Generic Selections, 2010

Thijs Alkemade and Alessandro Vermeulen,
Department of Information and Computing Sciences,
Utrecht University, 2010.

November 15, 2010

1 Introduction

When parsing an expression to an abstract syntax tree (AST), it is convenient to store the origin of the elements. This allows better error reporting, pretty printing and converting back and forth from a textual representation. However, cluttering the datatype of the AST with this meta-data is often not desirable. The Annotations package (Steenbergen et al. [2010] and Steenbergen [2010]) provides a way to store the boundaries a particular piece of the code had in the original source within the AST .

This paper describes an attempt at answering the question posed in Steenbergen et al. [2010] section 10.1. It is quoted below for convenience and answered in section 3.

> Dealing with changes to the tree is an interesting problem that we have not addressed yet. In particular, we have to make sure that the text positions stored in the annotated tree stay correct after an edit. A possible solution is to not store position information but the exact source code responsible for each subtree. From this, position information can be inferred.

In figures 1 and 2 you see an expression data structure and an example parse result as you may expect in vanilla Annotations.

2 Research Question

It might be required for some applications to transform the AST, for example for inserting, removing or refactoring code. This requires an update in the annotated tree. However, as each node of the tree stores its boundaries, any update which changes the size of a piece of code causes updates in all further parts of the tree. This can cause a lot of overhead if the updates are done often.

```
data ExprF rT
    = Add  rT rT
    |  Sub  rT rT
    |  Mul  rT rT
    |  Div  rT rT
    |  Num Int
  deriving (Eq,Show)
```

Figure 1: An expression data structure written in fix point notation

```
*ExprParser> parseExpr "1+ 5"
Right (In {out = Ann
  (Bounds {leftMargin = (0,0)
          ,rightMargin = (4,4)})
  (Add (In {out = Ann (Bounds {leftMargin = (0,0)
                              ,rightMargin = (1,1)})
                      (Num 1)})
       (In {out = Ann (Bounds {leftMargin = (2,3)
                              ,rightMargin = (4,4)})
                      (Num 5)})
  )}
)
```

Figure 2: An example parse result for the expression "1+ 5"

To avoid this problem, our goal was to investigate if it is possible to store the original source code instead of the boundaries in the annotations. This way the origin of each node is still present, but any update which changes the size of a node will not trigger updates in all other parts of the tree.

3 Research Contribution

In order to solve the aforementioned problem we changed the annotations. We set out to annotate the AST with just the source Tokens. However, the most important exposed functionality of the library relies on this positional information. In order to maintain this functionality we added more information to our annotations.

Next to the source tokens we store the prefix tokens. Those are the tokens that proceed the current expression. The storing of these tokens is handled by the provided parser combinators.

```
*ExprParser> parseExpr "1+ 5"
Right (In {out = Ann
  (Bounds {source = [TNum 1
                    ,TPlus
                    ,TNum 5]
               ,previous = []})
      (Add (In {out = Ann (Bounds {source = [TNum 1]
                                  ,previous = []})
                         (Num 1)})
         (In {out = Ann (Bounds {source = [TNum 5]
                                  ,previous = [TPlus
                                              ,TNum 1]})
                         (Num 5)})))})
```

With this information we recreated the *leftMargin* and *rightMargin* functions.

```
leftMargin :: Bounds s → Range
leftMargin (Bounds s bs) = (length bs,length bs)
rightMargin :: Bounds s → Range
rightMargin (Bounds s bs) = (length bs + length s
                             ,length bs + length s)
```

In this process however, the whitespace information in the ranges isn't available anymore. This is due to the processing that is done earlier where whitespace tokens are thrown away.

A more relevant issue with this change is that this is still not a solution. As there is local storage of positional information, in this case the source, the relevant nodes have to be updated to contain the new source.

Suppose that we add something at the beginning of our expression. In order to keep our local information correct we need to traverse the whole original AST to update the prefix tokens.

3.1 Storing the difference

Another solution might be to store relative positional information and the source. In this case updating the AST is relatively easy. Inserting can be done by adding a new expression element to the three without requiring any further house holding.

The downside of this solution is that lookups on the AST will be more expensive as you will have to calculate the absolute values from the diffs when traversing the tree. Also, as said before the currently available helper functions rely on having positional information available locally. Would you desire to keep the same functionality, you would need to come up with a way to compute this information. For this you most probably need the whole tree, and some method of finding the local node back again.

As we have no existing real-world examples of how the library is currently used it is difficult to provide a comparison of whether this solution would be an improvement or not.

4 Future work

4.1 Two separate annotations

As we have seen in the solutions discussed above, you either have to do a lot of work when updating the tree or a lot of work when performing a lookup on the tree. Therefore we propose a solution that would hopefully provide the best of both world plus a bit extra.

The idea is that you have one annotation that contains only the positional information and an annotation that contains only the source information. In this source annotated AST you can easily perform transformations as inserting an extra function, or perform other forms of refactoring.

Now if you would want to do a lot of transformations you'd transform your positionally annotated tree to a source annotated tree, perform your transformations, and convert back to the positionally annotated tree again.

4.2 Continuations

Another suggestion is to define transformations on the three in a continuation passing style, also described in Appel and Jim [1989]. This may provide the possibility of fusing (Coutts et al. [2007]) the updates. Resulting in less overhead for the updating of the tree structure.

One should take care to think of the consequences of sequencing two tree transformations as the first transformation performed may change the behaviour of the second transformation.

4.3 Providing a way to handle tree updates

The current Annotations library, Steenbergen [2010], does not provide any functionality to update a tree. In order to provide a satisfactory mechanism of updating the tree one should very likely first find solutions for the problems described above.

5 Related Work

It seems that there has not been a lot of research into generically annotating ASTs. An other example of the usage of Annotations is Generic Selections by Visser and Löh. Visser and Löh [2010] They present a mechanism to generically persist data in which they a way to perform actions at annotations.

6 Conclusion

As there are no examples where the library is used for updating trees, we can not compare which improvements outweigh their downsides. The library as it is relies a lot on the representation using ranges, which can not be translated efficiently to a form where updates can be done in constant time. We have suggested a number of improvements that can be used, depending on what is required of the tree.

References

A. W. Appel and T. Jim. Continuation-passing, closure-passing style. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '89, pages 293–302, New York, NY, USA, 1989. ACM. ISBN 0-89791-294-2. doi: http://doi.acm.org/10.1145/75277.75303. URL `http://doi.acm.org/10.1145/75277.75303`.

Duncan Coutts, Roman Leshchinskiy, and Don Stewart. Stream fusion: from lists to streams to nothing at all. *SIGPLAN Not.*, 42:315–326, October 2007. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1291220.1291199. URL `http://doi.acm.org/10.1145/1291220.1291199`.

Martijn van Steenbergen, 2010. URL `http://hackage.haskell.org/package/Annotations-0.1.1`.

Martijn van Steenbergen, José Pedro Magalhães, and Johan Jeuring. Generic selections of subexpressions. Technical Report UU-CS-2010-016, Department of Information and Computing Sciences, Utrecht University, 2010.

Sebastiaan Visser and Andres Löh. Generic storage in haskell. In *Proceedings of the 6th ACM SIGPLAN workshop on Generic programming*, WGP '10, pages 25–36, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0251-7. doi: http://doi.acm.org/10.1145/1863495.1863500. URL `http://doi.acm.org/10.1145/1863495.1863500`.