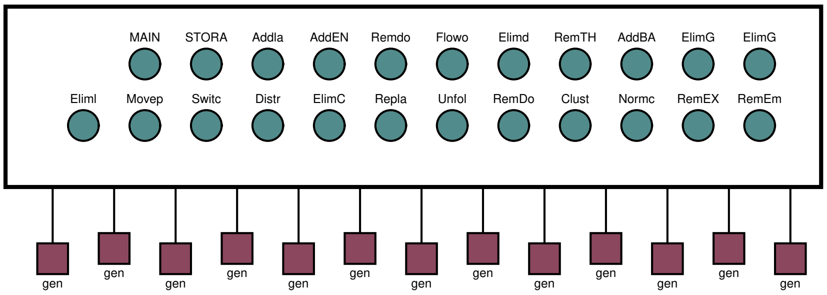
software
improvement
group



Source code transformations using the new ASF+SDF Meta-Environment

Hans Zaadnoordijk
March 2001

software
improvement
group

Source code transformations using the new ASF+SDF Meta-Environment

Hans Zaadnoordijk
March 2001

Masters Thesis

Informatica Instituut, Universiteit van Amsterdam
Sectie: Programmatuur
Afstudeerdocent: Prof. dr. P. Klint
Stagebegeleiders: Dr. S. Klusener & Prof. dr. C. Verhoef

Abstract

This masters thesis is about performing automatic source code transformation using the traversal functions in the new ASF+SDF Meta-Environment.

We will explain what these traversal functions are and how they should be used (by showing some example transformations on Cobol code). By means of an example on the toy language Pico, we will also show how much work can be saved by using the traversal functions.

We then use traversal functions to tackle a larger problem: the elimination of GOTOs in Cobol programs. This GOTO elimination has been performed before using the old ASF+SDF Meta-Environment, however the old ASF+SDF Meta-Environment turned out not to be powerful enough for such a complex transformation. We will create an implementation of the GOTO elimination which is powerful enough to even transform large Cobol programs.

Then we take the GOTO elimination one step further and create an algorithm which eliminates the need for user interaction to complete the GOTO elimination. We use this automated GOTO elimination algorithm to test the GOTO elimination on a large set of real-life Cobol programs from a large banking company.

Contents

Chapter 1

Introduction

1.1 Software maintenance

One of the most important parts of software engineering is not the construction of new code, but the maintenance of existing code. Code needs to be changed because bugs are found, requirements change, hardware changes, etc.

Software maintenance can be a difficult job, especially in the case of so-called legacy code, code that was written a long time ago and has been changed many times since then. Often the original author of legacy code is not around anymore, which means that the software maintainer must figure out what the code does by himself. In [5], page 535, it is stated that maintenance covers 49% of the costs of software.

To make software maintenance easier (and cheaper), it is useful to make sure the source code has certain nice properties ([3]). For example, a Cobol program is a lot easier to read when every `IF` statement is properly closed by an `END-IF` keyword, instead of a closing period (see section 3.1). Since older versions of Cobol (up to Cobol 74) didn't support the `END-IF` keyword, legacy code will often contain many non-properly closed `IF` statements.

1.2 The need for automatic transformations

Transforming source code (for example adding `END-IF` phrases) is a lot of work and can be dangerous. If the person performing the transformation makes a mistake, the program might (and probably will) no longer work correctly. Because of the amount of work involved in such transformations, because of the dangerous nature of such transformations and because such transformations are often not really necessary (only meant to improve readability), it is economically better not to perform them.

A solution could be automatic transformations. Once a system for performing automatic transformations has been developed, the transformations can be done fast and without risk (it is possible to prove that a set of transformation rules doesn't affect the way a program works and a transformation system makes no mistakes). In the case of automatic transformations, it *can* be profitable to perform a transformation. The development of a transformation system may be an expensive task, but after the system has been created, software maintenance becomes easier and thus cheaper. In the long run, automatic transformations *will* save money.

1.3 The ASF+SDF Meta-Environment

This masters thesis will discuss the creation of such automatic transformation systems using the new ASF+SDF Meta-Environment. One of the new features in this Meta-Environment is the possibility to use traversal functions. These traversal functions were designed to make creating transformation systems easier and to improve the performance of such systems. We will discuss the usefulness of the traversal functions and then use them to tackle the problem of eliminating GOTOs in Cobol programs.

1.4 Starting point and results of this research project

Starting point of the research project was the set of GOTO elimination rules by Sellink, Sneed and Verhoef ([16]) implemented using the 'old' ASF+SDF Meta-Environment. In this thesis we shortly discuss the migration of these rules to the 'new' ASF+SDF Meta-Environment.

The main contribution of this masters thesis is a generalization of the coordination algorithm (implemented as a ToolBus script) which enables the elimination of GOTOs on a large collection of source codes. We illustrate the application of this algorithm by a case study on a real-life banking system consisting of 85 sources with 2648 GOTO statements. The current version of the algorithm is able to eliminate 63% of all GOTOs. We will also show that extending the search patterns used in the transformation rules will increase the rate of eliminated GOTOs to at least 82%, possibly even 100%.

1.5 Structure of this masters thesis

Chapter 2 gives a short introduction to ASF+SDF and traversal functions. Section 2.5 shows the power of traversal functions by performing the same transformation twice: once *with* traversal functions, once *without* traversal functions.

We then show how to use the traversal functions in the new ASF+SDF Meta-Environment to create transformation tools on Cobol programs in chapter 3. Section 3.1 shows the creation of the 'Add `END-IF`' tool, section 3.2 shows the creation of the 'Nested `IF` to `EVALUATE`' tool.

Chapter 4 shows our approach to eliminating GOTOs in Cobol programs. In section 4.4 we implement the transformation rules of the GOTO elimination using the traversal functions in the new ASF+SDF Meta-Environment.

Using the implementation of these transformation rules, we create an algorithm to automatically perform the GOTO elimination in chapter 5. We implement this algorithm using the ToolBus in section 5.2.

Finally we test the GOTO elimination and discuss our results in chapter 6.

1.6 Acknowledgements

The research for this thesis has been supported by the Software Improvement Group (SIG). The supervisors of this project were Prof. dr. Chris Verhoef (VU) and dr. Steven Klusener (SIG). I would like to thank them for their support and suggestions both during the research and the writing of this thesis. I would also like to thank Prof. dr. Paul Klint (UvA, CWI) for his initiative to start this project and Jurgen Vinju (CWI) for his patience to answer my many questions about the ASF+SDF Meta-Environment and traversal functions.

Last but not least, I want to thank my sister for her moral support and my parents for making it possible for me to go to college and finish this study.

Chapter 2

ASF+SDF

This chapter gives a short introduction to ASF+SDF and the new ASF+SDF Meta-Environment, which we are going to use to develop tools for performing source code transformations. We will first give a global introduction and then focus on the built-in traversal functions, which make the transformations possible.

This chapter will also introduce some (naming-) conventions we are going to use throughout this masters thesis.

For more detailed information on the ASF+SDF Meta-Environment, see the user manual ([7]). For the latest release of the ASF+SDF Meta-Environment, see [6].

2.1 Introduction to ASF+SDF

[7] describes the ASF+SDF Meta-Environment as follows:

> The ASF+SDF Meta-Environment is an interactive development environment for the automatic generation of interactive systems for manipulating programs, specifications, or other texts written in a formal language.

The formalism that is used in the ASF+SDF Meta-Environment is ASF+SDF. It is a combination of the Algebraic Specification Formalism (ASF) and the Syntax Definition Formalism (SDF). ASF is a formalism that supports modularization and conditional equations. SDF has been developed to support the definition of lexical and context-free syntax.

The ASF+SDF Meta-Environment generates both scanners and parsers from ASF+SDF specifications as well as term rewriting systems. This makes it possible to execute ASF+SDF specifications, which in turn makes it possible to use the ASF+SDF Meta-Environment to perform source code transformations.

In order to perform a source code transformation, we first need an SDF specification of the grammar of the programming language the source code is written in. We then extend this specification with syntax-rules for functions that should perform a transformation. For example, we could extend the Cobol grammar `STDCOBOL`[1] (see [14]) with a function called `transform`. This function should take a Cobol program as

[1] `STDCOBOL` stands for 'Standard Cobol'. We call this grammar `STDCOBOL` to distinguish it from the grammar `EXTCOBOL`, which can be used to parse Cobol programs with embedded SQL and Cics commands.

argument and return a new Cobol program. The syntax definition of such a function in SDF looks like:[2]

```
module Transform

imports STDCOBOL

exports
  context-free syntax
    "transform" "(" Program ")" -> Program
```

The next step is to define the semantics of the function `transform` using ASF. This is done using labeled conditional equations. Each equation describes a rewrite step which can be performed if all conditions evaluate to 'true'. The execution of an ASF+SDF specification consists of applying equations leftmost innermost until no equation can be applied anymore.

The following example shows an equation and the necessary syntax definitions:

```
module ExampleFunction

exports
  sorts Word

  lexical syntax
    [a-z]+                        -> Word

  context-free syntax
    "ExampleFunction" "(" Word ")" -> Word

  variables
    "#Word"                [0-9]* -> Word

equations

  [1] #Word = hello
      ===============================
      ExampleFunction(#Word) = world
```

We have lexically defined the syntax of a sort `Word` to be something of one or more smallcase letters. `ExampleFunction` is defined as a function over sort `Word`. Equation `[1]` defines what function `ExampleFunction` should do. The equation has one condition: the word that is the argument of the function should be `hello`. If this condition evaluates to `true`, `ExampleFunction(#Word)` is rewritten to `world`.

Note that in the equation the token `#Word` is not part of the syntax we defined. It is a variable that can contain anything of sort `Word`.

[2] ASF+SDF allows us to define prefix functions without using quotes, as long as the functionname starts with a lower-case letter. So we could also write `transform(Program) -> Program` instead of `"transform" "(" Program ")" -> Program`.

2.2 Variables

As we have already seen, it is possible to use variables in the equations of an ASF+SDF module. A variable can take the place of anything that forms a sort, or a list of a sort. For example, variables can be of sort `Sentence`, `Stat`, `Stat*`, etc.

The syntax of variables is defined in a '`variables` section' of an ASF+SDF module. If, for example, we want to have a variable of sort `Sentence` from the Cobol grammar `STDCOBOL`, we can define it as follows:

```
module Transform

imports STDCOBOL

exports
  variables
    "#Sentence" [0-9]* -> Sentence
```

The variable declaration in this example declares a class of variables containing `#Sentence`, `#Sentence2`, `#Sentence0328`, etc. Each variable in this class ranges over sort `Sentence`.

Throughout this masters thesis, we follow the convention to name the variables after the sort they represent (since usually when transforming source code, there is no other logical alternative for naming variables), preceded by the symbol '`#`' and possibly followed by one or more numbers, as in the example above. The prefixing symbol '`#`' is not technically required; we use it to make it easier for the human reader to recognize variables in ASF+SDF equations.

From now on, we will not show variable declarations. The above mentioned naming convention makes it clear what is and what is not a variable.

2.3 Traversal functions

The ASF+SDF Meta-Environment has been under development for some years. During this time, it has also been used by a wide variety of users. The users and the developers of the ASF+SDF Meta-Environment have come up with many new ideas over the years. This resulted in a completely rewritten version of the ASF+SDF Meta-Environment, called the *new* ASF+SDF Meta-Environment. From now on with 'ASF+SDF Meta-Environment' we will mean the new ASF+SDF Meta-Environment.

One of the new features in the new ASF+SDF Meta-Environment is the possibility to use so called traversal functions ([8]). Because the functionality of traversal functions can be simulated using normal functions, we will sometimes speak of built-in traversal functions, especially in section 2.5, where we will compare the built-in traversal functions to simulated traversal functions.

Traversal functions are functions that traverse the parse tree of a text[3]. At every step of the traversal, a check is made whether there is an equation for the function that can be applied. If so, the equation is applied. If not, the traversal is continued.

Using traversal functions, we can apply a function to a whole program in order to make several changes to the program at lower levels, for example the level of statements. The following example shows how to use a traversal function on a Cobol program in order to change all statements in the program into `CONTINUE` statements:

[3]In the case of source code transformations, the text is a program.

```
module Stat2CONTINUE

imports STDCOBOL

exports

  context-free syntax
    "transform" "(" Program ")" -> Program {traverse}

  <variable declarations>

equations

  [1] transform(#Stat) = CONTINUE
```

The traversal starts at the top-sort, in this case `Program`. Since there is no equation that can be applied for this sort, the parse tree of the Cobol program is traversed until an equation can be applied for the sort `Stat`. Once a statement has been found, the equation is applied and the traversal doesn't continue downward (into the statement itself). Instead, the traversal backtracks and continues to traverse the rest of the program in order to find other statements that need to be transformed.

Since traversal functions traverse a parse tree and check for applicable equations at every level, traversal functions can be used to make transformations on different sorts in a program. For example, we could use one traversal function to transform both paragraphs and statements in a Cobol program.

It is also possible to define traversal functions with one or more extra arguments:

```
module Transform

imports STDCOBOL Integers

exports
  context-free syntax
    "transform" "(" Program "," Int ")" -> Program {traverse}
```

This example shows the syntax definition of a traversal function over sort `Program`, which takes one extra argument (an integer). This could be useful if for example we wanted to add 3 to all assignments of integers. After we added an equation that can perform such a transformation, we would simply call the function with 3 as the extra argument: `transform(<Program>, 3)` (where `<Program>` should be a Cobol program) would be rewritten to a new Cobol program where all assignments of integers have been changed. It is important to remember that the traversal is always performed over the first argument of a traversal function, in this case over sort `Program`.

Traversal functions are sort preserving. That means that any equation that defines the behaviour of a traversal function should rewrite from a certain sort to the same sort (for example `transform(#Stat)` should rewrite to something of sort `Stat`). It also means that in the syntax definition of a traversal function the output sort should be the same sort as the first argument of the function.

2.4 Accumulator functions

A special kind of traversal functions are the accumulator functions. These functions aren't used to transform source code, but to analyze source code[4]. Accumulator functions are defined in the same way as 'normal' traversal functions, except for the output sort. Like 'normal' traversal functions, the first argument of an accumulator function should be the top-sort of the traversal. However, in the case of accumulator functions, the output sort can be different from the sort of the first argument. To be more precise, the output sort should be the same as the sort of the second argument of the function.

As is the case with 'normal' traversal functions, a parse tree is traversed until an equation is found that can be applied to that level of the traversal. After this equation is applied, the traversal doesn't continue downwards, but backtracks and continues in other branches of the parse tree (as was the case for 'normal' traversal functions).

The next example shows an example of an accumulator function which counts `GO` statements in a Cobol program.

```
module Count-GO

imports STDCOBOL Integers

exports
  context-free syntax
    "count-go" "(" Program ")"          -> Int

hiddens
  context-free syntax
    "Count-GO" "(" Program "," Int ")" -> Int {traverse}

  <variable declarations>

equations

  [1] count-go(#Program) = Count-GO(#Program, 0)

  [2] #Stat = GO #Lab
      ================================
      Count-GO(#Stat, #Int) = #Int + 1
```

The actual accumulator function (`Count-GO`) is hidden. We use a normal function (`count-go`, with lower-case letters) with only one equation (labeled `[1]`) to call the hidden accumulator function with the start value `0` as second argument. The rewriting of additions of integers is handled by equations in the module `Integers`, which is imported by the example module.

2.5 The power of traversal functions

This section introduces Pico, a small toy language which is easy to understand and easy to define in ASF+SDF.

[4]Accumulator functions are also sometimes called analyzer functions, but to avoid confusion with simulated analyzer functions (see section 2.5) we prefer the name accumulator functions.

We will show that the built-in traversal functions in the new ASF+SDF Meta-Environment can be very useful by performing the same transformation on Pico programs twice; once using the built-in traversal functions, once without using them.

It is also possible to create a hand written accumulator function. However, simulating an accumulator function is very similar to simulating a traversal function. Therefore, we leave the details of writing a simulated accumulator function to the reader.

2.5.1 Definition of the Pico Language

Since Pico is a small toy language specifically meant to be used in examples, the definition of Pico is very simple. A Pico program consists of the keyword `begin`, followed by variable declarations, followed by a list of statements and finally the keyword `end`. In ASF+SDF:

```
"begin" DECLS {STATEMENT ";"}* "end" -> PROGRAM
```

This syntax-rule tells us that the list of statements consists of zero or more statements, separated by a semi-colon. This means that the last statement of the list is *not* followed by a semi-colon.

The variable declarations (`DECLS`) consist of the keyword `declare` followed by a list of declarations, followed by a semi-colon. In ASF+SDF:

```
"declare" {ID-TYPE ","}* ";" -> DECLS
```

We can see that the list of declarations consists of zero or more items of sort `ID-TYPE`, separated by a comma. Something of sort `ID-TYPE` binds each `PICO-ID` (variable) used in a Pico program to a `TYPE`:

```
PICO-ID ":" TYPE -> ID-TYPE
```

Pico has three simple statements: assignment, `if` and `while`:

```
PICO-ID := EXP                              -> STATEMENT
"if" EXP "then" {STATEMENT ";"}*
         "else" {STATEMENT ";"}* "fi"       -> STATEMENT
"while" EXP "do" {STATEMENT ";"}* "od"      -> STATEMENT
```

To complete the ASF+SDF definition of the Pico grammar, we need some more syntax rules. For example, we need to define what expressions (`EXP`) are, what a `PICO-ID` can look like and which `TYPE`s are allowed in Pico.

However, these details are not important for our example transformation, so we don't show them here. The complete specification of the Pico syntax is included in the official release of the ASF+SDF Meta-Environment ([6]).

2.5.2 Showing the power of traversal functions with Pico

With the toy language Pico we can show how powerful the built-in traversal functions of the new ASF+SDF Meta-Environment are.

Suppose we want to perform a transformation on Pico programs. As an example, we will replace `if` statements by `while` statements (the left column shows the situation before the transformation, the right column shows the situation after):

```
if <Exp> then                 while <Exp> do
  <Statements1>                 <Statements1>
else                          od
  <Statements2>
fi
```

Though this is a completely useless transformation, as an example it serves our purpose.

Transformation without traversal functions

We will first show how much work a simple transformation like this is if we don't use the built-in traversal functions. To make traversing lists of statements and lists of variable declarations easier (remember that it's not possible to have a list as output sort of a function), we made some small adjustments to the ASF+SDF specification of Pico: instead of writing {STATEMENT ";"}*, we introduced a new sort called STATEMENT-s which has the same meaning. This new sort is defined as:

```
                          -> STATEMENT-s
STATEMENT-p               -> STATEMENT-s
STATEMENT                 -> STATEMENT-p
STATEMENT ";" STATEMENT-p -> STATEMENT-p
```

So, something of sort STATEMENT-s can be nothing (in the case of zero statements), it can be one statement or it can be several statements separated by semi-colons. We made the same adjustment for type ID-TYPE-s.

These adjustments do not affect which Pico programs are correctly parsed, so the definition of Pico has not really been changed. However, we avoid using lists of a sort in order to make it easier to write our own traversal equations.

Since we want to perform the transformation on complete Pico programs, we start with a function on sort PROGRAM:

```
"if2while" "(" PROGRAM ")" -> PROGRAM
```

To start our 'manual traversal', we need a default equation that traverses into the different parts of the Pico program:

```
[default-01] #PROGRAM = begin
                          #DECLS
                          #STATEMENT-s
                        end
             ========================
             if2while(#PROGRAM) =
             begin
               if2while(#DECLS)
               if2while(#STATEMENT-s)
             end
```

We see that the function if2while must be overloaded, so the following two declarations are required:

```
"if2while" "(" DECLS       ")" -> DECLS
"if2while" "(" STATEMENT-s ")" -> STATEMENT-s
```

To complete our traversal equations, we need a declaration (syntax rule) for the function `if2while` over every sort in the Pico specification. We also need a default equation for each syntax rule in the specification. Our traversal ends when a syntax rule is reached which does not have any sorts on the left-hand side, for example:

```
"string" -> TYPE
```

The traversal also ends when a sort is defined by lexical syntax rules, in which case the function `if2while` over such a sort just returns the argument of the function:

```
[default-32] if2while(#PICO-NAT-CON) = #PICO-NAT-CON
```

In the case of Pico, to create a manual traversal of a Pico program we need 50 default equations. We use default equations because that way, the equation(s) that perform(s) the actual transformation overrule the traversal default equations. For the complete ASF+SDF module containing all 50 default equations necessary to traverse a Pico program, see appendix B.

Now that we have the default equations to traverse a Pico program, we only need one equation to perform the actual tranformation:

```
[1] #STATEMENT = if #EXP then
                   #STATEMENT-s1
                 else
                   #STATEMENT-s2
                 fi
    ============================
    if2while(#STATEMENT) =
    while #EXP do
      if2while(#STATEMENT-s1)
    od
```

Transformation with a traversal function

We now perform the same transformation using the built-in traversal functions of the new ASF+SDF Meta-Environment.

We will not need syntax rules for the function `if2while` over every sort in the Pico specification, but just one syntax rule for the top sort (in this case `PROGRAM`):

```
"if2while" "(" PROGRAM ")" -> PROGRAM {traverse}
```

This will take care of the traversal of Pico programs; we will not need the 50 default equations we needed when we didn't use traversal functions. We will also not need default equations to traverse a Pico program. The only equation that remains, is the above equation labeled `[1]`.

The use of traversal functions has reduced our Pico transformation tool from many syntax rules and 51 equations to just one syntax rule and just one equation. Pico is just a toy language. Real programming languages are a lot more complicated. A grammar specification of a real programming language consists of hundreds of syntax rules, which would also mean that we would need hundreds of default equations to traverse a program in such a language. This gives a good indication how much work we can save by using the built-in traversal functions.

2.5.3 Generation of default equations

The default equations we needed to manually perform a traversal, can be automatically generated. In fact, that approach has been succesfully used to perform source code transformations on Cobol programs in the past. For example, the GOTO elimination we will discuss in chapter 4 has been done with generated default equations similar to the default equations of section 2.5.2. The generated default equations of the GOTO elimination were more complicated than the default equations we showed, because they needed to carry extra information through the traversal. However, that is outside the scope of this chapter.

Generating such default equations also saves us the work of writing traversal equations. In that respect, the built-in traversal functions would not be necessary (just very handy). However, all those default equations take their toll on the ASF+SDF Meta-Environment. Though the GOTO elimination with generated equations was a success, it reached all limits of the ASF+SDF Meta-Environment and took a very long time to run.

The built-in traversal functions use the internal traversal of the ASF+SDF Meta-Environment. Because of this, performance of the built-in traversal functions is a lot better than the performance of traversals with generated default equations.

Another disadvantage of using generated default equations is the fact that it isn't possible anymore to use default equations for the actual transformation, since all possible uses of the function that performs the transformation already have a default equation. It is not possible to have two default equations for the same situation, since that would make the ASF+SDF module non-deterministic.

The built-in traversal functions do not 'use up' all possibilities for default equations, which gives ASF+SDF a 'third level of execution':

1. If a normal equation can be applied, it is applied.
2. If no normal equation can be applied, but a default equation *can* be applied, the default equation is applied.
3. If no normal equation and no default equation can be applied, the traversal function traverses one step further.

2.6 Remarks

2.6.1 Work under construction

ASF+SDF and the new ASF+SDF Meta-Environment are still work under construction. Even while the research for this masters thesis was done, some improvements were made to the ASF+SDF Meta-Environment, partially based on experience obtained by the research for this masters thesis.

Because of these improvements, some ASF+SDF code that worked fine in older versions of the ASF+SDF Meta-Environment, might not work in newer versions. Since some of the ASF+SDF code in this masters thesis was tested using older versions of the ASF+SDF Meta-Environment, this code might not work correctly in the current version. However, the code should be usable after some small adjustments.

2.6.2 A change in the definition of traversal functions

After the research for this masters thesis was done, a new version of the ASF+SDF Meta-Environment was released in which the syntax definition of traversal (and accumulator) functions has been changed. Instead of just defining the function for the top-level, the function should be defined for each sort over which the traversal function is used in the entire specification.

Using this new method of defining traversal functions, the first example of section 2.3 would look like:

```
module Stat2CONTINUE

imports STDCOBOL

exports
  context-free syntax
    "transform" "(" Program ")" -> Program {traverse}
    "transform" "(" Stat    ")" -> Stat    {traverse}
  variables
    "#Stat"              [0-9]* -> Stat

equations

  [1] transform(#Stat) = CONTINUE
```

The reason why this change was made, is that using the old style of syntax definition made it possible to apply traversal functions over each sort in a specification. Though this was very handy, it made parsetables of a syntax definition very large and therefore parsing a term was very slow. Also, the old method of defining traversal functions led to grammars with lots of ambiguities. Tests of executing ASF+SDF specifications using the new method of defining traversal functions have shown that this change has greatly improved the performance of executing ASF+SDF specifications which use traversal functions.

The downside of the change is that it has become necessary to manually specify over which sorts a traversal function will be applied in the equations, which makes using traversal functions a bit less user-friendly.

Chapter 3

Simple Cobol transformations

In this chapter we show two example transformations on Cobol programs. We discuss the creation of the tools 'Add `END-IF`' and 'Nested `IF` to `EVALUATE`' in great detail, in order to show how to use the built-in traversal functions of the new ASF+SDF Meta-Environment.

3.1 Adding `END-IF`

3.1.1 Why add `END-IF` keywords

Our first example is about adding `END-IF` keywords to `IF` statements.

Newer versions of Cobol (Cobol 85 and newer) use three different ways to terminate an `IF` statement:

- An `END-IF` keyword at the same level of nesting.
- An `ELSE` phrase associated with an `IF` statement at a higher level of nesting.
- A separator period (terminates all `IF` statements).

The first way is clearly the best way; the other two methods are confusing and make a Cobol program hard to understand for the human reader.

Unfortunately, up to Cobol 74, the `END-IF` keyword was not supported. This had more consequences than just the fact that it made code hard to read. Because an `IF` statement could only be closed by a separator period or by the `ELSE` phrase associated with an `IF` statement at a higher level of nesting, it wasn't possible to add statements in some places of a Cobol program.

Consider for example the following code fragment (the left column shows the fragment *without* using `END-IF` keywords, the right column shows the same code *with* `END-IF` keywords):

```
IF X = 1                        IF X = 1
  IF Y = 2                        IF Y = 2
    DISPLAY '2'                     DISPLAY '2'
  ELSE                            ELSE
    DISPLAY 'NOT 2'                 DISPLAY 'NOT 2'
ELSE                              END-IF
```

```
  DISPLAY ’NOT 1’.
```

```
ELSE
  DISPLAY ’NOT 1’
END-IF.
```

Suppose we want to add a statement (i.e. `DISPLAY ’1’`) inside the `THEN` phrase of the outer `IF` statement (so the statement should be before the second `ELSE` keyword), after the inner `IF` statement:

```
IF X = 1
  IF Y = 2
    DISPLAY ’2’
    DISPLAY ’1’
  ELSE
    DISPLAY ’NOT 2’
    DISPLAY ’1’
ELSE
  DISPLAY ’NOT 1’.
```

```
IF X = 1
  IF Y = 2
    DISPLAY ’2’
  ELSE
    DISPLAY ’NOT 2’
  END-IF
  DISPLAY ’1’
ELSE
  DISPLAY ’NOT 1’
END-IF.
```

In the right code fragment, we *can* add a statement where we want to. In the left code fragment, this is *not* possible. The only solution is code replication, as shown in the example.

In this small example program, it might not seem to be too big a problem to use code replication. However, in real-life problems, the replicated pieces of code would be much further apart. More importantly, the replicated pieces of code would not consist of just one statement, but of many lines of code. It is needless to say that code replication makes source code maintenance more difficult, dangerous and expensive.

Another example of why we would want to add `END-IF` keywords, is the case where we have two nested `IF` statements; an outer `IF` statement *with* an `ELSE` phrase and an inner `IF` statement *without* an `ELSE` phrase. Without an `END-IF` keyword, this is an impossible construction: there is no way for the compiler to know to which `IF` statement the `ELSE` phrase belongs. In fact, the compiler would assume that the `ELSE` phrase belongs to the inner `IF` statement.

A solution would be to use the `NEXT-SENTENCE` statement (the right column shows once again that there is no problem when we use `END-IF` keywords):

```
IF X = 1
  IF Y = 2
    DISPLAY ’2’
  ELSE
    NEXT-SENTENCE
ELSE
  DISPLAY ’NOT 1’.
```

```
IF X = 1
  IF Y = 2
    DISPLAY ’2’
  END-IF
ELSE
  DISPLAY ’NOT 1’
END-IF.
```

The solution with the `NEXT-SENTENCE` statement has been used very often in the past. So another transformation we might like to perform is removing `NEXT-SENTENCE` statements. However, we will not go into that problem in this masters thesis.

In the following sections we describe the generation of a tool in the new ASF+SDF Meta-Environment that adds `END-IF` keywords to all `IF` statements.

This tool is based on work done by Mark van den Brand, Alex Sellink and Chris Verhoef. They discuss their work in [10].

3.1.2 The Cobol grammar

The ASF+SDF specification of the 'Add END-IF' tool will turn out to be fairly simple. This simplicity is caused by the way IF statements are defined in the Cobol grammar we use (STDCOBOL, see [14]).

As we've seen, we can distinguish between properly closed IF statements (with an END-IF keyword) and non-properly closed IF statements. In the case of non-properly closed IF statements, the IF statement is always the last statement of a sentence, because the outer non-properly closed IF statement can only be closed by a separator period which closes the entire sentence.

```
Stat* IfNotClosed                  -> StatsOptIfNotClosed
Stat+                              -> StatsOptIfNotClosed

"IF" L-exp OptThen StatsOptIfNotClosed
     "ELSE" StatsOptIfNotClosed          -> IfNotClosed
"IF" L-exp OptThen StatsOptIfNotClosed  -> IfNotClosed

IfNotClosed "END-IF"                     -> Statx
Stat* IfNotClosed "."                    -> Sentence
```

The above ASF+SDF fragment shows the part of the Cobol grammar STDCOBOL where IF statements are defined. We see that an IF statement without an END-IF keyword is of sort IfNotClosed. If we add an END-IF keyword (IfNotClosed followed by "END-IF") we get a statement (sort Statx). So a properly closed IF statement can be part of a sentence, between other statements.

A non-properly closed IF statement optionally preceded by other statements and followed (closed) by a separator period forms a sentence (sort Sentence), which shows that a non-properly closed IF statement can only be at the end of a sentence.

The bodies of the THEN and ELSE phrases of an IF statement should be of sort StatsOptIfNotClosed (statements with an optional IfNotClosed at the end). Since a properly closed IF statement is of sort Statx, which in turn is injected into sort Stat (not shown in the above ASF+SDF fragment), the body of a THEN or ELSE phrase can contain multiple properly closed IF statements. If the body of a THEN or ELSE phrase contains a non-properly closed IF statement, it must once again be the last statement in the phrase.

3.1.3 The creation of the 'Add END-IF' tool

It is now easy to see what the 'Add END-IF' tool should do. Something of sort IfNotClosed should be followed by an END-IF keyword. Since something of sort IfNotClosed followed by END-IF is of sort Statx, this transformation cannot be done by a traversal function (remember that traversal functions are sort preserving).

To solve this problem, we apply the traversal function on a higher level in the parse tree of a Cobol program. The ASF+SDF fragment in section 3.1.2 shows that sort IfNotClosed can be part of two things: something of sort StatsOptIfNot-Closed or something of sort Sentence. This means that we will need two equations for our tool. One equation should transform something of StatsOptIfNotClosed if it contains a non-properly closed IF statement. The other equation should transform sentences which end with a non-properly closed IF statement:

```
module Add-END-IF
```

```
imports STDCOBOL

exports
  context-free syntax
    "aei" "(" Program ")" -> Program {traverse}

  <variable declarations>

equations

 [aei-1] #StatsOptIfNotClosed = #Stat*
                                #IfNotClosed,
         aei(#Stat*.) = #Stat*1.
         ====================================
         aei(#StatsOptIfNotClosed) =
         #Stat*1
         aei(#IfNotClosed)
         END-IF

 [aei-2] #Sentence = #Stat*
                     #IfNotClosed.,
         aei(#Stat*.) = #Stat*1.
         ===================================
         aei(#Sentence) =
         #Stat*1
         aei(#IfNotClosed)
         END-IF.
```

Function `aei` (Add End-If) is the traversal function that performs the transformation. As discussed in section 2.2, we have left out the variable declarations.

Note the condition `aei(#Stat*.)  = #Stat*1.` in both equations. What we actually wanted, was to apply function `aei` to the variable `#Stat*` and assign the result to variable `#Stat*1`. On first thought, we might write the condition as `aei(#Stat*) = #Stat*1` (without the periods). However, functions cannot have lists of sorts as output sort, so this condition isn't possible.

We solved this by adding the periods. This way, the function `aei` is applied over sort `Sentence`, not over a list of sort `Stat`. Since traversal functions always have the same sort as input and output, the output will also be of sort `Sentence`. Thanks to pattern matching, the assignment to variable `#Stat*1` still succeeds.

One might wonder why we apply the traversal function to the statements before the `IfNotClosed` in a sentence. This might seem contradictory to the fact that a non-properly closed `IF` statement can only be the last statement in a sentence. However, one of the earlier statements in the sentence could be a loop or a properly closed `IF` statement, of which the body contains a non-properly closed `IF` statement:

```
0001.
  DISPLAY '1'
  IF X = 1
    IF Y = 2
      DISPLAY '2'
    ELSE
      DISPLAY '3'
```

```
  ELSE
    DISPLAY '4'
  END-IF
  IF Z = 5
    DISPLAY '5'.
```

3.1.4 Some small examples

This section shows that the tool we created works the way we wanted it to work. We use some small examples to show this. The left column of each example shows the Cobol code *before* transformation, the right column shows the code *after* transformation.

`IF` closed by a separator period

Our first example shows that an `IF` closed by a separator period is transformed into an `IF` closed by an `END-IF`:

```
IF X=1                        IF X=1
  DISPLAY '1'                   DISPLAY '1'
ELSE                          ELSE
  DISPLAY '2'.                  DISPLAY '2'
                              END-IF.
```

`IF` closed by an `ELSE` from a higher `IF`

The second example shows that an `IF` statement that was closed by the `ELSE` phrase of a higher `IF` statement, gets it's own `END-IF` keyword:

```
IF X=1                        IF X=1
   DISPLAY '1'                  DISPLAY '1'
   IF X=2                       IF X=2
      DISPLAY '2'                 DISPLAY '2'
   ELSE                         ELSE
      DISPLAY '3'                 DISPLAY '3'
ELSE                            END-IF
   DISPLAY '4'.               ELSE
                                DISPLAY '4'
                              END-IF
```

Multiple nested `IF` statements closed by one separator period

The final example shows that multiple nested `IF` statements closed by one separator period all get their own `END-IF` keyword. This example shows the real power of adding the `END-IF` keyword using the ASF+SDF Meta-Environment. Our tool only uses two equations to get this job done, while other programming / scripting languages would use large codes to perform the `END-IF` addition (if it is possible at all)!

```
IF X=1                        IF X=1
  DISPLAY '1'                   DISPLAY '1'
  IF X=2                        IF X=2
    DISPLAY '2'                   DISPLAY '2'
    IF X=3                        IF X=3
      DISPLAY '3'                   DISPLAY '3'
```

```
IF X=4                        IF X=3
  DISPLAY '4'.                  DISPLAY '4'
                              END-IF
                            END-IF
                          END-IF
                        END-IF.
```

3.2 From Nested `IF` to `EVALUATE`

3.2.1 Why replace nested `IF` statements

Our second transformation example on Cobol programs is about transforming nested `IF` statements into `EVALUATE` statements.

Newer versions of Cobol support the `EVALUATE` statement. This statement takes a set of conditions and executes the statement(s) belonging to the first condition which is evaluated `true`. The `EVALUATE` statement can be compared to the `switch` statement in C, or the `Select Case` statement in (Visual) Basic. However, the `EVALUATE` statement is a bit more powerful than the `switch` statement or the `Select Case` statement, because of the `EVALUATE TRUE` construction. For a more detailed description of the `EVALUATE` statement, see almost any Cobol book (for example [13]) or on-line Cobol grammar ([18], [17]).

It is never *necessary* to use the `EVALUATE` statement, because the same functionality can also be achieved by using nested `IF` statements. However, using `EVALUATE` statements instead of nested `IF` statements improves the readability of a Cobol program. This is the reason why we want a tool that can detect and transform nested `IF` statements that could be replaced by an `EVALUATE` statement.

The next sections describe the creation of such a tool in the ASF+SDF Meta-Environment. We will first discuss which nested `IF` statements are suitable for transformation. Then we discuss which approach we will use to create the tool and finally we will create the actual tool.

3.2.2 Which nested `IF` statements to transform

To find out which nested `IF` statements can be transformed into an `EVALUATE` statement, we need to know what such a construction of nested `IF` statements looks like. We need to take into account that the `EVALUATE` statement can be of different forms.

The basic `EVALUATE` statement (right) and it's counterpart of nested `IF` statements (left) shows that the construction of nested `IF` statement should be of the form: an `IF` statement with in it's `ELSE` phrase just one statement, which in turn is an `IF` statement with in it's `ELSE` phrase just one statement, etc. The innermost `IF` statement can have more than one statement in it's `ELSE` phrase. These statements belong to the `WHEN OTHER` phrase of the `EVALUATE` statement:

```
IF X = 1                      EVALUATE X
  <statements>                  WHEN 1
ELSE                              <statements>
  IF X = 2                      WHEN 2
    <statements>                  <statements>
  ELSE                          WHEN 3
    IF X = 3                      <statements>
      <statements>              WHEN OTHER
```

```
    ELSE                             <statements>
      <statements>               END-EVALUATE
    END-IF
  END-IF
END-IF
```

Note that when the innermost IF statement has no ELSE phrase, the EVALUATE statement has no WHEN OTHER phrase.

The following example shows an example of an EVALUATE statement that uses the EVALUATE TRUE construction:

```
IF X = 1                     EVALUATE TRUE
  <statements>                 WHEN X = 1
ELSE                             <statements>
  IF Y > 2                     WHEN Y > 2
    <statements>                 <statements>
  END-IF                     END-EVALUATE
END-IF
```

This example shows that the conditions of the nested IF statements can be conditions over different variables. Also, it shows that the relation symbols in the conditions can be something else than the equality symbol. In those cases, the EVALUATE statement will use the EVALUATE TRUE construction.

We now know what kinds of nested IF statements can be transformed into an EVALUATE statement. We note that a non-nested IF statement could also be transformed into an EVALUATE statement, but that doesn't improve the readability of the Cobol program. Therefore, we will only transform *nested* IF statements.

3.2.3 Our approach

Though we could create a tool to transform nested IF statements into EVALUATE statements using only one traversal function, we choose to use a different approach. We use two separate traversal functions (so we actually create two transformations tools in one ASF+SDF module). Using two separate traversal function will decrease performance of the transformation, but it will increase the simplicity of the transformation tools, making it easier to understand and to develop. Our 'Nested IF to EVALUATE' tool will consist of these two functions:

- Function nif2eval will find all nested IF statements which are suitable for transformation and transform them into EVALUATE TRUE constructions.
- Function evalimprove will find all EVALUATE TRUE constructions that can be transformed into the basic form of an EVALUATE command (like the first example with EVALUATE X) and perform this transformation.

Our approach of using two traversal functions has another advantage. The second function will also rewrite EVALUATE TRUE constructions (if possible) that weren't created by the first function. So using two separate traversal functions extends the functionality of the transformation tool.

To apply the 'Nested IF to EVALUATE' transformation on a Cobol program, we will create an ASF+SDF term which looks like evalimprove(nif2eval(<Program>)), where <Program> should be the Cobol program to transform. After this term has been rewritten, the resulting term is the transformed Cobol program.

In order to keep our tool simple, we will assume that the Cobol program we get as input, only has properly closed IF statements. In other words, we assume that all IF statements are closed by a corresponding END-IF phrase. If the Cobol program we want to transform has IF statements that are not properly closed, we should first use the 'Add END-IF' tool described in section 3.1.

3.2.4 The creation of the 'Nested IF to EVALUATE' tool

Function nif2eval

We will first discuss the traversal function nif2eval, the function that finds the IF statements we want to transform and then transforms them.

In section 3.2.2 we discussed which IF statements to transform: all IF statements, where the ELSE phrase contains another IF statement, and nothing else. Our function nif2eval will have two equations: one for the case when the inner IF statement has *no* ELSE phrase and one for the case when the inner IF statement *does* have an ELSE phrase[1]. In the last case, it's possible that the ELSE phrase of the inner IF statement again contains only an IF statement. If so, the EVALUATE statement in the output of our tool will have one or more extra WHEN phrases. To create these phrases, we will need an extra function, which we call restWhen:

```
[eval1a] #Statx = IF #L-exp1 #OptThen
                    #Stat+1
                  ELSE
                    IF #L-exp2 #OptThen2
                      #Stat+2
                    ELSE
                      #Stat+3
                    END-IF
                  END-IF,
         #When+ = restWhen(#Stat+3),
         #Stat+11. = nif2eval(#Stat+1.),
         #Stat+12. = nif2eval(#Stat+2.)
         ===============================
         nif2eval(#Statx) =
         EVALUATE TRUE
           WHEN #L-exp1
             #Stat+11
           WHEN #L-exp2
             #Stat+12
           #When+
         END-EVALUATE
```

Function restWhen takes as input one or more statements (the body of the ELSE phrase of the inner IF statement) and returns one or more WHEN phrases. In order to be able to do this (the function can't return a list of sort When) we introduce a new sort Whens to the Cobol grammar which is defined as When+ -> When. A call to function restWhen will return something of this new sort. Since function restWhen is not a traversal function, it can return a different sort that it takes as argument.

[1] We don't show the equation for the case when the inner IF statement has no ELSE phrase, since this equation is almost identical to the equation for the case when the inner IF statement does have an ELSE phrase.

The statements in the THEN phrases of the IF statements and the statements in the ELSE phrase of the innermost IF statement should be traversed by our function nif2eval, in case they contain another nested IF structure that can be transformed into an EVALUATE statement. In other words, these statements need to be traversed in order to create nested EVALUATE statements.

Since the traversal functions in the ASF+SDF Meta-Environment don't work on lists of sorts (see section 2.3), we can't apply function nif2eval to the list of statements in the THEN phrases and the ELSE phrase of the innermost IF statement. We encountered the same problem in the 'Add END-IF' tool. We solved it by assigning the result of applying the traversal function to a list of statements to a variable (in the conditions of an equation). We then added a separator period to both sides of the condition, which made the call to the traversal function a call over sort Sentence instead of over a list of statements (see section 3.1.3). We use the same approach here.

Function restWhen

The function restWhen is only used in situations where we already know that we want to transform the nested IF statements into an EVALUATE statement. Therefore we don't have to check for nested IF statements (we know that we already are in a nested IF statement), we only have to check if the input list of statements consists of just one statement: an IF statement.

Function restWhen has three equations: one for the case of an IF statement with an ELSE phrase, one for the case of an IF statement without an ELSE phrase, and one default equation for all other cases:

```
[eval2] #Statx = IF #L-exp1 #OptThen
                   #Stat+1
                 ELSE
                   #Stat+2
                 END-IF,
        #When+ = restWhen(#Stat+2),
        #Stat+11. = nif2eval(#Stat+1.)
        ==============================
        restWhen(#Statx) =
        WHEN #L-exp1
          #Stat+11
        #When+

[eval2b] #Statx = IF #L-exp1 #OptThen
                    #Stat+1
                  END-IF,
         #Stat+11. = nif2eval(#Stat+1.)
         ==============================
         restWhen(#Statx) =
         WHEN #L-exp1
           #Stat+11

[default-eval] #Stat+1. = nif2eval(#Stat+.)
               ============================
               restWhen(#Stat+) =
               WHEN OTHER
                 #Stat+1
```

Function `evalimprove`

The first part of our tool (function `nif2eval`) is now finished. We can transform nested `IF` statements of the right form into `EVALUATE TRUE` structures. Function `evalimprove` should transform `EVALUATE` statements of the form `EVALUATE TRUE`, but only if all `WHEN` phrases are over the same expression and only if all `WHEN` phrases have the equality symbol as relation symbol. In order to check these conditions, we use two extra functions. Function `firstwhen` returns the first `WHEN` phrase of a list of `WHEN` phrases. Then function `allwhen` is used to check whether all `WHEN` phrases are over the same variable (sort `A-exp`) as the first `WHEN` phrase and whether all `WHEN` phrases have an equality as condition. If so, the function returns the boolean value `true`. If not, `false` is returned. In order to be able to use booleans, the module of our tool should import the existing module `Booleans`.

After all conditions have been checked, another function (called `convertwhens`) is used to transform the conditions of all `WHEN` phrases. For example: convertwhens(WHEN X = 2 <statements>) is transformed into WHEN 2 <statements>:

```
[ei-1] #Statx = EVALUATE TRUE
                  #When+
                END-EVALUATE,
       firstwhen(#When+) = WHEN #A-exp = #Lit-exp
                               #Stat+,
       allwhen(#When+, #A-exp) = true,
       #When+2 = convertwhens(#When+)
       ==========================================
       evalimprove(#Statx) =
       EVALUATE #A-exp
         #When+2
       END-EVALUATE
```

The equations of the functions `firstwhen`, `allwhen` and `convertwhens` are not shown, because they are pretty much straight-forward.

3.2.5 Example

The 'Nested `IF` to `EVALUATE` transformation tool is now finished. We show one example of what it can do. This example shows the creation of nested `EVALUATE` statements; one with the `EVALUATE TRUE` construction and one of the basic form:

```
IF X = 1                      EVALUATE X
  DISPLAY '1'                   WHEN 1
ELSE                              DISPLAY '1'
  IF X = 2                      WHEN 2
    DISPLAY '2'                   DISPLAY '2'
  ELSE                          WHEN OTHER
    DISPLAY '3'                   DISPLAY '3'
    IF Y > 4                      EVALUATE TRUE
      DISPLAY '4'                   WHEN Y > 4
    ELSE                              DISPLAY '4'
      IF Z = 5                      WHEN Z = 5
        DISPLAY '5'                   DISPLAY '5'
      END-IF                      END-EVALUATE
```

```
    END-IF                           END-EVALUATE
  END-IF
END-IF
```

Chapter 4

The elimination of GOTOs

In the previous chapters we've seen what traversal functions in the new ASF+SDF Meta-Environment are and what they can do (chapter 2). We've seen that it is easy to write source code transformation tools using traversal functions (chapter 3) and we've seen that the performance of the built-in traversal functions is a lot better than the performance of simulated traversal functions (section 2.5).

We will now use the built-in traversal functions to tackle a large transformation problem: the elimination of GOTOs. This transformation has been performed in the past using simulated traversal functions ([16]). However, the simulated traversal functions reached all limits of the old ASF+SDF Meta-Environment. Transforming a Cobol program took a long time and required a machine with lots and lots of memory. And even then, larger Cobol programs couldn't be handled.

We will take the old transformation rules and translate them for use with the built-in traversal functions. This gives us an opportunity to test the performance of the traversal functions and to test the GOTO elimination rules on a large set of Cobol programs.

This chapter introduces the problem of eliminating Cobol GO statements (also referred to as "GOTOs"). We discuss why it is important to eliminate GO statements and how to approach the problem of eliminating them. We give an overview of the GOTO elimination transformation rules[1] that were implemented using simulated traversal functions (see section 2.5) in the old ASF+SDF Meta-Environment by Sellink, Sneed and Verhoef.

The second part of this chapter is about the implementation of the transformation rules in the new ASF+SDF Meta-Environment. We will not go into the details of any one specific transformation rule, but we will show what problems we encountered while 'translating' the old tools and how we solved these problems.

4.1 Why eliminate GOTOs

In a Cobol program, normal control flow starts at the top of a program and continues down to the last statement at the bottom of the program. Cobol doesn't support the use of function calls (procedure calls), so control flow normally doesn't skip pieces of code.

[1] We use the term 'transformation rule' instead of 'transformation tool' to show that we are talking about what the transformations should do and not about how to implement them.

However, GOTOs change the normal control flow of a program. Control flow can jump anywhere in a program using GO statements. This makes it hard for the human reader of a program to determine how control flow progresses in a program. It is especially difficult to determine when a certain statement or piece of code is executed.

This makes maintenance of Cobol programs a difficult task. A piece of code should not be modified by someone who doesn't know when the piece of code is executed. Simple innocent changes to source code can have unexpected side effects when the changed piece of code turns out to be used for more than one task. This is why it is best to not use GOTOs. For more information on this topic, see [11].

In the past, GOTOs were widely used by programmers because software maintenance was not as big an issue as it has become nowadays. Also, older versions of Cobol (up to Cobol74) lacked certain features that could only be simulated using GO statements. Therefore, legacy Cobol code is usually full of GO statements[2]. In order to make software maintenance possible, it is necessary to eliminate those GOTOs and replace them by other constructs that are available in newer versions of Cobol.

Many techniques for eliminating GOTOs exist ([1], [2], [12]). However, most of these programs change the entire structure of the transformed source code, making the code almost impossible to read. The GOTO elimination algorithm of Sellink, Sneed and Verhoef [16] was designed to restructure a Cobol program, while changing as little as possible of the original program. This way, the resulting program is not only readable, but also maintainable.

4.2 Different types of GOTOs

GOTOs can be (and have always been) used to simulate many different programming constructs. In [9] we read:

> Already in the early days Cobol contained a GO TO statement. It has been used to simulate while constructs, conditional constructs, procedure calls, exit statements and more constructions that are nowadays standard in many programming languages, including COBOL85 dialects.

In general we can distinguish between two different types of GOTOs: local and distant ones.

4.2.1 Local GOTOs

Typical examples of local GOTOs are simulated conditional constructs (which often point to the next paragraph in the Cobol program) and simulated while constructs (which often point to the start of the current paragraph in a Cobol program).

Local GOTOs are relatively easy to eliminate. They can often be eliminated using just one ASF+SDF equation. The following example shows a simulated while construct (left) and it's transformed counterpart without GO statements (right):

```
PARLABEL.                    PARLABEL.
  IF X < 10                    PERFORM UNTIL NOT X < 10
```

[2]In chapter 6 we apply the GOTO elimination to a set of Cobol programs from a large banking company. This set of programs contains 49944 lines of code (including comments and blank lines) and 2648 GO statements. In other words, more than 5.3% of the lines of code in this set of programs contains a GO statement.

```
    DISPLAY X                      DISPLAY X
    MOVE X + 1 TO X                MOVE X + 1 TO X
    GO PARLABEL                  END-PERFORM.
  END-IF.
```

The problem with eliminating local GOTOs is that while constructs and conditional constructs can be simulated using GO statements in many different ways, depending among other things on programming style. Since we need an ASF+SDF equation for each different type of simulated construct when eliminating local GOTOs, we will get many equations. However, all those equations are *stand-alone* (just one equation to eliminate a certain type of simulated construct) and easy to understand.

Since it is virtually impossible to write an ASF+SDF module which can eliminate all types of simulated constructs (there is always a programmer who does things a little different from anything you ever encountered), we will use a module with equations for many types of constructs and add an equation whenever we encounter a 'new' type of simulated construct.

We will also use preprocessing to make eliminating GO statements a little easier. For example, applying the 'Add END-IF' tool (see section 3.1) to a Cobol program decreases the number of types of simulated constructs dramatically. We will no longer need equations for transforming simulated constructs using non-properly closed IF statements.

More information on the elimination of local GOTOs can be found in [9]. This paper handles eliminating local GOTOs using simulated traversal functions in the old ASF+SDF Meta-Environment (see section 2.5 for information on simulated traversal functions). Also see [1], [2] and [12] for more information on this topic.

4.2.2 Distant GOTOs

Distant GOTOs are more difficult to eliminate, because it's not possible to recognize a pattern. Distant GOTOs can occur anywhere in a program and point to anywhere in a program. Typical examples of distant GOTOs are simulated procedure calls, simulated exit statements and wild GOTOs. Wild GOTOs are GO statements that aren't used to simulate any type of construct but that are the result of sloppy programming. (Good software engineering includes preventing the use of GOTOs, since they make a program unstructured and hard to maintain.)

Most modern programming languages offer the possibility to write functions (procedures); pieces of code that aren't executed by normal control flow. Functions are executed when they are called in the source code of the main section of a program. After a function has been executed, control flow returns to the statement that called the function.

Since Cobol doesn't have support for functions, programmers often use GOTOs to simulate function calls. A piece of code (the function) would be made unreachable to control flow by placing a GOTO (or a STOP RUN statement) directly before this code. A function call would then consist of a GOTO to the piece of code that makes up the function. A variable could be used to tell the function where the function call was made from. Depending on this variable, the code of the function ends with a GOTO to the statement following the 'function call' (using a GO DEPENDING statement).

The following example shows such a simulated function call. Variable JUMPPOS is used to keep track of where the simulated function was called. Suppose the simulated function gets called in three places, then the variable JUMPPOS can have values 1, 2 and 3. The example shows the case where the variable has value 2. This value is used by the GO DEPENDING[3] statement at the end of the simulated function to return control flow to the place directly after the simulated function call. In this case, control flow is returned to the paragraph with the label RETURNPOSITION:

```
...
...
  MOVE 2 TO JUMPPOS
  GO SIM-FUNCTION.

RETURNPOSITION.
  <statements to be executed after function call>
...
...
  GO SOMEWHERE.

SIM-FUNCTION.
  <statements that make up simulated function>
  GO <some paragraph label>
     RETURNPOSITION
     <some more paragraph labels>
     DEPENDING JUMPPOS.
...
...
```

The elimination of distant GOTOs will be done by creating simulated function calls without the use of GO statements. This can be done by finding pieces of code (paragraphs) that are unreachable to control flow and that do not contain any GOTOs. These paragraphs are then moved to a place in the program after a STOP RUN statement. Instead of using GOTOs, we use the PERFORM statement to call such a paragraph. Using a PERFORM statement also takes care of returning to the place where the call was made.

4.3 The rules by Sellink, Sneed and Verhoef

In [16] Sellink, Sneed and Verhoef present an algorithm for the elimination of GOTOs in Cobol programs. Their algorithm consists of a collection of transformation rules which, when applied in the right order, eliminate GOTOs in the way we described above: local GOTOs are replaced by the constructs they simulate and distant GOTOs are eliminated by creating simulated procedure calls using PERFORM statements.

Besides creating simulated procedure calls, distant GOTOs are also eliminated by restructuring the Cobol program in such a way that some distant GOTOs become local GOTOs, which can be eliminated in a relatively easy way.

Sellink, Sneed and Verhoef use three different kinds of transformation rules: preprocessing rules, postprocessing rules and the actual GOTO elimination rules.

The preprocessing rules are used to make sure that the Cobol program to be transformed has certain properties which make the other tools simpler because less patterns

[3]The GO DEPENDING statement jumps to some paragraph depending on the value of a (numeric) variable. If this variable has value 1, a jump is made to the first label from the list of paragraph labels, if the variable has value 2, a jump is made the the second label, etc.

have to be recognized. A typical example of such a preprocessing rule is the addition of `END-IF` phrases (see section 3.1).

Postprocessing rules are necessary to clean up the transformed code. To keep the transformation rules as simple as possible, some 'unnecessary' Cobol constructions are introduced (like `CONTINUE` statement, empty sections and conditional expressions of the form `NOT(NOT(NOT(<condition>))))`. Postprocessing rules eliminate `CONTINUE` statements and empty sections, normalize conditional expressions, etc.

The actual GOTO elimination rules are the rules that do the real work. There is one rule to eliminate local GOTOs[4]. Some other transformation rules can, when applied in the right order, eliminate distant GOTOs.

The part of the GOTO elimination algorithm between the preprocessing and the postprocessing (the part that applies the actual GOTO elimination rules) will be referred to as the main loop of the algorithm.

The transformation rules by Sellink, Sneed and Verhoef were originally created to transform source code from one company. This source code had certain properties (due to programming style and the used Cobol version) which might not be true for other Cobol programs. Since the rules by Sellink, Sneed and Verhoef rely on some of these properties, we cannot correctly transform Cobol programs which don't have them.

We solved this problem by creating some extra preprocessing rules: 'Add labels', 'Remove `EVALUATE`' and 'Remove `NEXT-SENTENCE`'. We will first discuss these three transformation rules and then continue with the 'old' rules.

Most information (except for the information about the 'Add labels', 'Remove `EVALUATE`' and 'Remove `NEXT-SENTENCE`' transformation rules) in the following sections has been taken from [16].

4.3.1 Preprocessing rules

Add labels

One assumption the old transformation rules make is that the `PROCEDURE DIVISION` of a Cobol program consists of only `SECTION`s and that each section consists of only paragraphs.

Some versions of Cobol allow a programmer to start the `PROCEDURE DIVISION` with paragraphs or statements and to start `SECTION`s with statements. The following example shows a piece of source code with missing paragraph and `SECTION` labels. It also shows what the piece of code looks like after we applied the 'Add labels' transformation rule:

```
PROCEDURE DIVISION          PROCEDURE DIVISION

  DISPLAY '1'               FIRSTSEC-OF-PRG SECTION.
  DISPLAY '2'.
                            FIRSTPAR-OF-FIRSTSEC-OF-PRG.
CALCULATE SECTION.            DISPLAY '1'
                              DISPLAY '2'.
  MOVE X + 1 TO Y.
                            CALCULATE SECTION.
```

[4] Actually, there is one ASF+SDF module containing many equations, each equation for a different pattern. So it is more accurate to say that this one module handles many transformation rules. However, all these transformation rules serve the same purpose: replacing simulated constructs by the constructs themselves. Therefore we will refer to the rules as if we are talking about just one transformation rule.

```
DISPLAYRESULT.
  DISPLAY Y.                 FIRSTPAR-OF-CALCULATE.
                               MOVE X + 1 TO Y.

                             DISPLAYRESULT.
                               DISPLAY Y.
```

Remove EVALUATE

The ’Distribute’ and ’Cluster’ transformation rules make some assumptions on the form of EVALUATE statements. To be precise, they assume all EVALUATE statements are of the form:

```
EVALUATE <variablename>
  WHEN 1   <statements>
  WHEN 2   <statements>
  ...
  WHEN <n> <statements>
END-EVALUATE
```

EVALUATE statements of this form are created by the ’Eliminate GO DEPENDING’, so what the ’Distribute’ and ’Cluster’ rules actually assume is that the original Cobol program didn’t contain any EVALUATE statements.

To make certain that the ’Distribute’ and ’Cluster’ rules don’t perform illegal transformations, we created the preprocessing ’Remove EVALUATE’ transformation rule. It transforms EVALUATE statements into nested IF statements:

```
EVALUATE X                    IF X = 13
  WHEN 13                       DISPLAY '13'
    DISPLAY '13'              ELSE
  WHEN 87                       IF X = 87
    DISPLAY '87'                  DISPLAY '87'
  WHEN OTHER                    ELSE
    DISPLAY 'OTHER'               DISPLAY 'OTHER'
END-EVALUATE                    END-IF
                              END-IF
```

After the GOTO elimination is done, we use the ’Nested IF to EVALUATE’ rule from section 3.2 to transform the nested IF statements back into EVALUATE statements.

Remove NEXT-SENTENCE

This transformation rule eliminates NEXT-SENTENCE statements by replacing them by CONTINUE statements:

```
IF X = 1                      IF X = 1
  NEXT-SENTENCE                 CONTINUE
ELSE                          ELSE
  DISPLAY '2'                   DISPLAY '2'
END-IF                          DISPLAY '3'
DISPLAY '3'.                  END-IF.
DISPLAY '4'.                  DISPLAY '4'.
```

The NEXT-SENTENCE statement acts as a GOTO: control flow skips to immediately after the end of the current sentence. Since the 'Remove separator period' transformation rule changes the place where sentences end, the NEXT-SENTENCE statement might suddenly skip to a different place. Therefore we need to eliminate these statements. The following example shows an incorrect transformation which could occur if we don't eliminate NEXT-SENTENCE statements:

```
IF X = 1                        IF X = 1
  NEXT-SENTENCE                   NEXT-SENTENCE
ELSE                            ELSE
  DISPLAY '2'                     DISPLAY '2'
END-IF                          END-IF
DISPLAY '3'.                    DISPLAY '3'
DISPLAY '4'.                    DISPLAY '4'.
```

The left column shows a code fragment which displays 4 if X equals 1. The right column (which is the result of applying the 'Remove separator periods' transformation rule) doesn't display 4 if X equals 1.

Add END-IF

The 'Add END-IF' transformation rule (and the ASF+SDF implementation) has been discussed in detail in section 3.1. It adds the explicit scope terminator END-IF to IF statements.

Many of the following transformation rules rely on the fact that all IF statements are properly closed by an END-IF phrase. Therefore, this transformation rule should be one of the first rules to be applied.

Remove separator periods

This transformation rule (in short called 'RemDots') removes separator periods between statements:

```
0001.                           0001.
  DISPLAY '1'.                    DISPLAY '1'
  DISPLAY '2'.                    DISPLAY '2'
  DISPLAY '3'.                    DISPLAY '3'.
```

Many statements in Cobol can be ended with a separator period at wish. This transformation rule removes all unnecessary ones, creating syntactically more uniform code.

As was the case with the 'Add END-IF' rule, many of the following transformation rules rely on the fact that the 'Remove separator periods' transformation rule has already been applied. Therefore, this transformation rule should also be one of the first rules to be applied.

Other transformation rules might introduce unnecessary separator periods. Therefore, the 'RemDots' transformation rule is also used in the main loop, to assist the actual GOTO elimination rules.

Flow optimizer

This transformation rule optimizes the control-flow of `IF` statements:

```
IF X = 1                      IF X = 1 AND Y = 2 AND Z = 3
  IF Y = 2                      DISPLAY '123'
    IF Z = 3                  END-IF
      DISPLAY '123'
    END-IF
  END-IF
END-IF
```

We use this transformation rule to decrease the number of patterns we need to recognize when eliminating local GOTOs. However, since some transformation rules also make the control-flow of `IF` statements unnecessary complicated, this transformation rule is also used as a postprocessing rule.

Eliminate dead code

The 'Eliminate dead code' transformation rule eliminates certain types of dead code (code that is unreachable and therefore unnecessary). To be precise, this transformation rule eliminates code that appears below explicit jump instructions:

```
0001.                         0001.
  DISPLAY '1'                   DISPLAY '1'
  DISPLAY '2'                   DISPLAY '2'
  GO 0002                       GO 0002.
  DISPLAY '3'.
```

Since the algorithm for eliminating GOTOs moves code around in a Cobol program, it is possible that during the algorithm new dead code is created. Therefore, the 'Eliminate dead code' transformation rule is also used in the main loop, just like the 'Remove separator periods' transformation rule.

Remove `THEN`

In Cobol two statements can be connected to each other by the `THEN` keyword:

```
0001.                         0001.
  DISPLAY '1'                   DISPLAY '1'
  DISPLAY '2'                   DISPLAY '2'
  THEN                          DISPLAY '3'.
  DISPLAY '3'.
```

This keyword has no meaning; `DISPLAY '1' THEN DISPLAY '2'` is the same as `DISPLAY '1' DISPLAY '2'`. To decrease the number of patterns that need to be recognized by the GOTO elimination transformation rules, we use the 'Remove `THEN`' transformation rule to remove all occurrences of the `THEN` keyword.

Since traversal functions are sort preserving, a traversal function can't change one statement into a list of statements. In case we do wish to perform such a transformation, we change a statement into several statements connected by `THEN` keywords. Such a list of statements is of sort `Stat` (it is seen as one statement), which makes the transformation possible. We then use the 'Remove `THEN`' transformation rule to remove the `THEN` keywords.

The transformation rules of the GOTO elimination introduce THEN keywords. Therefore, this transformation rule will also be used in the main loop of our algorithm.

Add BAR **section**

As discussed in section 4.2.2, our GOTO elimination algorithm will simulate function calls by placing paragraphs of Cobol programs in a place that can't be reached by normal control-flow. We create such a place by creating a so called BAR section. This section contains a BAR paragraph, which consists of just the STOP RUN statement:

```
BAR SECTION.

BAR-PARAGRAPH.
  STOP RUN.
```

The 'Add BAR section algorithm' creates such a BAR section. It also creates so called SUBROUTINES sections behind the BAR section for each section in the original Cobol program. For example, if the original Cobol program has a section named CALCULATE SECTION, this transformation rule creates a section called CALCULATE-SUBROUTINES SECTION behind the BAR section.

Eliminate GO DEPENDING

This transformation rule eliminates GO DEPENDING statements by replacing them by EVALUATE statements containing a WHEN phrase for every label in the GO DEPENDING:

```
GO 0007                        EVALUATE X
   0002                          WHEN 1 GO 0007
   1202                          WHEN 2 GO 0002
   0309 DEPENDING X              WHEN 3 GO 1202
                                 WHEN 4 GO 0309
                               END-EVALUATE
```

It might seem counterproductive to introduce new GOTOs. However, the introduced GO statements can be eliminated by our GOTO elimination algorithm, while the GO DEPENDING statement could not.

4.3.2 GOTO elimination rules

Eliminate local GO

As discussed before, the 'Eliminate local GO' tool we are going to create actually consists of many equations which perform many transformation rules. However, because all these transformation rules have a similar purpose, we speak of just one transformation rule.

In section 4.2.1 we discussed a transformation rule that eliminates local GOTOs by replacing them by the constructs they simulate (a simulated while construct is replaced by the Cobol while construct PERFORM UNTIL NOT, etc.).

The 'Eliminate local GO' transformation rule does not only replace simulated constructs, it also eliminates other simple GOTOs, for example GO commands at the end of a paragraph which point directly to the next paragraph:

```
0001.                         0001.
  DISPLAY '1'                   DISPLAY '1'
  GO 002.
                              0002.
0002.                           DISPLAY '2'.
  DISPLAY '2'.
```

Though such GO statements are usually not present in the original Cobol code (because they serve no purpose at all), they might occur in the code after some steps in our GOTO elimination algorithm, due to moving code around.

Another type of transformation that is handled by this rule is shown in the following example:

```
0001.                         0001.
  DISPLAY '1'                   DISPLAY '1'.
  GO 0004.
                              0004.
0002.                           DISPLAY '4'
  DISPLAY '2'.                  GO 0006.

0003.                         0002.
  DISPLAY '3'                   DISPLAY '2'.
  GO 0005.
                              0003.
0004.                           DISPLAY '3'
  DISPLAY '4'                   GO 0005.
  GO 0006.
```

This example shows a transformation that can be described as follows: paragraphs that can be freely moved (because they end with a GOTO and are preceded by a GOTO) are moved to the place where a GOTO to this paragraph was used (if possible). The GOTO to that paragraph can then be eliminated.

This transformation actually eliminates distant GOTOs. However, since the transformation is simple (it can be implemented using only one equation), we include it in the 'Eliminate local `GO`' transformation rule rather than creating a separate tool. Since this means that the 'Eliminate local `GO`' transformation rule will also eliminate distant GOTOs, we will speak of the 'Eliminate `GO`' rule ('ElimGO') instead.

The 'Eliminate `GO`' transformation rule is the most complicated of all the rules involved in the GOTO elimination, because it performs so many tasks. However, each task on itself is simple and easy to implement.

Move paragraphs

This is the transformation rule that moves paragraphs of a Cobol program to the corresponding `SUBROUTINES` section behind the `BAR` section. Such a paragraph then becomes a 'simulated function' which can be called using the `PERFORM` statement:

```
EXAMPLE SECTION.              EXAMPLE SECTION.

0001.                         0001.
  DISPLAY '1'                   DISPLAY '1'
  GO 0003.                      PERFORM 0003
                                GO 0004.
```

```
0002.
  DISPLAY '2'.                0002.
  GO 0004.                      DISPLAY '2'
                                GO 0004.
0003.
  DISPLAY '3'.                0004.
                                DISPLAY '4'.
0004.                           GO 0006.
  DISPLAY '4'
  GO 0006.                    ...

...                           BAR SECTION.

BAR SECTION.                  BAR-PARAGRAPH.
                                STOP RUN.
BAR-PARAGRAPH.
  STOP RUN.                   EXAMPLE-SUBROUTINES SECTION.

EXAMPLE-SUBROUTINES SECTION.  0003.
                                DISPLAY '3'.
```

The 'Move paragraphs' transformation rule doesn't always decrease the number of GOTOs (in the example, the right column has the same number of GOTOs as the left column). However, it does change the structure of a Cobol program in a way which could cause distant GOTOs to become local GOTOs. These GOTOs can then be eliminated by the 'ElimGO' transformation rule.

This behaviour can be observed in the example: the `GO 0004` statement in paragraph `0002` is transformed from a distant GOTO (in the left column) into a local GOTO (in the right column).

A paragraph that can be moved must be free of GOTOs. The 'Move paragraphs' transformation rule uses an accumulator function called `CountGO` to check this. We will create a separate ASF+SDF module which contains this accumulator function, so that we can also use it as a separate tool to create statistics (see chapter 6).

Eliminate `CONTINUE`

Some of the transformation rules introduce `CONTINUE` statements in a Cobol program. This might be necessary to prevent 'empty bodies'. For example, when the only statement in the body of an `IF` statement is removed, an empty body remains. Since empty bodies are not allowed in Cobol programs, a `CONTINUE` statement is put in the place of the removed statement. This transformation rule eliminates these `CONTINUE` statements:

```
IF X = 1                      IF NOT (X = 1)
  CONTINUE                      DISPLAY '2'
ELSE                          END-IF
  DISPLAY '2'
END-IF
```

Though the 'Eliminate `CONTINUE`' transformation rule seems to be a typical post-processing rule, we use it inside the main loop of the GOTO elimination. The reason for this is that eliminating `CONTINUE` statements sometimes simplifies the structure of Cobol programs. This can lead to the creation of patterns that can be recognized by the 'ElimGO' transformation rule, while the old pattern (with CONTINUE statements) was *not* recognized.

Eliminate labels

Due to the elimination of GO statements, some paragraph labels in a Cobol program may become unnecessary (unreferenced in GO and PERFORM statements). This transformation rule eliminates these labels.

By eliminating unnecessary labels, distant GOTOs can become local GOTOs, as shown in the following example. This way, the 'ElimGO' transformation rule can eliminate those GOTOs:

```
0001.                           0001.
  IF X = 1                        IF X = 1
    GO 0003                         GO 0003
  END-IF.                         END-IF
                                  DISPLAY '2'.
0002.
  DISPLAY '2'.                  0003.
                                  DISPLAY '3'.
0003.
  DISPLAY '3'.
```

Switch paragraphs

This transformation rule is a variation of the 'Move paragraphs' rule. This one is used when a paragraph is not free of GOTOs. Such a paragraph is not moved to a place behind the BAR section, but to a different place in the program (if possible). This way, distant GOTOs might become local GOTOs, making it possible to eliminate them with the 'ElimGO' tranformation rule:

```
0001.                           0001.
  DISPLAY '1'                     DISPLAY '1'
  GO 0003.                        GO 0003.

0002.                           0003.
  DISPLAY '2'                     DISPLAY '3'
  GO 0007.                        GO 0008.

0003.                           0002.
  DISPLAY '3'                     DISPLAY '2'
  GO 0008.                        GO 0007.
```

The 'Switch paragraphs' is typicaly applied when the other transformation rules can't continue (when the GOTO elimination is 'stuck').

Distribute

This transformation rule optimizes EVALUATE and IF statements. For EVALUATE statements, it distributes common statements that occur in all the WHEN phrases outside the EVALUATE. For IF statements, it distributes common statements that occur in both the THEN and ELSE phrases outside the IF:

```
0001.
  IF X = 1
    DISPLAY '1'
    DISPLAY '3'
  ELSE
    DISPLAY '2'
    DISPLAY '3'
  END-IF.

0002.
  EVALUATE X
    WHEN 1
      DISPLAY '1'
      DISPLAY '4'
    WHEN 2
      DISPLAY '2'
      DISPLAY '4'
    WHEN 3
      DISPLAY '3'
      DISPLAY '4'
  END-EVALUATE.
```

```
0001.
  IF X = 1
    DISPLAY '1'
  ELSE
    DISPLAY '2'
  END-IF
  DISPLAY '3'.

0002.
  EVALUATE X
    WHEN 1
      DISPLAY '1'
    WHEN 2
      DISPLAY '2'
    WHEN 3
      DISPLAY '3'
  END-EVALUATE
  IF X >= 1 AND X <= 3
    DISPLAY '4'
  END-IF
```

When the common statements that are distributed are GOTOs, the 'Distribute' rule can create patterns that can be recognized by the 'ElimGO' transformation rule.

4.3.3 Postprocessing rules

Replace paragraphs

This transformation rule locates paragraphs in the SUBROUTINES sections behind the `BAR` section that are equal. It replaces all references to both paragraphs by references to one of the two.

We will consider paragraphs to be equal when the only difference between the two is the paragraph label:

```
EXAMPLE SECTION.

0001.
  PERFORM 0003.

0002.
  PERFORM 0004.

BAR SECTION.

BAR-PARAGRAPH.
  STOP RUN.

EXAMPLE-SUBROUTINES SECTION.

0003.
  DISPLAY '3'.

0004.
  DISPLAY '3'.
```

```
EXAMPLE SECTION.

0001.
  PERFORM 0003.

0002.
  PERFORM 0003.

BAR SECTION.

BAR-PARAGRAPH.
  STOP RUN.

EXAMPLE-SUBROUTINES SECTION.

0003.
  DISPLAY '3'.
```

Unfold paragraphs

This transformation rule replaces indirect function calls to direct function calls:

```
0001.                          0001.
  PERFORM 0002.                  PERFORM 0003.

BAR SECTION.                   BAR SECTION.

BAR-PARAGRAPH.                 BAR-PARAGRAPH.
  STOP RUN.                      STOP RUN.

EXAMPLE-SUBROUTINES SECTION.   EXAMPLE-SUBROUTINES SECTION.

0002.                          0003.
  PERFORM 0003.                  DISPLAY ’3’.

0003.
  DISPLAY ’3’.
```

Remove doubles

This transformation rule finds ’double paragraphs’, two paragraphs with the same label and the same contents. It then throws one of these two paragraphs away.

Cluster

This transformation rule clusters `WHEN` phrases of `EVALUATE` statements when the cases are equal:

```
EVALUATE X                     EVALUATE X
  WHEN 1                         WHEN 1
    PERFORM 0001                   PERFORM 0001
  WHEN 2                         WHEN 2 THRU 4
    PERFORM 0002                   PERFORM 0002
  WHEN 3                         WHEN 5
    PERFORM 0002                   PERFORM 0005
  WHEN 4                       END-EVALUATE
    PERFORM 0002
  WHEN 5
    PERFORM 0005
END-EVALUATE
```

Normalize conditional expressions

During the GOTO elimination algorithm, conditional expressions are changed. This might result in unnecessary complex expressions, like `NOT(NOT(NOT(X<>Y)))`. This transformation rule normalizes such conditional expressions. Our example would be normalized to `X=Y`.

Remove EXIT paragaphs

This transformation rule removes superfluous EXIT paragraphs. The EXIT statement is comment with the intention to mean that when the control is at the EXIT statement, it will leave the current section. Since we removed all GOTO logic, there are no jumps to EXIT paragraphs anymore so we remove such paragraphs.

Remove empty sections

The 'Add BAR section' transformation rule creates empty sections (behind the BAR section) that may or may not remain empty during the GOTO elimination algorithm. Sections that do remain empty, are removed by the 'Remove empty sections' transformation rule:

```
BAR SECTION.                    BAR SECTION.

BAR-PARAGRAPH.                  BAR PARAGRAPH.
  STOP RUN.                       STOP RUN.

EXAMP01-SUBROUTINES SECTION.  EXAMP02-SUBROUTINES SECTION.

EXAMP02-SUBROUTINES SECTION.  0002.
                                DISPLAY '2'.
0002.
  DISPLAY '2'.

EXAMP03-SUBROUTINES SECTION.
```

Nested IF to EVALUATE

In the preprocessing part of the GOTO elimination, we transformed all EVALUATE statements into nested IF statements using the 'Remove EVALUATE' rule. The reason we did this, was that otherwise the 'Distribute' and 'Cluster' transformation rules would possibly perform illegal transformations.

Now that the GOTO elimination is complete, it is safe to transform the nested IF statements back into EVALUATE statements as a postprocessing transformation rule. We use the 'Nested IF to EVALUATE' tool from section 3.2 for this purpose.

4.4 Translation to the new ASF+SDF Meta-Environment

To implement the above mentioned transformation rules in the new ASF+SDF Meta-Environment using the built-in traversal functions, we had to translate the 'old' implementations in the old ASF+SDF Meta-Environment using simulated traversal functions (see section 2.5).

The example in section 2.5 showed that translating simulated traversal functions to built-in traversal functions was not hard to do. In fact, the only things that had to be done were removing all generated equations and changing the syntax definition of the traversal functions.

In reality the translation was a little harder, due to the fact that the example of simulated traversal functions in section 2.5 was oversimplified. The real simulated traversal functions had a more complex syntax. Especially the simulated accumulator functions (analyzer functions) were harder to translate. However, the translation was still pretty straight-forward and did not require a lot of effort.

During the translation, some problems were encountered. The following sections sum up these problems and the solutions we used.

4.4.1 Lists

SDF does not allow a function to return a list of a sort. Since traversal functions are sort preserving, this means that it is not possible to use a traversal function to transform a list.

We've seen an example of this problem in chapter 3, where we wanted to apply a traversal function over the body of an `IF` statement, which is a list of statements (sort `Stat+`). We solved the problem by assigning the result of applying the traversal function to a variable (in a condition) and adding separator periods to both sides of the condition, so that the traversal function actually worked over sort `Sentence` (A list of statements followed by a separator period). For more detailed information on this solution, see chapter 3.

During the translation of the transformation rules of the GOTO elimination, it was necessary to apply traversal functions over all kinds of lists (`Stat+`, `Paragraph*`, `Section*`, etc.). Using the above mentioned trick for all these kinds of lists would make the translated implementation of the transformation rules extremely hard to understand. Therefore, we used a different solution.

To make it possible to call traversal functions over lists, we introduced new sorts in the Cobol grammar and replaced all occurrences of lists of sorts by the new sorts: `<sort>-s` instead of `<sort>*` and `<sort>-p` instead of `<sort>+`. The following ASF+SDF fragment shows an example of the changes we made to the grammar.

```
Stat* "." -> Sentence         Stat+      -> Stat-p
                              Stat*      -> Stat-s
                              Stat-p     -> Stat-s

                              Stat-s "." -> Sentence
```

4.4.2 Rewriting strategy

The algorithm of the GOTO elimination consists of applying the transformation rules in the right order. Since we created separate tools for all transformation rules, we have complete control over the order in which the rules are applied. However, this is not true for the 'ElimGO' transformation rule.

As discussed before, the 'ElimGO' tranformation rule is actually a collection of simple transformation rules, combined in one tool. The 'ElimGO' tool by Sellink, Sneed and Verhoef was designed in such a way that the transformation rules in it were applied in the right order. To get the same results in the translated tool in the new ASF+SDF Meta-Environment, we need to make sure that the equations in the ASF+SDF specification are applied in the same order. This turned out to be a problem, since the rewriting strategy of the simulated traversal functions in the 'old' tools was different than the rewriting strategy of the built-in traversal functions. This led to other equations being applied first, which led to a malfunctioning GOTO elimination algorithm.

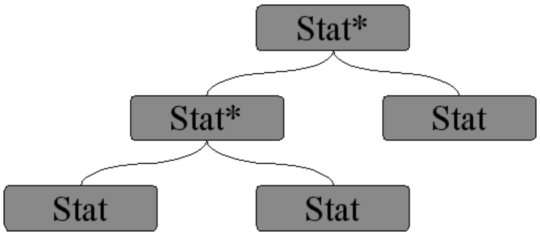


Figure 4.1: Simulated traversal of a list of statements.

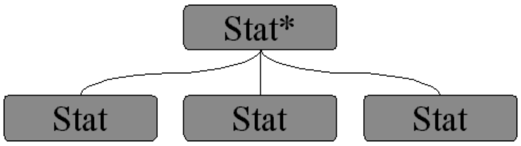


Figure 4.2: Built-in traversal of a list of statements.

The main problem was the way lists were traversed. In the simulated traversal functions, a list was traversed by splitting the list in two parts: the last element of the list and the rest of the list (figure 4.1).

The built-in traversal functions take a different approach. It breaks up a list into single elements; not into a smaller list and a last element (figure 4.2). This means that in order to call a traversal function over a list (using the newly introduced sorts from section 4.4.1), you have to specify a pattern for the complete list. Using the simulated traversal functions, you would only have to specify a pattern for the beginning of a list, up to the part you wish to transform.

This might not seem too big a difference. The following example shows how a simulated traversal function over lists might be translated to use a built-in traversal function and still have the same meaning:

```
[1] #Paragraph-s = #Paragraph*1 [1] #Paragraph-s = #Paragraph*1
                   #Lab.                            #Lab.
    ==========================                      #Paragraph*2
    transform(#Paragraph-s) =       ==========================
    transform(                      transform(#Paragraph-s) =
      #Paragraph*1                  transform(
    )                                 #Paragraph*1
                                      #Paragraph*2
                                    )
```

The translated version of the simulated traversal function in the example shows the same behaviour, as long as the list of paragraphs only needs to be transformed in one place. As soon as multiple transformations need to be made in one list of paragraphs, the order in which the transformations are made differs. This is caused by the fact that the built-in traversal functions follow a leftmost innermost rewriting strategy, while the simulated traversal functions simulate a rightmost innermost rewriting strategy.

Our final solution to this problem was not to change the translated equation, but instead add a default equation to 'override' the behaviour of the traversal equation:

```
[1] #Paragraph-s = #Paragraph*1
```

```
                  #Lab.
    ==========================
    transform(#Paragraph-s) =
    transform(
      #Paragraph*1
    )

[default-2] #Paragraph-s  = #Paragraph* #Paragraph,
            #Paragraph*2  = transform(#Paragraph*),
            #Paragraph2   = transform(#Paragraph),
            #Paragraph-s2 = #Paragraph*2 #Paragraph2
            ========================================
            transform(#Paragraph-s) = #Paragraph-s2
```

The default equation forces the traversal function to use a rightmost innermost rewriting strategy in the case of a list of paragraphs, just like it did in the case of simulated traversal functions. What we actually did is write a small part of the traversal of a Cobol program ourselves and letting the built-in traversal mechanism do the rest.

4.4.3 Default equations

In the case of simulated traversal functions, all default equations are 'used up'. They are used to perform the traversal of a Cobol program. Using the built-in traversal functions, it *is* possible to use default equations. However, using default equations for traversal functions can lead to counter-intuitive behaviour.

This problem was not encountered while translating the old transformation tools (since using simulated traversal functions, it wasn't possible to use default equations), but it was encountered while writing the extra pre- and postprocessing transformation tools.

The counter-intuitive behaviour is encountered when default equations are defined over higher sorts than normal equations. For example, we could write a tool that changes all `GO` statements into `CONTINUE` statements and eliminates sentences which don't contain any `GO` statements:

```
[1] #Stat = GO #Lab
    ==========================
    transform(#Stat) = CONTINUE

[default-2] transform(#Sentence) = .
```

Equation `[1]` changes `GO` statements into `CONTINUE` statements and if no normal equation can be applied, the default equation turns a sentence into an empty sentence (the remaining separator period can be removed by a separate tool).

This implementation does not work correctly, since the default equation is over a higher sort (`Sentence`) than the normal equation (`Stat`). Therefore, the traversal equation will always apply the default equation, which means that all sentences will be replaced by empty sentences. Even the sentences which *do* contain `GO` statements.

Though this counter-intuitive behaviour is not really a problem, it does make using default equations for traversal functions confusing sometimes.

Chapter 5

Making it all work together

Chapter 4 discussed the separate transformation rules that are necessary for the GOTO elimination. We also briefly discussed the implementation of the rules; we created a transformation tool for each transformation rule. The final step is using the separate tools in the right order.

The old implementation of the transformation rules (using simulated traversal functions, see section 2.5) was directed by the user. By means of a graphical user interface with a button for each rule, one could specify which transformation rule should be applied next. The resulting Cobol program was then shown, after which the user had to press another button to specify another transformation rule.

Because there was no algorithm for automatically applying the right transformation rules in order to eliminate GOTOs, the old GOTO elimination implementation has never been tested on a large set of Cobol programs. Another reason why the GOTO elimination was never tested on a large set of programs, was that the simulated traversal functions required so many resources that applying the GOTO elimination to a (large) program took many hours to complete.

Now that we have an implementation of the GOTO elimination using the built-in traversal functions, it becomes possible to apply the GOTO elimination to a large program in only minutes. This means that we can test the GOTO elimination on a large set of programs (which we will do in chapter 6).

This chapter discusses the algorithm we created that applies the transformation rules in the right order. We will first show what the algorithm should do and then show an implementation of this algorithm in the ToolBus, a system that can be used for tool coordination.

5.1 The Algorithm

In section 4.3 we already made a distinction between three different types of transformation rules: preprocessing, postprocessing and actual GOTO elimination rules.

This distinction leads to an algorithm with three parts. The first part is the preprocessing part. It consists of applying all preprocessing transformation rules to the Cobol program we want to eliminate GOTOs in. The preprocessing part of the algorithm is depicted in the left column of figure 5.1.

The postprocessing part (the last part) of the algorithm is equally simple. It consists of applying all postprocessing transformation rules to the Cobol program we want to

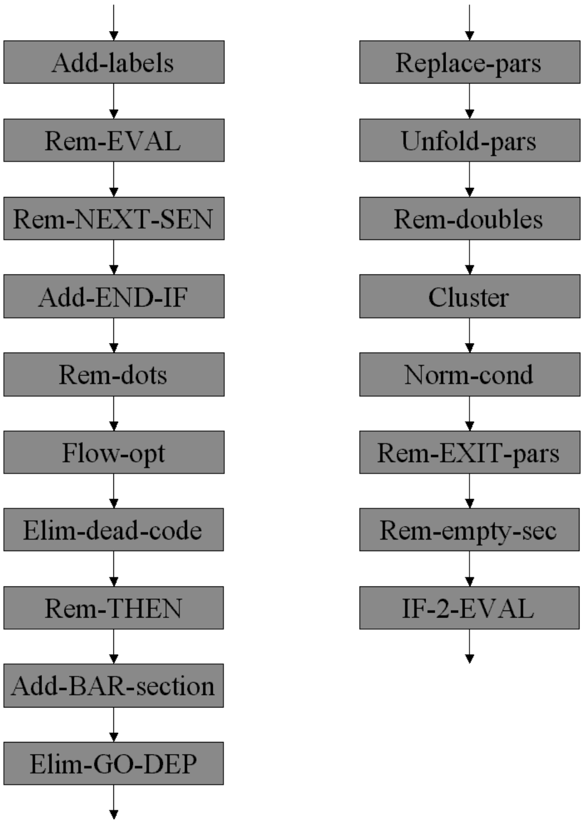


Figure 5.1: The preprocessing part (left) and the postprocessing part (right) of the GOTO elimination algorithm

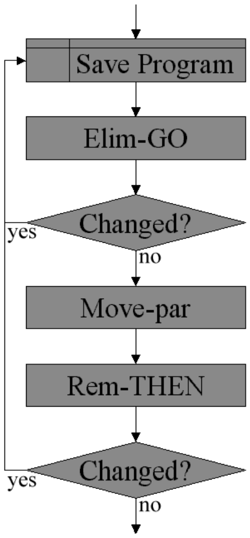


Figure 5.2: A first definition of the main loop of the GOTO elimination algorithm

eliminate GOTOs in. This part of the algorithm is shown in the right column of figure 5.1.

5.1.1 The main loop

The middle part of the algorithm is the part that performs the actual GOTO elimination. We will refer to this part as the main loop of the algorithm. As discussed in previous chapters, the main loop applies the 'ElimGO' rule to eliminate local (and other simple) GOTOs and the 'Move paragraphs' rule to create simulated function calls using the `PERFORM` statement. The movement of code in the 'Move paragraphs' rule can cause distant GOTOs to become local GOTOs, so after this rule has been applied, the 'ElimGO' rule should once more be applied.

This description of what the main loop should do, leads to the following definition of the main loop: Apply the 'ElimGO' rule until no more changes can be made. Then apply the 'Move paragraphs' rule. Repeat this until all GOTOs are removed. Figure 5.2 shows this definition of the main loop in a picture. We first save a copy of the Cobol program, so that we can compare this saved copy to the current version of the Cobol program. We use this comparison to determine whether any changes have been made by a certain transformation rule.

Note that in figure 5.2, we also use transformation rule 'Remove `THEN`' to remove the `THEN` statements that may have been introduced by the 'Move paragraphs' rule.

This definition of the main loop only works if the 'ElimGO' rule is eventually able to remove all local GOTOs. However, this will not be the case, since in order to keep the 'ElimGO' tool as simple as possible, we decided not to write equations for all possible patterns of local GOTOs. Instead, we relied on other transformation rules to restructure Cobol programs in such a way that patterns are created that *are* recognized by 'ElimGO'.

One of those transformation rules was 'Eliminate `CONTINUE`', a rule that eliminates `CONTINUE` statements which were introduced by the 'ElimGO' and 'Move para-

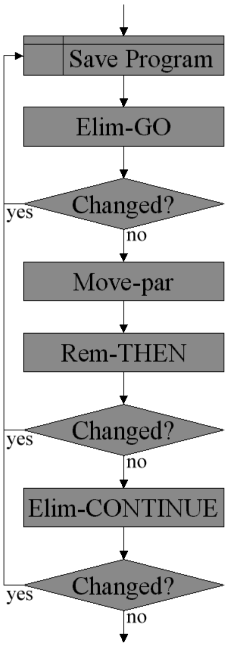


Figure 5.3: A second definition of the main loop of the GOTO elimination algorithm, this time with the 'Eliminate `CONTINUE`' transformation rule fitted in

graphs' rules. We fit the 'Eliminate `CONTINUE`' rule into the main loop as follows: if both the 'ElimGO' and 'Move paragraphs' rules cannot continue (applying them has no result), apply 'Eliminate `CONTINUE`' and check if 'ElimGO' can do something now. The main loop with the 'Eliminate `CONTINUE`' rule fitted in is depicted in figure 5.3.

Another transformation rule that should be fit into the main loop is the 'Eliminate labels' rule. Moving code around and eliminating `GO` statements, can lead to paragraph labels not being needed anymore. By eliminating these labels, distant GOTOs might become local GOTOs, which means 'ElimGO' might be able to eliminate them. The 'Eliminate labels' rule is fit into the main loop in a similar way as the 'Eliminate `CONTINUE`' rule: if the 'ElimGO', 'Move paragraphs' and 'Eliminate `CONTINUE`' rules cannot continue, apply 'Eliminate labels' and check if 'ElimGO' can do something now.

In the same way as the 'Eliminate `CONTINUE`' transformation rule, we also fit the 'Eliminate labels' and 'Distribute' transformation rules into the main loop. As was the case with 'Eliminate `CONTINUE`', 'Eliminate labels' 'Distribute' restructure certain aspects of a Cobol program in order to create patterns that might be recognized by 'ElimGO'.

5.1.2 Fitting in 'Switch paragraphs'

The 'Switch paragraphs' transformation rule moves code around so that distant GOTOs could become local GOTOs. However, if this transformation rule is applied more than once, it is possible that code is moved in circles. After several times of applying the 'Switch paragraphs' transformation rule, we get the same Cobol program we started with. Therefore, we need some mechanism to make sure our GOTO elimination algorithm doesn't infinitely loop.

One solution could be to exit the main loop as soon as all GOTOs have been eliminated. This would probably be the best solution, if we could be sure that the algorithm is indeed capable of eliminating all GOTOs. However, we can't be sure of that. As discussed before, our 'ElimGO' transformation tool will grow; we will add equations to it whenever we identify a pattern that we want to rewrite. Until the 'ElimGO' tool is complete (which it will never be, since there will probably always be a programmer who constructs something we've never seen before), we can't be sure that the GOTO elimination algorithm will eliminate all GOTOs.

So we need another solution to make sure the algorithm doesn't infinitely loop. Since the only tranformation rule which might run in circles is the 'Switch paragraphs' rule, we will limit the number of times this rule can be applied. The question that remains is: how many times should we allow the 'Switch pararaphs' transformation rule to be used?

By experimenting on source codes of which we knew it was possible to eliminate all GOTOs (source codes which have been transformed using the old tools with simulated traversal functions, see chapter 4.3), we came to the conclusion that 'Switch paragraphs' was only needed once, or not at all. Therefore we include a check in the main loop of the GOTO elimination algorithm: if the 'Switch paragraphs' rule has already been applied, don't apply it again.

Though this approach worked fine on the source codes we experimented on, it might not be good enough for all Cobol programs. Once the 'ElimGO' tool can recognize all necessary patterns to eliminate local and simple GOTOs in a certain source code and the GOTO elimination algorithm still isn't able to eliminate all GOTOs, it might be necessary to increase the number of times the 'Switch paragraphs' rule can be applied.

5.1.3 The final algorithm

With all the extensions we added to our algorithm, it looks like figure 5.4. The first step is preprocessing. After the preprocessing, we enter the main loop which starts with saving a copy of the program we're transforming. Then 'ElimGO' is applied, after which the program we're transforming is compared to the saved copy of the program. If the program has changed, the main loop starts over. If not, 'Move paragraphs' is applied, etc.

As we discussed,the picture shows that before 'Switch paragraphs' is applied, a check is made to see whether 'Switch paragraphs' has already been applied before. If so, 'Switch paragraphs' is skipped.

If none of the transformation rules in the main loop can do anything (after applying 'Distribute' the program is still the same as the saved copy), the algorithm performs the postprocessing step and exits.

5.2 The ToolBus script

The ToolBus ([4], [15]) is a software architecture intended for building cooperating, distributed applications. For example, the new ASF+SDF Meta-Environment was implemented using the ToolBus.

Using the ToolBus we can create a variable number of processes, which can communicate with each other and with external tools. The internal behaviour of these tools

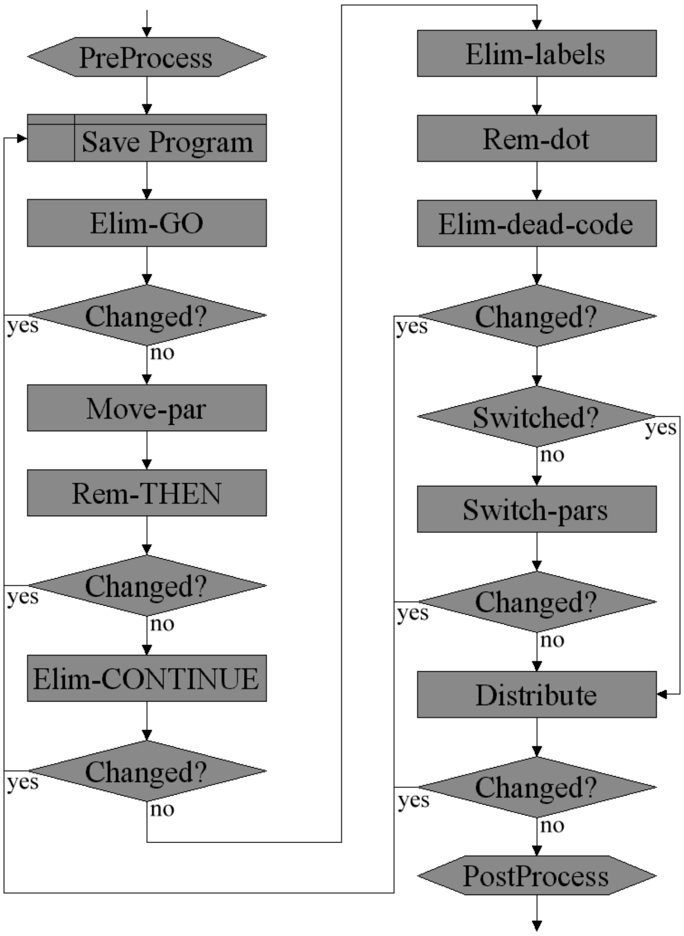


Figure 5.4: The GOTO elimination algorithm

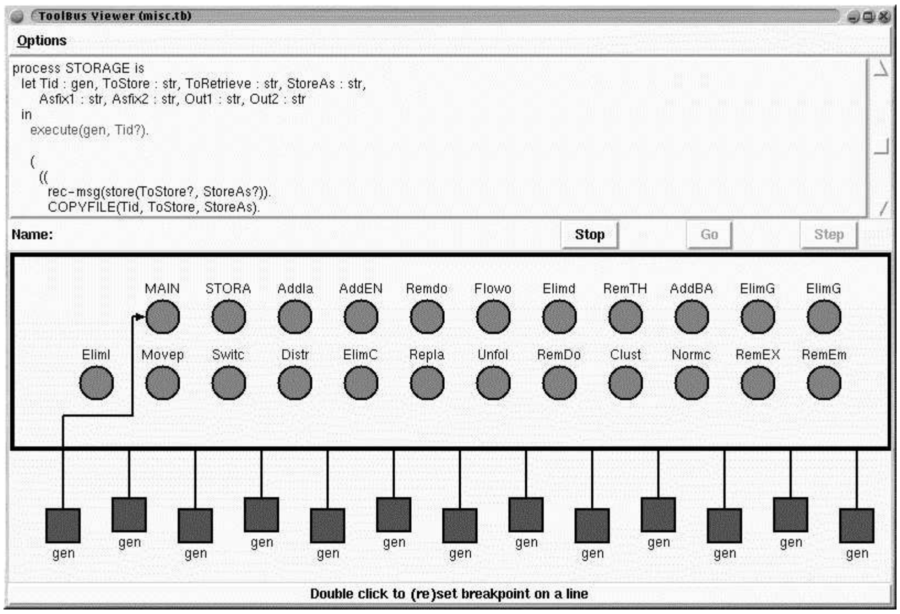


Figure 5.5: The ToolBus implementation of our GOTO elimination algorithm in action. The big rectangle shows the 'bus' containing the MAIN, STORAGE and transformation processes. Connected to the 'bus' are the transformation tools, shown as little squares. The figure shows the 'ToolBus Viewer', which can be used to monitor communications between processes and tools. Communications are shown using arrows.

is irrelevant; they may be implemented in different programming languages, be generated from specifications, etc. This means we can use the ToolBus to coordinate the separate transformation tools we created, which makes the ToolBus the ideal architecture to implement our GOTO elimination algorithm.

The implementation of the GOTO elimination algorithm consists of many processes:

- A MAIN process which is the actual implementation of the algorithm.
- A STORAGE process which is used to store, retrieve and compare Cobol programs.
- A separate process for each transformation tool.

The MAIN process

The MAIN process is a very simple process which opens the input Cobol file and sends it to STORAGE. Then the GOTO elimination algorithm is performed. Applying a transformation rule consists of sending a message to the process associated with that rule, telling the process to apply the rule to the Cobol program in STORAGE. The MAIN process then waits for a return message from the transformation process before it sends a message to the next transformation process. After the GOTO elimination algorithm has finished, the MAIN process retrieves the Cobol program from STORAGE and writes it to an output file. Finally, the MAIN process shuts down the ToolBus.

The STORAGE process

The STORAGE process can be used to store and retrieve two separate Cobol programs: the one we're transforming (the 'current Cobol program') and a saved copy which we need for the checks named 'Changed?' in the algorithm in figure 5.4. The STORAGE process can also be used to compare the two Cobol programs (the current and the saved programs).

The transformation processes

Each transformation process is associated with a different transformation rule. Since we have 24 different tranformation rules, we also have 24 transformation processes.

Each transformation process will wait for a message before doing anything. After a message has been received to tell the transformation process that it's transformation rule should be applied, the process retrieves the current Cobol program from STORAGE, sends it to the associated transformation tool, waits for the answer of the tool (the new Cobol program after the transformation rule has been applied) and sends the new program to STORAGE. The transformation rule will then start over and once again wait for a message before doing anything.

The following fragment of ToolBus Script shows an example of a transformation process, in this case the process for the 'Add labels' transformation rule:

```
tool gen is {command = "gen-adapter -addnewline"}

process Addlabels is
  let Tid : gen
  in
    execute(gen, Tid?).
    (
      rec-msg(addlabels).
      snd-msg(retrieve("ts.asfix", "current.asfix")).
      rec-msg(retrieved).
      EXECTOOL(Tid, "Add-labels", "add-labels").
      snd-msg(store("ts.asfix", "current.asfix")).
      rec-msg(stored).
      snd-msg(addlabels-done)
    )* delta
  endlet
```

We need to explain a few things about this process. In the version of the new ASF+SDF Meta-Environment that was available during the research for this masters thesis, it was not yet possible to compile an ASF+SDF specification which used traversal function. Therefore, we could not yet create stand-alone transformation tools and connect those tools to the ToolBus.

The only alternative was to perform transformations by calling `sglr` (to parse a Cobol program), `evaluator` (to perform the rewriting process) and `asource` (to convert a parse tree to readible text, in this case the new Cobol program) from command line. These three programs are parts of the ASF+SDF Meta-Environment and can be used to perform transformation without having to start the entire ASF+SDF Meta-Environment.

In order to be able to call `sglr`, `evaluator` and `asource`, we use a ToolBus tool called the `gen-adapter`. This is a tool that can be used to execute any Unix (or

Filename	Purpose
ts.trm	Input Cobol progam
ts.txt	Output Cobol program
ts.asfix	Cobol program we're working on (instead of sending the program in a message between processes)
current.asfix	Current program in STORAGE
saved.asfix	Saved copy of program in STORAGE

Table 5.1: Filenames used by the ToolBus implementation of the GOTO elimination algorithm

Linux) command as if it was a ToolBus tool itself. The output of the executed command is returned to the toolbus, so if we wanted to read a Cobol program into the ToolBus, we could execute the command '`cat program.cob`', where `program.cob` should contain the Cobol program.

However, the `gen-adapter` has some limitations on the size of the returned output. As it turned out, most Cobol programs are way too large for the `gen-adapter` to read the entire program into the ToolBus. Therefore, we used a different solution, namely working with files on disk. Table 5.1 shows which files we use for what purpose.

We can now explain what happens in the `Addlabels` process shown above. First, a message is send to STORAGE to retrieve the current Cobol program. What actually happens is that the current Cobol program is copied into `ts.asfix`. STORAGE then sends a message to the calling transformation process that the program has been retrieved, but the program itself is not sent in the message. Next a call to process `EXECTOOL` is made. This process simulates using a compiled ASF+SDF specification. It applies function `add-labels` from ASF+SDF module `Add-labels` on the Cobol program in `ts.asfix` using the `sglr`, `evaluator` and `asource` tools. The resulting Cobol program is once again placed in `ts.asfix`. The transformation process then 'sends the Cobol program back to STORAGE', by telling the STORAGE process to store file `ts.asfix` as the current Cobol program (`current.asfix`).

This form of implementation simulates the use of compiled ASF+SDF specifications. When it becomes possible to compile specifications with traversal functions, we would only have to make some changes to the STORAGE process and the transformation processes. The `Addlabels` process would then look something like:

```
tool addl is {command = "add-labels"}

process Addlabels is
  let Tid : addl, Program : str
  in
    execute(addl, Tid?).
    (
      rec-msg(addlabels).
      snd-msg(retrieve(current)).
      rec-msg(retrieved(Program?)).
      snd-eval("add-labels", Program).
      rec-value(result(Program?)).
      snd-msg(store(Program, current)).
```

```
      rec-msg(stored).
      snd-msg(addlabels-done)
    )* delta
  endlet
```

Instead of copying the file with the current Cobol program to a file called `ts.asfix`, the STORAGE process would send the Cobol program in a message to the transformation process. This process sends it to the compiled ASF+SDF specification with the command to apply a certain function to the program (instead of using `EXECTOOL` to call `sglr`, `evaluator` and `asource`). The compiled tool would return the transformed Cobol program to the transformation process, which sends it to STORAGE in a message (instead of telling the STORAGE process to copy the contents of `ts.asfix`).

5.3 A note on reparsing

Up to this point, we said that `ts.asfix`, `current.asfix` and `saved.asfix` contain Cobol programs. However, this is not entirely true. The files actually contain parse trees of Cobol programs (in asfix format, hence the file extension).

The reason we work with parse trees instead of Cobol programs in text format is that we are not interested in intermediate results of the GOTO elimination algorithm. Therefore, it isn't necessary to convert a parse tree to text and immediately parse that text again. We will simply store parse trees in STORAGE and only after the GOTO elimination is completed will we convert the parse tree to text.

We use the `apply-function` tool (which is part of the ASF+SDF Meta-Environment) to add a function symbol to a parse tree (for example: a parse tree with top sort `Program` can be changed by `apply-function` into a parse tree of the form `addlabels(<Program>)`, where `<Program>` is the old parse tree).

Reusing parse trees can lead to problems in certain situations. For more information, see section 6.2.1.

Chapter 6

Results and conclusions

In chapters 4 and 5 we created an implementation of the transformation rules of the GOTO elimination and the algorithm which applies these transformation rules in the right order. Using this implementation, we test the GOTO elimination on a large set of real-life Cobol programs from a large banking company.

This chapter will discuss the results of this test and present ideas for future enhancements of the GOTO elimination algorithm. Finally, we will present our conclusions on the use of traversal functions.

6.1 Testresults

6.1.1 The set of programs

We tested the GOTO elimination algorithm on a set of 85 Cobol programs. These programs are part of a project of ABN AMRO, a large Dutch banking company. 59 out of these 85 programs contain `GO` statements, so we only tested the GOTO elimination algorithm on these 59 programs.

Our test set of 59 programs consists of mostly small programs (an average of 846.5 lines of code per program, including comments and blank lines) and some larger programs (the largest program has 2771 lines of code). The set contains 2648 `GO` statements, which means an average of 44.9 GOTOs per program.

As discussed before, we did not expect the algorithm to be able to eliminate all GOTOs in the test set, since the 'ElimGO' transformation tool doesn't yet recognize all patterns of simulated constructs. Adding equations to the specification of the 'ElimGO' transformation tool would increase the number of eliminated GOTOs. However, what we would like to know is how many GOTOs can be eliminated by the algorithm as it is. Therefore, we don't make any changes to the 'ElimGO' tool.

6.1.2 Results

The results of applying the GOTO elimination algorithm to our test set of Cobol programs are shown in table 6.1. These results show that the GOTO elimination was able to eliminate an average of 63.7% of GOTOs per program.

By creating extra equations in the 'ElimGO' transformation tool, it should be possible to eliminate all local GOTOs (remember that local GOTOs are GOTOs to the

	loc	GOTOs b.	GOTOs a.	elim.	l. GOTOs	corr. elim.	par. b. BAR
	2702	269	115	57%	44	74%	58
	1982	171	95	44%	16	54%	44
	1000	61	24	61%	6	70%	17
	1312	54	24	56%	14	81%	25
	1021	77	32	58%	23	88%	35
	1714	84	38	55%	17	75%	38
	1328	65	34	48%	16	72%	22
	1031	69	35	49%	14	70%	23
	565	29	20	31%	12	72%	5
	743	108	60	44%	31	73%	31
	638	62	16	74%	11	92%	21
	1290	76	28	63%	9	75%	28
	1887	104	38	63%	30	92%	46
	1972	74	30	59%	14	78%	27
	406	12	7	42%	3	67%	4
	586	36	11	69%	3	78%	7
	409	11	7	36%	2	55%	1
	383	24	12	50%	4	67%	7
	248	7	3	57%	2	86%	1
	738	17	8	53%	6	88%	1
	145	2	0	100%	0	100%	1
	807	42	24	43%	9	64%	11
	727	24	7	71%	5	92%	10
	477	13	2	85%	1	92%	9
	1702	24	2	92%	2	100%	20
	576	12	2	83%	2	100%	4
	225	16	1	94%	1	100%	12
	291	11	3	73%	2	91%	1
	266	10	3	70%	2	90%	1
	753	30	6	80%	5	97%	16
	1194	43	18	58%	12	86%	16
	560	27	6	78%	4	93%	19
	881	37	10	73%	4	84%	20
	721	17	4	76%	4	100%	12
	2771	125	45	64%	19	79%	41
	630	58	17	71%	11	90%	14
	1008	64	30	53%	13	73%	21
	1949	124	43	65%	29	89%	50
	1545	56	28	50%	14	75%	16
	682	31	12	61%	4	74%	16
	498	15	2	87%	2	100%	10
	1176	65	42	35%	8	48%	11
	446	19	5	74%	1	79%	6
	431	26	8	69%	3	81%	12
	823	44	24	45%	14	77%	17
	341	18	10	44%	4	67%	5
	437	20	6	70%	4	90%	9
	516	24	9	63%	1	67%	10
	552	26	9	65%	1	69%	12
	1060	39	13	67%	11	95%	22
	342	3	1	67%	0	67%	0
	307	10	4	60%	2	80%	3
	395	34	7	79%	5	94%	13
	663	26	6	77%	4	92%	8
	827	63	26	59%	20	90%	33
	374	19	2	89%	2	100%	19
	333	9	3	67%	2	89%	8
	279	6	2	67%	2	100%	3
	279	6	2	67%	2	100%	3
Tot. / Av.	49944 / 846.5	2648 / 44.9	1081 / 18.3	- / 63.7%	508 / 8.6	- / 82.4%	955 / 16.2

Table 6.1: GOTO elimination statistics (loc = lines of code, GOTOs b. = number of GOTOs before transformation, GOTOs a. = number of GOTOs after transformation, elim. = percentage of GOTOs that was eliminated, l. GOTOs = number of remaining local GOTOs, corr. elim. = percentage of GOTOs that was eliminated if we could eliminate all local GOTOs, par. b. BAR = number of paragraphs behind BAR SECTION, Tot. = Total, Av. = Average)

current or to the next paragraph). To have an indication of the total percentage of GOTOs that would be eliminated if all local GOTOs could be eliminated, we counted the number of remaining local GOTOs in each test program. Assuming these GOTOs can be eliminated, we are already able to eliminate an average of 82.4% of GOTOs per program.

In reality, this percentage would be higher. Since eliminating local GOTOs makes it possible for the 'Move paragraphs' transformation rule to move pieces of code, new local GOTOs are created. These can then in turn be eliminated, etc. This would result in an even higher percentage of eliminated GOTOs.

We believe that in most cases it is possible to remove all GOTOs in a program simply by adding equations to the specification of the 'ElimGO' tool. In cases where this isn't possible, changing the number of times the 'Switch paragraphs' rule can be applied might be a solution.

6.1.3 Performance

Table 6.2 shows the execution times of the GOTO elimination algorithm per program, measured on a dedicated Intel Celeron 600 MHz processor, with 128 MB memory, running under Red Hat Linux 6.0. We see execution times ranging from 1 minute and 10 seconds (for the smallest program in our test set) to 35 minutes and 27 seconds (for one of the largest programs in the test set).

Note that the performance of the GOTO elimination algorithm can still be improved. See 'ideas for future work' in section 6.2.2 for some ideas on how to do this.

6.2 Conclusions

The research for this masters thesis had two purposes:

- Finding out how useful traversal functions are and what problems might be encountered when using them.
- Creating an implementation of the GOTO elimination algorithm which can be used to eliminate GOTOs in arbitrary Cobol programs.

We will now discuss our conclusions on these two subjects.

6.2.1 Traversal functions

We have already seen that traversal function can be very useful (see section 2.5); they save a lot of work and increase the performance of transformation tools. Though the traversal functions work the way they are supposed to, when using them one might encounter some problems. We will briefly discuss each of these problems.

Traversing lists

Since traversal functions are sort preserving and a function in ASF+SDF cannot have a list as output sort, traversal functions cannot be called over lists. We encountered this problem while creating the 'Add `END-IF`' and 'Nested `IF` to `EVALUATE`' tools in chapter 3. See section 3.1.3 for more information on this problem.

	loc	ex. time	loc/s
	2702	21:40	2.1
	1982	35:27	0.9
	1000	8:25	2.0
	1312	7:03	3.1
	1021	6:39	2.6
	1714	9:50	2.9
	1328	13:02	1.7
	1031	10:48	1.6
	565	5:55	1.6
	743	15:19	0.8
	638	4:04	2.6
	1290	11:50	1.8
	1887	12:35	2.5
	1972	11:39	2.8
	406	2:04	3.3
	586	10:43	0.9
	409	2:12	3.1
	383	7:56	0.8
	248	1:52	2.2
	738	2:36	4.7
	145	1:10	2.1
	807	4:38	2.9
	727	5:10	2.3
	477	4:07	1.9
	1702	9:31	3.0
	576	2:22	4.1
	225	1:28	2.6
	291	2:04	2.3
	266	1:52	2.4
	753	6:24	2.0
	1194	25:03	0.8
	560	4:54	1.9
	881	5:42	2.6
	721	12:10	1.0
	2771	18:38	2.5
	630	3:57	2.7
	1008	6:08	2.7
	1949	12:56	2.5
	1545	32:15	0.8
	682	10:18	1.1
	498	2:16	3.7
	1176	7:59	2.5
	446	6:13	1.2
	431	5:51	1.2
	823	6:40	2.1
	341	2:17	2.5
	437	3:01	2.4
	516	8:18	1.0
	552	7:15	1.3
	1060	33:46	0.5
	342	1:26	4.0
	307	1:37	3.2
	395	4:15	1.5
	663	3:39	3.0
	827	7:02	2.0
	374	2:49	2.2
	333	2:10	2.6
	279	2:14	2.1
	279	2:14	2.1
Tot. / Av.	49944 / 846.5	8:03:06 / 8:11	- / 2.2

Table 6.2: GOTO elimination performance statistics (loc = lines of code, ex. time = execution time in minutes, loc/s = lines of code per second)

Designing a grammar

In order to be able to parse programs written in a certain programming language, an ASF+SDF specification of the grammar of the programming language is required. While creating such a specification, one should introduce new sorts instead of using lists (`Stat-s` instead of `Stat*`, etc.) in order to avoid problems with traversing lists. We used such sorts during the development of the GOTO elimination tools. See section 4.4.1 for more information on this problem.

Default equations

The use of default equations to define the semantics of traversal functions can lead to counter-intuitive behaviour. This problem is described in more detail in section 4.4.3.

Priorities

One problem we have not discussed before is a problem on the level of parse trees. Certain transformations might give a correct result when converted to text, while the parse tree of the transformed program is not correct.

Suppose we have a slightly different definition of Cobol sentences than the definition we used in section 3.1. We now define sort `Sentence` to either be a list of statements followed by a closing period, or sort `StatsOptIfNotClosed` followed by a closing period. Since sort `StatsOptIfNotClosed` can consist of a list of statements and nothing else, this leads to ambiguities during parsing; it would not be clear whether a sentence consists of a list of statements, or of a `StatsOptIfNotClosed` (which also is a list of statements). These ambiguities can be solved by adding a priority rule to the specification of the Cobol grammar:

```
context-free syntax

  Stat-s "."                    -> Sentence
  StatsOptIfNotClosed "."       -> Sentence

  Stat-p                        -> StatsOptIfNotClosed
  Stat-s IfNotClosed            -> StatsOptIfNotClosed

context-free priorities

    Stat-s "."                  -> Sentence
  > StatsOptIfNotClosed "."     -> Sentence
```

The priority rule in this example means that in case an ambiguity is encountered while parsing a Cobol program, the choice should be made to use the rule `Stat-s "." -> Sentence` and not the rule `StatsOptIfNotClosed "." -> Sentence` to construct a `Sentence`.

Now suppose we apply the 'Add `END-IF`' transformation. This transformation has an equation for applying the used traversal function over sort `StatsOptIfNotClosed`. By adding an `END-IF` keyword to the `IfNotClosed` part of an item of sort `StatsOptIfNotClosed`, the `IfNotClosed` is tranformed into a `Stat` (statement), which means that the `StatsOptIfNotClosed` now consists of only statements (which fits the syntax rule `Stat-p -> StatsOptIfNotClosed`).

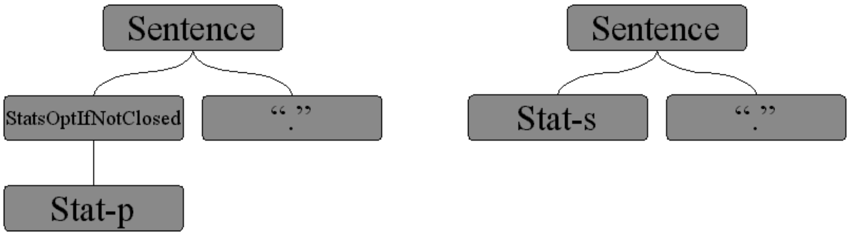


Figure 6.1: A parse tree that is forbidden by priority rules (left) and the correct alternative (right)

The result of this transformation is a `Sentence` which consists of a `StatsOptIfNotClosed` followed by a separator period. The `StatsOptIfNotClosed` consists of only statements. Figure 6.1 shows the resulting parse tree. Though this result looks correct when converted to text, the parse tree in incorrect; the priority rule of our example forbids this type of `Sentence` construction. The right picture in figure 6.1 shows what the resulting parse tree should look like.

If we decide to apply another transformation to the incorrect parse tree using the `apply-function` tool (see section 5.3), the transformation might fail. This problem could be solved by building explicit type casting into the transformation tool.

6.2.2 GOTO elimination

The GOTO elimination algorithm gave us an opportunity to test the GOTO elimination on a large set of programs. The results showed that as the algorithm is now, almost 2 out of 3 GOTOs can be eliminated.

Besides eliminating GOTOs, the GOTO algorithm also improves the structure of Cobol programs. By eliminating GOTOs, control flow inside a program is 'normalized': control flow starts at the first statement and progresses down through the program until it reaches the `BAR` section. When the `BAR` section is reached, the program terminates.

Control flow can jump to another point in a transformed program (with a `PERFORM` statement). However, the jump is always to a paragraph behind the `BAR` section. After this paragraph has been executed, control flow returns to the place of the jump.

The table which shows the results of the GOTO elimination test on a large set of programs (table 6.1) also shows how many paragraphs were moved to behind the `BAR` section. In other words, how many simulated function calls were recognized during the GOTO elimination.

The order of the paragraphs behind the bar section is not important. These paragraphs can be moved without consequences, as long as they stay behind the `BAR` section. It's also possible to insert new paragraphs behind the `BAR` section, which can then be called using `PERFORM` statements. This makes software maintenance a lot easier, since we always have a place where we can safely add new pieces of code.

We conclude that the GOTO elimination algorithm can help us to transform completely unstructured code into more structured code. However, the algorithm is not yet finished; it cannot yet be used to eliminate all GOTOs in an arbitrary Cobol program.

Ideas for future work

- We believe that it's possible to remove all GOTOs by adding equations to the specification of the 'ElimGO' tool and possibly changing the number of times the 'Switch paragraphs' transformation is applied. Once it becomes possible to eliminate *all* GOTOs in a program, the 'Switch paragraphs' transformation rule would no longer have to be limited in the number of times it is applied. Then the transformation rule would just be applied as long as GOTOs remain.
- The current implementation used by the GOTO elimination algorithm of the 'Add END-IF' and 'ElimGO' transformation rules suffer from the problem described under 'Priorities' in section 6.2.1. This makes it necessary to convert the parse tree of the Cobol program to text and reparse this text every time one of these transformation rules is applied. By solving this problem, the performance of the GOTO elimination would improve a great deal, since parsing a Cobol program takes a relatively large amount of time.
- When it becomes possible to compile ASF+SDF specifications which include traversal functions, the ToolBus implementation of the GOTO elimination algorithm can be changed (without too much effort) to use compiled tools instead of using the `gen-adapter`. This should further improve the performance of the GOTO elimination. Right now, eliminating GOTOs in one Cobol program takes between 2 (for small programs) and 30 minutes (for large programs)[1].
- Though we believe that all the transformation rules of the GOTO elimination perform only valid transformations (the resulting Cobol programs are semantically equal to the input programs), we did not prove it. Proving the soundness of the transformation rules is an important issue before the GOTO elimination can be used in real-life situations.

[1]The GOTO elimination algorithm was tested on a dedicated Intel Celeron 600 MHz processor, with 128 MB memory, running under Red Hat Linux 6.0.

Appendix A

Future issues

The creators of the ASF+SDF Meta-Environment are currently working on a number of improvements to the Meta-Environment. This appendix briefly mentiones some of these improvements and the effect they can have on the issues discussed in this masters thesis.

A.1 SDF2

SDF2 is the successor of SDF. One of the improvements SDF2 will bring is the possibility to have lists as output sort of a function. For more information on the problems that are caused by the inability to have lists as output sort of a function, see 'Traversing lists' in section 6.2.1.

A.2 Rewriting with full support for layout

Though we haven't discussed it in this masters thesis, the layout of the output of our ASF+SDF transformation tools was not correct.

The new ASF+SDF already supports rewriting with layout. The parts of a program that are not rewritten, keep their layout. The parts of a program that *are* rewritten, get the same layout as the equations of the ASF+SDF specification. As a consequence, the layout of the transformed parts of our Cobol program have incorrect indentation (among other things). Since source code comments are also defined as layout, transforming Cobol programs can also result in comments getting lost.

We solved the layout problem by using a pretty printer on our output Cobol programs. However, this means that also the parts of the programs that were not rewritten loose their original layout. To avoid this, we could use *markers* to specify which parts of the programs were transformed. We could then use a pretty printer which only changes the layout of code between the markers.

In future versions of the ASF+SDF Meta-Environment the control over the layout of rewritten terms will be improved. It will then be possible to 'reuse' the layout of the input term. In how far this will solve our layout problems remains to be seen.

A.3 Compilation of traversal functions

Currently it is not possible to compile ASF+SDF specifications which use traversal functions. In the future, this *will* be possible. Compiling such ASF+SDF specifications will probably greatly improve the performance of our transformation tools.

Appendix B

Specification of Pico example

This appendix contains the complete ASF+SDF specification of the example Pico transformation from section 2.5: changing `if` statements to `while` statements without using the built-in traversal functions of the new ASF+SDF Meta-Environment.

```
module WithoutTraversals

imports Pico-syntax

exports

  context-free syntax

    "if2while" "(" DECLS        ")" -> DECLS
    "if2while" "(" EXP          ")" -> EXP
    "if2while" "(" ID-TYPE      ")" -> ID-TYPE
    "if2while" "(" ID-TYPE-p    ")" -> ID-TYPE-p
    "if2while" "(" ID-TYPE-s    ")" -> ID-TYPE-s
    "if2while" "(" PICO-BOOL    ")" -> PICO-BOOL
    "if2while" "(" PICO-ID      ")" -> PICO-ID
    "if2while" "(" PICO-INT     ")" -> PICO-INT
    "if2while" "(" PICO-NAT     ")" -> PICO-NAT
    "if2while" "(" PICO-NAT-CON ")" -> PICO-NAT-CON
    "if2while" "(" PICO-STR-CON ")" -> PICO-STR-CON
    "if2while" "(" PICO-STRING  ")" -> PICO-STRING
    "if2while" "(" PROGRAM      ")" -> PROGRAM
    "if2while" "(" STATEMENT    ")" -> STATEMENT
    "if2while" "(" STATEMENT-p  ")" -> STATEMENT-p
    "if2while" "(" STATEMENT-s  ")" -> STATEMENT-s
    "if2while" "(" TYPE         ")" -> TYPE

  variables

    "#DECLS"                [0-9]* -> DECLS
    "#EXP"                  [0-9]* -> EXP
    "#ID-TYPE"              [0-9]* -> ID-TYPE
    "#ID-TYPE-p"            [0-9]* -> ID-TYPE-p
    "#ID-TYPE-s"            [0-9]* -> ID-TYPE-s
    "#PICO-BOOL"            [0-9]* -> PICO-BOOL
    "#PICO-ID"              [0-9]* -> PICO-ID
    "#PICO-INT"             [0-9]* -> PICO-INT
    "#PICO-NAT"             [0-9]* -> PICO-NAT
    "#PICO-NAT-CON"         [0-9]* -> PICO-NAT-CON
    "#PICO-STR-CON"         [0-9]* -> PICO-STR-CON
    "#PICO-STRING"          [0-9]* -> PICO-STRING
    "#PROGRAM"              [0-9]* -> PROGRAM
    "#STATEMENT"            [0-9]* -> STATEMENT
    "#STATEMENT-p"          [0-9]* -> STATEMENT-p
    "#STATEMENT-s"          [0-9]* -> STATEMENT-s
    "#TYPE"                 [0-9]* -> TYPE
```

```
equations

[default-01] #PROGRAM = begin
                          #DECLS
                          #STATEMENT-s
                        end
             ========================
             if2while(#PROGRAM) =
             begin
               if2while(#DECLS)
               if2while(#STATEMENT-s)
             end

[default-02] #STATEMENT-s =
             ========================
             if2while(#STATEMENT-s) =

[default-03] #STATEMENT-s  = #STATEMENT-p,
             #STATEMENT-p2 = if2while(#STATEMENT-p),
             #STATEMENT-s2 = #STATEMENT-p2
             ======================================
             if2while(#STATEMENT-s) = #STATEMENT-s2

[default-04] #STATEMENT-p  = #STATEMENT,
             #STATEMENT2   = if2while(#STATEMENT),
             #STATEMENT-p2 = #STATEMENT2
             =====================================
             if2while(#STATEMENT-p) = #STATEMENT-p2

[default-05] #STATEMENT-p = #STATEMENT;
                            #STATEMENT-p2
             ===========================
             if2while(#STATEMENT-p) =
             if2while(#STATEMENT);
             if2while(#STATEMENT-p2)

[default-06] #DECLS = declare #ID-TYPE-s ;
             =============================
             if2while(#DECLS) =
             declare if2while(#ID-TYPE-s) ;

[default-07] #ID-TYPE = #PICO-ID : #TYPE
             ===================================
             if2while(#ID-TYPE) =
             if2while(#PICO-ID) : if2while(#TYPE)

[default-08] #STATEMENT = #PICO-ID := #EXP
             ===================================
             if2while(#STATEMENT) =
             if2while(#PICO-ID) := if2while(#EXP)

[default-09] #STATEMENT = if #EXP then
                            #STATEMENT-s1
                          else
                            #STATEMENT-s2
                          fi
             ===========================
             if2while(#STATEMENT) =
             if if2while(#EXP) then
               if2while(#STATEMENT-s1)
             else
               if2while(#STATEMENT-s2)
             fi

[default-10] #STATEMENT = while #EXP do
                            #STATEMENT-s
                          od
             ==========================
             if2while(#STATEMENT) =
             while if2while(#EXP) do
               if2while(#STATEMENT-s)
             od

[default-11] #EXP  = #PICO-ID,
```

```
             #PICO-ID2 = if2while(#PICO-ID),
             #EXP2 = #PICO-ID2
             ===============================
             if2while(#EXP) = #EXP2

[default-12] #EXP  = #PICO-NAT-CON,
             #PICO-NAT-CON2 = if2while(#PICO-NAT-CON),
             #EXP2 = #PICO-NAT-CON2
             =========================================
             if2while(#EXP) = #EXP2

[default-13] #EXP  = #PICO-STR-CON,
             #PICO-STR-CON2 = if2while(#PICO-STR-CON),
             #EXP2 = #PICO-STR-CON2
             =========================================
             if2while(#EXP) = #EXP2

[default-14] #EXP = #EXP1 + #EXP2
             =================================
             if2while(#EXP) =
             if2while(#EXP1) + if2while(#EXP2)

[default-15] #EXP = #EXP1 - #EXP2
             =================================
             if2while(#EXP) =
             if2while(#EXP1) - if2while(#EXP2)

[default-16] #EXP = #EXP1 || #EXP2
             =================================
             if2while(#EXP) =
             if2while(#EXP1) || if2while(#EXP2)

[default-17] #EXP = ( #EXP2 )
             ===================
             if2while(#EXP) =
             ( if2while(#EXP2) )

[default-18] if2while(#PICO-STR-CON) = #PICO-STR-CON

[default-19] #PICO-STRING  = #PICO-STR-CON,
             #PICO-STR-CON2 = if2while(#PICO-STR-CON),
             #PICO-STRING2 = #PICO-STR-CON2
             =========================================
             if2while(#PICO-STRING) = #PICO-STRING2

[default-20] #PICO-STRING = #PICO-STRING1 || #PICO-STRING2
             ==================================================
             if2while(#PICO-STRING) =
             if2while(#PICO-STRING1) || if2while(#PICO-STRING2)

[default-21] #TYPE = natural
             =================
             if2while(#TYPE) =
             natural

[default-22] #TYPE = string
             =================
             if2while(#TYPE) =
             string

[default-23] #TYPE = nil-type
             =================
             if2while(#TYPE) =
             nil-type

[default-24] #PICO-BOOL = compatible (
                            #TYPE1
                          ,
                            #TYPE2
                          )
             =========================
             if2while(#PICO-BOOL) =
             compatible (
               if2while(#TYPE1)
```

```
             ,
               if2while(#TYPE2)
             )

[default-25] #PICO-BOOL = true
             ======================
             if2while(#PICO-BOOL) =
             true

[default-26] #PICO-BOOL = false
             ======================
             if2while(#PICO-BOOL) =
             false

[default-27] #PICO-BOOL = #PICO-BOOL1 | #PICO-BOOL2
             ============================================
             if2while(#PICO-BOOL) =
             if2while(#PICO-BOOL1) | if2while(#PICO-BOOL2)

[default-28] #PICO-BOOL = #PICO-BOOL1 & #PICO-BOOL2
             ============================================
             if2while(#PICO-BOOL) =
             if2while(#PICO-BOOL1) & if2while(#PICO-BOOL2)

[default-29] #PICO-BOOL = not ( #PICO-BOOL2 )
             ================================
             if2while(#PICO-BOOL) =
             not ( if2while(#PICO-BOOL2) )

[default-30] #PICO-BOOL = ( #PICO-BOOL2 )
             ================================
             if2while(#PICO-BOOL) =
             ( if2while(#PICO-BOOL2) )

[default-31] if2while(#PICO-ID) = #PICO-ID

[default-32] if2while(#PICO-NAT-CON) = #PICO-NAT-CON

[default-33] #PICO-NAT  = #PICO-NAT-CON,
             #PICO-NAT-CON2 = if2while(#PICO-NAT-CON),
             #PICO-NAT2 = #PICO-NAT-CON2
             =========================================
             if2while(#PICO-NAT) = #PICO-NAT2

[default-34] #PICO-NAT = #PICO-NAT1 -/ #PICO-NAT2
             ===========================================
             if2while(#PICO-NAT) =
             if2while(#PICO-NAT1) -/ if2while(#PICO-NAT2)

[default-35] #PICO-NAT = ( #PICO-NAT2 )
             ==========================
             if2while(#PICO-NAT) =
             ( if2while(#PICO-NAT2) )

[default-36] #PICO-INT  = #PICO-NAT,
             #PICO-NAT2 = if2while(#PICO-NAT),
             #PICO-INT2 = #PICO-NAT2
             ================================
             if2while(#PICO-INT) = #PICO-INT2

[default-37] #PICO-INT = + #PICO-NAT
             =======================
             if2while(#PICO-INT) =
             + if2while(#PICO-NAT)

[default-38] #PICO-INT = - #PICO-NAT
             =======================
             if2while(#PICO-INT) =
             - if2while(#PICO-NAT)

[default-39] #PICO-INT = #PICO-INT1 + #PICO-INT2
             =========================================
             if2while(#PICO-INT) =
             if2while(#PICO-INT1) + if2while(#PICO-INT2)
```

```
[default-40] #PICO-INT = #PICO-INT1 - #PICO-INT2
             ===========================================
             if2while(#PICO-INT) =
             if2while(#PICO-INT1) - if2while(#PICO-INT2)

[default-41] #PICO-INT = #PICO-INT1 * #PICO-INT2
             ===========================================
             if2while(#PICO-INT) =
             if2while(#PICO-INT1) + if2while(#PICO-INT2)

[default-42] #PICO-BOOL = #PICO-INT1 > #PICO-INT2
             ===========================================
             if2while(#PICO-BOOL) =
             if2while(#PICO-INT1) > if2while(#PICO-INT2)

[default-43] #PICO-BOOL = #PICO-INT1 >= #PICO-INT2
             ============================================
             if2while(#PICO-BOOL) =
             if2while(#PICO-INT1) >= if2while(#PICO-INT2)

[default-44] #PICO-BOOL = #PICO-INT1 < #PICO-INT2
             ===========================================
             if2while(#PICO-BOOL) =
             if2while(#PICO-INT1) < if2while(#PICO-INT2)

[default-45] #PICO-BOOL = #PICO-INT1 <= #PICO-INT2
             ===========================================
             if2while(#PICO-BOOL) =
             if2while(#PICO-INT1) <= if2while(#PICO-INT2)

[default-46] #PICO-INT = ( #PICO-INT2 )
             ===========================================
             if2while(#PICO-INT) =
             ( if2while(#PICO-INT2) )

[default-47] #ID-TYPE-s =
             =========================
             if2while(#ID-TYPE-s) =

[default-48] #ID-TYPE-s  = #ID-TYPE-p,
             #ID-TYPE-p2 = if2while(#ID-TYPE-p),
             #ID-TYPE-s2 = #ID-TYPE-p2
             ===================================
             if2while(#ID-TYPE-s) = #ID-TYPE-s2

[default-49] #ID-TYPE-p  = #ID-TYPE,
             #ID-TYPE2 = if2while(#ID-TYPE),
             #ID-TYPE-p2 = #ID-TYPE2
             ==================================
             if2while(#ID-TYPE-p) = #ID-TYPE-p2

[default-50] #ID-TYPE-p = #ID-TYPE,
                          #ID-TYPE-p2
             ============================
             if2while(#ID-TYPE-p) =
             if2while(#ID-TYPE),
             if2while(#ID-TYPE-p2)



[1] #STATEMENT = if #EXP then
                   #STATEMENT-s1
                 else
                   #STATEMENT-s2
                 fi
    ============================
    if2while(#STATEMENT) =
    while #EXP do
      if2while(#STATEMENT-s1)
    od
```

Bibliography

[1] Z. Ammarguellat. *A control-flow normalization algorithm and its complexity*. IEEE Transactions on Software Engineering, 18(3):237–251, March 1992.

[2] E.Ashcroft and Z. Manna, *The translation of "goto" programs to "while" programs*. In Proceedings of IFIP Congres 71, Amsterdam, 1972.

[3] C. Babcock, *Restructuring eases maintenance*. Computerworld, page 19 and 22, November 1987.

[4] J.A. Bergstra and P. Klint, *The Discrete Time ToolBus – a software coordination architecture*. Science of Computer programming, 31(2-3):205-229, 1998.

[5] B.W. Boehm, *Software Engineering Economics*. Prentice Hall, 1981.

[6] Mark van den Brand et al., *The* ASF+SDF *Meta-Environment*. Available at http://www.cwi.nl/projects/MetaEnv/.

[7] M.G.J. van den Brand and P. Klint, ASF+SDF *Meta-Environment User Manual*. Available at http://www.cwi.nl/projects/MetaEnv/.

[8] M.G.J. van den Brand, P. Klint and J. Vinju, *Term Rewriting with Traversal Functions*. CWI report, to appear, 2001.

[9] Mark van den Brand, Alex Sellink and Chris Verhoef, *Control Flow Normalization for COBOL/CICS Legacy Systems*. Available at http://www.cs.vu.nl/~x/cfn/cfn.html.

[10] Mark van den Brand, Alex Sellink and Chris Verhoef, *Generation of components for software renovation factories from context-free grammars*. In I.D. Baxter, A. Quilicy and C. Verhoef, editors, *proceedings of the fourth working conference on reverse engineering*, pages 144-153, 1997. Available at http://www.cs.vu.nl/~x/trans/trans.html.

[11] E.W. Dijkstra, *Go to statement considered harmful*. Communications of the ACM 11, 147-148. 1968.

[12] Robert E. Filman, Ronals A. Gray and Paul Morris, *GOTO Removal Based On Regular Expressions*.

[13] IBM, *COBOL Language Reference (for OS/390 & VM, IBM COBOL Set for AIX, and VisualAge for COBOL)*. See http://www.s390.ibm.com/bookmgr-cgi /bookmgr.cmd/BOOKS/IGYLR201/CCONTENTS.

[14] Merijn de Jonge, Eelco Visser, Joost Visser, editors, *The Grammar Base*. See http://www.program-transformation.org/gb/index.html.

[15] P. Klint, *A Guide To ToolBus Programming*. Available in the 'Completa' release of [6].

[16] Alex Sellink, Harry Sneed, Chris Verhoef, *Restructuring of COBOL/CICS Legacy Systems*, an extended abstract has been published in the third Conference on Maintenance and Reengineering (CSMR'99), March 1999, Amsterdam. Available at http://www.cs.vu.nl/~x/res/res.html, see also http://www.cs.vu.nl/~x/systolic/systolic.html for a detailed presentation of the GOTO-elimination algorithm.

[17] Alex Sellink and Chris Verhoef, *An evolutionary Cobol grammar*, which can be found in The Grammar Base ([14]).

[18] *VS COBOL II grammar Version 1.0.3*, 1999. Available at http://www.cs.vu.nl/~x/grammars/vs-cobol-ii.