Doucmentation of the FOSA ruler project

Mark Snyder, mark.snyder@gmail.com
John van Schie, jcschie@cs.uu.nl

1 Language Design

There were two major motivations behind the language's design. The first motivation is to embrace common language paradigms and terminology. By doing this we hope to minimize the ramp up time for new users. The second motivation is to create a malleable syntax that can be easily adapted to different applications. For this reason, we added a directives hook that makes it possible to annotate rulesets, rules, and judgments with application specific information.

Another major goal of this language is to address some of the software engineering challenges brought about by the ever increasing EHC code size. We use Java style inheritance to decompose different versions of the system and a combination of the *private* directive and the *uses* clause to manage data visibility between layers.
In our prototype implementation interfaces and implementations are tied to each other by name and the inheritance relationship is expressed in the interface specification itself. One possible improvement to our prototype would be to extract the inheritance information and keep the inheritance hierarchy separate from the interface and implementation files. By externalizing this information we make it possible to experiment with different layer assemblies.

To make the above discussion concrete, we show a portion of the Known type layer specification below.

```
layer Known extends Equation
  interface checkSubs
    params
    uses
      in    iSubs     : FIOut
      out   oSubs     : FIOut
    pattern "iSubs ~> oSubs"
```

2 Haskell Output

2.1 Haskell output overview

Currently our compiler is limited to generating Haskell code. To be more precise, for each ruleset that complies to an interface, a function is generated. Such a function has as many cases as there are rules within the ruleset.

So when we have a ruleset definition like this:

```
ruleset typeInference implements exprTypeInference
  rule typeInference_Int = ...
  rule typeInference_Var = ...
```

will be translated to the following haskell function skeleton:

```
exprTypeInference (CInt x) = ...
exprTypeInference (CVar x) = ...
```

Generating Haskell presents its own challenges. Unlike AG code fragments, Haskell code fragments, such as pattern matches, are order sensitive. For this reason we preserve the order of rules within a ruleset when we generate Haskell. The problem becomes more complicated when a ruleset is merged with its cousin from the layer below. In particular, it isn't clear where to place newly introduced rules within the order of the lower ruleset.

In the most pathological cases, an explicit rule sequence will have to be provided. In our prototype we preserve the order of lower layers, inserting new layers in the "holes" between pre-existing rule positions. We also provide an "overwrite" directive which makes it possible to completely replace a ruleset in the lower layer with the upper layer's version.

One possible extension to the existing language definition would be additional syntax that allows users to specify the exact rule order in the upper layer. The decision to extend the language in this way must be balanced against the aesthetic cost of cluttering up the syntax.

2.2 The Haskell translation scheme

The translation scheme from our ruler-like language is implemented via an attribute grammar (`PrettyHaskell.ag`), so changes to the Haskell output should be made there. The scheme is quite straightforward and can be summarized as follows.

For the *post judgment* of a rule, we create a new case with the `In` and `InOut` parameters (and their values) as input parameters. The body of the case consists of a let expression. The bindings of the let expression are all *pre judgments*. For each pre judgment a binding is created. The value of the binding will be

the `InOut` and `Out` parameters tupled of the judgment and the `In` and `InOut` parameters will be the parameters of the call to the interface of the pre judgment. That leaves us with the result of the let-expression. This wil simply be the `InOut` and `Out` parameters of the post judgment tupled.

This can be seen in the following example

```
-- File Equation.inf
layer Equation
  interface checkSubs
    params
         in    iSubs     : FIOut
         out   oSubs     : FIOut
      pattern "iSubs ~> oSubs"

  interface fitsIn
    params
         in    guard     : Bool
         in    typeLeft  : Ty
         in    typeRight : Ty
         out   subs      : FIOut
      pattern "guard ==> typeLeft ~= typeRight"
```

The code above specifies the shape of rules in the Equation layer. It serves the same purpose as the interface construct in Java. This specification describes two functional abstractions, checkSubs which is used to 'hide' a functioncall, and the fitsIn interface, which is the interface for the unification of two types. The patterns noted in the interface, although required by the parser, are not used yet. Their purpose is to specify a more compact concrete syntax through which judgments can be specified.

```
-- File Equation.impl
implementation of Equation
 ruleset fitsIn implements fitsIn
  rule fitsIn_Ty_Con implements fitsIn
    post fitsIn.R
           | guard     = "x==y"
           | typeLeft  = "Ty_Con x"
           | typeRight = "Ty_Con y"
           | subs      = "emptyFO {foTy = typeRight}"

  rule fitsIn_Ty_Any implements fitsIn (symmetric)
    post fitsIn.R
           | typeLeft  = "Ty_Any"
           | typeRight = "typeRight"
           | subs      = "emptyFO {foTy = typeRight}"
```

```
rule fitsIn_Ty_App implements fitsIn
  pre fitsIn.PreArgs
        | typeLeft  = "ta2"
        | typeRight = "ta1"
        | subs      = "ffo"
      fitsIn.PreFun
        | typeLeft  = "tf1"
        | typeRight = "tf2"
        | subs      = "afo"
      checkSubs.CheckSubs
        | iSubs     = "[ffo,afo,emptyFO {foTy = Ty_App (foTy ffo) (foTy afo)}]"
        | oSubs     = "res"
  post fitsIn.R
         | typeLeft  = "Ty_App tf1 ta1"
         | typeRight = "Ty_App tf2 ta2"
         | subs      = "res"
```

In the above code, we see that three rules: fitsIn_Ty_Con, fitsIn_Ty_Any and fitsIn_Ty_App are specified. For each interface file there must be a corresponding implementation file (.impl). We briefly review the meaning behind these rules in the paragraphs below.

fitsIn_Ty_Con tries to unify two constructors. The special parameter guard is used as a keyword for the Haskell prettyprinter. Every parameter that is named guard will be added to the guard expression of the function case. So for this rule we expect a case that will be guarded by $x == y$ and as body has an let expression without bindings.

fitsIn_Ty_Any has the directive *symmetric*. This means that if the input of the case is mirrored, we get another desired case. In this case we want one case to match the Ty_Any constructor to the left and one constructor to match the case where the Ty_Any is at the right. The symmetric directive accomplishes this.

fitsIn_Ty_App is a rule with pre judgments. One of these judgments is of the interface `checkSubs`. So for this rule we expect a let expression with three bindings, two recursive and one binding that is utilizing the checkSubs function.

So we run our compiler tool and get the following output. Please note that the order of the rules is preserved.

```
-- File Equation.hs
module Equation where
fitsIn typeLeft@(Ty_Con x) typeRight@(Ty_Con y) | x==y  =
  let
  in ( emptyFO {foTy = typeRight} )
```

```
fitsIn typeLeft@(Ty_Any) (typeRight)  =
  let
  in ( emptyFO {foTy = typeRight} )
fitsIn (typeRight) typeLeft@(Ty_Any)  =
  let
  in ( emptyFO {foTy = typeRight} )

fitsIn typeLeft@(Ty_App tf1 ta1) typeRight@(Ty_App tf2 ta2)  =
  let
    ( ffo ) = fitsIn (ta2) (ta1)
    ( afo ) = fitsIn (tf1) (tf2)
    ( res ) = checkSubs ([ffo,afo,emptyFO {foTy = Ty_App (foTy ffo) (foTy afo)}])
  in ( res )
```

We will now look at how the Equation layer is extended in order to specify the know type rules that correspond to the unification logic performed by EHC version 2. Our interface does not need to be extended (no new parameters are needed) but we do declare those parameters that the Known layer will use from the layer below (all parameters). This redundancy, the copying of information from the layer below, is introduced so that developers won't be forced to inspect several files in the interface hierarchy in order to determine what parameters are available to the implementation. Please inspect the `Known.inf` file in order to see how inheritance and parameter usage is expressed.

The implementation refinements for the known types unification logic are displayed below. Only two rules are specified in this implementation - one which is a new addition to the ruleset and the other which overwrites the original rule in the Equation layer. The latter is annotated with the *overwrite* directive. The rest of the rules are inherited, without interference, from the Equation layer.

```
implementation of Known
 ruleset fitsIn implements fitsIn

  rule fitsIn_Ty_Var_Ty_Var implements fitsIn
    post fitsIn.R
      | guard     = "v1==v2"
      | typeLeft  = "Ty_Var v1"
      | typeRight = "Ty_Var v2"
      | subs      = "emptyFO {foTy = typeLeft}"

  rule fitsIn_Ty_App implements fitsIn (overwrite)
    pre fitsIn.PreFun
          | typeLeft  = "tf1"
          | typeRight = "tf2"
          | subs      = "ffo"
        fitsIn.PreArgs
          | typeLeft  = "(foCnstr ffo) |=> ta2"
```

```
          | typeRight = "(foCnstr ffo) |=> ta1"
          | subs      = "afo"
        checkSubs.CheckSubs
          | iSubs     = "[ffo,afo
                         ,emptyFO { foTy = Ty_App ((foCnstr afo) |=> foTy ffo)
                                                  (foTy afo)
                                  , foCnstr = (foCnstr afo) |=> (foCnstr ffo)}]"
          | oSubs     = "res"
    post fitsIn.R
           | typeLeft  = "Ty_App tf1 ta1"
           | typeRight = "Ty_App tf2 ta2"
           | subs      = "res"
```

The compiler output for this implementation is shown below.

```
module Known where

fitsIn typeLeft@(Ty_Var v1) typeRight@(Ty_Var v2) | v1==v2  =
  let
  in ( emptyFO {foTy = typeLeft} )

fitsIn typeLeft@(Ty_Con x) typeRight@(Ty_Con y) | x==y  =
  let
  in ( emptyFO {foTy = typeRight} )

fitsIn typeLeft@(Ty_Any) (typeRight)  =
  let
  in ( emptyFO {foTy = typeRight} )
fitsIn (typeRight) typeLeft@(Ty_Any)  =
  let
  in ( emptyFO {foTy = typeRight} )

fitsIn typeLeft@(Ty_App tf1 ta1) typeRight@(Ty_App tf2 ta2)  =
  let
    ( ffo ) = fitsIn (tf1) (tf2)
    ( afo ) = fitsIn ((foCnstr ffo) |=> ta2) ((foCnstr ffo) |=> ta1)
    ( res ) = checkSubs ([ffo,afo
                         ,emptyFO { foTy = Ty_App ((foCnstr afo) |=> foTy ffo)
                                                  (foTy afo)
                                  , foCnstr = (foCnstr afo) |=> (foCnstr ffo)}])
  in ( res )
```

Note the order. The new rule gets translated as a case at the top, while the overwritten function gets its original postion.
All these examples can be found in *.impl and *.inf files of this directory.

2.3 Generalizing the Output Target

We realize that programming an AG so that it generates Haskell output directly is not an ideal solution. This decision was motivated by time constraints and not design principles. If future work is to be done on this language, we think that adapting Ruler's template framework would be a worthwhile investment.

3 Restoring Type Safety

One of the most obvious differences between our prototype and the Ruler language is our template based approach. We allow function calls, boolean expressions, etc. to be expressed as text. This choice was an expedient one based largely upon time constraints and it comes at a high price, since we give up front-end type checking in the process. Aside from time constraints, this approach was taken in reaction to Ruler's relation construct which we found peculiarly cumbersome.

Another possible improvement to this prototype would be to replace the textual code fragments with Haskell expressions - only supporting a small subset of Haskell, of course. In order to decide whether this is a viable idea, one would have to review the unification code for more complex layers and determine whether these rules can be expressed naturally in a tiny subset of Haskell.