

# C++ disambiguation in Transformers

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

LRDE seminar, May 21, 2003



# Table of Contents

<b>Program transformation</b> .....	<b>3</b>
StrategoXT program transformation tools .....	4
Global view .....	5
The Stratego term rewriting system .....	6
Example: unfor (abstract syntax) .....	7
Example: unfor (concrete syntax) .....	8
<b>C++ disambiguation filters</b> .....	<b>9</b>
namespace .....	10
declaration .....	12

## Table of Contents

---

declarator .....	14
specifier .....	17
disambiguate .....	18
<b>Use case: the typeof operator</b> .....	<b>19</b>
Implementation .....	20
<b>Known issues</b> .....	<b>21</b>
The ISO/IEC C++ standard grammar .....	22
The C preprocessor .....	23
User friendliness .....	24
<b>Conclusion</b> .....	<b>25</b>
<b>References</b> .....	<b>26</b>

# 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>

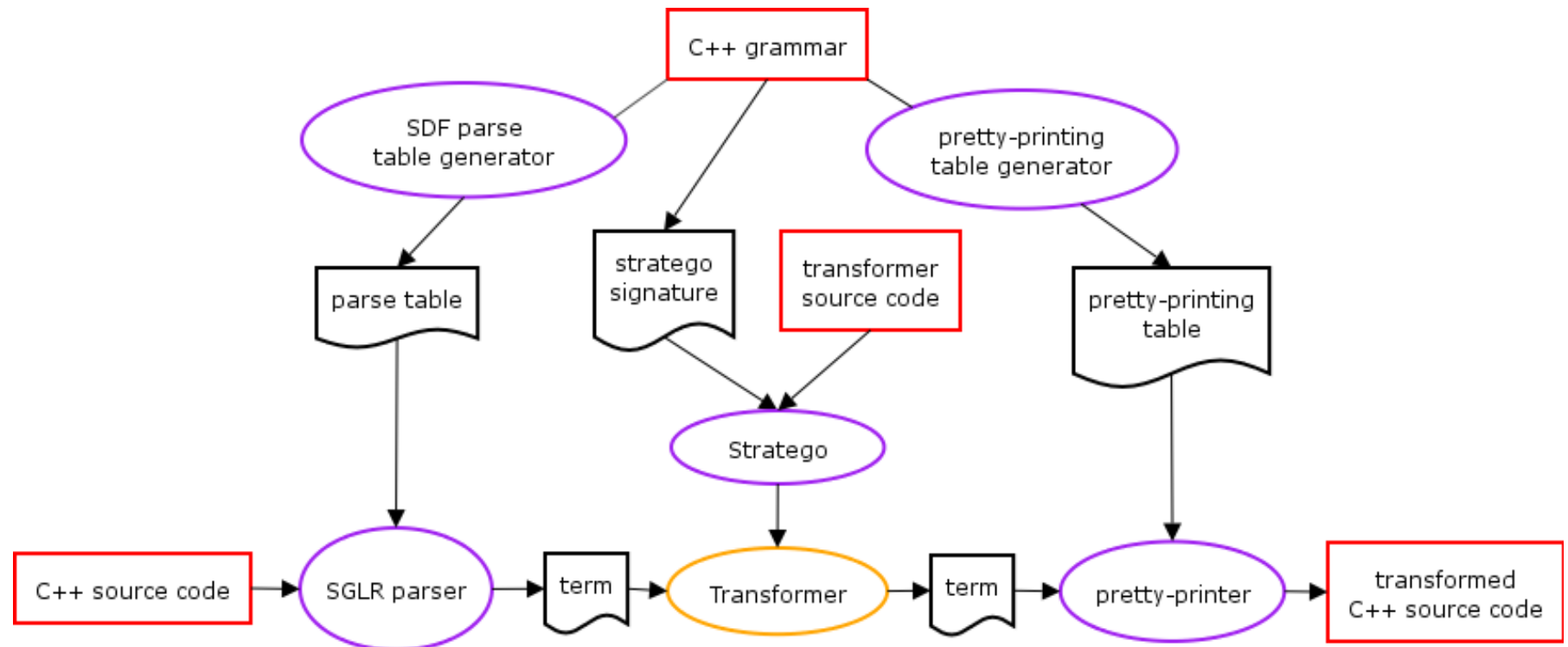
**sdf** syntax definition formalism ([Heering et al., 1989](#))

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

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

**gpp** generic pretty printer ([Jonge, 2000](#))

# Global view



## 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

## Example: unfor (abstract syntax)

```
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
```

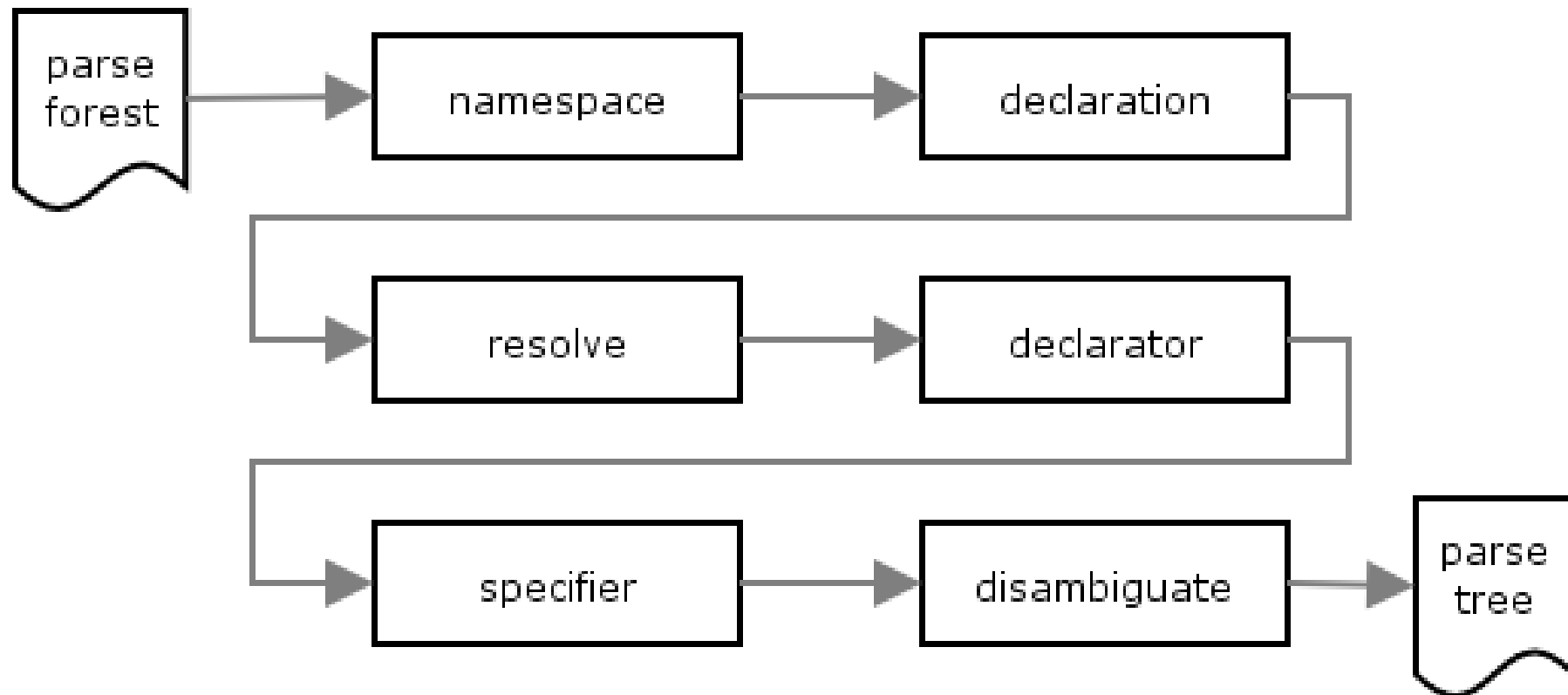
```
rules
```

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

```
strategies
```

```
cpp-unfor = iowrap (bottomup (try (Unfor)))
```

## C++ disambiguation filters



---

# 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**

# declaration

## C++ grammar

simple-declaration:

decl-specifier-seq(opt) init-declarator-list(opt)

```
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])
```

## declaration filter implementation

### 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**

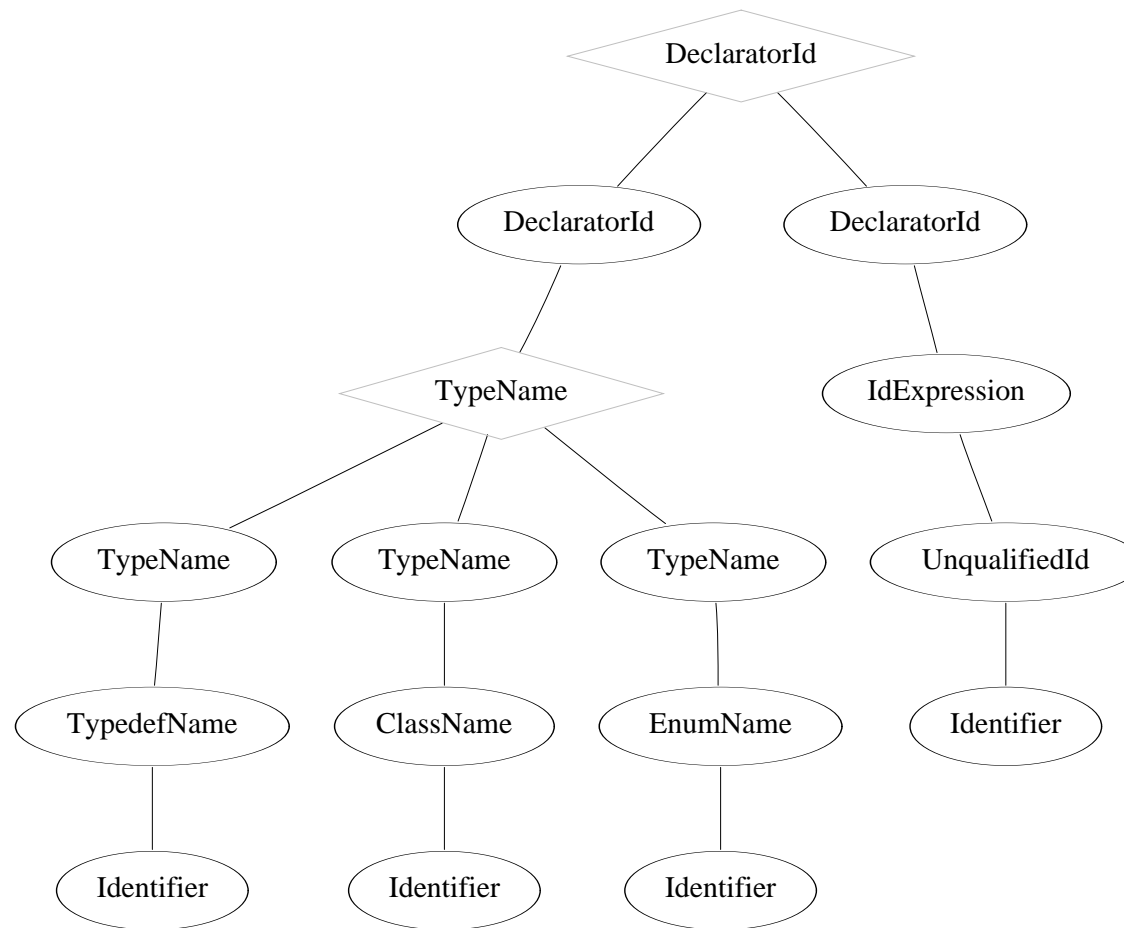
**end top-down traversal**

---

## declarator

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**

## specifier

```
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**

# disambiguate

*{build environments}*

**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**

*{disambiguate}*

**top-down traversal**

**for each** ambiguous node  $a(a_1, a_2, \dots, a_n)$  **do**

keep  $a_i$  such as  $disambiguate(a_i)$  is successful

**end for**

**for each** symbol  $s$  seen with kind  $k$  **do**

find namespace  $n$  in which  $s$  is declared

**if**  $E_n \vdash s : k' \neq k$  **then**

fail

**end if**

**end for**

**for each** local 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**

**for each** scope in namespace  $n$  **do**

save and restore  $E_n$  properly

**end for**

**end top-down traversal**

## Use case: the `typeof` operator

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

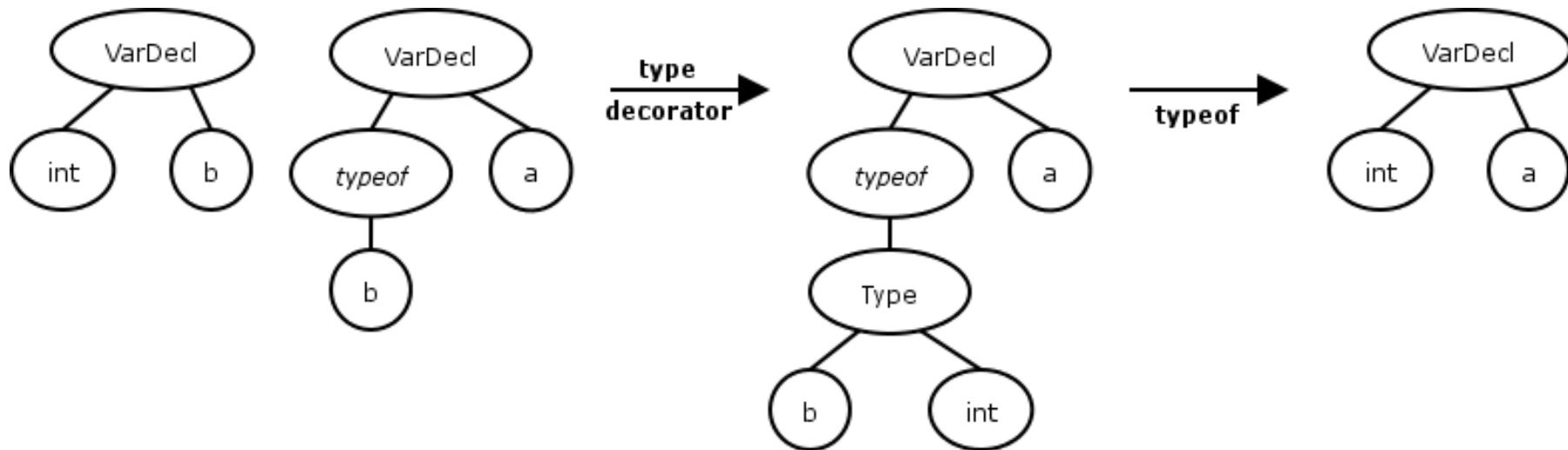
Arithmetic operator for a numeric array template in which the element types of the operands are mixed. ([Vandervoorde and Josuttis, 2002](#))

```
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.

# Implementation

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



The implementation needs to know about types  $\Rightarrow$  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  $\Rightarrow$  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.

## The C preprocessor

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

## Conclusion

To do:

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

Transformers:

<http://www.lrde.epita.fr>

## References

Bagge, O. S., Haverlaan, M., and Visser, E. (2001). CodeBoost: A framework for the transformation of C++ programs. Technical Report UU-CS-2001-32, Institute of Information and Computing Sciences, Utrecht University.

C++98 (1998). ISO/IEC 14882:1998 (E). *Programming languages - C++*.

Heering, C., Hendriks, P. R. H., Klint, P., and Rekers, J. (1989). The syntax definition formalism SDF - reference manual -. *SIGPLAN Notices*, 24(11):43–75.

Jonge, M. d. (2000). A pretty-printer for every occasion. In Ferguson, I., Gray, J., and Scott, L., editors, *Proceedings of the 2nd International*

## References

---

*Symposium on Constructing Software Engineering Tools (CoSET2000)*, pages 68–77. University of Wollongong, Australia.

Vandervoorde, D. and Josuttis, N. M. (2002). *C++ Templates The Complete Guide*. Addison-Wesley.

Visser, E. (1997). Scannerless generalized-lr parsing.

Visser, E. (2001). Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In Middeldorp, A., editor, *Rewriting Techniques and Applications (RTA'01)*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–361. Springer-Verlag.