CS-Note* 47

Bill J. Ellis
June 13, 2003

Program Transformation with XT and STRATEGO

Contents

*Notes in this series are for ϵ-baked ideas, for $1 \geq \epsilon \geq 0$. Only exceptionally should they be cited or distributed outwith the NuSPADE project (EPSRC GR/R24081).

1 Introduction

In CS-note 45 it is noted that program provers need access to the program source code. In practical terms this access involves parsing SPARK into a pseudo-language, called here PSEUDO-SPARK. Currently, the approach outlined in CS-note 40 has been pursued. This involves creating a parser LW-SPARK-PARSER, using LEX and YACC to parse SPARK into a large intermediate Prolog term. This term is transformed into PSEUDO-SPARK by a separate system called SUBPROG-SPIDER. The separate system is also responsible for brining together all of the different files (intermediate terms) to create the complete PSEUDO-SPARK as the information needed to consider a subprogram is rarely contained in the same file. This is limiting in a few points:

- **Weak intermediate language -** The intermediate language outputted from LW-SPARK-PARSER and read by SUBPROG-SPIDER is not well defined.
- **The third system -** SUBPROG-SPIDER performs two operations. The transformation of the intermediate term into PSEUDO-SPARK should be viewed as a separate operation that is much more tightly coupled to LW-SPARK-PARSER.
- **Distance from SPARK -** The SPARK language will change. When it does, this will require changes to the LEX and YACC files of SUBPROG-SPIDER. More problematically, it will require changes to the intermediate language and the behaviour of LW-SPARK-PARSER.
- **Poor implementation -** At the core of SUBPROG-SPIDER is a very inefficient term rewriting engine. However, although inefficient, it remains viable on most programs.

Following CS-note 40 it was found that the system it describes is essentially a *program transformation* system, in both behaviour and general design. A tool already exist for this purpose called STRATEGO [5]. This note explores the applicability of STRATEGO to the problem of accessing source code for program analysis.

2 Getting started

2.1 Documentation

STRATEGO is described in various documents available at the project web site [1]. In particular [4, 3] provide a practical introduction to the system. An extremely helpful note [2] was found on the web. This is used as the basis for the example given in §5[1].

2.2 Installing

The most recent version of STRATEGO is packaged alongside a suite of program transformation modules called XT. All of the components must be installed to gain a working STRATEGO system. The following source files were downloaded from the web site [1]:

```
aterm-1.6.7.tar.gz
sdf2-1.5.tar.gz
StrategoXT-0.9.tar.gz
```

These are unpacked and installed through the usual `gunzip`, `tar -xf`, `configure`, `make`, `make install`. The STRATEGO web site includes a shell script to perform the whole process, and install the system in the current directory.

STRATEGO is also available as red hat RPMs. These have also been installed and tested[2]. If using STRATEGO long term RPMs and a proper install, outside a home directory, may be more suitable.

3 About XT and STRATEGO

XT is described as a bundle of program transformation tools. The current distribution of XT contains 114 separate binary programs. These programs are combined to perform operations such as building parsers, managing grammars and display terms. Together, these programs can be combined to create program transformation systems. STRATEGO is one, albeit large, component of the XT bundle. STRATEGO is a term rewriting language, building directly upon many of the low level programs available within XT.

There are two main tasks of the program transformation system under consideration:

[1] Note that, `src/ast/Eval.r` on page 7 should read `src/ast/Exp.r`.

[2] On my home computer. RPM install requires root access, which I do not have at Heriot-Watt.

- **Parser -** Parse the SPARK code into a suitable term for analysis. This involves creating a grammar Syntax Definition Formalism (SDF) file. From this SDF file the XT tools can be used to automatically build a parser. This is very similar to LEX and YACC, but the SDF offers a slightly higher level of abstraction.
- **Transformation -** Transform the parsed term into an new output term. This involves applying a number of rewrite rules to the term. First this involves creating an Abstract Syntax (AS[3]) file to describe the terms being manipulated. The AS file for the input grammar can be automatically extracted from the grammar SDF. However an AS file will need to written manually to cover the structure of the output term. Secondly, a Transform Rule (TR) file is created. The rules in the TR file are essentially rewrite rules. Thirdly, a Transformation Strategy (TS) file is created to specify how the rules in TR are to be applied. Finally a Main Transformation (MT) file is required to link the AS, TR and TS files together into a complete program transformation system.

Thus to build the program transformation system their is a need to produce:

- Syntax Definition Formalism (SDF) file.
- Abstract Syntax (AS) file. (The AS file for the input term can be automatically generated from the SDF).
- Transformation Rule (TR) file.
- Transformation Strategy (TS) file.
- Main Transformation (MT) file.

The structure of the program transformation system under consideration is as below:

```
                         ~~~~~~
                         |Code|
                         ~~~~~~
                             |
                             V
~~~~~~~~~~~~~            --------
|Grammar SDF|- - - - - ->|Parser|
~~~~~~~~~~~~~            --------
     |                       |
                             V
     |                    ~~~~~~
                          |Term|
     |                    |    |
                          ~~~~~~
     |                       |
     V                       V
~~~~~~~~~~~~~       ----------------    ~~~~~~~~~~~~~~~~
|Abstract   | - - ->|Transformation|<- -|Main          |
|Syntax     |       |              |    |Transformation|
~~~~~~~~~~~~~       |              |    ~~~~~~~~~~~~~~~~
                    |              |
~~~~~~~~~~~~~~~~    |              |
|Transformation|- ->|              |
|Rules         |    |              |
~~~~~~~~~~~~~~~~    |              |
                    |              |
~~~~~~~~~~~~~~~~    |              |
|Transformation|- ->|              |
|Strategy      |    |              |
~~~~~~~~~~~~~~~~    ----------------
                             |
                             V
                        ~~~~~~~~
                        |Result|
                        |Term  |
                        ~~~~~~~~
```

[3]Every STRATEGO file has the extension STR. For clarity, extra names are introduced in this note.

4 My first STRATEGO system

This example is copied more or less verbatim from the early sections of [2]. It demonstrates the build process of STRATEGO on an ambiguous[4] expression grammar.

4.1 SDF grammar file

It is common to present the SDF file as a collection of modules. Here the grammar is separated into three modules, shown in figures 1, 2 and 3.

```
module Main

imports
    Exp
```

Figure 1: File: Main.sdf

```
module Exp
    imports
        N
    exports
        sorts Exp
        context-free syntax
            N -> Exp {cons("N")}
            "(" Exp ")" -> Exp {cons("Paren")}
            Exp "*" Exp -> Exp {cons("Mul")}
            Exp "/" Exp -> Exp {cons("Div")}
            Exp "%" Exp -> Exp {cons("Mod")}
            Exp "+" Exp -> Exp {cons("Add")}
            Exp "-" Exp -> Exp {cons("Sub")}
            "-" Exp -> Exp {cons("Minus")}
```

Figure 2: File: Exp.sdf

```
module N
    exports
        lexical syntax
            [0-9]+ -> N
```

Figure 3: File: N.sdf

The role of each module is described briefly below:

- Figure 1 (module `Main`) is the root of the SDF definition. This references all of the modules which define the grammar.
- Figure 3 (module `N`) declares the syntax of numbers. It is common to collate all of the lexical syntax into a single module. Note that only module `Exp` imports module `N`. Module `Main` does not need to know about module `N`.
- Figure 2 (module `Exp`) defines a small grammar for arithmetic expressions. The grammar is essentially BNF, presented backwards. The `cons` keywords describes how to create the result of the parsing. `cons` is a special keyword which creates a tuple with the supplied name and the arguments of the associated grammar rule.

[4] The ambiguities are iteratively removed in [2].

Note that the grammar of module `Exp` is ambiguous. It is ambiguous as their is no operator precedence ($1{+}2{*}3 = \{(1{+}2){*}3\} \vee \{1{+}(2{*}3)\}$). Further, associtivity has not been defined ($1{+}2{+}3 = \{(1{+}2){+}3\} \vee \{1{+}(2{+}3)\}$). Their are STRATEGO ways for eliminating such ambiguities. However, these are not of much interest to our problem. The SPARK grammar made available by Praxis is, by definition, unambiguous. Their is no need to explore the STRATEGO approach to resolving grammar ambiguities as we know that our grammar is not ambiguous.

4.2 Build a parser

4.2.1 Create single definition file

`pack-sdf` takes the root of an SDF module tree and searches through all of the modules to create a single complete definition. By default this definition is outputted as a large ASFIXTREE. ASFIXTREE is the main internal structure used by STRATEGO. It is possible to store ASFIXTREE in both text and binary form.

```
pack-sdf -i Main.sdf -o Main.asfixtree
```

The ASFIXTREE is a large nested term that is quite hard to read. Fortunately, `asfix-yield` can be used to translate any ASFIXTREE into its neat textual SDF form. As this is now describing the whole grammar it is called a Grammar Definition (DEF) file.

```
asfix-yield -i Main.intdef -o Main.def
```

The resulting DEF file `Main.def` is shown in figure 4.

```
definition
module N
    exports
        lexical syntax
            [0-9]+ -> N


module Exp
    imports
        N
    exports
        sorts Exp
        context-free syntax
            N -> Exp {cons("N")}
            "(" Exp ")" -> Exp {cons("Paren")}
            Exp "*" Exp -> Exp {cons("Mul")}
            Exp "/" Exp -> Exp {cons("Div")}
            Exp "%" Exp -> Exp {cons("Mod")}
            Exp "+" Exp -> Exp {cons("Add")}
            Exp "-" Exp -> Exp {cons("Sub")}
            "-" Exp -> Exp {cons("Minus")}

module Main

imports
    Exp
```

Figure 4: File: Main.def

4.2.2 Create table for parsing

`sdf2table` takes a DEF file and creates a Parse Table (TBL) file. This is essentially the same as the shift-reduce table used by YACC.

sdf2table -l -v -m Main -i Main.def -o Main.tbl	
-l = display statistic information	
-v = verbose mode	
-m name = parse table is generated for module name	

4.2.3 Test the parser

Create a file containing the term to be parsed. Note that this does not have any layout text[5]. Layout will be considered in the next example.

```
echo -n "1+2*(3/2)" > testmyfirstexample
```

Parsing is achieved through `sglr`. This takes as input the TBL file, describing the behaviour of the parser. The input text is parsed generating an output ASFIXTREE file as default stored in a binary format.

sglr -p Main.tbl -i testmyfirstexample -o result.asfixtree	
-p file = use parse table file	

The `asfixbinary` is converted into a readable Abstract Syntax Tree (AST) through `implode-asfix`.

```
implode-asfix -i result.asfixtree -o result.ast
```

The result of the parse is shown in figure 5. Note that a parsing ambiguity was detected. All of the possible interpretations are returned as a list inside `amb(...)`.

```
amb([Mul(Add(N("1"),lit("+"),N("2")),lit("*"),Paren(lit("("),Div(N("3"),lit("/"),
N("2")),lit(")"))),Add(N("1"),lit("+"),Mul(N("2"),lit("*"),Paren(lit("("),
Div(N("3"),lit("/"),N("2")),lit(")"))))])
```

Figure 5: File: result.ast

[5] `echo -n` does not leave a trailing newline.

4.3 Build a transformer

4.3.1 Create an abstract syntax description

`sdf-to-sig` takes a DEF file and generates corresponding AS files. Note that no output file is specified as the abstract syntax files are stored in files mirroring those modules in the DEF file.

```
sdf-to-sig -i Main.def
```

Applying `sdf-to-sig` to `Main.def` creates three AS files shown in figures 7, 6 and 8.

```
module Main
imports Exp
```

Figure 6: File: Main.str

```
module Exp
imports N
signature
  constructors
    N     : N -> Exp
    Paren : Exp -> Exp
    Mul   : Exp * Exp -> Exp
    Div   : Exp * Exp -> Exp
    Mod   : Exp * Exp -> Exp
    Add   : Exp * Exp -> Exp
    Sub   : Exp * Exp -> Exp
    Minus : Exp -> Exp
```

Figure 7: File: Exp.str

```
module N
```

Figure 8: File: N.str

The STRATEGO documentation is a little unclear if AS files are necessary for the system to operate. Early versions of STRATEGO operated without them. It is argued that the AS files should be created as they cleanly describe the grammars as processed by STRATEGO. Further, STRATEGO offers some helpful error messages where providing rules in a TR file that mismatch with the grammar described in the corresponding AS files.

4.3.2 Create transformation rules

Each rule in a TR file is of the form:

```
Name: LHS -> RHS
```

Variables in the `LHS` not known to be concrete lexical items are treated as variables, being instantiated during pattern matching in the same style as Prolog. These variables may be referred to in constructing the `RHS`.

A TR file is given in figure 9. The rule `SillyRule` transforms any addition into the result `42`. While not being of any practical value, this serves to demonstrate the rewriting process.

4.3.3 Create transformation strategies

The strategies describe how these rules are applied to perform the transformations. The strategies are expressed using a powerful collection of connecting constructs, similar to the formulation of powerful tactics from tactics and tacticals in tactic based theorem provers. The strategies are described in detail in the STRATEGO documentation [3].

A TS file is given in figure 10. This applies `SillyRule` on the input term term from the bottom up. It will continue to repeat this operation until the entire term is traversed from the bottom up and no changes are made.

```
module Eval
    imports
        Exp

rules
    SillyRule: Add(x,y) -> N("42")
```

Figure 9: File: Rules.str

```
module Strategies
    imports
        Rules

strategies
    eval = repeat(somebu(SillyRule))
```

Figure 10: File: Strategies.str

4.3.4 Create main transformation

The MT file describe the actual behaviour of the program transformation system. Outputting the resulting term is not default in STRATEGO thus it must be explicitly declared inside the main strategy. Note that sophisticated users of STRATEGO may not employ such crude output and instead develop a pretty printer for their language[6].

A MT file is given in figure 11. The `lib` module is supplied as part of STRATEGO and includes the definition of the `stdio` function, used here for straight forward textual output.

```
module Transform

    imports
        lib
        Strategies

strategies
    main=stdio (eval)
```

Figure 11: File: Transform.str

4.3.5 Create the transformation program

The transformation program is created by compiling the MT file.

```
sc --verbose 2 -i Transform.str -o mytransformer
--verbose N = Higher value of N give more compilation details
```

4.3.6 Test the transformation program

The parsing and transformation can be performed in a single command by using pipes (`|`).

```
echo -n "1+2+3" | sglr -2 -p Main.tbl | implode-asfix | ./mytransformer > tranresult.ast
```

The result of the parse is shown in figure 12. Note how the transformations have modified two ambiguous parsed structures into the same term.

[6] STRATEGO has support for the "box-notation" used to express pretty printing algorithms.

```
amb([N("42"),N("42")])
```

Figure 12: File: tranresult.ast

5 Moving onto SPARK

From experience developing SUBPROG-SPIDER the most difficult part of the required program transformation is dealing with sequences of statements and conditional statements. These two features have been successfully implemented within STRATEGO to demonstrate the applicability of the approach in general.

5.1 The code

5.1.1 The SDF files

```
module Main
imports
    Program
    Layout
```

File: Main.sdf

```
module Program
    imports
        Syntax
        Layout
    exports
        sorts Program
        context-free syntax

      Gifstatement -> Program {cons("Cprogram")}

      Tif Gcondition Tthen
          Gsequenceofstatements
          Gelsifpart
          Gelsepart
      Tend Tif Tsemicolon -> Gifstatement  {cons("Cifstatement")}

      Gelsifpart
          Telsif Gcondition Tthen
              Gsequenceofstatements -> Gelsifpart {cons("Celseifpart1")}

              -> Gelsifpart {cons("Celseifpart2")}

      Telse Gsequenceofstatements -> Gelsepart {cons("Celsepart1")}

              -> Gelsepart {cons("Celsepart2")}

      Gsequenceofstatements Gstatement -> Gsequenceofstatements {cons("Csequenceofstatements1")}
      Gstatement -> Gsequenceofstatements {cons("Csequenceofstatements2")}

      Tthreedots -> Gstatement {cons("Cstatement")}

      Tthreedots -> Gcondition {cons("Ccondition")}
```

File: Program.sdf

```
module Layout
    exports
        lexical syntax
            [\ \n] -> LAYOUT
        context-free restrictions
        LAYOUT? -/- [\ \n]
```

File: Layout.sdf

```
module Syntax
    exports
        lexical syntax
            "if" -> Tif
            "then" -> Tthen
            "else" -> Telse
            "elsif" -> Telsif
            "end" -> Tend
            ".a." -> Tthreedots
            ".b." -> Tthreedots
            ".c." -> Tthreedots
            ".d." -> Tthreedots
            ".e." -> Tthreedots
            ".f." -> Tthreedots
            ".g." -> Tthreedots
            ".h." -> Tthreedots
            ".i." -> Tthreedots
            ".j." -> Tthreedots
            ";" -> Tsemicolon
```

File: Syntax.sdf

5.1.2 The AS files

```
module Abstractsyntax
imports
    Program
    Output
    Construct
```

File: Abstractsyntax.str

```
module Main
imports Program Layout
```

File: Main.str

```
module Program
imports Syntax Layout
signature
  constructors
    Cprogram               : Gifstatement -> Program
    Cifstatement           : Tif * Gcondition * Tthen * Gsequenceofstatements *
                             Gelsifpart * Gelsepart * Tend * Tif *
                             Tsemicolon -> Gifstatement
    Celseifpart1           : Gelsifpart * Telsif * Gcondition * Tthen *
```

```
                              Gsequenceofstatements -> Gelsifpart
    Celseifpart2            : Gelsifpart
    Celsepart1              : Telse * Gsequenceofstatements -> Gelsepart
    Celsepart2              : Gelsepart
    Csequenceofstatements1 : Gsequenceofstatements * Gstatement -> Gsequenceofstatements
    Csequenceofstatements2 : Gstatement -> Gsequenceofstatements
    Cstatement              : Tthreedots -> Gstatement
    Ccondition              : Tthreedots -> Gcondition
```

File: Program.str

```
module Layout
```

File: Layout.str

```
module Syntax
```

File: Syntax.str

```
module Output
imports
    Program
signature
    constructors
        //If statments.
        Pif      : Gcondition * Pcodelist * Pcodelist -> GPif

        //Sequence of statments.
        Pcodelist: Cstatement * Pcodelist -> GPcodelist
        Pcodelist: Pemptycodelist -> GPcodelist
        Pemptycodelist: GPemptycodelist
```

File: Output.str

```
module Construct
imports
    Program
signature
    constructors
        IF-CON-unfold : Celseifpart1 * Celsepart1 -> GIF-CON-unfold
        IF-CON-unfold : Celseifpart1 * Celsepart2 -> GIF-CON-unfold
        IF-CON-unfold : Celseifpart2 * Celsepart1 -> GIF-CON-unfold
        IF-CON-unfold : Celseifpart2 * Celsepart2 -> GIF-CON-unfold

        SOS-CON-unfold : Csequenceofstatements1 * Pcodelist -> GSOS-CON-unfold
        SOS-CON-unfold : Csequenceofstatements2 * Pcodelist -> GSOS-CON-unfold
```

File: Construct.str

5.1.3 The TR (And some embedded strategy) files

```
module IF
    imports
        Abstractsyntax

rules

//No elsif
IF-RUL-noelsif:
Cifstatement(a1,a2,a3,a4,Celseifpart2,a6,a7,a8,a9) ->
Pif(a2,a4,a6)

//Has elsif
IF-RUL-yeselsif:
Cifstatement(a1,a2,a3,a4,Celseifpart1(b1,b2,b3,b4,b5),a6,a7,a8,a9) ->
Pif(a2,a4,IF-CON-unfold(Celseifpart1(b1,b2,b3,b4,b5),a6))

//Unfold the last elsif
IF-RUL-nestedclose:
IF-CON-unfold(Celseifpart2, a2) ->
a2

//Unfold nested elsif (Push the else part deeper)
IF-RUL-nesteddown:
IF-CON-unfold(Celseifpart1(b1,b2,b3,b4,b5),a2) ->
IF-CON-unfold(b1, Pif(b3,b5,a2))

//Tidy up the else parts.
IF-RUL-tidynoelse:
Pif(a1,a2,Celsepart2) ->
Pif(a1,a2,Pemptycodelist)

IF-RUL-tidyyeselse:
Pif(a1,a2,Celsepart1(b1,b2)) ->
Pif(a1,a2,b2)

strategies

IF-STR-tidyone = oncetd(IF-RUL-tidyyeselse + IF-RUL-tidynoelse)

IF-STR-nestedstart = oncetd(IF-RUL-yeselsif)
IF-STR-nesteddown = repeat(oncetd(IF-RUL-nesteddown))
IF-STR-nestedclose = oncetd(IF-RUL-nestedclose)
IF-STR-nestedone = IF-STR-nestedstart ;
                   IF-STR-nesteddown ;
                   IF-STR-nestedclose ;
                   IF-STR-tidyone

IF-STR-nestedall = repeat(IF-STR-nestedone)

IF-STR-singledeal = oncetd(IF-RUL-noelsif)
IF-STR-singleone = IF-STR-singledeal ;
                   IF-STR-tidyone
IF-STR-singleall = repeat(IF-STR-singleone)

IF-STR-all = IF-STR-nestedall ;
             IF-STR-singleall
```

File: IF.str

```
module SOS
    imports
        Abstractsyntax

rules

//Sequence of statements

SOS-RUL-nestedtop:
Csequenceofstatements1(a1,a2) ->
SOS-CON-unfold(a1,Pcodelist(a2,Pemptycodelist))

SOS-RUL-nesteddown:
SOS-CON-unfold(Csequenceofstatements1(b1,b2), a2) ->
SOS-CON-unfold(b1,Pcodelist(b2,a2))

SOS-RUL-nestedfinal:
SOS-CON-unfold(Csequenceofstatements2(b1), a2) ->
Pcodelist(b1,a2)

SOS-RUL-singletop:
Csequenceofstatements2(a1) ->
Pcodelist(a1,Pemptycodelist)

strategies

SOS-STR-nestedstart = oncetd(SOS-RUL-nestedtop)
SOS-STR-nesteddown = repeat(oncetd(SOS-RUL-nesteddown))
SOS-STR-nestedfinal = oncetd(SOS-RUL-nestedfinal)
SOS-STR-nestedone = SOS-STR-nestedstart ;
                    SOS-STR-nesteddown ;
                    SOS-STR-nestedfinal
SOS-STR-nestedall = repeat(SOS-STR-nestedone)

SOS-STR-singleone = oncetd(SOS-RUL-singletop)
SOS-STR-singleall = repeat(SOS-STR-singleone)

SOS-STR-all = SOS-STR-nestedall ; SOS-STR-singleall
```

File: SOS.str

5.1.4 The TS file

```
module Strategies
    imports
        Abstractsyntax
        IF
        SOS

strategies
    eval = SOS-STR-all ; IF-STR-all
```

File: Strategies.str

5.1.5 The MT file

```
module Transform
    imports
```

```
        lib
        Strategies

strategies
    main=stdio (eval)
```

File: Transform.str

5.2 Examples

5.2.1 Example 1

Input:

```
echo -n "if.a.then.b.else.e.end if;" | \
sglr -2 -t -v -p Main.tbl | \
implode-asfix -S | \
transformer
```

Result:

```
Cprogram(Pif(Ccondition(".a."),Pcodelist(Cstatement(".b."),Pemptycodelist),
Pcodelist(Cstatement(".e."),Pemptycodelist)))
```

5.2.2 Example 2

Input:

```
echo -n "if.a.then.b.elsif.c.then.d.elsif.e.then.f.else.g.end if;" | \
sglr -2 -t -v -p Main.tbl | \
implode-asfix -S | \
transformer
```

Result:

```
Cprogram(Pif(Ccondition(".a."),Pcodelist(Cstatement(".b."),Pemptycodelist),Pif(Ccondition(".c."),
Pcodelist(Cstatement(".d."),Pemptycodelist),Pif(Ccondition(".e."),Pcodelist(Cstatement(".f."),
Pemptycodelist),Pcodelist(Cstatement(".g."),Pemptycodelist)))))
```

5.2.3 Example 3

Input:

```
echo -n "if.a.then.b..c..d.else.e.end if;" | \
sglr -2 -t -v -p Main.tbl | \
implode-asfix -S | \
transformer
```

Result:

```
Cprogram(Pif(Ccondition(".a."),Pcodelist(Cstatement(".b."),Pcodelist(Cstatement(".c."),
Pcodelist(Cstatement(".d."),Pemptycodelist))),Pcodelist(Cstatement(".e."),Pemptycodelist)))
```

5.3 Comments

- **Layout -** The file **Layout.sdf** describes the syntax matched by the special lexical item `LAYOUT`. It is implicit that `LAYOUT` can occur between every item in the grammer[7]. This can cause problems if `LAYOUT` greedily consumes too much layout at empty grammar rules. This is solved by restricting the layout consumption of `LAYOUT` using `LAYOUT? -/- [\ \n]`.
- **Component strategies -** The strategy for processing `if` statements is separately packaged to the strategy for processing sequences of statements. This aids debugging as it is straight forward to include and modify the strategies being used.

6 The remaining problems

The more difficult parts of the program transformation is implemented above. It should be relatively straight forward to enrich the grammar to full SPARK adding new transformation strategies accordingly. Thus the remaining issue in developing the SPARK program transformation system is deciding on its output. This is not a trivial task and will require dealing with:

- **Unique names -** The PSEUDO-SPARK will need a unique name for each construct. Although SPARK ensures a unique name for each construct exists, the names used to refer to the same construct may differ depending on scope. It is necessary to establish unique names for PSEUDO-SPARK constructs and use these consistently throughout all packages, nested packages and child packages.
- **Notations -** Notations are needed to describe the full SPARK language. This includes describing subprogram calls, multi-dimensional arrays, records and structures of structures (for example arrays of arrays).
- **Post processing -** STRATEGO will output a transformed term. It is still necessary to collate these terms together, following the SUBPROG-SPIDER approach, to produce the complete PSEUDO-SPARK.

7 Conclusions

STRATEGO is a powerful program transformation system which appears quite capable of supporting the conversion from SPARK to PSEUDO-SPARK. In particular, it describes the process at a high level, allowing for a clear and structured approach. The next phase in development is to establish the output desired from the STRATEGO part of program analysis and accordingly extend the above example to all of SPARK.

References

[1] Stratego. `http://www.stratego-language.org/`.

[2] Quôc Peyrot. Tiny expression evaluator using stratego. `http://www.lrde.epita.fr/~chojin/calc_tut.pdf/`.

[3] Eelco Visser. *The Stratego Reference Manual.* Institute of Information and Computing Sciences, Utrecht University, 0.5 edition, 2000.

[4] Eelco Visser. *The Stratego Tutorial.* Institute of Information and Computing Sciences, Utrecht University, 0.5 edition, 2000.

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

[7] In **Exp "*" Exp -> Exp** it is implicit that this means **Exp LAYOUT? "*" LAYOUT? Exp -> Exp**.