Proxima Extension Manual

Joost Verhoog
jverhoog@cs.uu.nl

February 7, 2005

1 Introduction

In this manual we describe how to extend the Proxima generic editor. The running example that we use will be editing AG datatypes in Proxima. We assume knowledge of Haskell, the UU Attribute Grammar system and parser combinators.

The first thing that you need to do is install Proxima. The homepage for the Proxima project can be found at http://www.cs.uu.nl/research/projects/proxima/. Follow the installation manual to install Proxima on your local system. All file and folder descriptions that we use will be relative to the location that you install Proxima to.

Extending Proxima basically consists of the following three steps:

1. Extend the existing document type with the document type that you want to edit.
2. Write a presentation for this extension.
3. Write the inverse of this presentation; the parser.

We will discuss these three steps in the following three chapters. A last chapter is dedicated to using knowledge about the document to provide the user with specific information.

2 Extend the document type

The document type can be found in proxima/DocumentType.hs. The .hs extension suggests that it is a haskell-file, but in fact it is haskell extended with two extra features:

1. Data constructors have labelled fields, just like UU-AG datatypes.
2. You can add extra labelled fields between curly brackets at the end of a construcor declaration, which will be insterted as the first fields of the constructor.

We choose to add an extra declaration to the document type, the AG declaration.

```
data Decl = ...
          | AGDecl elems : [AGElem]
            {idD : IDD idP0 : IDP idP1 : IDP idP2 : IDP}
```

This AG declaration hase some AG Elements as its children. It hase four extra children. These children store extra information needed by Proxima. We will come back to this in section 3. For now, note that there is one IDD child, and a number (in this case 2) of IDP children.

An AG Element is an AG Data-type which consist of a datatype name and some alternatives. The AG Data-type can be found in the sources of the AG System. We use the concrete syntax, to make presentation and parsing easier.

```
data AGElem = AGData name : Ident
                     alts   : [AGAlt]
              {idD : IDD idP0 : IDP}
```

Again, we have an extra IDD field, and some IPD fields.

An alternative has a name, and some fields, which consist of a name and a type.

```
data AGAlt = AGAlt name :    Ident
                   fields :  [AGField]
             {idD : IDD idP0 : IDP}
data AGField  = AGField name : Ident
                        tp   : Ident
             {idD : IDD idP0 : IDP}
```

From the document type, several other files can be generated. All these files have `_Generated` in their names. All these files have a line

—– GENERATED PART STARTS HERE. DO NOT EDIT ON OR BEYOND THIS LINE —–

in them, above which you can edit freely. The generated part is put below this line. The generator program can be found in `proxima/src/generator`. Run `make` in this folder, and then `generate DocumentType.hs`. This generates, for example, AG Datatypes for the datatypes that you have just created:

```
DATA Decl
       ...
   |   AGDecl idD : IDD idP0 : IDP idP1 : IDP idP2 : IDP elems : List_AGElem
DATA AGElem
   |   AGData idD : IDD idP0 : IDP name : Ident alts : List_AGAlt
   |   HoleAGElem
   |   ParseErrAGElem Node Presentation
DATA AGAlt
   |   AGAlt idD : IDD idP0 : IDP name : Ident fields : List_AGField
   |   HoleAGAlt
   |   ParseErrAGAlt Node Presentation
DATA AGField
   |   AGField idD : IDD idP0 : IDP name : Ident tp : Ident
   |   HoleAGField
   |   ParseErrAGField Node Presentation
```

As you can see, the IDD and IDP fields written between curly brackets before, have been added as the first children of the alternatives. For each datatype, two extra alternatives have been added:

1. A Hole, to indicate that there is currently no value. This is used while editing to serve as a placeholder for a value that will be typed there.
2. A ParseErr, to indicate that a parse error has originated at this point. It saves the old tree and the old presentation, so they can be reused after the parse error has been resolved.

For each list in the document type, two extra datatypes are generated, a `List_` which can be either a true list, or a hole or a parse error, and an actual list `ConsList_`, with the usual Cons and Nil alternatives. In our document type we have three lists, so the following six datatypes are generated:

```
DATA List_AGElem
    |   List_AGElem idd : IDD elts : ConsList_AGElem
    |   HoleList_AGElem
    |   ParseErrList_AGElem Node Presentation
DATA ConsList_AGElem
    |   Cons_AGElem head : AGElem tail : ConsList_AGElem
    |   Nil_AGElem
DATA List_AGAlt
    |   List_AGAlt idd : IDD elts : ConsList_AGAlt
    |   HoleList_AGAlt
    |   ParseErrList_AGAlt Node Presentation
DATA ConsList_AGAlt
    |   Cons_AGAlt head : AGAlt tail : ConsList_AGAlt
    |   Nil_AGAlt
DATA List_AGField
    |   List_AGField idd : IDD elts : ConsList_AGField
    |   HoleList_AGField
    |   ParseErrList_AGField Node Presentation
DATA ConsList_AGField
    |   Cons_AGField head : AGField tail : ConsList_AGField
    |   Nil_AGField
```

Now we need to specify how this document type is presented. We will describe this in the next section.

3 Presentation

The presentation of the document is written in AG code, in the file

`proxima/src/presentation/PresentationAG.ag`

We need to specify the attribute pres, which is the presentation of our document. For this we have several functions:

- `row' :: [Presentation] -> Presentation`, put several presentations next to eachtother.
- `key :: IDP -> String -> Presentation`, the presentation of a keyword.
- `sep :: IDP -> String -> Presentation`, the presentation of a separator.

At each node, we do the following with its presentation:

- `loc`, to save a copy of the tree for reuse.
- `parsing` or `structural` to indicate whether or not the presentation can be edited
- `presentFocus`, to present the focus.

For our document type, we want, for example, to present the document describing a binary tree with integers in its leafs as

```
AG {
        DATA  Tree
         |  Branch  left:Tree right:Tree
         |  Leaf    int:Int
}
```

Now the important thing to observe is that the whitespace used is not part of the specification of our presentation. We want to present it with the whitespace that the user has specified. That is why we save whitespace in the document, and reuse it in our presentation. The whitespace is saved in the IDP fields of our datatype. Indicating it in our example with underscores (`_`), we get:

```
_AG_{
        _DATA  _Tree
         _|  _Branch  _left_:_Tree _right_:_Tree
         _|  _Leaf    _int_:_Int
_}
```

Observe that each identifier (e.g. Tree), each keyword (e.g. DATA) and each node in our tree has to save the whitespace before it. Whitespace information is stored in IDP's (IDentifiers of Presentation). So, AGDecl needs to save 3 IDP's, (`_AG_{ ... _}`), AGElem 1 (`_DATA ...`), AGAlt 1 (`_| ...`), AGField 1 (`... _: ...`) and Ident 1 (`_...`). This is exactly how many IDP's were specified in the document type for these nodes.

Now we continue with the specification of the presentation of our extention to the document type. For the AG Declaration we get:

```
SEM Decl
  | AGDecl
    lhs.pres = loc (DeclNode @self @lhs.path)
      $ parsing
      $ presentFocus @lhs.focusD @lhs.path
      $ row' ([key (mkIDP @idP0 @lhs.pIdC 0) "AG"
               , sep (mkIDP @idP1 @lhs.pIdC 1) "{"]
               ++ @elems.press
               ++ [sep (mkIDP @idP2 @lhs.pIdC 2) "}"])
```

With this we describe that the presentation of an AG Declaration is the keyword AG, the separator , the presentation of the elements and a separator . The function mkIDP reuses the whitespace information that has been saved in de IDP child if possible, and otherwise genererates new whitespace.

We present the focus at the current location with `presentFocus`. The presentation can be edited, thus `parsing`. We save the old tree in a DeclNode, with the function `loc`. These Node datatype for each node of the document are generated by the generator.

Now the rest of the presentation follows exactly the same pattern:

```
SEM AGElem
  | AGData
    lhs.pres = loc (AGDataNode @self @lhs.path)
      $ parsing
      $ presentFocus @lhs.focusD @lhs.path
      $ row′ ([key (mkIDP @idP0 @lhs.pIdC 0) "DATA"
             , @name.pres]
              ++ @alts.press)
SEM AGAlt
  | AGAlt
    lhs.pres = loc (AGAltNode @self @lhs.path)
      $ parsing
      $ presentFocus @lhs.focusD @lhs.path
      $ row′ ([key (mkIDP @idP0 @lhs.pIdC 0) "|"
             , @name.pres]
              ++ @fields.press)
SEM AGField
  | AGField
    lhs.pres = loc (AGFieldNode @self @lhs.path)
      $ parsing
      $ presentFocus @lhs.focusD @lhs.path
      $ row′ [ @name.pres
             , key (mkIDP @idP0 @lhs.pIdC 0) ":"
             , @tp.pres]
```

Now the presentation is finished, and we can generate Haskell source for it by running `make` in `proxima/src/presentation`.

Now that we can present our document, we need to specify the inverse: parsing. We discuss parsing in the next section.

4 Parsing

The parser is written in proxima/src/presentation/ProxParser.hs. We extend the parser to cope with the new document type. On parsing, we reuse the old document . For this, the generator has generated reuse function. For each constructor, a corresponding reuse function is defined. Its first parameter takes a list of all whitespace tokens that will be reused, in tokenNodes. As other parameters it takes a maybe for each field of the constructor. If a field must be reused, it can be passed, and Nothing is used for not reusing the field.

Te write the parser, we have the following functions available:

1. `pKey` parses a keyword, and returns the whitspace before it.
2. `parseIdent` parses an identifier.
3. `parseUIdent` parses an identifier starting with an uppercase character.
4. `pList` parses a list.
5. Generated functions `toConsList_` for each generated `ConsList_` to convert a list to a `ConsList_`.

We arrive at the following parser for our AG Declaration:

```
parseDecl = ...
<|> (λtk1 tk2 elems tk3 →
        reuseAGDecl [tokenNode tk1, tokenNode tk2, tokenNode tk3]
                    Nothing
```

```
                              (Just $ tokenIDP tk1)
                              (Just $ tokenIDP tk2)
                              (Just $ tokenIDP tk3)
                              (Just elems))
                  <$> pKey "AG"
                        <*> pKey "{"
                        <*> parseList_AGElem
                        <*> pKey "}"
```

The rest of our parser follows exactly the same pattern:

```
parseAGElem =
  (λtk1 name alts →
    reuseAGData [tokenNode tk1]
                Nothing
                (Just $ tokenIDP tk1)
                (Just name)
                (Just alts))
  <$> pKey "DATA"
        <*> parseUIdent
        <*> parseList_AGAlt
parseAGAlt =
  (λtk1 name fields →
    reuseAGAlt [tokenNode tk1]
               Nothing
               (Just $ tokenIDP tk1)
               (Just name)
               (Just fields))
  <$> pKey "|"
        <*> parseUIdent
        <*> parseList_AGField
parseAGField =
  (λname tk1 tp →
    reuseAGField [tokenNode tk1]
                 Nothing
                 (Just $ tokenIDP tk1)
                 (Just name)
                 (Just tp))
  <$> parseIdent
    <*> pKey ":"
    <*> parseUIdent
```

The parsers for lists can reuse their whitspace, but we don't do that here.

```
parseList_AGAlt =
  (λelems →
    reuseList_AGAlt []
                    Nothing
                    (Just $ toConsList_AGAlt elems))
  <$> pList parseAGAlt
parseList_AGField =
  (λelems →
    reuseList_AGField []
                      Nothing
```

```
                              (Just $ toConsList_AGField elems))
            < $ > pList parseAGField
```

Now we can present and parse our document, and we have a working editor. In the next section, we will improve our presentation to indicate errors.

5 Improving the presentation

We can use the fact that we know the structure of the editor to our advantage, and present extra information that can be derived from the tree to help the user. Let's take the types of fields as an example. When the user gives a field a type that has not been defined, we want to indicate this. For this, we collect all defined datatypes in attributes, and pass them down. (We assume that we have the datatypes from the prelude in `preludeDatas`)

```
ATTR List_AGElem ConsList_AGElem AGElem
       [ | | datas USE{++}{[]} : {[String]}]
SEM AGElem
  |  AGData lhs.datas = [ @name.str]
ATTR List_AGElem ConsList_AGElem AGElem
       List_AGAlt    ConsList_AGAlt    AGAlt
       List_AGField ConsList_AGField AGField
       [alldatas : {[String]} | | ]
SEM Decl
  |  AGDecl elems.alldatas = preludeDatas ++ elems.datas
```

Now we know all possible types that can be used at an `AGField`, and we can define a function that gives an error squiggle under the type if it is not defined. (squiggly is a function that takes a color, and does this)

```
SEM AGField
  |  AGField loc.tpCheck = if @tp.str ∈ @lhs.alldatas
                           then id
                           else squiggly error3Color
```

We apply this function to our presentation, modifying the presentation of an AGField to

```
SEM AGField
  |  AGField
       lhs.pres = loc (AGFieldNode @self @lhs.path)
          $ parsing
          $ presentFocus @lhs.focusD @lhs.path
          $ row' [ @name.pres
                 , key (mkIDP @idP0 @lhs.pIdC 0) ":"
                 , @tpCheck @tp.pres]
```