C++ disambiguation in Transformers

Clément Vasseur <clement.vasseur@lrde.epita.fr>

LRDE seminar, May 21, 2003
# Table of Contents

## Program transformation

- StrategoXT program transformation tools .......................................................... 4
- Global view ........................................................................................................ 5
- The Stratego term rewriting system .................................................................. 6
- Example: unfor (abstract syntax) ......................................................................... 7
- Example: unfor (concrete syntax) ...................................................................... 8

## C++ disambiguation filters

- namespace ......................................................................................................... 10
- declaration ......................................................................................................... 12
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>declarator</td>
<td>14</td>
</tr>
<tr>
<td>specifier</td>
<td>17</td>
</tr>
<tr>
<td>disambiguate</td>
<td>18</td>
</tr>
<tr>
<td><strong>Use case: the <code>typeof</code> operator</strong></td>
<td>19</td>
</tr>
<tr>
<td>Implementation</td>
<td>20</td>
</tr>
<tr>
<td><strong>Known issues</strong></td>
<td>21</td>
</tr>
<tr>
<td>The ISO/IEC C++ standard grammar</td>
<td>22</td>
</tr>
<tr>
<td>The C preprocessor</td>
<td>23</td>
</tr>
<tr>
<td>User friendliness</td>
<td>24</td>
</tr>
<tr>
<td><strong>Conclusion</strong></td>
<td>25</td>
</tr>
<tr>
<td><strong>References</strong></td>
<td>26</td>
</tr>
</tbody>
</table>

C++ disambiguation in Transformers, Clément Vasseur - LRDE seminar, May 21, 2003
Program transformation

source code → transformation → source code

• compilation
• optimization
• synthesis
• refactoring
• migration
• normalization
StrategoXT program transformation tools

Eelco Visser - Institute of Information and Computing Sciences (Utrecht)
http://www.stratego-language.org

**sdf** syntax definition formalism (*Heering et al.*, 1989)

**sglr** scanner-less generalized LR parser (*Visser*, 1997)

**stratego** language for the specification of transformation systems (*Visser*, 2001)

**gpp** generic pretty printer (*Jonge*, 2000)
The Stratego term rewriting system

⇒ modular specification of transformation systems
⇒ paradigm of rewriting strategies

• abstract and concrete syntax tree manipulation

• simplification / desugaring

• optimization

• code generation

• application generation

• program analysis
module cpp-unfor

imports lib Cxx

rules

Unfor : for (i, Some (c), Some (e), s) ->
StatementSeq-opt ([i, while (c,
  StatementSeq-opt ([s,
    Expression-opt (Some (e))])))])

strategies

cpp-unfor = iowrap (bottomup (try (Unfor)))
Example: unfor (concrete syntax)

module cpp-unfor

imports lib Cxx

rules

Unfor : | [ for (e1 ; e2 ; e3) s ] |
-> | [ { e1; while (e2) { s e3; } } ] |

strategies

cpp-unfor = iowrap (bottomup (try (Unfor)))
C++ disambiguation filters

The diagram shows the process of C++ disambiguation filters:

1. Parse forest
2. Namespace
3. Declaration
4. Resolve
5. Declarator
6. Specifier
7. Disambiguate
8. Parse tree
namespace

C++ grammar

namespace-name:
  original-namespace-name
  namespace-alias

original-namespace-name:
  identifier

namespace-alias:
  identifier
namespace filter algorithm

\[ E \leftarrow \{\} \]

**top-down traversal**

```
for each ambiguous definition \( d \) of namespace \( n \) do
    if \( n \in E \) then
        \( d \) is an extension of \( n \)
    else
        \( d \) is the original definition of \( n \)
        \( E \leftarrow E \cup \{n\} \)
    end if
end for
```

end top-down traversal
C++ grammar

simple-declaration:
    decl-specifier-seq(opt) init-declarator-list(opt)

```cpp
int foo = 0;       // not ambiguous
int foo, bar;      // not ambiguous
foo bar;           // ambiguous: ([foo bar], []) or ([foo], [bar])
typedef foo bar;   // ambiguous: ([typedef foo bar], []) or ([typedef foo], [bar])
foo bar = 0;       // not ambiguous
foo bar, baz;      // not ambiguous
class A {};        // not ambiguous
class A { } a;     // ambiguous: ([class A { } a], []) or ([class A { }], [a])
```
top-down traversal
   for each ambiguous declaration $d$ do
      for each branch $b(b_s, b_d)$ of $d$ do
         $b_s$ is the list of specifiers carried by $b$
         $b_d$ is the list of declarators carried by $b$
         if $b_d = []$ then
            remove $b$ from $d$
         end if
      end for
   end for
end for
end top-down traversal
The grammar distinguishes several types of identifiers: expressions, types, classes, enumerations, ...

typedef-name:
    identifier
class-name:
    identifier
template-id
enum-name:
    identifier
template-name:
    identifier
declarator filter implementation

top-down traversal

for each declaration \( d(d_s, d_d) \) do

\( d_s \) is the list of specifiers carried by \( d \)

\( d_d \) is the list of declarators carried by \( d \)

for each declarator \( x \) of \( d_d \) do

keep the expression branch of \( x \)

end for

end for

end top-down traversal
typedef foo bar; // not ambiguous
typedef foo::bar baz; // ambiguous: [foo::bar] or [foo, ::bar]
typedef foo::bar::baz qux; // ambiguous: [foo::bar::baz] or [foo, ::bar::baz] or ...

top-down traversal
for each ambiguous list of specifiers \( l \) do
    for each ambiguous branch \( l_i \) of \( l \) do
        compute the number of non-trivial specifiers \( k_i \)
    end for
    keep \( l_i \) such as \( k_i = 1 \)
end for
end top-down traversal
C++ disambiguation filters

{build environments}
{disambiguate}

top-down traversal

for each definition of namespace or class \( n \) do

\( E_n \leftarrow \{\} \)

end for

for each declaration \( d \) of symbol \( s \) do

\( s \) is declared in namespace \( n \)
\( s \) is of kind \( k \)

\( E_n \leftarrow E_n \cup \{s : k\} \)

end for

end top-down traversal
Use case: the `typeof` operator

```cpp
template <typename T1, typename T2>
Array<??> operator+ (Array<T1> const& x,
                     Array<T2> const& y);
```

Arithmetic operator for a numeric array template in which the element types of the operands are mixed. (Vandervoorde and Josuttis, 2002)

```cpp
template <typename T1, typename T2>
Array<typeof(T1()+T2())> operator+ (Array<T1> const& x,
                                     Array<T2> const& y);
```

typeof: produces a compile-time entity from an expression, than can act as the name of a type.
Use case: the typeof operator

**Implementation**

```c
int b;
typeof(b) a;
```

The implementation needs to know about types ⇒ the type-decorator should be usable from the transformation code.
Known issues

- The ISO/IEC C++ standard grammar does not use advanced SDF features
- Preprocessed files easily become too big
- Some usability problems
The ISO/IEC C++ standard grammar

- Stay as close as possible to the standard ISO/IEC C++ grammar (C++98, 1998)
- ISO/IEC C++ grammar written using BNF notation
- BNF is a simple notation ⇒ not using SDF advanced features

**Example:** priorities are enforced by using a lot of intermediate rules, while SDF supports a context-free priorities section to define priorities between grammar rules.
preprocessed C++ code \(\Rightarrow\) enormous files when using the STL headers

iostream: 27000 lines of highly ambiguous code

**problem:** sglr can’t handle too many ambiguities

- CodeBoost (Bagge et al., 2001) only parses a small section of the source code, assuming *unknown* symbols are OK to work with.

- pre-parse the headers and keep their environments for later use.
User friendliness

- From the disambiguation filters programmer point of view: disambiguation filters are written using parse tree syntax
  ⇒ they can be rewritten using abstract tree syntax, or even using concrete syntax

- From the transformation programmer point of view: the term constructor names are automatically generated
  ⇒ the programmer is expected to manipulate symbols with no meaning

- From the end-user point of view: there is no line numbers in terms
  ⇒ it is impossible to give a precise location when an error occurs
To do:

- Split the *disambiguate* filter
- Work out the preprocessor problem
- Start to implement useful transformations

Transformers:
http://www.lrde.epita.fr
References


