where Var is $String$:

```
data Term
     | Id   ident  :: {Var}
     | App  lt, rt :: Term
     | Abs  var    :: {Var}
            term   :: Term
```

Types are also defined using AG; a type is either a type variable, concretely implemented as numbers, or an arrow type:

```
data Type
     | AtomType at     :: {TVar}
     | FunType  lt, rt :: Type
```

and $TVar$ is implemented with integer numbers.

We define the attributes of the grammar for lambda terms:

```
attr Term
   inh ctx        :: Ctx
   inh termType   :: Type
   chn maxTVar    :: TVar
   syn tsubst     :: TSubst
```

The result of type inference will be a substitution to apply on the initial type, and we implement it with a synthesized attribute **tsubst**. As we saw in the previous example, given a term we assume that it has a type, this assumed type is implemented wit an inherited attribute **termType**. We'll have a context for assigning types to variables too, so we define the attribute **ctx** and because is necessary to generate fresh type variables, we define the chained attribute **maxTVar**, so when we need to get a fresh type, it will be the following type variable to the $maxTVar$. This attribute is chained because is necessary to pass the value in a transversal way.

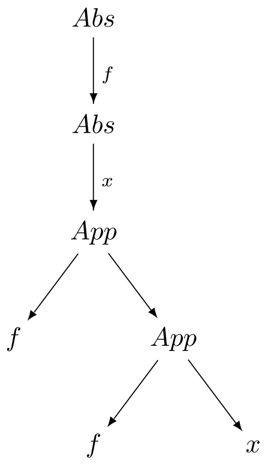
To see how the implemented algorithm works consider the next term:

$$\lambda\, f.\lambda\, x.f\ (f\ x)$$

which is defined in Haskell:

```
Abs "f" (Abs "x" (App (Id "f") (App (Id "f") (Id "x"))))
```

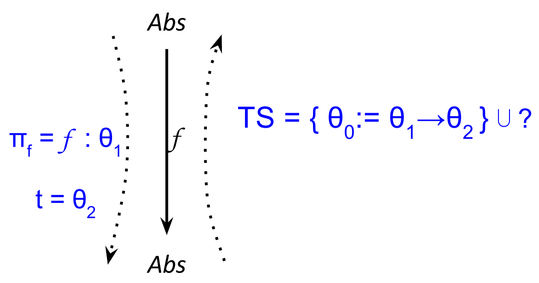
or in a tree way:



$$\begin{tikzcd}[column sep=small]
 & Abs \arrow[d, "f"] & & \\
 & Abs \arrow[d, "x"] & & \\
 & App \arrow[dl] \arrow[dr] & & \\
f & & App \arrow[dl] \arrow[dr] & \\
 & f & & x
\end{tikzcd}$$

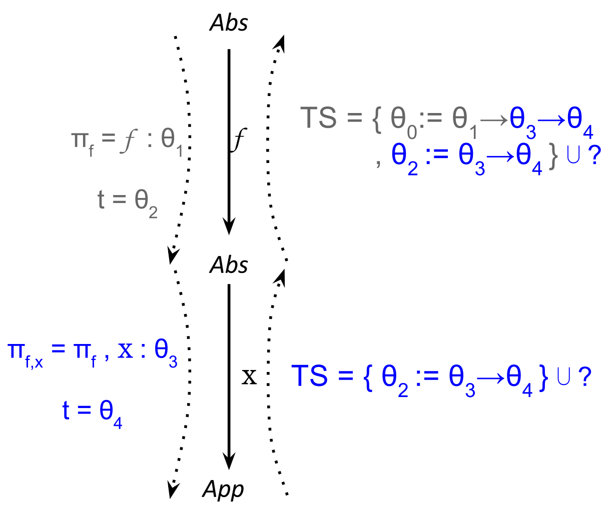
For the sake of clarity in explanation of the example, we omit some details, for example the value of attribute $maxTVar$ needed to generate fresh variables won't be considered.

The first node is an abstraction. We have to define the value of the inherited attributes on child and to calculate the syntesized attribute in function of it.



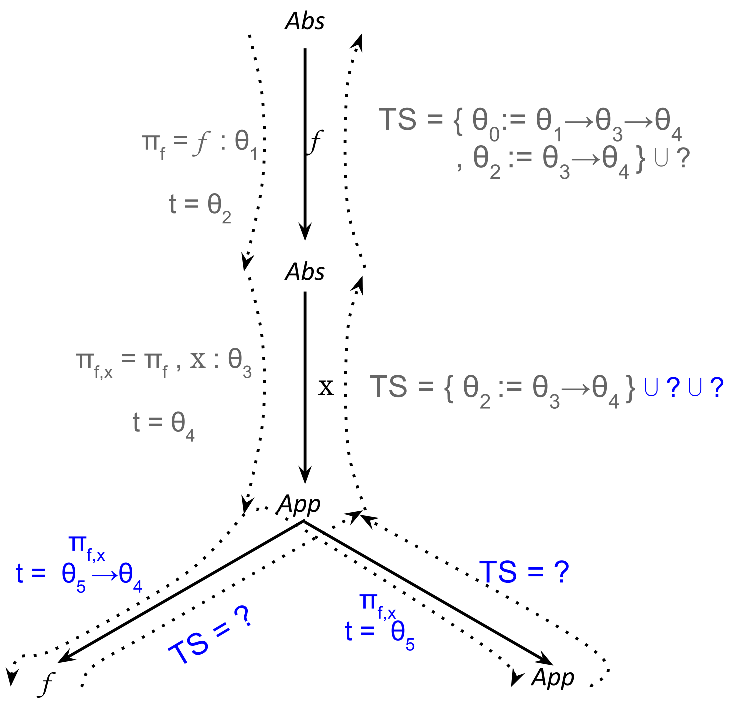
As we assume for this term the type θ, we must replace by $\theta_1 \to \theta_2$ and we define the values of the inherited attributes: the context will contain the pair $f : \theta_1$ and the asummed type on child will be θ_2. The substitution will contain $\theta : \theta_1 \to \theta_2$ and the substitution obtained from child. In the graphics we'll complete the syntesized attribute as we calculate the attributes in subterms.

In the next step, the node is an abstraction too, so we perform the same procedure:


Abs
$\pi_f = f : \theta_1$
$t = \theta_2$
f
TS = { $\theta_0 := \theta_1 \to \theta_3 \to \theta_4$
, $\theta_2 := \theta_3 \to \theta_4$ } ∪ ?
Abs
$\pi_{f,x} = \pi_f , x : \theta_3$
$t = \theta_4$
x
TS = { $\theta_2 := \theta_3 \to \theta_4$ } ∪ ?
App

θ_2 will be replaced by $\theta_3 \to \theta_4$, we add to the context $x : \theta_3$ and the type asummed to the subterm is θ_4. We update the substitution in the first node adding the obtained one in this subterm, and we perform the replacement $\theta_2 := \theta_3 \to \theta_4$ on it, so we can ensure that the variables occurring on the left side of substitution don't occur on the right side. We perform unification of types when necessary.

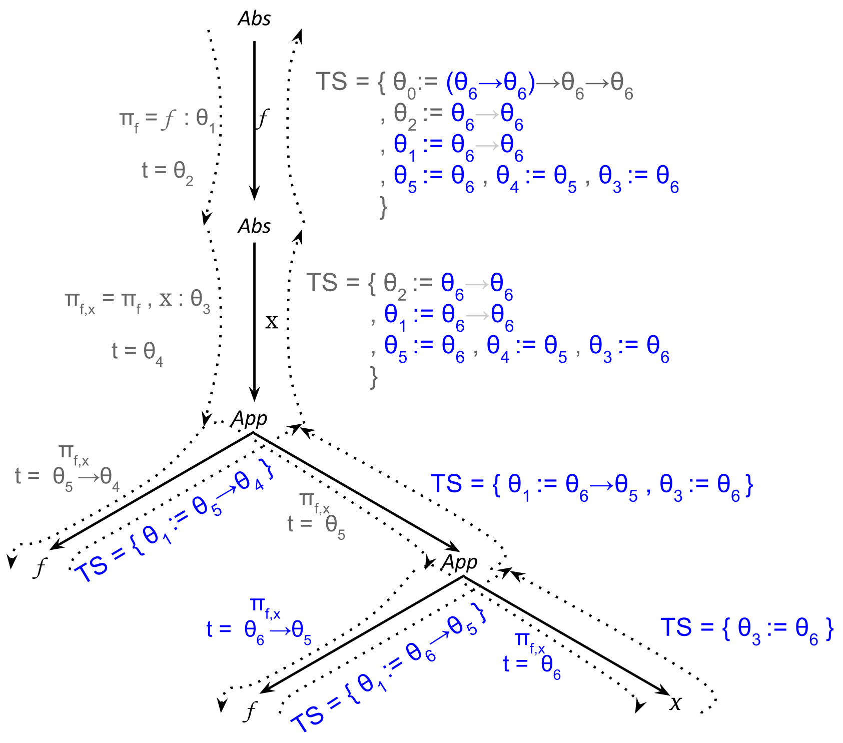
Now we analize the next node which is an application. According to the corresponding typing rule, if the type of the application is θ_4, the operator should have an arrow type $\theta_5 \to \theta_4$ while the operand should have θ_5. We update the values:



The substitution on an application will be the union of the substitutions in subterms, so we don't complete this yet.

Now we have two terms to analize. The left is the identifier with variable y and asummed type is $\theta_5 \to \theta_4$ but in the context we have $f : \theta_1$. So, we need to add $\theta_1 := \theta_5 \to \theta_4$ to the substitution.

The right term is another application with type θ_5, so we realize the same procedure:



In the last image we obtain the substitution resulting of type inference. We can see that when two substitution are concatenated the replacements are performing online and if is necessary, unification is performed. $\theta_5 \to \theta_4$ is unified with $\theta_6 \to \theta_5$ so we add $\theta_5 := \theta_6$ and $\theta_4 := \theta_5$ to the substitution. Substitutions and unification was implemented using Attribute Grammars too.

The final result is the substitution on the first node of the term. And the inferred type is the result of applying that substitution on the initial type θ. So we obatin the type $(\theta_6 \to \theta_6) \to \theta_6 \to \theta_6$.

3. Conclusions

In the first section we explained the "repmin" problem, which is introduced in several papers about Attribute Grammars. A way of thinking about syntax and semantics is by initial algebras and we are studying it at the moment using category theory. Because of this we try to apply some of those concepts in the implementation of repmin problem and we obtained an equivalent solution to those found in the bibliography. Nevertheless there are concepts that we need to understand in a better way.

To apply the power of Attribute Grammars we chose Type inference of lambda calculus because we are studying it in our phd and we think that is an interesting problem to solve. We found the main difficulty when we implemented unification and substitution and finally we found a solution using Attribute Grammars too, but we didn't explain this solution in this report due to the lack of time.

Finally, we decided to write this “attempt of report” to practice writing at the beginning our phd and we know that we need to improve this. We apologize for the many mistakes that you might have found.