Running Ometa Parsers Backwards for Source to Source Translation

Ted Kaehler and Alessandro Warth

This material is based upon work supported in part by the National Science Foundation under Grant No. 0639876. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

VPRI Memo M-2008-001

Viewpoints Research Institute, 1209 Grand Central Avenue, Glendale, CA 91201 t: (818) 332-3001 f: (818) 244-9761

Running OMeta Parsers Backwards for Source to Source Translation

by Ted Kaehler and Alessandro Warth

http://www.vpri.org/pdf/ParseBackwards-RM-2008-001.pdf

This material is based upon work supported by the National Science Foundation under Grant No. 0639876.

Abstract

Translation of source code from one computer language to another is normally done by using a parser to convert the text to an Abstract Syntax Tree (AST), and then a pretty printer to convert the tree to the target language. Going in both directions between two languages therefore requires four components: two parsers and two pretty printers. We present a technique for running the rules of a parser in the reverse direction to convert an AST to source code. This makes it possible to translate programs back and forth between two languages using only two parsers. Some of the parser's rules must be written in a restricted form, so that our system can automatically extract templates from them. The parser must be written using a PEG-based parser generator so we can analyze the parse tree of a rule.

View Source Code in Any Language

A valuable property of a programming environment is the ability to display the source code in any of several languages. This is one of the goals of the STEPS project [2]. The code is represented internally as an Abstract Syntax Tree (AST) or as code in a single reference language. The running example in this paper uses Squeak Smalltalk as the reference language. The alternative language is a scripting language called **Boldly**Code. Designed as the scripting language for a HyperCard-like personal information system, **Boldly**Code uses text emphasis to denote syntactic forms, and is derived from Mark Lentczner's Glyphic Codeworks [3]. In **Boldly**Code, message selector keywords are in bold, unary messages are underlined, and comments are in italics.

Fgure 1 below shows the normal way that systems do translation between source code languages. The path from **Boldly**Code to Smalltalk is separate from the path in the reverse direction. Two parsers and two pretty printers are needed.

The parsers in our prototype are written in OMeta [1], an object-oriented language for pattern matching. OMeta is based on a variant of Parsing Expression Grammars [4], and the approach described in this paper should be applicable to parsers written using any PEG-based parser generator.

Figure 2 shows the simplification that results if the parsers are able to run in reverse. The pretty printers are not needed for this step. Adding a parser for a third language enables translation between it and any other language already present. If the formats of the trees are identical, a tree from the third language can be run backwards through either of the other two parsers. If the formats are different, the tree will need to be transformed before it can be reverse parsed.

An advantage is that when one of the languages is modified, and its parser changes, no corresponding change needs to be made in its pretty printer, since there is no pretty printer. For a given language, its pretty printer and its parser are very different in structure and code. This is a weakness, since on some level they are inverses of each other. By making the parser run backwards and

eliminating the pretty printer, less code must be written, it is easier to maintain, and the solution is more beautiful.

In our example system, the step from Smalltalk (on the right) to the Abstract Syntax Tree is accomplished by the Smalltalk parser. The final representation of the parse tree is an array of arrays, in the format of Yoshiki Ohshima's S-expressions. Each array S-expression is one node in a tree. The list elements in an S-expression are its subnodes. Each node also has a property list, which is valuable for storing extra properties and information that do not qualify as subnodes. One S-expression tree represents one method or one expression in a do-it. This representation was designed for the purpose of serializing Smalltalk code so that it could be put in a file.

Figure 1. Paths to translate between two languages.

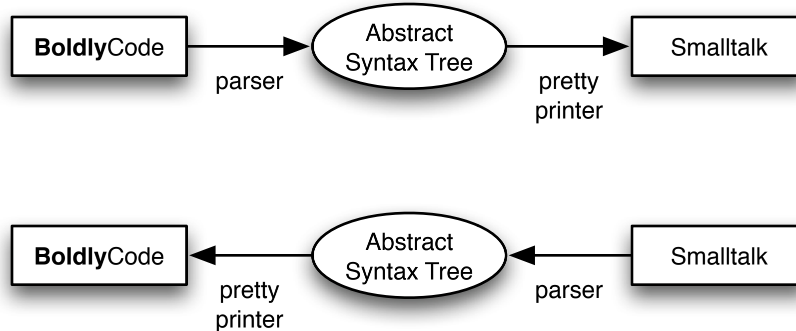
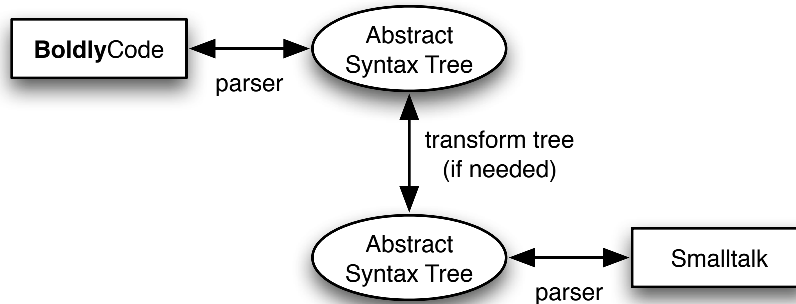


Figure 2. Running a parser backwards removes the need for the pretty printers.



Following the path from right to left in Figure 3, we perform several transformations on the S-expression tree. Generic transformations are:

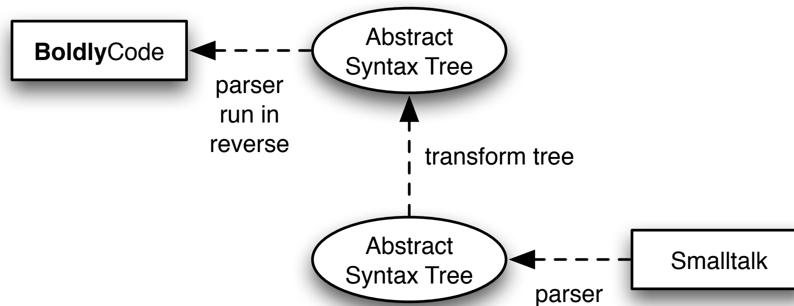
• Install a property in each node that points to its parent node. This allows a program to get data from nodes that are above it in the tree. The routine that uses operator precedence to insert parentheses will need to compare the current node with the node above.

• The important information inside a node is called different things in different kinds of nodes. We pass over the tree and install the most important data of each node in a property called 'value'. These include the selector for a message send, the value of a number or string in a literal, and the name of a variable. This value will become the text of this node in the target language. If we redesigned the tree knowing that is would be used to run a parser backwards, we would not need to do this pass over the tree.

• Mark nodes that must be enclosed in parentheses due to operator precedence. Ask the language class, in this case **Boldly**Code, for the precedence of each selector.

• Conditionals in our parse trees are represented differently than message sends. Pass over the tree converting conditionals into message sends. This only applies to pure message-send languages.

Figure 3. The path explained as an example in this paper.



Some transformations are specific to the properties of the target language, **Boldly**Code.

• The S-expression node for a message send treats the selector as a single name, such as #copyFrom:to:. **Boldly**Code, like Smalltalk, breaks up selectors into keywords. Pass over the tree inserting a list of keyword-argument pairs below each message send node.

• Treat the header of a message definition in the same way. Break up its selector into a list of keyword-variable pairs.

• When a block has arguments, gather the input argument into a node that holds list of blockArg nodes.

• Mark each statement that is inside a block, so that a period can be added after it. This is to distinguish the expressions that are statements from expressions that are arguments to message sends.

• Mark statements in the main body of the method in the same way.

Reversing a Parser Rule

The heart of the matter is our technique for making an OMeta rule run backwards. This is actually very simple. Consider the parser rule [5]:

```
assignment ::=
        <spaces> <name>:x <tok ':='> <msgExpr>:y          =>>  #('#assignment' 'x' 'y')
|       <msgExpr>
```

This rule recognizes a statement such as

```
        fragment := aString copyFrom: 5 to: 10.
```

and creates a parse tree node for it. To run this rule backwards, the input is a parse node such as:

```
        (assignment
           (variable  value='fragment'  text=' fragment' )
           (sendKeyword  text=' aString copyFrom: 5 to: 10'  (... subnodes ...)))
```

Note that we compute the output text from the leaf nodes up. When we consider a node, we have already computed the text in each of its subnodes.

To translate that node into the text of the statement, we need a template and a map; these are automatically extracted from the parsing rules using the analyzeRule routine, which is described below. The template often corresponds one-to-one with the left side of the parser rule.

```
        ( ' '  (var 'x')  ' :='  (var 'y') )
```

and we compute a map

((1,2)   (2,4)).
The goal is to substitute the subnodes of the assignment node into the template. Since we process the tree bottom-up, the subnodes already have translated strings that are expressions in the target language. The subnodes are (variable ' fragment') and (sendKeyword ' aString copyFrom: 5 to: 10'). The first pair in the map (1,2) says to take the text of the first subnode and install it in the second position of the template. The pair (2,4) says to install the text of the second subnode, ' aString copyFrom: 5 to: 10', into position four of the template, replacing #(var 'y').

The analyzeRule routine analyzes a parser rule to extract its template and map.   It only runs once after the rule is created or modified. Rules in OMeta are themselves parsed with an OMeta parser. AnalyzeRule obtains the parse tree of the rule itself, and examines the left side.

```
(and
    (apply 'spaces')
    (assign 'x'  (apply 'name'))
    (apply 'tok' '':='')
    (assign 'y'  (apply 'msgExpr'))
```

From this it generates the template #(' ' (var 'x') ' :=' (var 'y')). The right side of the rule is also in the parse tree of the rule. It is:

```
(specialAction  ('#assignment' 'x' 'y'))
```

It contain information about the order that the variables will appear in the node. From that it is easy to construct the map, ((1,2)   (2,4)).

Parsers are renowned for their flexibility and generality, and the OMeta parser is especially capable. AnalyzeRule can only handle relatively simple rules. Luckily, most programming languages can be parsed with relatively simple rules. The author of a parser can break up complex rules into simple ones, so that these can be understood by analyzeRule and can then be run backwards.
We have already mentioned how we transform the S-expression tree in order to get its nodes to match the righthand sides of rules. After applying three generic transformations and five more that are specific to **Boldly**Code, we traverse the tree depth-first. This means that all subnodes of a node have already been processed when we consider a node.

- If a node is a leaf and has no subnodes, use its value property as if it were a subnode.
- Use the name of the node such as 'assignment' to look up a rule, and use the template and map to create a text string for the node.
- If no rule is found, simply concatenate the texts of each of the subnodes.

After doing this for every node in the tree, the resulting text in the top node is the program translated to the target language.

More Examples

We'd like to translate the Smalltalk method:

```
ruleNameArity: msgName
    | na |
    na := msgName numArgs.
    na = 0 ifTrue: [
        UnaryCount := UnaryCount + 1.
        ^ #sendUnary].
    na > 1 ifTrue: [^ #sendKeyword].
    ^ msgName last == $: ifTrue: [#sendKeyword] ifFalse: [#sendBinary].
```

into the **Boldly**Code method:

```
ruleNameArity  msgName.
    use na.
```

```
na := msgName numArgs.
na =  0 ifTrue [
        UnaryCount := UnaryCount + 1.
        return #sendUnary].
na >  1 ifTrue [ return #sendKeyword].
return msgName last ==  $:  ifTrue [ #sendKeyword] ifFalse [ #sendBinary].
```

These two methods look quite similar. However, because the S-expression tree adopts the conventions of a more traditional programming language, converting the tree into **Boldly**Code forces us to be quite general. Notice that **Boldly**Code uses bold and underline to denote keyword selectors and unary message send selectors.

Let's follow how a text emphasis change is extracted from the parser and applied to a syntactic element when the parser is run in reverse. In line three of the method, the variable msgName receives the message numArgs. The parse tree node for the messge selector is:

```
(unarySymbol  value='numArgs'  selector='numArgs' )
```

The parser has a rule:

```
unary ::=
    <spaces> #underlined <name>:uu #ununderlined       =>> #('#unarySymbol' 'uu')
```

AnalyzeRule has examined the rule and created:

```
template:     ( ' '  underlined  (var 'uu')  ununderlined )
map:          ( (1,3) )
```

The rule reverser reads the map and substitutes 'numArgs' into the template to get:

```
( ' '  underlined  'numArgs'  ununderlined )
```

The process of converting the template to text reads the symbols underlined and ununderlined, and produces the text ' numArgs'.

As a further example, a block in the source language such as

```
[UnaryCount := UnaryCount + 1.
 ^ #sendUnary]
```

arrives as the node:

```
(block
   (assignment  text='UnaryCount := UnaryCount + 1.'  (...subnodes...))
   (return  text=' return #sendUnary.'
      (literal  value='#sendUnary')))
```

The parser rule for a block itself is:

```
block ::=
    <tok '['> <blockArgs>:ba <blockExprs>:es <tok ']'>      =>> #('#block' 'ba' 'es')
|   <tok '['> <blockExprs>:es <tok ']'>                     =>> #('#block' '#noTemps' 'es')
```

From the second line, AnalyzeRule created:

```
template:     ( ' ['  (var 'es')  ' ]' )
map:          ( (1,2) )
listSuffix:   '/r/t'
```

The list suffix of return tab was added to all subnodes by the tree transformer for block nodes. The translated code looks better when statements are on separate lines in the code. Because there is a listSuffix, the rule reverser knows that a list of subnodes will be installed wherever the variable is in the template. The reverser produces text:

```
[UnaryCount := UnaryCount + 1.
  return #sendUnary.
]
```

Four kinds of parser rules can be reversed. These are:
1. A rule with mixed constants and sub-expressions. Assignment is an example of this.
2. A rule with emphasis change in it. UnarySymbol is an example.
3. A rule with an undetermined number of sub-nodes, such as the statements in a block. Constant text such as linefeed can be inserted between the elememts.
4. A parse tree node that does not match any rule. The text of the subnodes is collected and concatenated.

Together these cover enough variety of parsing to show an Abstract Syntax Tree in a variety of source languages.

Commentary

The goal of this paper is to explain how to run an OMeta parser in reverse. In general, this is impossible, since a parser written in OMeta can have arbitrary Smalltalk expressions on both the left and right sides of a rule. Many expressions and many rules are not invertible in the strict sense. However, several factors make it possible to run a parser in reverse in almost all cases.

- An Abstract Syntax Tree has all the information needed to create running code. It has complete information about the program. Specifically, it has all of the components of the program in order. This order is the execution order, and is almost always the order in which these items appear in the source code.
- When a parser rule has several alternative choices that can match, how can this be run backwards? Often, several variations in syntax have the same meaning. The parser funnels these to a single kind of tree node. In the reverse direction, only one of the variations needs to be used, since it is a legal statement of the meaning. The reverse parser can ignore the other cases.
- When two languages have differences in order, such as message selectors divided into keywords versus as a single name, these differences can be expressed in the parse tree. Explicitly converting from one tree form to another encapsulates large differences of this kind.
- Some parser rules are impossible to run backwards. The answer is to restrict which rules can be used. The author of the parser rules can stay within a subset of all possible rules. With the fallback of simply concatenating the subnodes of a node, and with tree restructuring, it appears that most parsers will be invertible.

Being able to run parsers backwards suggests that code can be kept in Abstract Syntax Trees and viewed in whatever language the user desires. A worthwhile goal would be to have a universal standard form for Abstract Syntax Trees. This format would encompass a large variety of programming languages. A program expressed in the standard form could be easily converted into source code in each of the languages.

Since programming languages are different from one another, a universal Abstract Syntax Tree format needs to encompass the semantics of many programming languages. This is not easy. If an idiom in language A is translated into language B, it may expand into longer code. If that is translated back into language A, we would like it to be identical to the original code. It will be difficult to recognize the construct in language B, and put it back into the same Abstract Syntax Tree that language A would use. The system will need to recognize idioms and put them back into a simple form. Trying to find a universal standard form for Abstract Syntax Trees is a worthy research topic.

Conclusion

A modified OMeta parser can run backwards, and this simplifies the task of translating from one programming language to another.

References

1. **OMeta: an Object-Oriented Language for Pattern Matching**, Alessandro Warth, Ian Piumarta. http://vpri.org/pdf/OMeta_TR-2007-003.pdf

2. **Steps Toward the Reinvention of Programming (First Year Progress Report)**, A. Kay, I. Piumarta, K. Rose, D. Ingalls, D. Amelang, T. Kaehler, Y. Ohshima, C. Thacker, S. Wallace, A. Warth, T. Yamamiya. http://vpri.org/pdf/steps_TR-2007-008.pdf

3. **The GlyphicScript language Scripting Manual**, Mark Lentczner. 1995. http://www.ozonehouse.com/mark/cw/Scripting%20Manual.pdf More info here: http://www.ozonehouse.com/mark/codeworks.html

4. **Parsing Expression Grammars: a recognition-based syntactic foundation**, B. Ford. In POPL ’04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pages 111–122.

5. OMeta experts will notice that the action portion of the rule is in an unusual syntax. The normal way to express the right side of a rule is compiled to Smalltalk code. The conversion to Smalltalk is done too eagerly, and loses information.

<spaces> <name>:x <tok ':='> <msgExpr>:y   => [{#assign. x. y}]

The routine that reads a rule and creates the template and map needs to examine the parse tree of the rule itself. The new syntax

<spaces> <name>:x <tok ':='> <msgExpr>:y   =>> #('#assignment' 'x' 'y')

preserves the structure of the right hand side, and allows it to be analyzed. We modified OMeta itself to accept this new syntax for a rule.