

# A Software Restructuring Tool for Oberon

A THESIS  
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE  
OF THE UNIVERSITY OF STELLENBOSCH  
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS  
FOR THE DEGREE OF  
MASTER OF COMMERCE



By  
Johannes J. Eloff  
December, 2001

Supervised by: Professor P.J.A. de Villiers

# Declaration

I the undersigned hereby declare that the work contained in this thesis is my own original work and has not previously in its entirety or in part been submitted at any university for a degree.

# Abstract

Software *restructuring* is a form of perfective maintenance that modifies the structure of a program's source code. Its goal is increased maintainability to better facilitate other maintenance activities, such as adding new functionality or correcting previously undetected errors.

The modification of structure is achieved by applying transformations to the source code of a software system. Software engineers often attempt to restructure software by manually transforming the source code. This approach may lead to undesirable and undetectable changes in its behaviour. Ensuring that manual transformations preserve functionality during restructuring is difficult; guaranteeing it is almost impossible.

One solution to the problem of manual restructuring is automation through use of a *restructuring tool*. The tool becomes responsible to examine each transformation and determine its impact on the software's behaviour. If a transformation preserves functionality, it may be applied to produce new source code. The tool only automates the application of transformations. The decision regarding which transformation to apply in a specific situation still resides with the maintainer.

This thesis describes the design and implementation of a restructuring tool for the *Oberon* language, a successor of *Pascal* and *Modula-2*, under the *PC Native Oberon* operating system. The process of creating an adequate abstraction of a program's structure and its use to apply transformations and generate new source code are investigated. Transformations can be divided into different classes: *Scoping*, *Syntactic*, *Control flow* and *Abstraction* transformations. The restructuring tool described in this thesis contains implementations from all four classes. Informal arguments regarding the correctness of each transformation are also presented.

# Opsomming

Die *herstrukturering* van programmatuur is daarop gemik om die struktuur van 'n program se bronkode te wysig. Hierdie strukturele veranderings dien in die algemeen as voorbereiding vir meer omvangryke onderhoudsaktiwiteite, soos byvoorbeeld die toevoeging van nuwe funksionaliteit of die korrigering van foute wat voorheen verskuil was.

Die verandering in struktuur word teweeggebring deur die toepassing van transformasies op die bronkode. Programmatuur-ontwikkelaars voer dikwels sulke transformasies met die hand uit. Sulke optrede kan problematies wees indien 'n transformasie die funksionaliteit, in terme van programgedrag, van die programmatuur beïnvloed. Dit is moeilik om te verseker dat bogenoemde metode funksionaliteit sal behou; om dit te waarborg is so te sê onmoontlik.

'n Oplossing vir bogenoemde probleem is die outomatisering van die herstruktureringsproses deur die gebruik van gespesialiseerde programmatuur. Hierdie programmatuur is in staat om die nodige transformasies toe te pas en terselfdertyd funksionaliteit te waarborg. Die keuse vir die toepassing van 'n spesifieke transformasie lê egter steeds by die programmeerder.

Hierdie tesis bespreek die ontwerp en implementering van programmatuur om bronkode, geskryf in *Oberon* (die opvolger van *Pascal* en *Modula-2*), te herstrukturer. Die skep van 'n voldoende abstrakte voorstelling van bronkode, die gebruik van sodanige voorstelling in die toepassing van transformasies en die reprodusering van nuwe bronkode, word bespreek. Transformasies kan in vier breë klasse verdeel word: *Bestek*, *Sintaks*, *Kontrolevloei* en *Abstraksie*. Die programmatuur wat ontwikkel is vir hierdie tesis bevat voorbeelde uit elkeen van die voorafgenoemde klasse. Informele argumente word aangebied om die korrektheid van die onderskeie transformasies te staaf.

# Acknowledgements

I gladly acknowledge the help of several people who made this project feasible:

- Pieter de Villiers, for introducing me to the world of software engineering and all its challenges.
- My parents, for giving me the opportunity to study and always showing an active interest in my work.
- Christine du Toit, my fiancé, for her support and encouragement throughout the project.
- Pieter Muller and Patrik Reali (both from ETH Zürich), for providing valuable technical information regarding *Oberon* and the *OP2* compiler.

# Contents

<b>Abstract</b>	<b>iii</b>
<b>Opsomming</b>	<b>iv</b>
<b>Acknowledgements</b>	<b>v</b>
<b>1 Introduction</b>	<b>1</b>
1.1 The subject of this thesis . . . . .	3
1.2 The outline of this thesis . . . . .	4
<b>2 Program Representation and Transformations</b>	<b>5</b>
2.1 Transformation and Restructuring . . . . .	5
2.1.1 Manual Restructuring . . . . .	6
2.1.2 Automated Restructuring . . . . .	7
2.2 Program Representation Structures . . . . .	8
2.2.1 The Abstract Syntax Tree . . . . .	8
2.2.2 The Control Flow Graph . . . . .	8
2.2.3 The Program Dependence Graph . . . . .	9
2.2.4 The Unified Inter-procedural Graph . . . . .	9
2.3 Summary . . . . .	10

<b>3</b>	<b>The Context Entity Graph</b>	<b>11</b>
3.1	Definitions . . . . .	11
3.2	Constructing the CEG . . . . .	12
3.2.1	Language Restrictions . . . . .	13
3.2.2	The Module Context . . . . .	14
3.2.3	The Import Context . . . . .	15
3.2.4	Constant Entities . . . . .	16
3.2.5	Type Entities . . . . .	20
3.2.6	Variable Entities . . . . .	24
3.2.7	The Procedure Context . . . . .	24
3.2.8	Statements . . . . .	26
3.2.9	The Predefined Context . . . . .	29
3.3	Summary . . . . .	31
<b>4</b>	<b>Transformations on the CEG</b>	<b>32</b>
4.1	Preliminary Definitions . . . . .	32
4.1.1	The Context Path . . . . .	33
4.1.2	Dependencies . . . . .	35
4.1.3	Searching . . . . .	35
4.2	Transformation Definitions . . . . .	37
4.2.1	The Rename Transformation . . . . .	38
4.2.2	The Remove Transformation . . . . .	40
4.2.3	The Move Transformation . . . . .	42
4.2.4	The CASE-IF Transformation . . . . .	46
4.2.5	The IF-CASE Transformation . . . . .	46

4.2.6	The Exchange Statement Transformation . . . . .	51
4.2.7	The Create Procedure Transformation . . . . .	53
4.3	Transformation Examples . . . . .	55
4.3.1	Example: Syntactic Transformation . . . . .	55
4.3.2	Example: Control Flow Transformation . . . . .	58
4.3.3	Example: Abstraction Transformation . . . . .	58
4.4	Transformation Dependencies . . . . .	63
4.5	Updating Dependency Information within the CEG . . . . .	63
4.6	Summary . . . . .	64
<b>5</b>	<b>Evaluation and Conclusion</b>	<b>65</b>
5.1	Implementation and Results . . . . .	65
5.2	User Interface . . . . .	68
5.3	Evaluation of Transformations . . . . .	69
5.3.1	Syntactic Transformations . . . . .	69
5.3.2	Control Flow Transformations . . . . .	69
5.3.3	Abstraction Transformations . . . . .	69
5.4	Evaluation of the CEG . . . . .	71
5.5	Future Work . . . . .	72
5.6	Conclusion . . . . .	73
<b>A</b>	<b>Oberon Language Definition</b>	<b>74</b>
A.1	Module . . . . .	74
A.2	Declarations . . . . .	74
A.3	Statements . . . . .	75
A.4	Expressions and Operators . . . . .	76

A.5 Identifiers, Numbers and Strings . . . . .	76
<b>B Selected Algorithms</b>	<b>77</b>
B.1 The Move Transformation . . . . .	77
B.2 The Exchange Statement Transformation . . . . .	80
B.3 The Create Procedure Transformation . . . . .	80

# List of Figures

1	CEG structure representing a module context . . . . .	15
2	<i>Oberon</i> implementation for parsing a module . . . . .	16
3	CEG structure representing an import context . . . . .	17
4	CEG structure representing a constant entity . . . . .	19
5	<i>Oberon</i> implementation for parsing declarations . . . . .	21
6	CEG structure representing a simple type declaration . . . . .	22
7	CEG structure representing a complex type declaration . . . . .	23
8	<i>Oberon</i> implementation for parsing types . . . . .	24
9	CEG structure representing a variable declaration . . . . .	25
10	CEG structure representing a procedure context . . . . .	26
11	CEG structure representing a statement entity. . . . .	27
12	CEG structure representing a statement context. . . . .	28
13	<i>Oberon</i> implementation for parsing a statement sequence. . . . .	29
14	<i>Oberon</i> implementation for parsing specific statements. . . . .	30
15	Example of a complete <i>Oberon</i> module and its CEG. . . . .	34
16	Algorithm used to construct the context path of a node . . . . .	35
17	The algorithm for function <i>Node</i> . . . . .	36
18	The algorithm for function <i>LocalNode</i> . . . . .	36

19	The algorithm for function <i>GlobalNode</i> . . . . .	37
20	<i>Oberon</i> example to illustrate rename transformation . . . . .	38
21	Algorithm for the $\delta_{rename}$ transformation. . . . .	40
22	<i>Oberon</i> implementation for the $\delta_{rename}$ transformation . . . . .	41
23	Algorithm for the $\delta_{remove}$ transformation. . . . .	42
24	<i>Oberon</i> example to illustrate move transformation . . . . .	43
25	Algorithm for the $\delta_{move}$ transformation. . . . .	45
26	Example of a CASE and IF statement . . . . .	47
27	Implementation to evaluate expression structure . . . . .	48
28	Implementation to decompose expressions into sub-expressions . . . . .	49
29	Creation of decomposition nodes . . . . .	49
30	Locating common sub-expressions . . . . .	50
31	The relationship between the CEG and a dominator tree . . . . .	52
32	Algorithm for the $\delta_{exchange}$ transformation. . . . .	54
33	Algorithm for the $\delta_{createprocedure}$ transformation. . . . .	56
34	Original module that awaits restructuring. . . . .	57
35	<i>Oberon</i> module after removing a constant declaration. . . . .	57
36	Original module before control flow restructuring. . . . .	59
37	Restructuring by abstracting code into subroutines. . . . .	61
38	<i>Oberon</i> module after applying an abstraction transformation. . . . .	62
39	Implementation structure of NORT. . . . .	67
40	Hypertext environment provided by NORT . . . . .	70
41	CheckLocal algorithm . . . . .	77
42	CheckGlobal algorithm . . . . .	78

43	CheckCurrent algorithm . . . . .	78
44	CheckDependenciesMove algorithm . . . . .	78
45	CheckStatements algorithm . . . . .	79
46	CheckAll algorithm . . . . .	79
47	SingleEntryExit algorithm . . . . .	81
48	CheckDependenciesProc algorithm . . . . .	82

# List of Tables

1	The predefined context . . . . .	31
2	Breakdown of implementation into modules. . . . .	66
3	Breakdown showing memory requirements of the CEG for various modules. . . . .	66
4	Breakdown of nodes into the four major node classes for various modules. . . . .	67

# Chapter 1

## Introduction

Implementing software correctly and efficiently is difficult. The evolutionary nature of software, combined with the imperfections that occur during implementation are manifested by the need for software maintenance. Gallagher defines software maintenance as

“... the process of designing and integrating consistent changes to an existing software system.”

and states that the impact these changes have on a system is not always clear to the maintainer [18]. Maintenance is an expensive activity. Studies have confirmed that in some cases it may consume up to 80% of an organisation’s budget, limiting resources to develop new software [8]. Finding more efficient ways to maintain software is important and requires knowledge of the various activities and their underlying roles within the development cycle. The maintenance process can be broken down into the following three activities:

- *Corrective maintenance* is defined as the correction of errors in operational systems. These errors can usually be attributed to poorly designed and inadequate test cases that failed to identify them during the development and testing phase of the software [8, 39].
- *Perfective maintenance* includes those activities that aim to increase the efficiency of a system or make the system more maintainable for the programmers once it is operational [8, 39].

- *Adaptive maintenance* occurs when changes in a software system are prompted by changes in its operating environment. Functionality that was not anticipated during the initial design is usually incorporated into the system during this activity [8, 39].

The application of more stringent testing procedures will reduce errors and should alleviate the pressure to perform corrective maintenance. Little can be done to reduce adaptive activities since this part of the maintenance equation is often driven by end-user needs, industry trends and competition. Perfective maintenance impacts directly on both corrective and adaptive activities. It is generally accepted that software degrades over time. Keeping a system structured will not only reduce the time required to apply corrective measurements when the need arises, but will also simplify the process of integrating new features into a system.

Restructuring is a form of perfective maintenance and strives to increase the maintainability of a system. Arnold defines software restructuring as the modification of software to make it easier to understand and less susceptible to errors [2]. The modification is achieved by applying transformations to the source code to alter its structure. Change in structure often implies change in behaviour. This problem can be addressed by applying meaning preserving transformations to a program. Meaning preserving transformations ensure that the transformed system is functionally equivalent to its predecessor, given the same set of inputs.

Manual restructuring is error prone since the maintainer has no way to guarantee that the changes he<sup>1</sup> introduced will preserve the system's behaviour. The responsibility to guarantee functional equivalence can be shifted from the maintainer when using a restructuring tool. The tool becomes responsible for determining whether or not a specific transformation invalidates the condition of functional equivalence. It is important to note that a restructuring tool only automates the application of transformations; the decision to apply a specific transformation ultimately resides with the maintainer.

Several restructuring tools for a variety of programming languages exist today. Amongst these are Griswold's tool for *Scheme* [19], **CStructure** for the *C* language [30], **Elbereth** for *Java* [24], **SPRUCE** for *Pascal* [26] and the **Refactoring Browser** for *SmallTalk* [37]. A diverse collection of issues surrounding restructuring are addressed by the various tools. Tools such as **Elbereth** attempt to aid the maintainer in designing and planning maintenance activities by visualising a system's structure using the Star-Diagram.

---

<sup>1</sup>“He” should be read as “he or she” throughout this thesis.

The **CStructure** tool, on the other hand, implements specialised transformations including statement re-ordering based on virtual control flow analysis. Approaches vary between tools, but the goal is consistent: Developing efficient techniques for software maintenance.

Empirical results indicate that restructuring tools impact positively on software maintenance. Griswold's experiments indicate that tool-aided restructuring reduces both errors and maintenance cost [19]. Markosian et al. showed that the time required to re-engineer a large software module could be reduced from 20 person-weeks to only 4 person-hours by using a restructuring tool [27]. Case studies based on the **Design Maintenance System** (DMS) show that clone detection, a method by which duplicate code is identified, can locate and remove up to 10% redundancy in systems [4].

## 1.1 The subject of this thesis

The goal of this thesis is the implementation of a prototype tool for restructuring software modules written in *Oberon*<sup>2</sup> called *Native Oberon Restructuring Tool* (NORT). Although the prototype was designed specifically for the *PC Native Oberon* system, it will be possible to port it to other *Oberon* platforms. The choice of *Oberon* is motivated by two factors. First, there is currently no restructuring tool available for *Oberon*. Second, *Oberon* is a general purpose programming language and as such poses similar problems that restructuring tools for other languages are facing.

A number of secondary goals will also be addressed in this thesis and include the following:

1. The use of a single program representation structure that can facilitate both transformations and the generation of new source code.
2. Describing the various transformations implemented in NORT and comparing them with similar transformations found in other tools.
3. Proposing, describing and motivating new transformations as possible future research.

---

<sup>2</sup>Oberon is the name of both an operating system and programming language.

## 1.2 The outline of this thesis

An overview of implementation methodologies for transformations is presented in Chapter 2. The relationship between program transformations and program representation is also examined. An outline of the advantages and disadvantages of the respective structures is presented.

The representation structure used in the implementation of NORT is examined in Chapter 3. This structure is introduced through various short examples, allowing the reader to familiarise himself with the representation and its relationship with constructs in the *Oberon* language.

The transformations supported by NORT will be examined in Chapter 4. A notation to describe transformations is also introduced. Arguments will be presented to illustrate that the transformations enforce functional equivalence.

A discussion on the implementation of NORT is presented in Chapter 5. An evaluation of the tool and suggested future work are also presented.

## Chapter 2

# Program Representation and Transformations

Successfully applying transformations to a program requires a robust method to manipulate the structure of the program interactively. This method should also facilitate the generation of source code because restructuring operations are essentially source-to-source translations. This chapter does not present a comprehensive survey of software maintenance and restructuring. A more focused approach is taken and only work closely related to the issues surrounding restructuring tools are discussed.

The concepts of *transformation* and *restructuring* are described in Section 2.1, followed by a discussion of different implementation methodologies and the use of program representation structures as a means to facilitate transformations. Various structures are examined in more detail in Section 2.2 and include the *abstract syntax tree* (AST), *control flow graph* (CFG), *program dependence graph* (PDG) and the *unified inter-procedural graph* (UIG).

### 2.1 Transformation and Restructuring

A transformation  $T$  is a function that receives a source program  $P$  as input and produces a new source program  $P'$ .  $P'$  is said to be functionally equivalent to  $P$  (written as  $P' \equiv P$ ) iff  $P'$  exhibits identical behaviour to  $P$  for all defined inputs of  $P$ .  $T$  is said to be a meaning preserving transformation if  $P \equiv P'$  [20]. Transformations in object-oriented systems are often referred to as refactorings [17, 34, 36]. Fowler defines

a refactoring as a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour [17]. This definition is similar to that of a meaning preserving transformation.<sup>1</sup>

Transformations form the basis of restructuring and describe a single action that improves structure and preserves meaning. Restructuring, on the other hand, is a preparatory process in which transformations are applied at regular intervals to realize certain goals and obtain what is deemed a *well structured system*, suitable for comprehensive maintenance operations.

### 2.1.1 Manual Restructuring

Various methods exist whereby one can restructure a system. Manual restructuring is perhaps the most common technique for implementing transformations, although not very efficient. Griswold reached three conclusions based on observations that were made during an experiment where a group of programmers were given the task of restructuring a small program by hand [19]:

1. People are inconsistent. The subjects in the experiment used various combinations of **Copy-Paste** and **Cut-Paste** techniques to restructure the program, but were inconsistent as far as the application of these techniques were concerned.
2. People make mistakes. Many of the subjects committed small syntactic and semantic errors which can easily be avoided when taking an automated, tool-based approach.
3. Manual restructuring is haphazard. Many of the subjects focused locally, ignoring the global implications of their actions. Again, these problems can be avoided by taking a tool-based approach.

The number of errors being introduced during manual restructuring can be restricted through regular testing [17]. However, as Dijkstra so eloquently remarked:

“...program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence.” [12]

---

<sup>1</sup>*Refactoring* and *transformation* are treated as equivalent terms for the purpose of this thesis.

### 2.1.2 Automated Restructuring

Alternatively, the transformation process may be automated to realize restructuring goals. A tool based on the direct manipulation of the source text of a program is perhaps the simplest solution for implementing transformations. A transformation that renames a variable translates into a **Search-and-Replace** operation, a function found in most text editors. Difficulties exist in obtaining information about semantic properties when manipulating the source text of a program directly [20]. Renaming a variable is simple compared to an operation that allows one to re-order statements, possibly changing the behaviour of the program and its functionality. Tools based on direct text manipulation are therefore not an effective solution to satisfy the requirements of automation.

Techniques such as term rewriting can also be used to implement transformations. Term rewriting is the specification of a system in terms of recursive equations. These equations are transformed into new, but equivalent equations using a process of folding and unfolding [9]. Burstall and Darlington's rewrite system was originally designed to facilitate the creation and maintenance of software. The authors state that:

“The overall aim of our investigation has been to help people to write correct programs which are *easy to alter*.” [9]

Although restricted to applicative (functional) languages [13], term rewriting has been successfully applied in the development of transformation tools for legacy COBOL systems [43].

A third method for implementing transformations is the manipulation of program representation structures. These structures not only record the syntactic structure of a program, but often provides additional information to aid with tasks such as the analysis of semantic properties. While the majority of representation structures have been around since the early days of compilers, others have only recently emerged.

The information these structures provide are used by restructuring tools to determine the validity of a transformation and the impact it will have on the functionality of a program. A further advantage is that they can be used to generate new source code once a transformation has been applied.

Program representation structures are often based on graphs that can be visualised. The visualisation can either be achieved by using standard graph layout algorithms [38] or by using more specialised visualisation methods, such as the Star-Diagram [6], to

focus on specific properties. Tools such as *Aspect Browser* (AB) creates a special map of the program source to assist with the planning of maintenance operations [21]. The major advantage of visualisation lies in the abstract reasoning being presented to the maintainer, allowing him to choose the best transformation for a specific situation.

## 2.2 Program Representation Structures

Understanding the nature of the various program representation structures is crucial when selecting a structure capable of meeting the requirements of a specific restructuring tool. Each structure provides a number of advantages and disadvantages when compared to other structures. This section will briefly examine four representation structures and discuss their role in restructuring tools.

### 2.2.1 The Abstract Syntax Tree

The AST is the closest representation of the program source, as the graph is primarily constructed through the application of the production rules of the language's syntax [1, 46]. Being a close representation of the source code makes it ideal to generate new source code, but difficult to obtain information required to perform meaning preserving transformations.

This problem has recently been addressed by the application of virtual control flow and demand driven data analysis [29]. Virtual control flow analysis is a method through which control successor and predecessor expressions can be computed on demand. The AST, combined with demand driven data analysis, eliminates the need to maintain other complex structures [29]. The `CStructure` transformation tool makes use of this technique [29].

### 2.2.2 The Control Flow Graph

A CFG is a directed graph whose nodes are composed of basic blocks with directed edges between two nodes  $N_1$  and  $N_2$  if control flows directly from  $N_1$  to  $N_2$  [7, 44]. Traditionally, the CFG is used as a form of intermediate representation to facilitate optimisation operations within a compiler [1, 7].

Although it is impossible to generate accurate source code from the CFG alone, it

does provide the necessary information to ensure meaning preserving transformations. The CFG must be used in conjunction with other structures, such as the AST, to generate source code. Maintaining multiple structures introduces complexities into a restructuring tool, such as defining mapping functions to relate one structure to another or keeping the various structures synchronised between transformations [29, 19].

### 2.2.3 The Program Dependence Graph

A PDG represents a program as a graph where the vertices are statements or predicate expressions and the edges connecting these vertices denote the control and data dependencies on which the execution of the vertices depend [15]. Since both control and data dependencies are represented by the PDG, it requires the construction of a CFG and some extensive data flow analysis. PDGs are typically used as a form of intermediate program representation in optimising compilers [7, 32] or in designing program slicers [15]. The PDG also introduces several disadvantages, such as the calculation of information that may never be used [29]. However, combining the PDG and AST allows for the implementation of meaning preserving transformations [19]. Griswold's tool for restructuring *Scheme* programs was based on manipulating the PDG.

### 2.2.4 The Unified Inter-procedural Graph

The UIG addresses the issue of multiple representation structures to some extent and reduces information redundancy created by maintaining multiple structures [22]. The UIG combines the features of the call graph, program summary graph, inter-procedural graph and system dependence graph into a single structure. The nodes in a call graph represent individual procedures and record information related to call sites and parameters. The program summary graph extends the call graph, allowing flow sensitive data analysis across procedures. The inter-procedural graph can be used for more specialized analysis, but does not provide any control dependence information. The system dependence graph is a variation of the call graph, capable of representing control dependencies.

The construction of the unified inter-procedural graph requires that all of the aforementioned structures are built. Using only the UIG to develop a restructuring tool may not be the most efficient method, especially given the overhead incurred by constructing the various graphs. However, the UIG was developed for a large software

maintenance environment that, apart from restructuring, included activities such as testing and debugging and should be evaluated in terms of its operating environment.

## 2.3 Summary

Examination of the representation structures seems to reveal that restructuring tools are based on data structures whose original design were not prompted by the need to perform source-to-source transformations. These structures were designed with different goals in mind, such as generating optimal code or computing minimum sub-programs to identify the relationship of a specific entity within a system in the case of a program slicing tool.

Morgenthaler makes the following remark with regards to the various program representations:

“Since none of these data structures explicitly contain all the information required by restructuring transformations, the question of how to obtain the remaining information must still be answered.” [29]

Chapter 3 introduces the program representation structure employed by NORT. This structure is not an answer to the question posed by Morgenthaler, but perhaps a step towards finding one.

## Chapter 3

# The Context Entity Graph

Transformations in NORT are applied by manipulating a program representation structure called the *context entity graph* (CEG). The construction of the CEG and its relationship with the *Oberon* language is examined in this chapter. The structure fulfils two important functions: First, it facilitates the application of a transformation and provides the necessary information to determine if the transformation is valid. Second, the CEG is an accurate representation of the original program and can be used to generate new source code after every restructuring operation.

A description of the CEG is presented in Section 3.1. A more formal definition will be provided in Chapter 4. The construction of the CEG is examined in Section 3.2.

### 3.1 Definitions

The CEG is a decorated graph based on the abstract syntax tree (AST). It is primarily composed of nodes representing entities and contexts. Apart from being used to represent the structure of a program, it also records information regarding specific dependencies that may exist between various nodes in the graph.

An *entity* constitutes an indivisible language construct or declaration that may not encapsulate other entities and must impose a single execution path if it represents a statement. Examples of entities include constant, type and variable declarations as well as simple executable statements such as assignments, procedure calls, EXIT and RETURN statements.

A *context* extends the entity concept and encapsulates the declaration of other entities. Contexts are also used to represent complex type structures or multiple execution paths in the case of compound and selection statements. Examples of contexts include modules, records, procedure declarations, compound statements such as **WHILE** and **LOOP** and selection statements such as **IF** and **CASE**.

## 3.2 Constructing the CEG

The nodes inside the CEG not only represent the contexts and entities that exist within a program, but also the dependencies that may exist between them. Some dependencies are computed while the CEG is constructed, whilst others are computed on demand when applying specific transformations. The CEG is a dynamic structure and may either grow or shrink during the application of transformations. For example, removing a constant declaration would result in the deletion of the nodes in the CEG representing its declaration.

The mechanisms responsible for constructing the CEG are embedded within the parser of **NORT**. The parser itself is based on a typical recursive descent algorithm [1], similar to the implementation used by various *Oberon* compilers such as **OP2** and **00C2** [7, 45, 47]. Extending the parser of the **OP2** compiler, as was done in the case of a program slicing tool developed at Linz [42], would follow the design philosophy of *Oberon*, but was rejected for the following reasons:

1. The **OP2** compiler dynamically constructs and collapses scopes during parsing. The requirements imposed by the transformation functions supported in **NORT** require that all scopes within a program are visible (open).
2. Certain optimisations, such as constant folding, may destroy information required to reproduce accurate source code. Ultimately, a restructuring tool performs source-to-source translations guided by rules that attempt to preserve functional equivalence. Consider the constant declaration `DiskSpace = Blocks*BlockSize` where `Blocks = 1440` and `BlockSize = 1024`. The reproduced source after constant folding becomes `DiskSpace = 1474560`. The original code can no longer be reproduced because of a simple optimisation. The loss in meaning incurred in this example is minimal, but may increase when more complex systems are considered.

3. The OP2 compiler would require extensive modifications to include the necessary structures to represent dependency relationships and remove unnecessary information used during code generation.
4. The OP2 compiler is designed to be portable with support for multiple machine architectures [10]. However, many variations of Oberon (both the language and operating system) are in existence today, including different compiler implementations. Separating NORT from a specific compiler implementation, and more specifically, the program representation used by such an implementation, results in an independent tool and greater portability.

The various constructs of *Oberon* will be discussed to show how the CEG is constructed. Each section not only examines the formal language definition of the construct, but also provides short, concrete examples to illustrate the relationship between the language and the CEG. A short discussion regarding the implementation of each construct is presented at the end of every section.

### 3.2.1 Language Restrictions

Only a subset of the *Oberon* language is currently supported by NORT. The subset was chosen to facilitate the testing and evaluation of the CEG structure in terms of source-to-source transformations and the application of demand driven analysis techniques. The following restrictions were imposed:

- Pointer declarations are allowed, but entities containing references to pointers may result in undefined behaviour when transformations involving data flow analysis are applied. Points-to analysis is a technique that can be used to perform data analysis on expressions and statements involving pointers by identifying aliases within a program. An alias is created when two l-values within a statement or expression reference the same memory location [29].

Various points-to analysis algorithms exist, but are often not suited for demand driven techniques because of their exhaustive nature and complexity [29, 40, 41]. However, near linear time algorithms do exist and may be considered for inclusion in future implementations [3, 41].

- Type extended records are not supported. *Oberon* implements objects as type extended records, resulting in the creation of both static and dynamic types. Type

extended records are evaluated by using the `WITH` guard along with the `IS` guard test while methods are implemented using type bound procedures and procedure variables [35, 47]. Procedure variables are similar to function pointers found in *C* and *C++* and would therefore require the implementation of points-to analysis algorithms to facilitate data and code abstraction transformations involving type extended records.

### 3.2.2 The Module Context

The unit of compilation in *Oberon* is called a module [47] and is represented by the module context. An example of a module context is illustrated in Figure 1 along with its EBNF definition. The context in which an entity exists is called the defining or parent context. The `Context` edge of a node always points to its defining context. Apart from the predefined context (discussed in Section 3.2.9), no other context exists outside the module because `NORT` restricts transformations to a single module. This restriction is imposed by turning the `Context` edge of a module into a self-loop.

Labelled, directed edges point to the various contexts and entities that exist within the module. These edges represent the `DeclarationSequence` and `ImportList` productions contained in the EBNF. Constant declarations are denoted by the `ConstantEntity` edge, type declarations by the `TypeEntity` edge, and so forth.

Modules in *Oberon* may export items to grant other modules access to them. The collection of items exported by a module is called a definition [35]. A module must import other modules to gain access to their definitions. The `ImportContext` edge in Figure 1 represents the definitions of all the imported modules.

Finally, each module contains an optional statement sequence represented by the `Body` edge of the module context. This edge points to the first statement if a statement sequence is present.

### Implementation

The construction process is initiated by calling the `Module` function located in the `NORTParser` module. The function's implementation is shown in Figure 2. An empty module context containing the name of the module being parsed is created in line 7. Import contexts are generated if necessary (lines 9 through 13), followed by the various

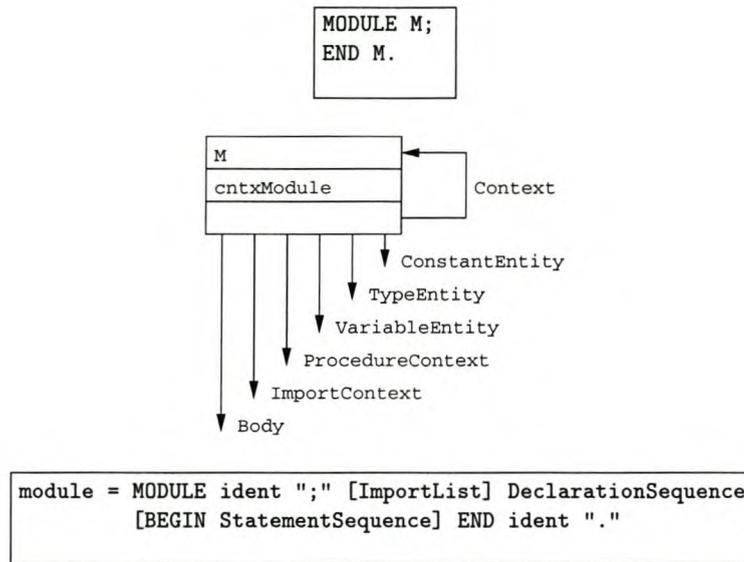


Figure 1: The top part of the figure contains an empty *Oberon* module called **M** along with its CEG. The EBNF definition for a module is presented at the bottom of the figure. The **Context** edge forms a self-loop because, apart from the predefined context, no contexts exist outside the module.

declarations (line 14) and a possible statement sequence (lines 15, 16 and 17). Most of the procedures in the parser require the current context as a parameter. This not only results in efficient searching, but also ensures that entities are created within the correct parent context. The completed CEG is returned in line 19 once the source program has been parsed.

### 3.2.3 The Import Context

Module definitions are stored in a symbol file that is generated during compilation and incrementally updated when changes occur. *Oberon* implements a variation of the *object model* (OM) for its symbol files that allows the addition of new symbols to the module definition without invalidating existing clients of the module [11]. The symbol file of each imported module is examined by **NORT** to obtain the module's definition and create an equivalent import context. The **SYSTEM** module is an exception to this rule. This module is platform dependent and used to perform specific low-level operations. The import context for the **SYSTEM** module is generated internally by **NORT**.

Figure 3 illustrates module **M2** importing module **M1** along with the corresponding

```

1. PROCEDURE Module*(): NCEG.CEGNode;
2. VAR
3.   ModuleContext: NCEG.CEGNode;
4. BEGIN
5.   NCEG.InitializeStats; NS.GetToken(Token);
6.   Expect(ND.symModule); Expect(ND.symIdent);
7.   ModuleContext := NCEG.CreateModuleContext(PreviousToken.Name);
8.   Expect(ND.symSemi);
9.   IF Token.Type = ND.symImport THEN
10.    NS.GetToken(Token); Import;
11.    WHILE Token.Type = ND.symComma DO Import END;
12.    Expect(ND.symSemi)
13.  END;
14.  DeclarationSequence(ModuleContext);
15.  IF Token.Type = ND.symBegin THEN
16.    NS.GetToken(Token); StatementSequence(ModuleContext, ModuleContext.Body)
17.  END;
18.  Expect(ND.symEnd); Expect(ND.symIdent); Expect(ND.symStop);
19.  RETURN ModuleContext
20. END Module;

```

Figure 2: The `Module` function is the starting point of the restructuring process. A CEG is returned upon successfully parsing an *Oberon* module.

EBNF definition. An import context for `M1` is created and connected to `M2` by its `ImportContext` edge. Note that the structure of an import context closely resembles that of the module context because it represents a module definition.

## Implementation

The list of imported modules is parsed by the `Import` procedure called in line 10 of Figure 2. Procedure `Import` calls `CreateImportContext` once it has identified the imported module. Procedure `CreateImportContext` is responsible for parsing the definition of the imported module (located within the symbol file section of its object file) and creating an import context node that corresponds with its definition. The import context is returned and inserted into the `ImportContext` edge of the module context. This process is repeated until contexts for all the imported modules have been created.

### 3.2.4 Constant Entities

Figure 4 depicts the CEG for two constant declarations along with the *Oberon* implementation and the corresponding EBNF. The first constant declaration within a

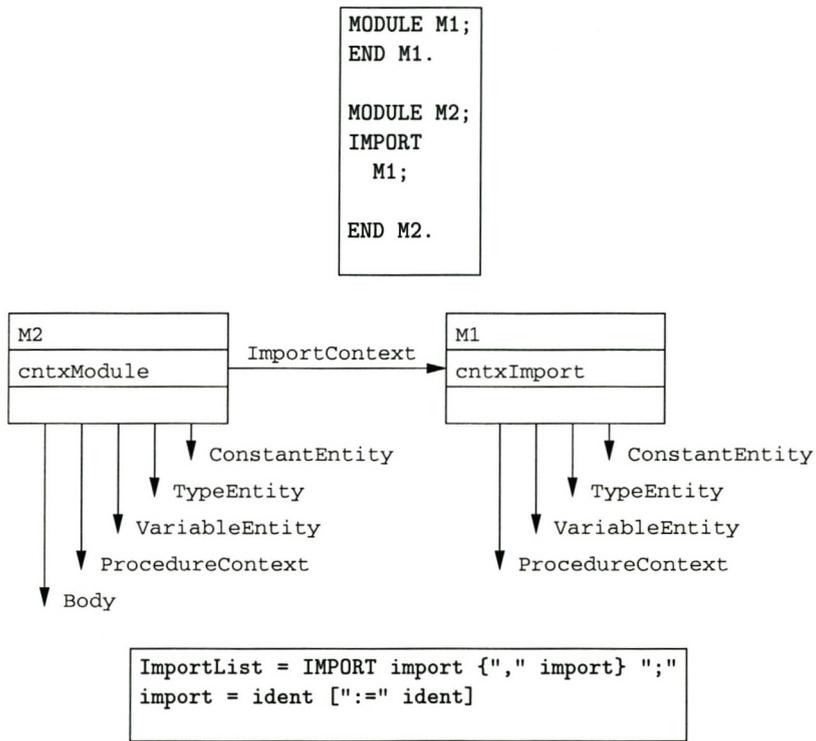


Figure 3: Module M1 is imported by M2. The import context created from the definition of M1 is connected to the module context of M2 through the `ImportContext` edge.

context is indicated by the `ConstantEntity` edge. The remaining declarations are located by traversing the `Next` edge of each constant entity. Every constant contains an edge labelled `Expression` that points to the root of the expression tree associated with the constant.

Constant `A` contains an edge labelled `DependencySet`. This edge is present whenever an entity is referenced by other entities within a module. The `DependencySet` edge points to a list of reference dependency nodes. These nodes in turn point to the actual entity that generated the reference. In this example, constant `B` references `A` by way of the expression, `A*2`. The reference dependency node therefore points to `B`.

The expression tree associated with `B` in Figure 4 contains a node with an edge labelled `Designator` that points to an anonymous designator node. The anonymous designator in turn points to the node containing the declaration of `A`. This level of indirection is required to create a uniform structure that can accommodate both single and qualified identifiers and is based on the EBNF definition of a designator. Qualified identifiers are used when referring to imported entities or record fields. for example `R.x.y.z`. A designator node is created for every reference to an identifier. The designator nodes also ensure that correct source code is generated by always obtaining the name of an identifier from its declaration.

## Implementation

Constant declarations are handled within the `DeclarationSequence` procedure. The excerpt in Figure 5 shows the implementation for constant declarations (lines 10 through 24). The identifier name is used to perform a local search within the current context to determine whether similarly named identifier already exists (line 13, `FindLocalCEGNode`). If not, a constant entity, along with its expression tree, is created and inserted into the `ConstantEntity` edge of the current context (lines 15 through 21). Lines 25 through 31 handle the remaining declarations for types, variables and procedures.

Dependency nodes are generated during parsing. Consider the declaration of `B` in Figure 4 again. A constant entity for `B` will be created in node at line 19 of Figure 5. This entity is then passed as a parameter to procedure `Expression` (line 20). Eventually the expression routine will locate the node representing `A`. At this point, a dependency node pointing to `B` is created and inserted into the dependency set of `A` after which the expression tree is completed and finally inserted into the `Expression` edge of `B`.

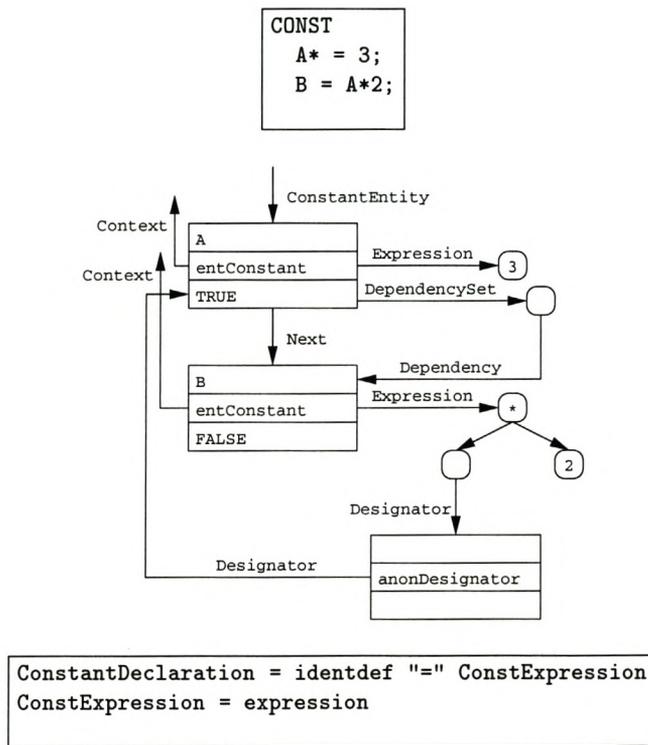


Figure 4: Two constant declarations exhibiting a dependency relationship. The boolean fields of the nodes indicate whether they are marked for export. This is indicated by placing the export mark \* after the identifier being exported (the declaration of A in this case).

The completed constant is then inserted into the `ConstantEntity` edge of the current context.

### 3.2.5 Type Entities

Type entities are created for type declarations. Figure 6 shows the CEG for two simple type declarations along with the EBNF definition.

Type names have an associated type called a type dependency. In Figure 6, `X` has a type dependency on the predefined type entity `INTEGER` while `Y` exhibits a type dependency on `X`. This dependency is indicated by the dependency set of `X`.

Apart from type dependencies, each type entity also records information about its own structure. Both `X` and `Y` are user defined types (indicated by the `typUserDefined` field in the node) while `INTEGER` is associated with a simple type structure (indicated by `typSimple`).

Figure 7 illustrates a type declaration for a record. The internal fields of the record are represented by field entities. The defining context of a field entity is the record structure in which it is defined. Note that the node type of `R` is `entType` even though it represents a context. The declaration of `R` represents an entity, but the type structure associated with its type dependency represents a context. Type entities and contexts are distinguished based upon their structural type (`typRecord` in the case of the type dependency associated with `R`).

### Implementation

Type declarations are parsed by the `DeclarationSequence` procedure in Figure 5. This procedure is only responsible for creating the type entity and inserting it into the CEG. The type dependency is constructed by procedure `Type` shown in Figure 8. Consider the type declaration of `Y = X` used in Figure 6. Procedure `DeclarationSequence` will create a type entity for `Y` and call `Type`, passing both the context and newly created type entity as parameters. Procedure `Qualident` (called in line 7) is responsible for locating `X` and creating a dependency node for `X` pointing to `Y`. If `X` does not exist it is assumed to be a forward type declaration and handled accordingly (lines 8 through 13). The node returned in line 14 will be inserted into the `TypeDependency` edge of the type entity created by `DeclarationSequence`. Record, array, pointer and procedure

```

1. PROCEDURE DeclarationSequence(VAR context: NCEG.CEGNode);
2. VAR
3.   export: BOOLEAN;
4.   modname, name: ND.Name;
5.   typedep, type, var, var2, pc, node: NCEG.CEGNode;
6. BEGIN
7.   WHILE (Token.Type = ND.symConst) OR (Token.Type = ND.symType) OR
8.     (Token.Type = ND.symVar) OR (Token.Type = ND.symProcedure) DO
9.     export := FALSE;
10.    IF Token.Type = ND.symConst THEN
11.      NS.GetToken(Token);
12.      WHILE Token.Type = ND.symIdent DO
13.        name := Token.Name; node := NCEG.FindLocalCEGNode(context, name);
14.        IF node = NIL THEN
15.          NS.GetToken(Token); export := FALSE;
16.          IF Token.Type = ND.symMul THEN export := TRUE; NS.GetToken(Token) END;
17.          IF export & (context.Type # NCEG.cntxModule) THEN Error(NE.errNotLevel0) END;
18.          Expect(ND.symEqual);
19.          node := NCEG.CreateConstantEntity(context, name, export, NIL);
20.          Expression(node.Expression, context, node, ExprConst, 0);
21.          NCEG.InsertCEGNode(context.ConstantEntity, node)
22.        ELSE Error(NE.errDupIdent) END;
23.        Expect(ND.symSemi)
24.      END
25.    ELSIF Token.Type = ND.symType THEN
26.      (* Type declarations *)
27.    ELSIF Token.Type = ND.symVar THEN
28.      (* Variable declarations *)
29.    ELSIF Token.Type = ND.symProcedure THEN
30.      (* Procedure declarations *)
31.    END
32.  END
33. END DeclarationSequence;

```

Figure 5: Implementation for parsing declarations and constructing the corresponding nodes within the CEG. Lines 10 through 24 show the implementation for constant declarations. Type, variable and procedure declarations are handled in a similar fashion.

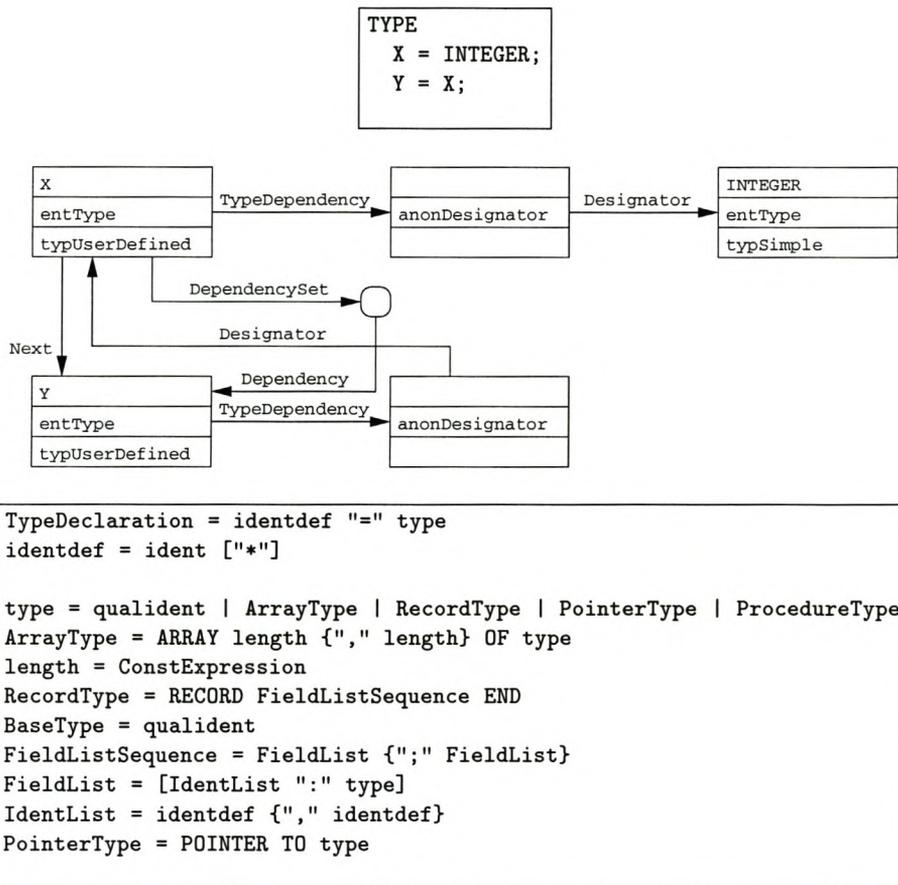


Figure 6: An example of a simple type declaration. Note that the dependency Y exhibits on X is indicated by the dependency set of X.

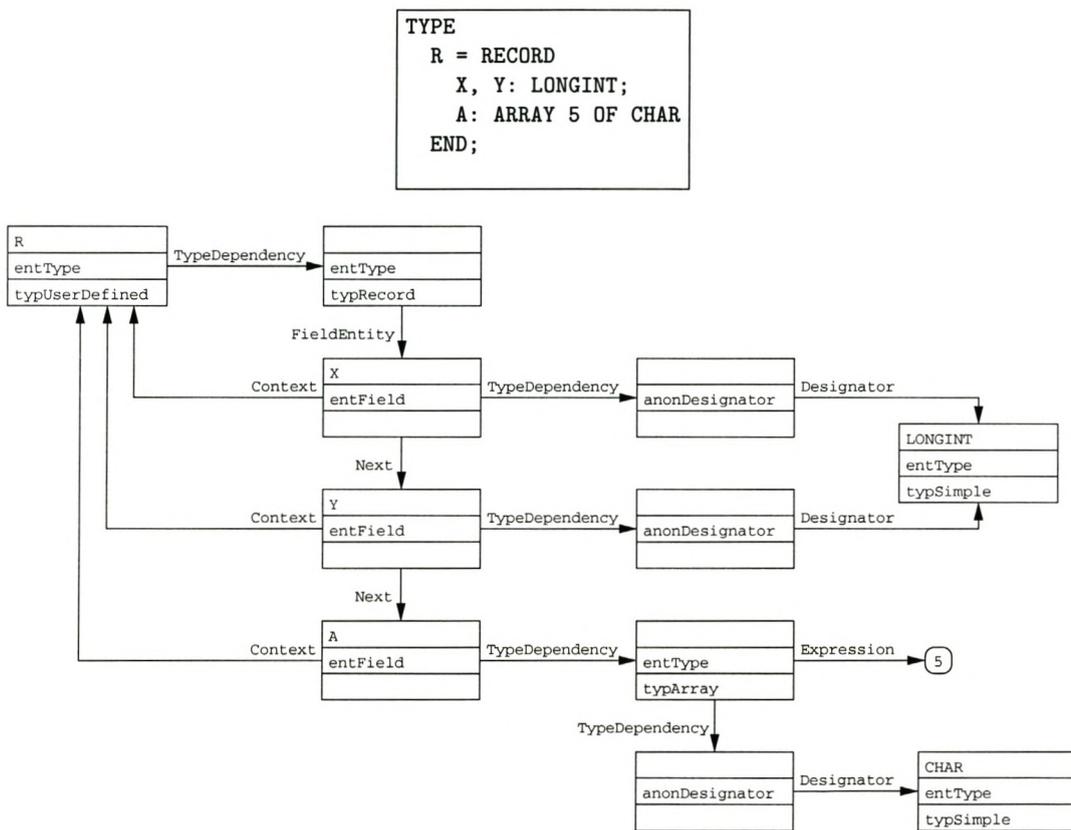


Figure 7: CEG representation for a RECORD. Each field inside the record has a separate type dependency. The structure for representing an ARRAY is illustrated by the record field A.

```

1. PROCEDURE Type(context, typedep: NCEG.CEGNode; forward: BOOLEAN): NCEG.CEGNode;
2. VAR
3.   basetype, type, node: NCEG.CEGNode;
4.   name: ND.Name;
5. BEGIN
6.   IF Token.Type = ND.symIdent THEN
7.     name := Token.Name; node := Qualident(context, typedep);
8.     IF (node = NIL) & (forward) THEN
9.       type := NCEG.CreateTypeEntity(context, name, NIL, NCEG.typForward, FALSE);
10.      NCEG.InsertReferenceDependency(type.DependencySet,
11.                                     NCEG.CreateReferenceDependency(typedep));
12.      NCEG.InsertCEGNode(context.TypeEntity, type);
13.      node := NCEG.CreateDesignator(type, NIL)
14.    END;
15.    RETURN node
16.  ELSIF Token.Type = ND.symArray THEN
17.    (* Create ARRAY type *)
18.  ELSIF Token.Type = ND.symRecord THEN
19.    (* Create RECORD type *)
20.  ELSIF Token.Type = ND.symPointer THEN
21.    (* Create POINTER type *)
22.  ELSIF Token.Type = ND.symProcedure THEN
23.    (* Create PROCEDURE type *)
24.  END
25. END Type;

```

Figure 8: Type dependencies are constructed by procedure `Type`. Both the context and type entity are passed as parameters to ensure that reference dependencies are updated correctly.

types are handled in a similar fashion.

### 3.2.6 Variable Entities

The CEG for a variable declaration is illustrated in Figure 9. The type dependency of a variable declaration is treated similar to that of a type declaration. The `VariableEntity` edge of a module or procedure context points to the first variable declaration within the context.

### 3.2.7 The Procedure Context

Procedure declarations are treated as contexts because they may contain declarations and usually have a body consisting of statements. Figure 10 shows the CEG for a function procedure with two parameters, one of which is a reference parameter (indicated

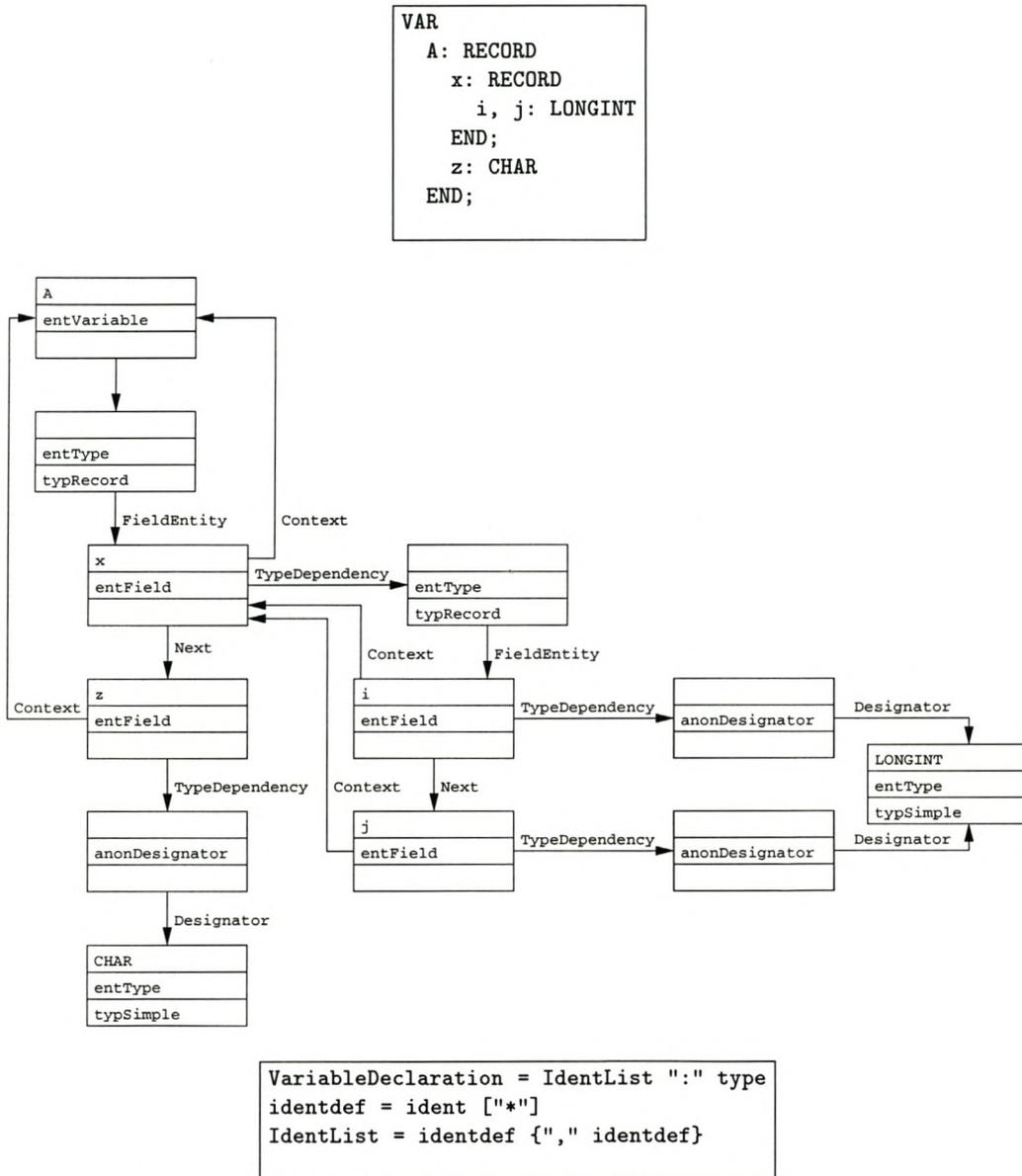


Figure 9: CEG for a variable declaration. The end nodes of the type dependency edges are treated as anonymous structures.

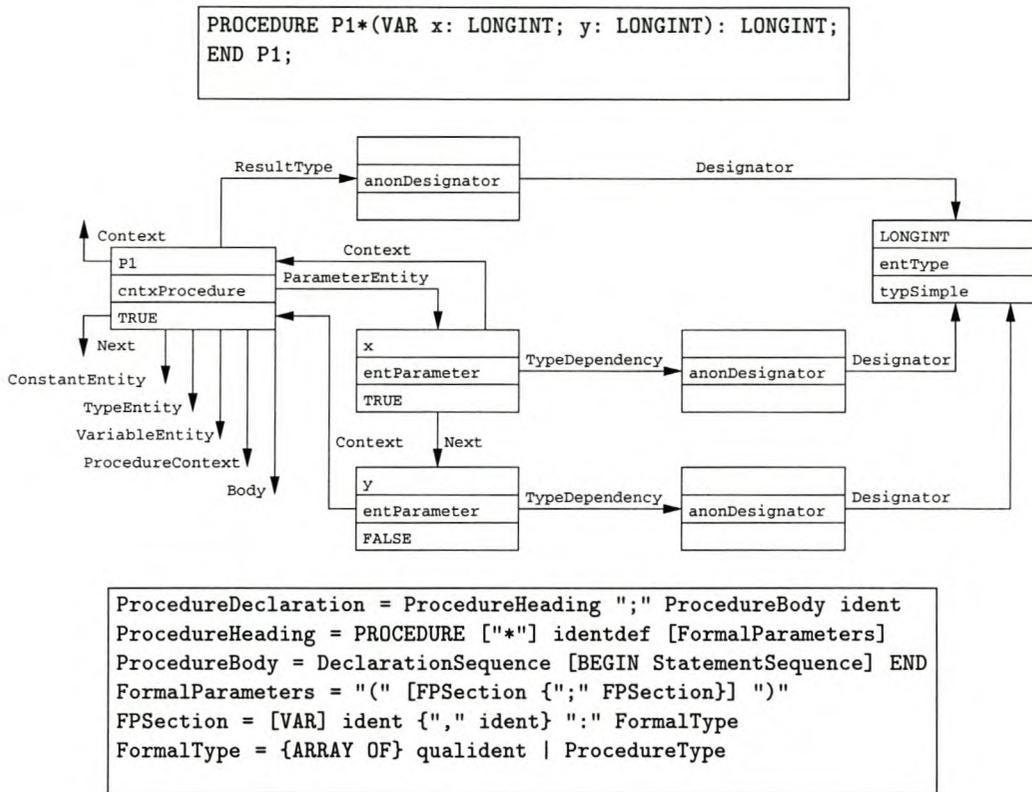


Figure 10: A simple function procedure. The structure of a procedure context is similar to that of the module context with regards to how declarations are represented. The boolean field of each parameter is set to TRUE if it was declared as a reference parameter.

by the VAR construct preceding its name). The EBNF for a procedure declaration is presented at the bottom of the figure.

The declarations inside the procedure are treated in a similar fashion to those inside a module context. Local definitions are located through the ConstantEntity, TypeEntity, VariableEntity and ProcedureContext edges. The Body edge points to the first statement of the procedure. The parameters are located through the ParameterEntity edge. The result type for a function procedure is located through the ResultType edge.

### 3.2.8 Statements

An Oberon statement can either be classified as an entity or as a context. Statement entities include those statements that neither contain statement sequences (such as a

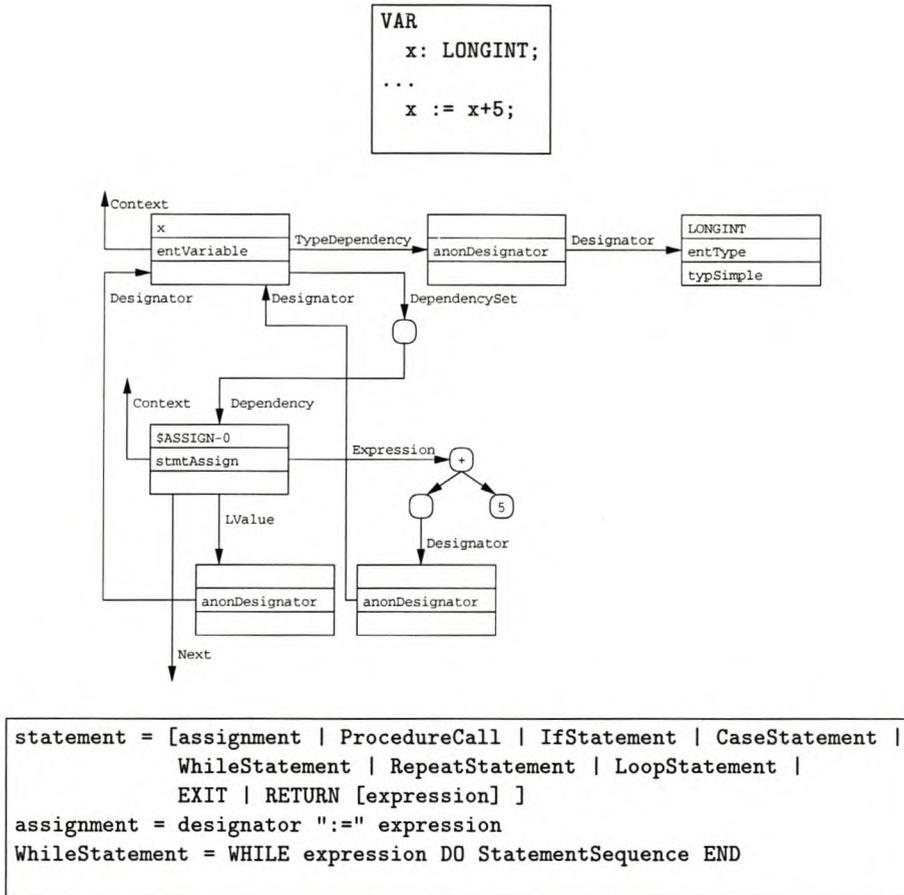


Figure 11: Assignments are classified as a statement entities because they can not contain other statements and represent a single execution path.

LOOP or WHILE) nor multiple control paths (such as a CASE or an IF). Given this definition, valid statement entities include assignments, procedure calls, EXIT and RETURN statements.

Unlike declared entities and contexts, statements do not have names. However, NORT will assign a name to every statement to assist the maintainer in identifying specific statements. The name of a statement is composed of a \$, followed by the type of the statement and a numerical designation. The \$ ensures that the statement name is illegal and therefore can not be confused with declared entities within a program. For example, the first assignment will be named \$ASSIGN-0, the second \$ASSIGN-1 and so forth. An example of an assignment statement is illustrated in Figure 11.

Compound and selection statements are viewed as statement contexts rather than

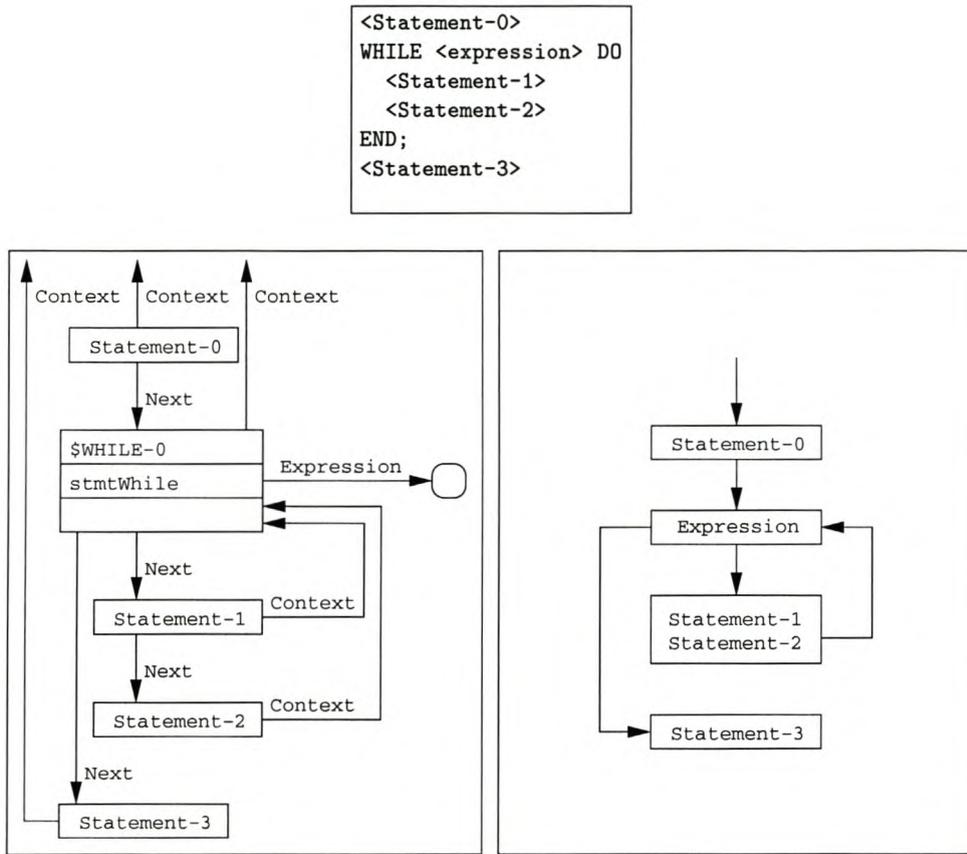


Figure 12: The CEG for a **WHILE** statement is illustrated on the left side of the figure. The right side shows a typical control flow graph and assumes that **Statement-1** and **Statement-2** share the same block. Notice the similarities in structure between the CEG and CFG.

statement entities and introduce a secondary function of contexts: The identification of basic blocks. A basic block is defined as a sequence of consecutive statements in which control enters at the beginning and leaves at the end without the possibility of halting or branching except at the end [1]. Figure 12 illustrates the CEG for a **WHILE** statement accompanied by its control flow graph (CFG). Notice the resemblances between the context edges of the CEG (left) and the control edges of the CFG (right).

```

1. PROCEDURE StatementSequence(context: NCEG.CEGNode; VAR body: NCEG.CEGNode);
2. BEGIN
3.   NCEG.InsertCEGNode(body, Statement(context));
4.   WHILE Token.Type = ND.symSemi DO
5.     NS.GetToken(Token);
6.     NCEG.InsertCEGNode(body, Statement(context))
7.   END
8. END StatementSequence;

```

Figure 13: Statements within a statement sequence are parsed, nodes are created by the `Statement` function procedure and then inserted into the `Body` edge.

### Implementation

Statements are processed by two procedures. Procedure `StatementSequence`, shown in Figure 13, is responsible for parsing statements one at a time (lines 3 and 6) and inserting them into the `Body` edge of the current context. The `context` parameter ensures that statements are created within the correct context.

The different statement types are parsed, and their CEG nodes constructed by procedure `Statement` in Figure 14. The excerpt shows the processing of a `WHILE` statement. The statement is created in a temporary node (`stmt`) in line 13 and its expression inserted in line 14. The statement is passed as a parameter to procedure `Expression`, ensuring that every identifier in the expression will receive a dependency node pointing to the statement. Procedure `StatementSequence` is called in line 16, passing the newly created `stmt` as parameter. This method ensures that the defining context for statements inside a compound statement points to the compound statement. The `stmt` node is returned and inserted into the context specified by procedure `StatementSequence` (Figure 13, lines 3 and 6). Other statement types are handled in a similar fashion by procedure `Statement`.

#### 3.2.9 The Predefined Context

*Oberon*, like many other languages, contains a number of predefined constructs including built-in procedures, functions, types and constants. These predefined constructs are located within the predefined context. Unlike other entities, the predefined context does not have a defining context. This context is usually searched when an entity could not be located within the CEG. Table 1 lists the entities defined in this context.

```
1. PROCEDURE Statement(context: NCEG.CEGNode): NCEG.CEGNode;
2. VAR
3.   designator, elsif, else, stmt: NCEG.CEGNode;
4. BEGIN
5.   IF Token.Type = ND.symIdent THEN
6.     (* Assignment or procedure call *)
7.   ELSIF Token.Type = ND.symCase THEN
8.     (* CASE statement *)
9.   ELSIF Token.Type = ND.symIf THEN
10.    (* IF statement *)
11.  ELSIF Token.Type = ND.symWhile THEN
12.    NS.GetToken(Token);
13.    stmt := NCEG.CreateStatement2(context, NCEG.stmtWhile);
14.    Expression(stmt.Expression, context, stmt, ExprOther, 0);
15.    Expect(ND.symDo);
16.    StatementSequence(stmt, stmt.Body);
17.    Expect(ND.symEnd)
18.  ELSIF Token.Type = ND.symRepeat THEN
19.    (* REPEAT statement *)
20.  ELSIF Token.Type = ND.symLoop THEN
21.    (* LOOP statement *)
22.  ELSIF Token.Type = ND.symExit THEN
23.    (* EXIT statement *)
24.  ELSIF Token.Type = ND.symReturn THEN
25.    (* RETURN statement *)
26.  ELSIF Token.Type = ND.symFor THEN
27.    (* FOR statement *)
28.  END;
29.  RETURN stmt
30. END Statement;
```

Figure 14: Procedure `Statement` is responsible for parsing individual statements and constructing the necessary nodes to represent each statement within the CEG.

Identifier	Type	Identifier	Type
CHAR	Type Entity	ABS	Procedure Context
BOOLEAN	Type Entity	ODD	Procedure Context
SHORTINT	Type Entity	CAP	Procedure Context
INTEGER	Type Entity	ASH	Procedure Context
LONGINT	Type Entity	LEN	Procedure Context
REAL	Type Entity	MAX	Procedure Context
LONGREAL	Type Entity	MIN	Procedure Context
SET	Type Entity	SIZE	Procedure Context
TRUE	Constant Entity	ORD	Procedure Context
FALSE	Constant Entity	CHR	Procedure Context
SHORT	Procedure Context	ENTIER	Procedure Context
LONG	Procedure Context	INC	Procedure Context
HALT	Procedure Context	DEC	Procedure Context
COPY	Procedure Context	NEW	Procedure Context

Table 1: Various types, constants and procedures are located within the predefined context.

### 3.3 Summary

The *context entity graph* (CEG) was introduced in this chapter. The structure is based on the AST and used in the application of transformations and generation of new source programs. The EBNF definition of *Oberon*, accompanied by short excerpts of *Oberon* code, was used to explain the structure and function of various nodes within the CEG. Specific issues regarding the implementation and construction of the CEG were also discussed. The application of transformations, based on the manipulation of the CEG, is examined in Chapter 4.

## Chapter 4

# Transformations on the CEG

A variety of transformations are implemented in NORT. A notation to define, abstract and describe the transformations is introduced in Section 4.1. The function of the notation is twofold: First, it facilitates the discussion of the implementation without delving into the details of the *Oberon* source code. Second, it supports the arguments regarding the correctness of the various transformations.

Transformation classes are defined in Section 4.2, followed by a detailed discussion of each transformation, along with informal arguments regarding its correctness. Excerpts from the *Oberon* implementation will sometimes accompany the discussion to clarify issues or illustrate specific concepts.

Examples illustrating the application of transformations are presented in Section 4.3 and a brief discussion of transformation dependencies is presented in Section 4.4. A discussion on how the CEG is updated to reflect the result of a transformation is presented in Section 4.5, followed by a brief summary in Section 4.6.

### 4.1 Preliminary Definitions

The CEG of a module is a labelled, directed graph  $G = (V, E)$  containing a set  $V$  of vertices (nodes) and a set  $E$  of labelled edges. Vertices represent data structures that contain many internal fields. The specific fields of a vertex are denoted by placing them between parentheses. For example,  $v(\textit{type})$  refers to the *type* field of  $v$ .

Every vertex  $v \in V$  belongs to a certain class,  $v(\textit{class}) \in \mathcal{C}$ , where  $\mathcal{C}$  is a set denoting

node classes. Four major node classes are identified:

- *CEG* nodes are used to represent entities such as variables declarations and statements.
- *Expression* nodes are used to construct the expression trees that sometimes accompany a *CEG* node. For example, the expression of a **WHILE** statement.
- *Expression List* nodes are used to represent sequences of expression trees. For example, the actual parameters of a procedure call are represented by *expression list* nodes.
- *Dependency* nodes are used to represent the dependency relationships that exist between nodes.

Every vertex  $v \in V$  has an associated type,  $v(\text{type}) \in \mathcal{T}$ , to distinguish different nodes belonging to the same class. For example, the type for a *CEG* node representing a variable declaration is *entVariable*, whereas the type for a node representing an **IF** statement is *stmtIf*. Similarly, the type associated with *expression* nodes can be used to distinguish between various operators, values and so forth.

Enumerated types are used to denote node types and are prefixed with *cntx*, *ent* or *stmt*. Let  $\mathcal{S} \subset \mathcal{T}$  denote those node types that represent statements and  $\mathcal{S}_c \subset \mathcal{S}$  those statements that are also contexts. Consider a restricted language definition that only allows assignments and **WHILE** statements. Then  $\mathcal{S} = \{\text{stmtAssign}, \text{stmtWhile}\}$  and  $\mathcal{S}_c = \{\text{stmtWhile}\}$ .

A selection of algorithms not discussed in this chapter is presented in Appendix B. These algorithms are used by the transformation functions to evaluate specific conditions.

#### 4.1.1 The Context Path

The defining or parent context for a vertex  $v$  is given by  $v(\text{context})$ . In Figure 15,  $x(\text{context}) = P$  and  $P(\text{context}) = M$ . Certain transformations require that the encapsulating contexts of nodes are examined. A procedure called *ContextPath* provides the necessary information. It takes a node  $v$  and returns a sequence of vertices that represents a path from  $v$  to the module context. The path stops at the module context

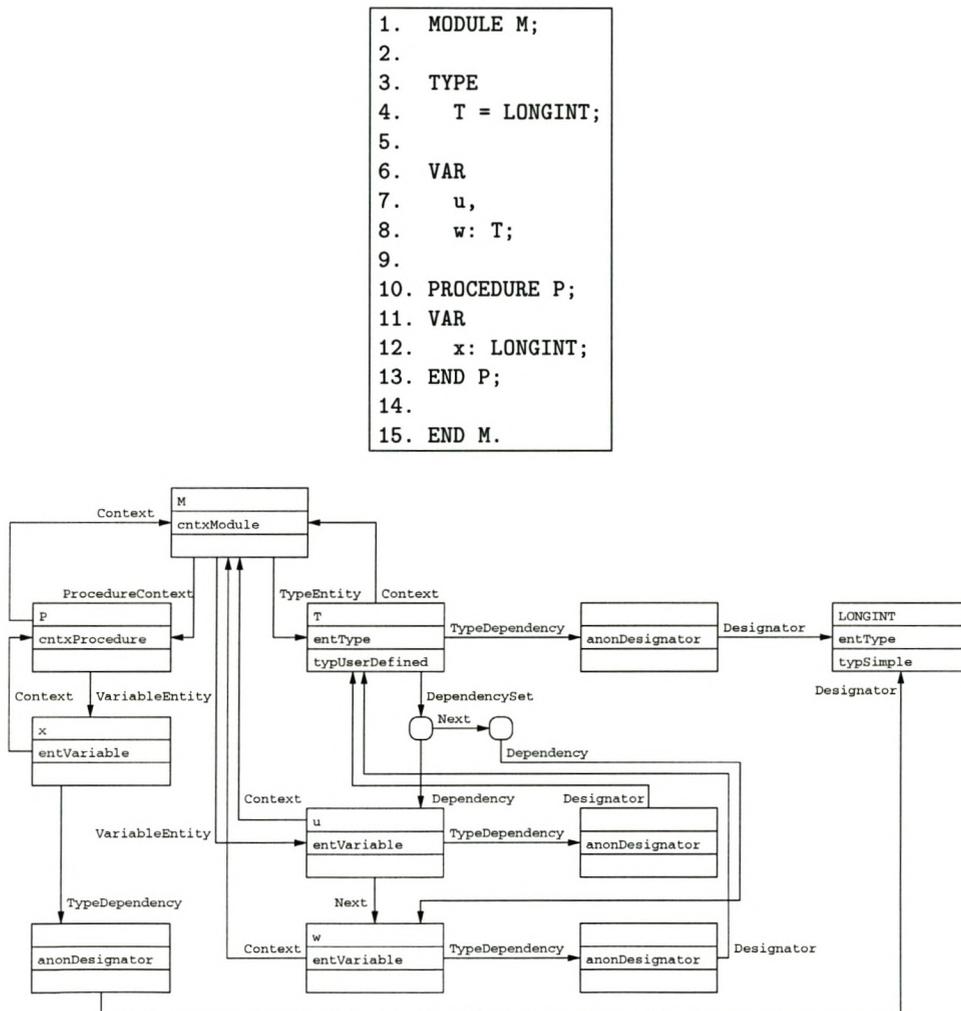


Figure 15: Example of a complete *Oberon* module and its CEG.

```

proc ContextPath(v, path)  $\equiv$ 
  if v(type) = ctxModule then InsertLast(path, v)
  else InsertLast(path, v);
        ContextPath(v(context), path)
  fi
end

```

Figure 16: Algorithm used to construct the context path of a node

because the **Context** edge of a module is always a self-loop (See Section 3.2.2). The sequence of vertices returned in *path* is called the *context path* of *v*. Given the module in Figure 15,  $ContextPath(\mathbf{x}) = \{\mathbf{P}, \mathbf{M}\}$ . The algorithm used to construct the *context path* is presented in Figure 16.

### 4.1.2 Dependencies

The dependency set of a vertex *v* is denoted by  $v(dependencyset)$ , its elements denoting vertices in the CEG that reference *v*. Given the type declaration of **T** in line 4 of Figure 15 together with the variable declarations of **u** and **w** (lines 6 and 7), both of type **T**, then  $T(dependencyset) = \{\mathbf{u}, \mathbf{w}\}$ . In some cases it is necessary to determine the vertices referenced by a vertex *v* (the opposite of the dependency set). This information is provided by the  $ReferenceDependency(v)$  function. This function returns a set containing all the vertices referenced by *v*. Continuing with the example in Figure 15,  $ReferenceDependency(\mathbf{u}) = \{\mathbf{T}\}$  and  $ReferenceDependency(\mathbf{w}) = \{\mathbf{T}\}$ .

### 4.1.3 Searching

Conducting efficient search operations within the CEG is important. Transformation functions must be able to quickly locate the necessary information to determine the validity of an operation. Two functions called *LocalNode* and *GlobalNode* are provided to facilitate local and global search operations within the CEG. The algorithms for these functions are presented in Figure 18 and Figure 19. The efficiency of a search operation is increased by providing hints using the *search* field of the current context. Each vertex  $v \in V$  denoting a context specifies a set  $v(search)$ , describing the edges of *v* that may be traversed when a search operation is performed. For example, a node representing a procedure will indicate that its **ParameterEntity** edge may be searched in addition to any other edges it specifies. Likewise, the *search* field of a record will

```

proc Node(first, id)  $\equiv$ 
  while (first  $\neq$  NIL)  $\wedge$  (first(name)  $\neq$  id) do
    first  $\leftarrow$  first(next)
  od;
  return first
end

```

Figure 17: Function *Node* searches through a linked list of nodes for a node called *id*. If the node cannot be found it returns *NIL*.

```

proc LocalNode(root, id)  $\equiv$ 
  v  $\leftarrow$  NIL;
  if srcConstant  $\in$  root(search) then v  $\leftarrow$  Node(root(constantentity), id) fi
  if (srcType  $\in$  root(search))  $\wedge$  (v = NIL) then v  $\leftarrow$  Node(root(typeentity), id) fi
  if (srcVariable  $\in$  root(search))  $\wedge$  (v = NIL) then v  $\leftarrow$  Node(root(variableentity), id) fi
  if (srcParameter  $\in$  root(search))  $\wedge$  (v = NIL) then v  $\leftarrow$  Node(root(parameterentity), id) fi
  if (srcField  $\in$  root(search))  $\wedge$  (v = NIL) then v  $\leftarrow$  Node(root(fieldentity), id) fi
  if (srcProcedure  $\in$  root(search))  $\wedge$  (v = NIL) then v  $\leftarrow$  Node(root(procedurecontext), id) fi
  if (srcImport  $\in$  root(search))  $\wedge$  (v = NIL) then v  $\leftarrow$  Node(root(importcontext), id) fi
  return v
end

```

Figure 18: Function *LocalNode* performs a local search for a node called *id* in the context specified in *root*.

indicate that its `FieldEntity` edge may be searched.

The *LocalNode*(*root*, *id*) function searches for a node *v* called *id*. The search starts at *root* (which always represents a context) and traverses all the edges specified in *root*(*search*). Search operations are localised within the context denoted by *root*. The named node is deemed non-existent if *LocalNode* returns *NIL*.

The *GlobalNode*(*root*, *id*) function extends the search from *root* to its encapsulating contexts by traversing the nodes in *ContextPath*(*root*). A local search is performed in each context. *GlobalNode* continues until it reaches the module context. The search will continue into the predefined context if the named node could not be located inside the module context. The named node is deemed non-existent if *GlobalNode* returns *NIL*.

```

proc GlobalNode(root, identid)  $\equiv$ 
  v  $\leftarrow$  NIL;
  while (root  $\neq$  NIL)  $\wedge$  (v = NIL) do
    if (srcConstant  $\in$  root(search))  $\wedge$  (v = NIL) then v  $\leftarrow$  Node(root(constantentity), id) fi
    if (srcType  $\in$  root(search))  $\wedge$  (v = NIL) then v  $\leftarrow$  Node(root(typeentity), id) fi
    if (srcVariable  $\in$  root(search))  $\wedge$  (v = NIL) then v  $\leftarrow$  Node(root(variableentity), id) fi
    if (srcParameter  $\in$  root(search))  $\wedge$  (v = NIL) then v  $\leftarrow$  Node(root(parameterentity), id) fi
    if (srcProcedure  $\in$  root(search))  $\wedge$  (v = NIL) then v  $\leftarrow$  Node(root(procedurecontext), id) fi
    if (srcImport  $\in$  root(search))  $\wedge$  (v = NIL) then v  $\leftarrow$  Node(root(importcontext), id) fi
    if (srcPredefined  $\in$  root(search))  $\wedge$  (v = NIL) then v  $\leftarrow$  Node(root(typeentity), id)
    else root  $\leftarrow$  root(context)
  fi
  od
  return v
end

```

Figure 19: Function *GlobalNode* performs a local search for a node called *id* in the context specified in *root*. If the node can not be located, the search is extended to include the parent context of *root*.

## 4.2 Transformation Definitions

Transformation functions are denoted by the  $\delta$  symbol, for example  $\delta_{rename}$ . A short description accompanies each transformation along with an explanation of any required parameters. Each transformation specifies a number of conditions that must be met before it may be applied. These conditions ensure that the transformation preserves the meaning of the program being restructured. The transformations implemented in NORT are classified according to the four categories (classes) in Bowdidge's classification [6]. These categories are:

- *Scoping transformations.* This class of transformations alters the location where an entity is declared within the program and is usually achieved through a *move* operation. An example would be moving a constant out of a procedure to make it a global declaration within the module.
- *Syntactic transformations.* This class of transformations alter specific characteristics of entities within a program. An example of a syntactic transformation would be renaming a variable or converting a **CASE** statement into an equivalent **IF** statement.
- *Control flow transformations.* Control flow transformations allow one to alter the program order of statements. It is usually achieved through a process of *exchange* or *move* operations.

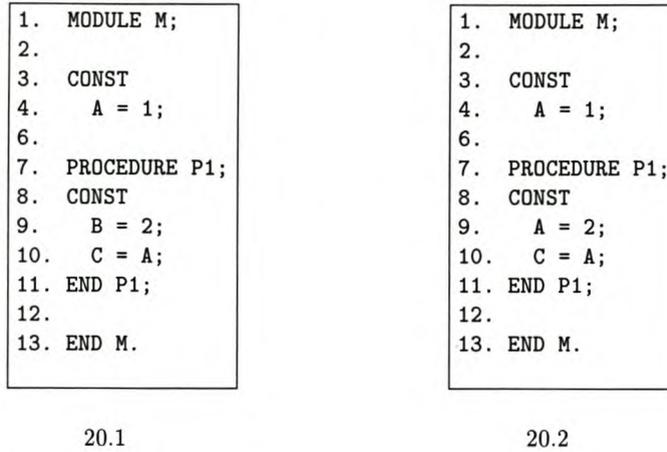


Figure 20: Renaming a constant declaration may alter the functionality of a module by changing existing dependencies.

- *Abstraction transformations.* These transformations apply to both code and data. Code abstraction allows one to replace a statement sequence by a procedure call. Data abstraction is often used when complex object-oriented hierarchies require restructuring, but may also be used on simple abstract data types.

#### 4.2.1 The Rename Transformation

The  $\delta_{rename}(v, newname)$  transformation is a syntactic transformation and may be used to change the name of a declared entity such as a variable or procedure declaration. Figure 20 illustrates the effect of this transformation when the constant declaration B (line 9 in Figure 20.1) is renamed to A (line 9 in Figure 20.2). It is clear from this example that the transformation is illegal because it would not preserve the meaning of the program (the value of C in line 10 has changed from 1 to 2). Various conditions are tested by the  $\delta_{rename}$  transformation to ensure that the new program is functionally equivalent to its predecessor.

The transformation attempts to assign  $newname$  to  $v(name)$ . The algorithm for the transformation is illustrated in Figure 21. The implementation of  $\delta_{rename}$  is remarkably simple and is illustrated in Figure 22. The implementation contains an additional parameter, **Result**, which is used for error reporting and may be ignored during the discussion of the transformation. Execution of the transformation depends on satisfying

the following conditions:

1. Only entities associated with a declaration, such as a variable or procedure, may be renamed (line 2 in Figure 21, line 8 in Figure 22). Statements may not be renamed, even though names are assigned internally by NORT. This process was described in Section 3.2.8.
2. Although renaming an entity or context to itself will have no effect on the CEG, it is deemed an invalid operation (line 3 in Figure 21, line 9 in Figure 22). Transformations are always aborted at the earliest possible point to eliminate unnecessary tests and increase efficiency.
3. The new name must be valid according to the EBNF definition of an identifier to ensure that the compiler will accept the new source code (line 4 in Figure 21, line 10 in Figure 22).
4. The semantic rules of *Oberon* state that two identifiers sharing the same name may not be declared within the same scope [25, 35]. The current scope must be examined to determine whether an entity called *newname* has already been declared. If such an entity exists then the transformation is illegal because it would create a duplicate identifier (line 5 in Figure 21, lines 11 through 14 in Figure 22).
5. Renaming declared entities may introduce false dependencies. A false dependency is created if a node  $w$  depends on a node  $u$  and a transformation alters this so that  $w$  now depends on  $z$  instead. Figure 20 illustrated how a rename operation may introduce false dependencies. Renaming B to A alters the dependency that C exhibited on the global declaration of A (line 4 in Figure 20.1). It now depends on the local declaration of A (line 9 in Figure 20.1), changing the value of C from 1 to 2. To prevent situations like this, the last condition of  $\delta_{rename}$  must determine whether renaming  $v$  would introduce a false dependency.

This condition is examined last because of its complexity. A global search is performed to determine if a node called *newname* exists. If so, a context path is constructed for every dependency in the dependency set of the node. If the context of  $v$  lies within this path the transformation will introduce a false dependency. This condition is tested in lines 6 through 13 by the algorithm in Figure 21 and the implementation is illustrated in lines 15 through 26 in Figure 22.

```

1 proc Rename(v, newname) ≡
2   if v(type) ∈ S then ERROR fi;
3   if v(name) = newname then ERROR fi;
4   if ¬ValidName(newname) then ERROR fi;
5   if LocalNode(v(context), newname) ≠ NIL then ERROR fi;
6   u ← GlobalNode(v(context), newname);
7   if u ≠ NIL then
8     dep ← u(dependencyset);
9     while dep ≠ NIL do
10      if v(context) ∈ ContextPath(dep) then ERROR fi
11      dep ← dep(next)
12    od
13  fi;
14  v(name) ← newname
15 end

```

Figure 21: Algorithm for the  $\delta_{rename}$  transformation.

If these conditions are met, then *newname* may be assigned to *v*(*name*) (line 14 in Figure 21, line 27 in Figure 22).

## 4.2.2 The Remove Transformation

The  $\delta_{remove}(v)$  transformation is used to delete declared entities such as procedures and variables. The transformation will remove *v* from the CEG and is classified as a syntactic transformation. The algorithm for the transformation is illustrated in Figure 23. The following conditions must be satisfied to execute the transformation:

1. Entities that are exported may not be deleted (line 2 in Figure 23). It is impossible to determine if an exported declaration is referenced by another module without examining all other modules in the system. A conservative approach is taken whereby it is assumed that exported declarations were created with the intent to be used by other modules.
2. If the entity being removed is referenced by other entities, the transformation will be invalid since its removal will break the dependencies exhibited by these entities (line 3 in Figure 23). Nodes within the CEG automatically track references made by other nodes to themselves through their dependency set. An empty dependency set implies that the node is not referenced and may be removed.
3. The  $\delta_{remove}$  transformation is currently limited to constant, type, variable and procedure declarations (line 4 in Figure 23).

```

1. PROCEDURE Rename*(VAR this: NCEG.CEGNode; name: ND.Name;
2.                   VAR Result: LONGINT);
3. VAR
4.   node: NCEG.CEGNode;
5.   ds: NCEG.ReferenceDependency;
6.   CP: NCEG.ContextPath;
7. BEGIN
8.   IF NCEG.IsStatement(this) THEN Result := NE.errRenameStmt; RETURN END;
9.   IF this.Name = name THEN Result := NE.errRenameSelf; RETURN END;
10.  IF ~ValidName(name) THEN Result := NE.errRenameInvalidName; RETURN END;
11.  IF NCEG.FindLocalCEGNode(this.Context, name) # NIL THEN
12.    Result := NE.errRenameDupName;
13.    RETURN
14.  END;
15.  node := NCEG.FindCEGNode(this.Context, name);
16.  IF node # NIL THEN
17.    ds := node.DependencySet;
18.    WHILE ds # NIL DO
19.      NCEG.BuildContextPath(ds.Dependency, CP);
20.      IF NCEG.ContextInPath(this.Context, CP) THEN
21.        Result := NE.errRenameInvalidDep;
22.        RETURN
23.      END;
24.      ds := ds.Next
25.    END
26.  END;
27.  this.Name := name
28. END Rename;

```

Figure 22: *Oberon* implementation for the  $\delta_{rename}$  transformation. Efficiency is improved by arranging the conditions according to their complexity and exiting the transformation on the first condition that fails.

```

1 proc Remove(v) ≡
2   if v(export) = TRUE then ERROR fi;
3   if v(dependencyset) ≠ ∅ then ERROR fi;
4   if v(type) ∉ {entConstant, entType, entVariable, cntxProcedure} then ERROR fi;
5   if v(type) = entConstant then RemoveNode(v, v(context)(constantentity))
6   elsif v(type) = entType then RemoveNode(v, v(context)(typeentity))
7   elsif v(type) = entVariable then RemoveNode(v, v(context)(variableentity))
8   elsif v(type) = cntxProcedure then RemoveNode(v, v(context)(procedurecontext)) fi
9 end

```

Figure 23: Algorithm for the  $\delta_{remove}$  transformation.

If these conditions are met then  $v$  may be deleted from the CEG (lines 5 through 8 in Figure 23).

### 4.2.3 The Move Transformation

The  $\delta_{move}(v, newcontext)$  transformation is a scoping transformation that moves a declaration from one scope to another. The transformation removes  $v$  from its current context and places it inside context  $newcontext$ . The algorithm for this transformation is illustrated in Figure 25.

The following conditions must hold before the transformation may be applied:

1. The destination context must either be a procedure or module context because only declarations may be moved (line 2 in Figure 25). This is a limitation of the current implementation and may be extended in future versions to facilitate data abstraction operations.
2. The node being moved must represent a constant, type, variable or procedure declaration (line 3 in Figure 25).
3. An entity may not be moved if its current context and destination context are the same. Allowing the transformation to proceed will have no effect on the program structure, but will introduce unnecessary work and decrease efficiency (line 4 in Figure 25).
4. Moving an entity may not introduce a duplicate identifier within the destination context because it will violate the semantic rules of *Oberon* and the new source code will be rejected by the *Oberon* compiler. For example, moving C (line 20 in

```
1.  MODULE M;  
2.  CONST  
3.    A = 1;  
4.    B = 2;  
5.    C = A+B;  
6.  
7.  PROCEDURE P1;  
8.    CONST  
9.      D = 10;  
10.  
11.  PROCEDURE P2;  
12.    CONST  
13.      D = 20;  
14.      E = D+5;  
15.    END P2;  
16.  END P1;  
17.  
18.  PROCEDURE P2;  
19.    CONST  
20.      C = 10;  
21.      D = 5;  
22.    END P2;  
23.  
24.  END M.
```

24.1

```
1.  MODULE M;  
2.  CONST  
3.    A = 1;  
4.    B = 2;  
5.    C = A+B;  
6.  
7.  PROCEDURE P1;  
8.    CONST  
9.      D = 10;  
10.     C = 10;  
11.  
12.  PROCEDURE P2;  
13.    CONST  
14.      D = 20;  
15.      E = D+5;  
16.    END P2;  
17.  END P1;  
18.  
19.  PROCEDURE P2;  
20.    CONST  
21.      D = 5;  
22.    END P2;  
23.  
24.  END M.
```

24.2

Figure 24: The scope of a declaration may be changed by moving the declaration out of its current context and into a new context. The constant declaration of *C* in procedure *P2* (left figure) has been moved to procedure *P1* (right figure).

Figure 24.1) from P2 to M is illegal because M already contains a declaration called C (line 5 in Figure 24.1). This condition is evaluated in line 5 in Figure 25.

5. Any dependencies  $v$  exhibits on other entities must still be satisfied from the destination context. If not, the transformation is aborted (line 7 in Figure 25).
6. The transformation must ensure that dependencies exhibited by other entities on  $v$  are still satisfied after  $v$  has been moved. This implies that *newcontext* must be in the context path of every entity that depends on  $v$ . For example, moving A (line 3 in Figure 24.1) to P1 is not legal. The constant declaration of C (line 5 in Figure 24.1) depends on A, but after the move operation, A will be declared within a deeper scope and the dependency of C can no longer be satisfied. This condition is evaluated in line 8 in Figure 25.
7. Moving  $v$  to *newcontext* may introduce false dependencies and is considered last because of the number of computations it requires. Unlike the  $\delta_{rename}$  transformation,  $\delta_{move}$  also examines false dependencies from a global perspective. A global perspective is required because an entity changing contexts can potentially affect all program scopes. The first part of this condition tests whether global false dependencies are introduced. The second part of this condition examines whether the local dependencies exhibited by  $v$  will be falsely satisfied in *newcontext*. Consider moving the constant declaration of E (line 15 in Figure 24.1) from its current context, P2, to procedure P1. The value of E will change from 25 to 15 because the local dependency of E on D can be satisfied in P1, except that it is a different constant also called D. This condition is evaluated by procedure *CheckCurrent* in line 9 in Figure 25.

Moving a procedure declaration complicates the last three conditions. In this case, every constant, variable, type and procedure declaration, as well as every parameter, the result type (in case  $v$  is a function procedure) and every statement within  $v$  must be examined to determine whether any dependencies will become invalid. The overhead for such a transformation is substantial, but offers the user an efficient way to either localise or globalise procedures and functions using a single operation. This transformation is supported by NORT (line 11 in Figure 25). If all the conditions are satisfied, then  $v$  may be moved to *newcontext* (lines 13 through 29 in Figure 25). Appendix B describes the various sub-algorithms used by  $\delta_{move}$  in Figure 25.

```

1 proc Move(v, newcontext) ≡
2   if newcontext(type) ∉ {ctxModule, ctxProcedure} then ERROR fi;
3   if v(type) ∈ {entConstant, entType, entVariable, ctxProcedure} then ERROR fi;
4   if v(context) = newcontext then ERROR fi;
5   if LocalNode(newcontext, v(name)) ≠ NIL then ERROR fi;
6   if v(type) ≠ ctxProcedure then
7     CheckLocal(v, newcontext);
8     CheckGlobal(v, newcontext);
9     CheckCurrent(v, newcontext);
10  else
11    CheckAll(v, newcontext);
12  fi;
13  if v(type) = entConstant then
14    RemoveNode(v, v(context)(constantentity));
15    v(next) ← NIL;
16    InsertNode(newcontext(constantentity), v);
17  elseif v(type) = entType then
18    RemoveNode(v, v(context)(typeentity));
19    v(next) ← NIL;
20    InsertNode(newcontext(typeentity), v);
21  elseif v(type) = entVariable then
22    RemoveNode(v, v(context)(variableentity));
23    v(next) ← NIL;
24    InsertNode(newcontext(variableentity), v);
25  elseif v(type) = ctxProcedure then
26    RemoveNode(v, v(context)(procedurecontext));
27    v(next) ← NIL;
28    InsertNode(newcontext(procedurecontext), v);
29  fi;
30 end

```

Figure 25: Algorithm for the  $\delta move$  transformation.

#### 4.2.4 The CASE-IF Transformation

The  $\delta_{caseif}(s)$  transformation is a syntactic transformation that transforms a CASE statement into an IF statement. The only condition that must be satisfied is  $s(type) = stmtCase$ . Usually IF statements are converted into CASE statements. However, it is sometimes desirable to convert CASE statements to IF statements. For instance, a CASE statement that only contains a few cases will probably result in better executable code if it is re-written as an IF statement. Although most compilers should be able to recognise this, making changes to the structure can perhaps present a more pleasing look.

The transformation process is based on rewriting the case labels into expressions and combining them with the CASE expression to form the expressions for the IF and ELSIF clauses. The expression generated from the first case becomes the expression of the IF clause. The expressions generated from the remaining case labels are assigned to ELSIF clauses. The ELSE clause of the CASE statement becomes the ELSE clause of the IF statement. The structure of the case labels determines the structure of the expressions in the IF statement. A case label may assume one of three distinct structures in *Oberon*:

1. Single case labels. Given a case expression  $x$  and a single label  $l$ , the expression for an IF or ELSIF clause becomes  $x = l$ .
2. Comma separated labels. Given a case expression  $x$  and a sequence consisting of  $n$  comma separated labels  $l_1, l_2, \dots, l_n$ , the transformed expression becomes  $(x = l_1) OR (x = l_2) OR \dots OR (x = l_n)$ .
3. Ranges. Given a case expression  $x$  and a range bounded by two labels  $l_1..l_2$ , the new expression will become  $(x \geq l_1) \& (x \leq l_2)$ .

Figure 26 contains a CASE statement and an equivalent IF statement. The CASE statement contains an example of each label structure to illustrate the process of expression transformation.

#### 4.2.5 The IF-CASE Transformation

The  $\delta_{ifcase}(s)$  transformation is a syntactic transformation that transforms an IF statement into a CASE statement. This operation is more complex than the CASE-to-IF

```

1.  CASE i*2 OF
2.    0: ProcedureA |
3.    1, 3, 4: ProcedureB |
4.    5..8, 10: ProcedureC
5.  ELSE ProcedureD END
6.
7.  IF (i*2 = 0) THEN ProcedureA
8.  ELSIF (i*2 = 1) OR (i*2 = 3) OR (i*2 = 4) THEN ProcedureB
9.  ELSIF ((i*2 >= 5) & (i*2 <= 8)) OR (i*2 = 10) THEN ProcedureC
10. ELSE ProcedureD END

```

Figure 26: Example of an *Oberon* CASE statement. The IF statement is equivalent to the CASE and is the result of applying the  $\delta_{caseif}$  transformation to the CASE statement.

transformation. Any CASE statement may be re-written as an IF statement. However, not every IF statement can be re-written as a CASE statement.

The complex nature of the transformation warrants a more detailed discussion to convey the general ideas behind the operation before presenting the conditions that must be satisfied. Successfully converting an IF into a CASE requires that one of the following three expression structures be identified within every expression in both the IF and ELSIF clauses:

- Single equality test, for example  $x = l$ .
- Multiple equality tests separated by OR operators, for example  $(x = l_1) OR (x = l_2) OR \dots OR (x = l_n)$ .
- Expressions bounded by  $\geq$  and  $\leq$ , separated by  $\&$ , for example  $(x \geq l_1) \& (x \leq l_2)$ .

Any expression that fails to meet these requirements will result in the termination of the transformation. Once validated, the expressions of the IF and ELSIF clauses are broken into sub-expressions. These sub-expressions are used to identify a common sub-expression that will become the case expression. The remaining sub-expressions are used to form the labels of the CASE statement. The structure of the case labels is based on the operators associated with the root of each sub-expression.

The following conditions must hold for the transformation to be valid:

1. The statement selected for conversion must be an IF statement. Therefore,  $s(type) = stmtIf$ .

```

1. PROCEDURE ExpressionStructureValid(expression: NCEG.ExpressionNode): BOOLEAN;
2. BEGIN
3.   IF expression # NIL THEN
4.     IF expression.Type = NCEG.exprEqual THEN RETURN TRUE
5.     ELSIF expression.Type = NCEG.exprOr THEN
6.       RETURN ExpressionStructureValid(expression.Left) &
7.         ExpressionStructureValid(expression.Right)
8.     ELSIF expression.Type = NCEG.exprAnd THEN
9.       IF (expression.Left # NIL) & (expression.Right # NIL) THEN
10.        RETURN (expression.Left.Type = NCEG.exprGreaterEqual) &
11.          (expression.Right.Type = NCEG.exprLessEqual)
12.       ELSE RETURN FALSE END
13.     ELSE RETURN FALSE END
14.   ELSE RETURN FALSE END
15. END ExpressionStructureValid;

```

Figure 27: *Oberon* implementation responsible to determine if the structure of an expression meets the criteria when transforming an IF statement into a CASE statement.

2. The expression structure of the IF clause is examined to determine if it conforms to one of the three structures previously mentioned.
3. The expression of every ELSIF clause must also be examined. Again, the transformation will terminate on the first expression that fails to meet the structural criteria.

The *ExpressionStructureValid* function is used to evaluate the structure of every expression in the IF and ELSIF clauses. This evaluation is based on the structural forms identified in Section 4.2.4. The *Oberon* code used to evaluate the structure of an expression is illustrated in Figure 27.

The function in Figure 27 takes the root of an expression tree and returns TRUE if the structure is valid, and returns FALSE otherwise. The expression is deemed valid if the node type of the root denotes the = operator (line 4). If the root of the expression contains the OR operator (line 5) then the expression is valid only if both the left and right sub-expressions are valid (line 6 and 7). If the root contains the & (AND) operator (line 8), then the expression is valid only if there exists a left and right child (line 9) and the root of the left sub-expression contains the >= operator and the root of the right sub-expression contains the <= operator (lines 10 and 11).

If the structural test is passed, the expressions contained in the IF statement are broken down into sub-expressions by the procedure in Figure 28 and placed in *decomposition*

```

1. PROCEDURE BreakExpression(expression: NCEG.ExpressionNode;
2.     VAR exprlist: NCEG.ExpressionList);
3. BEGIN
4.   IF expression # NIL THEN
5.     IF (expression.Type = NCEG.exprEqual) OR
6.       (expression.Type = NCEG.exprLessEqual) OR
7.       (expression.Type = NCEG.exprGreaterEqual) THEN
8.       NCEG.InsertExpressionList(exprlist, NCEG.CreateExpressionList(expression))
9.     ELSIF (expression.Type = NCEG.exprOr) OR (expression.Type = NCEG.exprAnd) THEN
10.      BreakExpression(expression.Left, exprList);
11.      BreakExpression(expression.Right, exprList)
12.    END
13.  END
14. END BreakExpression;

```

Figure 28: *Oberon* implementation for the `BreakExpression` procedure that is responsible for breaking an expression into a linked list of sub-expressions.

```

1. BreakExpression(this.Expression, IFexprlist);
2. InsertDecomposition(decomp, CreateDecomposition(IFexprlist, this.Body));
3. node := this.Elsif;
4. WHILE (node # NIL) DO
5.   ELSIFexprlist := NIL;
6.   BreakExpression(node.Expression, ELSIFexprlist);
7.   InsertDecomposition(decomp, CreateDecomposition(ELSIFexprlist, node.Body));
8.   node := node.Elsif
9. END;

```

Figure 29: The expressions contained in the IF statement are broken down into sub-expressions. These expressions are combined with the statement sequence associated with each clause to form decomposition nodes.

nodes along with the original statement sequence associated with the corresponding clause in the IF statement. The decomposition nodes are then used to construct the labels of the CASE statement.

The transformation process will be described with references to its implementation in *Oberon*. Figure 29 shows how the expressions are broken down and the decomposition nodes created. The IF clause is broken down (line 1) after which the decomposition node is created (line 2). Next, every ELSIF clause is examined, breaking its expression into a number of sub-expressions and creating a decomposition node for the clause (lines 3 through 9).

The transformation then proceeds to construct the case statement as illustrated in

```

1. CaseExpression := decomp.ExprList.Expression.Left;
2. decompList := decomp;
3. WHILE decompList # NIL DO
4.   exprList := decompList.ExprList;
5.   WHILE exprList # NIL DO
6.     IF ~NCEG.ExpressionEqual(CaseExpression, exprList.Expression.Left) THEN
7.       Result := NE.errIfCaseCommonExpr;
8.       RETURN
9.     END;
10.    exprList := exprList.Next
11.  END;
12.  decompList := decompList.Next
13. END;

```

Figure 30: The decomposition nodes are examined to determine if a common sub-expression exists. If so, it will become the case expression.

Figure 30. The left sub-expression of the first decomposition node becomes the case expression (line 1). This expression is compared with the left sub-expression of every decomposition node to determine if it is a common sub-expression (lines 2 through 13). The transformation exits if the test for equality fails (lines 6 through 9). The `ExpressionEqual` function is very conservative and will not attempt to transform an expression to determine if it is equivalent to another. For example, the expression  $a+1 = b$  is equivalent to  $b-1 = a$ , but `ExpressionEqual` is not capable of recognizing this.

A possible future extension to the `ExpressionEqual` function may include the conversion of expressions into tripples to allow sub-expression elimination and simple algebraic transformations to test for equivalence and regroup non-constant entities to form the case expression.

The final step in the transformation involves the construction of the `CASE` statement node and its labels. The labels, as stated earlier, are created from the decomposition nodes. The structure of the `CASE` labels are based on the operators denoted by the root of every sub-expression in each of the decomposition nodes. Finally, the `IF` statement is removed and the `CASE` statement inserted into the CEG.

### 4.2.6 The Exchange Statement Transformation

The  $\delta_{exchange}(s_1, s_2)$  function is a control flow transformation and will attempt to exchange statement  $s_1$  with  $s_2$ . This transformation is based on the `move_statement` operation found in `CStructure` [29]. However, there are two important differences between these two transformations. First,  $\delta_{exchange}$  does not perform any alias analysis, so transformations involving pointers are not considered. Second, whereas `move_statement` altered the position of a single statement,  $\delta_{exchange}$  exchanges two statements and therefore constitutes two move operations. The transformation provides a more efficient way in which to reorganise code, but incurs a penalty in terms of overhead due to the additional data analysis required.

Control flow transformations can have far reaching effects on a program. Control and data flow analysis is used to determine if it will affect the program's behaviour. The transformation makes use of flow-insensitive data analysis using definition and use sets. This results in a more conservative transformation than would otherwise be possible when performing flow sensitive data analysis. A number of concepts used during analysis is defined before examining the conditions imposed by the  $\delta_{exchange}$  transformation.

- A *definition* of a variable  $x$  is a statement that assigns (or may assign) a value to  $x$  [1]. Definitions may also be the result of ambiguous statements where a variable is defined due to a pointer reference or through a call-by-reference parameter. For example, the statement  $x := y+z$  is said to *define*  $x$  and *use*  $y$  and  $z$ .
- A *program component* is any node in the CEG, and refers to its entire sub-tree as a unit [29].
- Component  $A$  is *flow dependent* on component  $B$  if a definition in  $A$  is later used in  $B$  [7, 29]. Consider two consecutive statements  $x := y$  and  $z := x$ . A flow dependence exists between these statements because the second statement depends on the definition of  $x$  in the first.
- An *anti-dependence* exists between two components  $A$  and  $B$  if exchanging  $A$  and  $B$  would result in the creation of a flow dependence [7, 28, 29]. The two consecutive statements  $z := x$  and  $x := y$  exhibits an anti-dependence.
- An *output dependence* exists between two components  $A$  and  $B$  if both  $A$  and  $B$  contain a definition of the same variable [7, 28, 29]. An example of an output

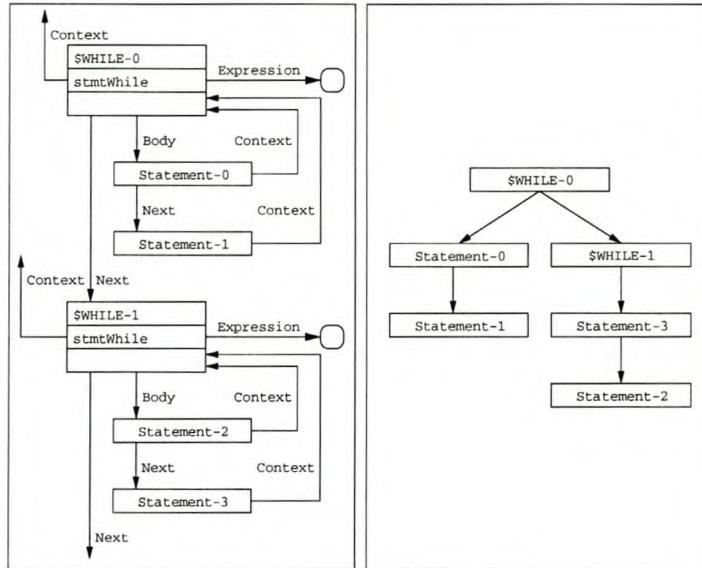


Figure 31: The left side of the figure shows two WHILE statements, each containing two statements. The right side shows the dominator tree for the two statements. Exchanging **Statement-1** with **Statement-2** would change their control dominators.

dependence would be the two statements  $x := y$  and  $x := z$ .

- As mentioned in Section 3.2.8, the context edge of statement entities may be used to represent characteristics of a control flow graph (CFG). A statement context may also act as a control dominator in a limited capacity. A node  $v$  is said to be dominated by a node  $w$  if every path from the initial node in the flow graph to  $v$  goes through  $w$  [1]. The node in the CEG describing the statement context becomes the dominator of the statements it contains. This concept is illustrated in Figure 31.
- The definition set of a node is constructed by the  $DEF(v)$  function. For example, given the statement  $x := a+b+c$ , the definition for the assignment would be  $\{x\}$ . The use set of a node is constructed by the  $USE(v)$  function. For example, given the statement  $x := a+b+c$ , the use set for the assignment would be  $\{a, b, c\}$ .

The following conditions must be examined before the transformation may be applied:

1. RETURN and EXIT statements may not be exchanged because it will alter the control flow and prevent the execution of the other statements being exchanged

(lines 2, 3 and 4 in Figure 32). This condition is strengthened later to ensure that the statements being exchanged have a single entry and exit point.

2. Statements located in different control regions may not be exchanged because it will alter the conditions under which they execute by changing the control dominator of the statements (line 5 in Figure 32).
3. Compound statements containing EXIT or RETURN statements may not be exchanged because it will also effect the control dominators of the statements being exchanged. Both  $s_1$  and  $s_2$  must have a single entry and single exit point (line 6 in Figure 32).
4. The previous conditions only examined control dependencies. The data dependencies must now be tested. Three dependency checks are required to determine the influence of re-ordering the statements. The transformation is aborted if either a flow, output or anti-dependence is detected (lines 7, 8 and 9 in Figure 32).
5. As stated earlier, the  $\delta_{exchange}$  transformation effectively moves two statements. It is possible that  $s_1$  and  $s_2$  are not adjacent. In this case the statement sequence between  $s_1$  and  $s_2$  must also be examined to determine if a flow, output or anti-dependence exists (lines 10 through 19 in Figure 32).

If these conditions are met, then  $s_1$  and  $s_2$  may be exchanged.

#### 4.2.7 The Create Procedure Transformation

The  $\delta_{createprocedure}(S, local, procname)$  function may be used to replace a number of consecutive statements by a procedure call and is defined as an abstraction transformation. The transformation replaces a selected sequence  $S$ , consisting of  $n$  consecutive statements, with a call to procedure  $proc$ . If  $S$  is embedded within a procedure context and  $local = \text{TRUE}$ , then  $proc$  will be created locally within this procedure context. Otherwise,  $proc$  is created globally within the module context.

The new procedure will be called  $procname$  and contain the statement sequence  $S$ . The transformation automatically determines if the parameters of  $proc$  require to be declared as reference or value parameters. The individual statements in  $S$  are denoted as  $s_1, s_2, \dots, s_n$ . The algorithm for this transformation is presented in Figure 33. The following conditions must hold to ensure that the transformations will preserve the meaning of the program:

```

1 proc ExchangeStatement( $s_1, s_2$ )  $\equiv$ 
2   if ( $s_1 \notin S$ )  $\vee$  ( $s_2 \notin S$ ) then ERROR  $\underline{f}$ ;
3   if ( $s_1(\text{type}) \in \{\text{stmtExit}, \text{stmtReturn}\}$ )  $\vee$ 
4     ( $s_2(\text{type}) \in \{\text{stmtExit}, \text{stmtReturn}\}$ ) then ERROR  $\underline{f}$ ;
5   if  $s_1(\text{context}) \neq s_2(\text{context})$  then ERROR  $\underline{f}$ ;
6   if  $\neg(\text{SingleEntryExit}(s_1)) \wedge (\text{SingleEntryExit}(s_2))$  then ERROR  $\underline{f}$ ;
7   if  $USE(s_2) \cap DEF(s_1) \neq \emptyset$  then ERROR  $\underline{f}$ ;           flow dependence
8   if  $USE(s_1) \cap DEF(s_2) \neq \emptyset$  then ERROR  $\underline{f}$ ;           anti-dependence
9   if  $DEF(s_1) \cap DEF(s_2) \neq \emptyset$  then ERROR  $\underline{f}$ ;           output dependence
10   $s \leftarrow s_1(\text{next})$ ;
11  while  $s \neq s_2$  do
12    if  $USE(s) \cap DEF(s_1) \neq \emptyset$  then ERROR  $\underline{f}$ ;           flow dependence
13    if  $USE(s) \cap DEF(s_2) \neq \emptyset$  then ERROR  $\underline{f}$ ;           anti-dependence
14    if  $USE(s_1) \cap DEF(s) \neq \emptyset$  then ERROR  $\underline{f}$ ;           anti-dependence
15    if  $USE(s_2) \cap DEF(s) \neq \emptyset$  then ERROR  $\underline{f}$ ;           flow dependence
16    if  $DEF(s) \cap DEF(s_1) \neq \emptyset$  then ERROR  $\underline{f}$ ;           output dependence
17    if  $DEF(s) \cap DEF(s_2) \neq \emptyset$  then ERROR  $\underline{f}$ ;           output dependence
18     $s \leftarrow s(\text{next})$ 
19  od;
20 end

```

Figure 32: Algorithm for the  $\delta_{exchange}$  transformation.

1. The new name must be a valid identifier according to the EBNF definition for `ident` to ensure that the compiler will accept the procedure declaration (line 2 in Figure 33).
2. The entities in the sequence must all be valid statements and share the same context. This does not imply that the selected statements reside in the same block. However, it does ensure that the selection is taken from either a single compound statement or procedure body (lines 3 through 8 in Figure 33).
3. The context, *procontext*, for the new procedure, *proc*, must now be determined, based on whether the procedure must be created locally (nested) within a procedure or globally within the module (lines 9 through 18 in Figure 33).
4. The destination context must now be examined to determine if an entity called *procname* already exists. If so, the transformation is illegal because it would create a duplicate identifier (line 19 in Figure 33).
5. The statement sequence may contain references to other procedures or variables called *procname*. If this is the case, then the transformation will invalidate these dependencies (lines 20 through 24 in Figure 33).
6. The next step involves examining every procedure call within *S* to determine whether it can be reached from the new context, *procontext* (lines 25 through 29

in Figure 33). At this point an empty procedure context, *proc*, called *procname* may be created so that  $proc(context) = procontext$  (line 30 in Figure 33).

7. Type dependencies of variable and type declarations, as well as constant declarations, must be reachable from the new context. Variable declarations are not examined because they will become the parameters of the procedure call. This condition is examined in lines 31, 32 and 33 of Figure 33.

The transformation may now be applied as follows:

1. The variables shared between the use and definition sets of *S* are removed from *use* to avoid duplicating parameters (line 34 in Figure 33).
2. Global variables and constants are removed from *use* and *def* to limit the number of parameters passed to the new procedure (lines 35 and 36 in Figure 33).
3. A node representing the procedure call, *proccall*, must now be created along with the necessary formal and actual parameters. The parameters are obtained from *def* and *use*. If a variable denotes a structured data type, then the variable is passed as a parameter instead of an individual field or index of the variable (lines 37 through 40 in Figure 33).
4. Move the statements in *S* into *proc* and insert *proccall* in the predecessor of  $s_1$  (lines 41, 42 and 43 in Figure 33).
5. Insert *proc* into *procontext* (lines 44 in Figure 33).

### 4.3 Transformation Examples

Three examples are examined in this section to illustrate the detail of the transformations discussed in Section 4.2.

#### 4.3.1 Example: Syntactic Transformation

An example of a syntactic transformation is illustrated using the  $\delta_{remove}$  operation. Figure 34 contains an *Oberon* module that requires restructuring, the goal being the removal of the constant declaration of **C** in line 6. The conditions that apply to this transformation, outlined in Section 4.2.2, must now be examined:

```

1 proc CreateProcedure( $S$ , local, procname)  $\equiv$ 
2 if  $\neg$ ValidName(newname) then ERROR fi;
3  $s \leftarrow s_1$ ;
4 while ( $s \neq NIL$ )  $\wedge$  ( $s(context) = s_1(context)$ )  $\wedge$ 
5 ( $s(type) \in S$ ) do
6  $s \leftarrow s(next)$ 
7 od;
8 if  $s \neq NIL$  then ERROR fi;
9 procontext  $\leftarrow s_1(context)$ ;
10 if local then
11 while procontext(type)  $\neq$  cntxModule do
12 procontext  $\leftarrow$  procontext(context)
13 od
14 else
15 while procontext(type)  $\notin$  {cntxModule, cntxProcedure} do
16 procontext  $\leftarrow$  procontext(context)
17 od
18 fi;
19 if LocalNode(procontext, procname)  $\neq$  NIL then ERROR fi;
20  $s \leftarrow s_1$ ;
21 while  $s \neq NIL$  do
22 if NewControlDep( $s$ , procname) then ERROR fi;
23  $s \leftarrow s(next)$ 
24 od;
25  $s \leftarrow s_1$ ;
26 while  $s \neq NIL$  do
27 if  $\neg$ ProcCallValid( $s$ , ContextPath(procontext)) then ERROR fi;
28  $s \leftarrow s(next)$ 
29 od;
30 proc  $\leftarrow$  CreateProcedure(procontext, procname);
31 use  $\leftarrow$  USE( $S$ ); def  $\leftarrow$  DEF( $S$ );
32 CheckDependenciesProc(use, ContextPath(proc));
33 CheckDependenciesProc(def, ContextPath(proc));
34 use  $\leftarrow$  use  $\setminus$  def;
35 RemoveGlobals(use, ContextPath(procontext));
36 RemoveGlobals(def, ContextPath(procontext));
37 proccall  $\leftarrow$  NewNode(stmtProcedureCall,  $s_1(context)$ );
38 proccall(procedurecall)  $\leftarrow$  proc;
39 CreateFormal(proc, def, use);
40 CreateActual(proccall, def, use);
41 pre  $\leftarrow$  Predecessor( $s_1$ );
42 MoveStatements( $S$ , proc(body));
43 InsertNode(proccall, pre);
44 InsertNode(procontext(procedurecontext), proc)
45 end

```

Figure 33: Algorithm for the  $\delta_{createprocedure}$  transformation.

```

1. MODULE Example1;
2.
3. CONST
4.   A = 1;
5.   B = A;
6.   C = A+B;
7. END Example1.

```

Figure 34: Original module that awaits restructuring.

```

1. MODULE Example1;
2.
3. CONST
4.   A = 1;
5.   B = A;
6. END Example1.

```

Figure 35: The `Example1` module after removing the constant declaration of `C` in line 6 of Figure 34 by applying the  $\delta_{remove}$  transformation.

1.  $C(\text{export}) = \text{FALSE}$ . This condition holds because `C` is not exported.
2.  $C(\text{dependencyset}) = \emptyset$  because no entity in `Example1` references `C`.
3.  $C(\text{type}) = \text{entConstant}$ . Only certain types of entities may be deleted, including constant declarations.

New source code will be generated once the node representing the declaration is removed. The source code for the new module is presented in Figure 35.

The  $\delta_{remove}$  transformation also illustrates the dependency relationship that sometimes exists between restructuring operations. Again, consider the module presented in Figure 34, but assume that the restructuring goal was the removal of `A` (line 4). Conditions 1 and 3 still hold, but not condition 2. The transformation is rejected because  $A(\text{dependencyset}) = \{B, C\}$ . Constant `A` can only be removed if `C` and `B` are removed first to ensure that the dependency set of `A` is empty. Transformation dependencies, and how to deal with them, are briefly discussed in Section 4.4.

### 4.3.2 Example: Control Flow Transformation

An example of a control flow transformation is presented using the  $\delta_{exchange}$  transformation. Figure 36 contains an *Oberon* module that must be restructured by exchanging the first `WHILE` statement (`$WHILE-0`) with the second (`$WHILE-1`). The conditions described in Section 4.2.6 must now be examined.

1. Neither `$WHILE-0` nor `$WHILE-1` are `EXIT` or `RETURN` statements.
2. Both `WHILE` statements share the same context `P1` and fall under the same control region.
3. Neither `$WHILE-0` nor `$WHILE-1` contain `EXIT` or `RETURN` statements, satisfying the condition that statements must have a single entry and exit point.
4. The definition and use sets for `$WHILE-0` and `$WHILE-1` must now be computed.  $USE(\$WHILE-0) = \{x\}$ ,  $DEF(\$WHILE-0) = \{x\}$ ,  $USE(\$WHILE-1) = \{y, z\}$  and  $DEF(\$WHILE-1) = \{y, z\}$ .
  - $USE(\$WHILE-1) \cap DEF(\$WHILE-0) = \emptyset$  so no flow dependence exists
  - $USE(\$WHILE-0) \cap DEF(\$WHILE-1) = \emptyset$  so no anti-dependence exists
  - $DEF(\$WHILE-0) \cap DEF(\$WHILE-1) = \emptyset$  so no output dependence exists
5. There is one statement between `$WHILE-0` and `$WHILE-1` that must be examined. Computing its use and definition sets yield  $USE(\$ASSIGN-4) = \{x\}$  and  $DEF(\$ASSIGN-4) = \{z\}$ .  $USE(\$ASSIGN-4) \cap DEF(\$WHILE-0) = \{x\}$  indicating that a flow dependence exists between `$WHILE-0` and `$ASSIGN-4`. The transformation will therefore be aborted.

### 4.3.3 Example: Abstraction Transformation

An example of an abstraction transformation is illustrated using the  $\delta_{createprocedure}$  operation. Figure 37 contains a module that awaits restructuring. The goal is to replace the `FOR` statement (lines 19 through 23) by a procedure call. The internal name assigned by `NORT` to this statement is `$FOR-0`. The new procedure will be called `InitializeUsers` and must be created within the module context. The conditions outlined in Section 4.2.7 must now be examined.

```
MODULE Example2;

CONST
  N = 10;

PROCEDURE P1;
VAR
  x, y, z: LONGINT;
BEGIN
  x := 0;          (* \$$ASSIGN-0 *)
  y := 0;          (* \$$ASSIGN-1 *)
  z := 0;          (* \$$ASSIGN-2 *)
  WHILE x < N DO (* \$$WHILE-0 *)
    x := x+1      (* \$$ASSIGN-3 *)
  END;
  z := x;          (* \$$ASSIGN-4 *)
  WHILE z > 0 DO (* \$$WHILE-1 *)
    y := y+1;     (* \$$ASSIGN-5 *)
    z := z-1      (* \$$ASSIGN-6 *)
  END
END P1;

END Example2.
```

Figure 36: An *Oberon* module that must be restructured by exchanging the two `WHILE` statements. The internal statement names generated by `NORT` have been included as comments.

1. `InitializeUsers` is a valid identifier according to *Oberon's* EBNF definition.
2.  $\$FOR-0(type) = stmtFor$  and is a valid selection. The second part of the condition tests whether all the statements in  $S$  share the same context and is satisfied because there is only one statement in  $S$ .
3. It was assumed that the user elected to place the new procedure within the module context ( $local = FALSE$ ).  $procontext \leftarrow \$FOR-0(context)$  so  $procontext = CreateDatabase$ . After  $procontext$  is adjusted it points to the module context, `Example3`.
4. The search to locate an entity called `InitializeUsers` is carried out over all the edges specified in `Example3(search)`. However, such a node does not exist, and the transformation will continue to the next condition.
5. The reference dependency set of  $\$FOR-0$  is computed and includes the dependencies obtained from the three assignments nested inside  $\$FOR-0$  (lines 20, 21 and 22).  $ReferenceDependency(\$FOR-0) = \{i, MaxUsers, Database.Name, Database.Age, Database.ID\}$ . The transformation continues to the next condition because none of the reference dependencies are called `InitializeUsers`.
6. There are no procedure calls within  $S$ . An empty procedure context  $proc$  is created.  $proc(name) \leftarrow "InitializeUsers"$  and  $proc(context) \leftarrow Example3$ .
7.  $use = \{i, MaxUsers\}$  and  $def = \{i, Database.Name, Database.Age, Database.ID\}$ . All the record fields exhibit a type dependency on `LONGINT`. The `LONGINT` entity is declared in the predefined context and can be reached from the new procedure. The `MaxUsers` constant can also be reached because it was declared globally within the module.

The transformation may now be performed:

1.  $use = \{i, MaxUsers\}$  and  $def = \{i, Database.Name, Database.Age, Database.ID\}$ . Since  $i$  is an element of both  $use$  and  $def$ , it may be removed ( $use = \{MaxUsers\}$ ).
2. `MaxUsers` is a global constant, reachable from procedure `InitializeUsers`, and may therefore be removed ( $use = \emptyset$ ).
3. A node,  $proccall$ , representing a procedure call to `InitializeUsers` is created. The formal parameters of  $proc$  and actual parameters for  $proccall$  must now be

```

1.  MODULE Example3;
2.
3.  CONST
4.    MaxUsers = 100;
5.
6.  TYPE
7.    UserDesc = RECORD
8.      Name: ARRAY 32 OF CHAR;
9.      Age, ID: LONGINT
10.    END;
11.
12.   UserList = ARRAY MaxUsers OF UserDesc;
13.
14.  PROCEDURE CreateDatabase*;
15.  VAR
16.    i: LONGINT;
17.    Database: UserList;
18.  BEGIN
19.    FOR i := 0 TO MaxUsers-1 DO
20.      Database[i].Name[0] := 0X;
21.      Database[i].Age := 0;
22.      Database[i].ID := 0;
23.    END;
24.    (* ... remaining code of procedure ... *)
25.  END CreateDatabase;
26.
27.  END Example3.

```

Figure 37: *Oberon* module that awaits restructuring. The goal is to replace the FOR statement by a procedure call.

created. Since *use* is empty, only *def* will be considered. The first parameter added to both *proc* and *proccall* is *i*. The next parameter is *Database* (recall that the transformation removes the qualified fields when working with structured data types). The elements remaining in *def* all qualify *Database* and will not be added to the parameter lists.

4. \$FOR-0 is removed from its original context and placed within the body of procedure *InitializeUsers*, while *proccall* is inserted into the body of procedure *CreateDatabase*.
5. The procedure context, *proc*, is inserted into *Example3(procedurecontext)*. The restructured module is presented in Figure 38.

```
1.  MODULE Example3;
2.
3.  CONST
4.    MaxUsers = 100;
5.
6.  TYPE
7.    UserDesc = RECORD
8.      Name: ARRAY 32 OF CHAR;
9.      Age, ID: LONGINT
10.    END;
11.
12.    UserList = ARRAY MaxUsers OF UserDesc;
13.
14.  PROCEDURE InitializeUsers(VAR i: LONGINT; VAR Database: UserList);
15.  BEGIN
16.    FOR i := 0 TO MaxUsers-1 DO
17.      Database[i].Name[0] := 0X;
18.      Database[i].Age := 0;
19.      Database[i].ID := 0
20.    END
21.  END InitializeUsers;
22.
23.  PROCEDURE CreateDatabase*;
24.  VAR
25.    i: LONGINT;
26.    Database: UserList;
27.  BEGIN
28.    InitializeUsers(i, Database);
29.    (* ... remaining code of procedure ... *)
30.  END CreateDatabase;
31.
32.  END Example3.
```

Figure 38: Module `Example3` after replacing the original `FOR` statement in procedure `CreateDatabase` by the procedure call `InitializeUsers`. The elimination of parameters results in an almost optimal solution. `i` could have been passed as a value parameter without affecting the program's behaviour, but since it is modified by the `FOR` loop it is treated as a reference parameter.

## 4.4 Transformation Dependencies

Dependency relationships are not limited to the entities within a program. Restructuring operations may also exhibit this characteristic. The example presented in Section 4.3.1 showed how the removal of a constant can depend on the prior removal of other constants.

Roberts defined a method whereby a chain of refactorings (transformations) could be re-ordered according to their inter-dependencies and separated into sub-chains. By identifying those transformations within a chain that can commute<sup>1</sup> and repeatedly re-ordering them, separate chains can be generated that are independent of each other, thereby simplifying complex restructuring operations [37].

## 4.5 Updating Dependency Information within the CEG

Transformations not only alter the structure of a program, but also influence the dependencies that exist between nodes within the program representation structure. Using a single program representation structure such as the CEG also requires *post-transformation* synchronisation. For example, removing a declaration of a variable may leave unwanted references to the declaration in the dependency sets of other nodes within the CEG.

One method to address this problem would be to synchronise the CEG after every transformation by updating all the affected dependency sets to correctly reflect the current state of the program. However, this operation may prove to be expensive, depending on the nature of the restructuring operation. In the worst case, the process of updating dependencies will require a complete traversal of the CEG to locate all the affected nodes. Furthermore, the operations to update a specific dependency may entail various actions on the individual nodes to remove and re-connect certain vertices, operations which themselves may be expensive.

A simple, and less computationally intensive solution, was implemented to address this problem. Once a user elects to accept a transformation, the restructured code is parsed to produce a new CEG. Not only does this solve the problem, but also guarantees that all the dependencies are correct.

---

<sup>1</sup>As a rule of thumb, two transformations do not commute if either one creates the conditions on which the other depends.

## 4.6 Summary

Restructuring a program by simply applying transformations is not sufficient. Transformations must preserve the meaning of the program being restructured. This chapter introduced a notation to abstract and define the transformation functions implemented in NORT. The transformations were classified and various conditions were presented to ensure that meaning is preserved during restructuring operations. Examples were also presented to illustrate the function of the various conditions during the application of transformations.

## Chapter 5

# Evaluation and Conclusion

The primary goal of this thesis was to design and implement a prototype tool to restructure *Oberon* source code. A number of secondary objectives were also defined at the outset of this project, as described in Section 1.1.

Results from the implementation and various trial runs of the prototype are discussed in Section 5.1. The environment, specifically the user interface, is discussed in Section 5.2. The supported transformations are evaluated in Section 5.3, followed by a discussion of the CEG and possible future work in Sections 5.4 and 5.5. Concluding remarks are presented in Section 5.6.

### 5.1 Implementation and Results

The *Oberon* system rests upon two principles: *simplicity* and *clarity* [45]. It is important to keep the principles of *Oberon* in mind when reviewing the results of NORT and comparing it with the results from other systems because of the economics of scale involved. In Unix, for instance, the gcc compiler<sup>1</sup> constitutes a fairly large application spanning more than 120 files, totalling more than 300,000 lines<sup>2</sup> of code. In comparison, the entire *Oberon* system<sup>3</sup> is composed of 250 modules and represents less than 200,000 lines<sup>4</sup> of code.

---

<sup>1</sup>Information obtained from gcc, release 2.95.1.

<sup>2</sup>Line count based only on \*.c files and includes comments.

<sup>3</sup>The system described here is based on the PC Native 31.07.2000 release and includes the kernel, device drivers, network support, compiler and Gadgets framework. No other utilities or applications were included.

<sup>4</sup>Line count based only on \*.Mod files and includes comments.

Module	Function	Lines	Executable (bytes)
NORTDefs	Global variables, etc.	95	47
NORTDebug	Debugging routines	102	1848
NORTErrors	Error reporting	209	2803
NORTCEG	Manipulation of CEG nodes	1209	16291
NORTOM	Object Model parser	555	12351
NORTGadgets	General Gadgets manipulation	166	2380
NORTCEGgadgets	CEG Gadgets implementation	147	2305
NORTSourceGen	Source code generator	542	11591
NORTScanner	Scanner	279	5445
NORTParser	Parser	791	11648
NORTTransformations	Transformation functions	1002	13544
NORT	Application front-end	625	9063
Total		5722	89316

Table 2: Breakdown of implementation into modules.

Table 2 presents an overview of the various modules used by NORT while Figure 39 illustrates the implementation structure of NORT.

Unlike traditional systems where the size of an application can easily end up totalling tens or even hundreds of thousands of lines of code, modules in *Oberon* often only reach a few hundred, or sometimes thousand, lines of code. This leads to a considerable reduction in the amount of resources required to restructure a module. Table 3 contains the results of various modules used to evaluate NORT. Most important is the memory requirements of the CEG structure shown in the last column.

Module	Function	Lines	Statements	Procedures	CEG size
Kernel	Oberon kernel	2663	1815	144	1.65 MB
Disks	Block devices interface	320	169	51	0.23 MB
ATADisks	ATA disk driver	1275	1038	118	1.06 MB
OPO	Compiler back-end	850	559	103	0.58 MB
NORTCEG	Manipulation of CEG nodes	1209	791	119	0.82 MB

Table 3: Breakdown showing memory requirements of the CEG for various modules.

A breakdown of the various node classes created during the construction of the CEG for a specific module is presented in Table 4. Node sizes are fixed, although some data generated by expression nodes may grow dynamically, as is the case when working with literate string constants. A CEG node requires 176 bytes, expression nodes 44 bytes (excluding string constants), while dependency and expression list nodes each require 8 bytes. The design of the various node classes is not optimal. Some improvements,

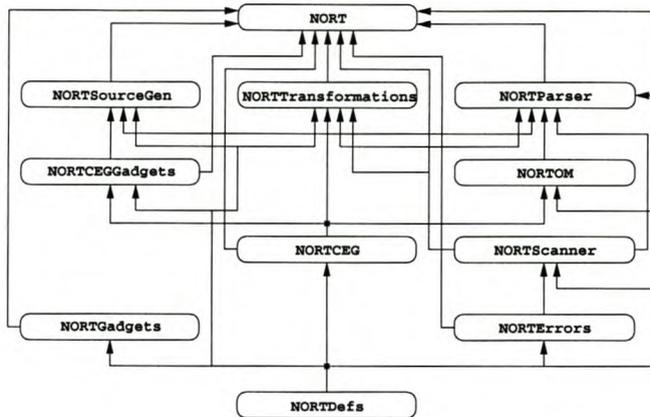


Figure 39: Implementation structure of NORT.

such as generalising the use of certain edges within the CEG node, can be adopted to reduce memory requirements even further.

The majority of CEG nodes generated represent designator nodes and on average accounted for 50% of the CEG nodes. In some cases, such as the `Kernel` module, the designator nodes accounted for 60% of the nodes (4938 out of 8177 nodes). Refinement of the designator nodes should dramatically affect the memory consumption of NORT. Designator nodes are currently implemented using the CEG node type. It should be possible to redesign the nodes so that the memory requirements of designator nodes are reduced to around 8 bytes per node (a single pointer to the CEG node representing the identifier being referenced and an additional field to create a linked list when representing qualified identifiers). This may complicate the implementation of some algorithms by introducing an additional node class, but can be justified given the gains in terms of memory requirements.

Module	CEG Nodes	Expression Nodes	Dependency Nodes	Expression Lists
<code>Kernel</code>	8177	5396	4868	2071
<code>Disks</code>	1204	554	693	194
<code>ATADisks</code>	5403	3128	2836	1243
<code>OPD</code>	2882	1794	1893	750
<code>NORTCEG</code>	4185	2177	2768	720

Table 4: Breakdown of nodes into the four major node classes for various modules.

## 5.2 User Interface

A restructuring tool should provide a comfortable environment for the user, allowing access to information required to plan and execute restructuring operations, while hiding unnecessary details [17, 23]. A number of user interfaces were considered during the development of the project, including the use of visual abstractions such as the *Star-Diagram* [6, 24]. After careful consideration it was decided to design an interface based on hypertext, a specialised form of *active text*. Active text extends traditional text by providing sensitive areas for user interaction and may encapsulate a variety of information [31]. Hypertext is often used to link separate but related pieces of information [31, 33]. In NORT, this link is created between specific syntactic components of the language in the source code and their representation within the CEG.

Source code, containing links to nodes within the CEG, is produced by NORT after parsing an *Oberon* module. The links are created by embedding visual objects into the text. These objects are based on Gadgets, a component framework for *Oberon* [16]. The components (objects) of the Gadget framework are also called gadgets and represent persistent objects that exist within the run-time environment of *Oberon* [16]. The gadgets embedded within the source code are visually similar to normal text and are distinguished based on their bold typeface. Figure 40 contains a view of the environment provided by NORT and shows the source code of a module containing active text.

Each gadget represents a node within the CEG that may be selected for a restructuring operation. The algorithm responsible for generating the source code from the CEG is also responsible for inserting links to connect each component with its corresponding node in the CEG. This framework presents a number of advantages:

- There is no need to visualise the graph structure of the CEG. The user manipulates it indirectly through the visual components embedded within the source code. The user associates restructuring operations with the programming language and not a complex graph structure.
- Search operations to locate vertices in the CEG corresponding to syntactic units are not required. Every component the user may select is directly linked to a specific vertex of the CEG.
- The user can inspect a specific component, should he desire to do so. Again, little

overhead is incurred because of the direct link that exists between an active text component and a vertex within the CEG.

## 5.3 Evaluation of Transformations

A number of transformations have been implemented in NORT and arguments pertaining to their correctness have been put forward. Whether these arguments are sufficient to ensure meaning preserving transformations during sustained use within a software development environment remains to be seen.

### 5.3.1 Syntactic Transformations

A number of syntactic transformations were implemented and it was illustrated that basic transformations could be facilitated by the CEG. In addition to these basic operations, more complex syntactic transformations such as  $\delta_{ifcase}$  were implemented to illustrate the potential of the CEG.

Although useful, the  $\delta_{ifcase}$  transformation is limited in identifying valid IF statements that may be converted into CASE statements. This can be attributed to the conservative algorithm used for the evaluation of expressions. Converting the expression tree to three address format before attempting sub-expression elimination should yield better results.

### 5.3.2 Control Flow Transformations

The implementation of the  $\delta_{exchange}$  transformation is based on the demand driven techniques used by CStructure [29] and, despite the lack of alias analysis, showed that the CEG is capable of supporting similar functionality. The transformation also provided an extension of the `move_statement` transformation, allowing the relocation of two statements simultaneously.

### 5.3.3 Abstraction Transformations

The  $\delta_{createprocedure}$  transformation showed how demand driven analysis, based on *definition* and *use* sets, can be applied to identify and limit the number of parameters

The screenshot shows the NORT application window with a menu bar (File Operations, Transform, Rename, Create Procedure, Misc, Options) and the title "NORT Native Oberon Restructuring To". The main area is divided into four sections:

- Source Text:** Contains the original source code for `Registerx` and `Unregisterx` procedures. The `Registerx` procedure is defined with parameters `d: Device` and `num`. It includes a loop to assert that device names are unique and another loop to copy device data into a table. The `Unregisterx` procedure is also defined.
- Transformed Text:** Shows the transformed version of the source code. The parameter `d` is replaced by `device`. The `ASSERT` and `NEW` calls are updated to use the transformed names.
- Messages:** Displays two messages: "Source file successfully initialized" and "Transformation has been successful".
- CEG Information:** Provides metadata for the transformed code, including:
  - Name: `d`
  - Parent Context: `Register`
  - CEG Type: `Parameter Entity`
  - Dependency Set: `{ $PROC-1, $ASSIGN-16 }`
  - Reference Dependencies: `{ Device }`
  - Context Path: `{ Register, Disks }`

Figure 40: Hypertext, based on components designed using the Gadgets framework, is embedded within the source code generated from the CEG by NORT. The hypertext components are indicated by the bold typeface.

required when abstracting a sequence of statements into a procedure. Several improvements can still be made, such as identifying unnecessary parameters that may be declared as local variables or generating reference parameters to handle large data structures more efficiently.

Data abstraction is important, not only during the design and implementation of software, but also during maintenance. The lack of data abstraction transformations in NORT needs to be addressed. This can be accomplished by including transformations to support operations such as the creation of abstract data types.

## 5.4 Evaluation of the CEG

One of the objectives described in Section 1.1 was the use of a single program representation structure. The design of the CEG is closely related to the syntactic and semantic properties of the *Oberon* language. However, it will be possible to use the general context-entity relationship to represent other languages such as *C*. Modelling object oriented languages, such as *C++* and *Java*, can only be investigated once the CEG has been successfully used to model the object oriented structures of *Oberon*.

The CEG can easily be constructed during parsing and supports the computation of information necessary to implement a wide variety of transformations. It also provides the functionality to facilitate demand driven analysis.

However, program representation is not limited to the syntactic units of a language. A simple example should suffice: Programmers often use comments to describe and annotate their ideas. The CEG lacks the functionality of associating program constructs with specific comments. For example, a programmer may design a complex data structure and place a comment nearby to describe its function and that of specific fields. How should a program representation structure handle an abstraction operation that removes certain fields to create a new abstract data type? Should it try to analyse and associate specific comments with the data being abstracted and generate new comments? Should it depend on guidance from the user or simply do nothing? Issues such as these must also be addressed in the search to find a single program representation structure.

## 5.5 Future Work

A number of issues not addressed in this thesis require further investigation and include the following:

1. The prototype of NORT does not support the object orientation model of type extension and message passing employed by *Oberon*. The inclusion of type extended records, their representation within the CEG and methods to facilitate transformations on such records must still be examined.

Extending the CEG to accommodate type extended records should not pose a great problem. The addition of an edge to represent the base type of a record should suffice. The issues surrounding dynamic and static types, especially in context of the **WITH** guard statement, the **IS** relation and general alias analysis, justify an investigation to determine whether demand driven analysis, based on the CEG, can provide sufficient information to implement abstraction transformations related to object oriented frameworks.

2. The context-entity relationship provides a reasonable solution for restructuring a single software module. Applications developed within the *Oberon* environment may demand extending this relationship to model larger, more complete software systems. The *Oberon* compiler is a good example of a program composed of a number of modules. Allowing the user to perform restructuring operations across modules may result in improving the overall structure of the program and not just a single module. It may prove a challenging problem to represent and analyse the inter-module dependencies that exist within such a system or to apply demand driven analysis techniques across modules. One solution that may prove successful is to extend the context-entity relationship by creating a system context that encapsulates various module contexts, representing a complete software system. Another approach may be to incorporate the *module interconnection graph* (MIG) described by Kang and Bieman [23].
3. A variety of transformations were implemented in NORT, of which some were trivial, and others more complex. Analysing the needs of programmers and those responsible for software maintenance will result in the development of more specialised transformations to aid the user in functioning more efficiently and productively.

Clone detection, a technique used to identify duplicate code and abstract each occurrence into a subroutine call, should receive special consideration. Given the fact that clone detection can be implemented using the AST and the focus placed on demand driven techniques, makes this an ideal transformation to be considered for future implementations [5].

## 5.6 Conclusion

The project is deemed successful, although some work is still required to produce a tool that may be applied routinely during software development within the *Oberon* environment. This thesis has contributed by providing insight into the design and implementation process of a restructuring tool, specifically the representation structure and the design of transformations based on such a structure. This project also confirmed the results presented by Morgenthaler [29] that a single representation structure is feasible for developing a restructuring tool. Various problem areas have been identified and possible solutions and improvements have been proposed as future research.

To conclude: Martin Feather, one of the advocates of transformation systems during the 1970's, recently remarked:

“... the one big thing I, and perhaps others, didn't know was today's many and varied applications of transformation.” [14]

Software maintenance is important, but so are all the other activities that form part of the infamous software life cycle. Applying the correct tool to a specific problem is just as important as understanding the problem. A restructuring tool is one of many tools at the disposal of a programmer today and should not be used in isolation. It must be combined with other tools to realize its full potential in producing high quality software and furthering our understanding of large software systems.

## Appendix A

# Oberon Language Definition

The syntax definition of the subset of the Oberon language supported by NORT is presented in extended Backus-Naur Formalism (EBNF) based on the definition in [35]. The [ and ] symbols denote that the enclosed sentential form is optional while { and } denote repetition (possibly 0 times). Changes in the language definition occur from time to time. The current definition is contained in the `OberonReport.html` file which accompanies the Native Oberon distribution<sup>1</sup>. A formal specification describing the semantics of Oberon may be found in [25].

### A.1 Module

```
module = MODULE ident ";" [ImportList] DeclarationSequence
        [BEGIN StatementSequence] END ident "."
```

```
ImportList = IMPORT import {"," import} ";"
import = ident [":=" ident]
```

### A.2 Declarations

```
DeclarationSequence = {CONST {ConstantDeclaration ";" } |
                       TYPE {TypeDeclaration ";" } |
                       VAR {VariableDeclaration ";" } }
```

---

<sup>1</sup>Available through anonymous ftp at <ftp://ftp.inf.ethz.ch/pub/ETHOberon/>

```

        {ProcedureDeclaration ";" }
ConstantDeclaration = identdef "=" ConstExpression
ConstExpression = expression
TypeDeclaration = identdef "=" type
VariableDeclaration = IdentList ":" type
identdef = ident ["*"]

type = qualident | ArrayType | RecordType | PointerType | ProcedureType
ArrayType = ARRAY length {" ," length} OF type
length = ConstExpression
RecordType = RECORD FieldListSequence END
FieldListSequence = FieldList {";" FieldList}
FieldList = [IdentList ":" type]
IdentList = identdef {" ," identdef}
PointerType = POINTER TO type

ProcedureDeclaration = ProcedureHeading ";" ProcedureBody ident
ProcedureHeading = PROCEDURE ["*"] identdef [FormalParameters]
ProcedureBody = DeclarationSequence [BEGIN StatementSequence] END

FormalParameters = "(" [FPSection {";" FPSection}] ")"
FPSection = [VAR] ident {" ," ident} ":" FormalType
FormalType = {ARRAY OF} qualident | ProcedureType

```

### A.3 Statements

```

statement = [assignment | ProcedureCall | IfStatement | CaseStatement |
            WhileStatement | RepeatStatement | LoopStatement |
            ForStatement | EXIT | RETURN [expression] ]
assignment = designator ":=" expression
ProcedureCall = designator [ActualParameters]
StatementSequence = statement {";" statement}
IfStatement = IF expression THEN StatementSequence
              { ELSIF expression THEN StatementSequence }
              { ELSE StatementSequence }
              END
CaseStatement = CASE expression OF case {"|" case}
              {ELSE StatementSequence} END

```

```

case = [CaseLabelList ":" StatementSequence]
CaseLabelList = CaseLabels {"," CaseLabels}
CaseLabels = ConstExpression [".." ConstExpression]
WhileStatement = WHILE expression DO StatementSequence END
RepeatStatement = REPEAT StatementSequence UNTIL expression
LoopStatement = LOOP StatementSequence END
ForStatement = FOR designator ":@" expression TO expression DO
                StatementSequence END

```

## A.4 Expressions and Operators

```

expression = SimpleExpression [relation SimpleExpression]
relation = "=" | "#" | "<" | "<=" | ">" | ">=" | IN
SimpleExpression = ["+" | "-"] term {AddOperator term}
AddOperator = "+" | "-" | OR
term = factor {MulOperator factor}
MulOperator = "*" | "/" | DIV | MOD | "&"
factor = number | CharConstant | string | NIL | set | "(" expression ")" |
        designator [ActualParameters] | "~" factor
set = "{" [element {"," element}] "}"
element = expression [".." expression]
ActualParameters = "(" [ExpList] ")"
ExpList = expression {"," Expression}

```

## A.5 Identifiers, Numbers and Strings

```

ident = letter {letter | digit}
number = integer | real
integer = digit {digit} | digit {hexDigit} "H"
real = digit {digit} "." {digit} [ScaleFactor]
ScaleFactor = ("E" | "D") ["+" | "-"] digit {digit}
hexDigit = digit | "A" | "B" | "C" | "D" | "E" | "F"
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
CharConstant = "" character "" | digit {hexDigit} "X"
string = "" {character} ""

```

## Appendix B

# Selected Algorithms

The algorithms presented in Sections 4.2.3, 4.2.6 and 4.2.7 rely on a number of algorithms that perform intermediary calculations, such as examining dependencies. Selected algorithms are presented here to clarify details pertaining to the implementation of the aforementioned transformation functions.

### B.1 The Move Transformation

The  $\delta_{move}$  transformation (described in Section 4.2.3) relies on a number of algorithms that perform the necessary dependency checks. Local, global and current dependencies are examined by the algorithms in Figures 41, 42 and 43 respectively. The algorithm in Figure 46 is used when a whole procedure is moved. The CEG is recursively traversed to ensure that every entity and context within the procedure are examined.

```

1 proc CheckLocal(w, c)  $\equiv$ 
2   r  $\leftarrow$  ReferenceDependency(w);
3   while r  $\neq$  NIL do
4     if  $\neg r(\textit{dependency})(\textit{context}) \in \textit{ContextPath}(c)$  then ERROR fi;
5     r  $\leftarrow$  r(next)
6   od
7 end

```

Figure 41: Algorithm used to examine the local dependencies of entities when applying the  $\delta_{move}$  transformation.

```

1 proc CheckGlobal( $w, c$ )  $\equiv$ 
2    $r \leftarrow w(\text{dependencyset});$ 
3   while  $r \neq \text{NIL}$  do
4     if  $\neg c \in \text{ContextPath}(r(\text{dependency}))$  then ERROR fi;
5      $r \leftarrow r(\text{next})$ 
6   od
7 end

```

Figure 42: Algorithm used to examine the global dependencies of entities when applying the  $\delta_{move}$  transformation.

```

1 proc CheckCurrent( $w, c$ )  $\equiv$ 
2    $u \leftarrow \text{GlobalNode}(c, w(\text{name}));$ 
3   if  $(u \neq \text{NIL}) \wedge (u \neq c)$  then
4      $r \leftarrow u(\text{dependencyset});$ 
5     while  $r \neq \text{NIL}$  do
6       if  $c \in \text{ContextPath}(r(\text{dependency}))$  then ERROR fi;
7        $r \leftarrow r(\text{next})$ 
8     od
9   fi;
10   $r \leftarrow \text{ReferenceDependency}(w);$ 
11  while  $r \neq \text{NIL}$  do
12     $u \leftarrow \text{LocalNode}(c, u(\text{dependency})(\text{name}));$ 
13    if  $(u \neq \text{NIL}) \wedge$ 
14       $(u(\text{name}) \neq r(\text{dependency})(\text{name})) \wedge$ 
15       $(u \neq r(\text{dependency}))$  then ERROR fi;
16     $r \leftarrow r(\text{next})$ 
17  od
18 end

```

Figure 43: Algorithm to examine the existing dependencies of entities when applying the  $\delta_{move}$  transformation.

```

1 proc CheckDependenciesMove( $w, c$ )  $\equiv$ 
2   while  $w \neq \text{NIL}$  do
3     CheckLocal( $w, c$ );
4     CheckGlobal( $w, c$ );
5     CheckCurent( $w, c$ );
6      $w \leftarrow w(\text{next})$ 
7   od
8 end

```

Figure 44: Algorithm used to recursively evaluate all the dependencies of an entity when applying the  $\delta_{move}$  transformation.

```

1 proc CheckStatements(w, c)  $\equiv$ 
2   while w  $\neq$  NIL do
3     if w(type) = stmtAssign then CheckLocal(w, c)
4     elsif w(type) = stmtIf then
5         CheckLocal(w, c);
6         CheckStatements(w(body), c);
7         elsif  $\leftarrow$  w(elsif);
8         CheckStatements(w(else), c);
9     elsif w(type) = stmtCase then
10        CheckStatements(w(else), c);
11    elsif (w(type) = stmtWhile)  $\vee$  (w(type) = stmtRepeat)  $\vee$ 
12        (w(type) = stmtFor) then
13        CheckLocal(w, c);
14        CheckStatements(w(body), c)
15    elsif w(type) = stmtLoop then
16        CheckStatements(w(body), c)
17    elsif w(type) = stmtReturn then
18        CheckLocal(w, c)
19    fi;
20    w  $\leftarrow$  w(next)
21  od
22 end

```

Figure 45: Algorithm used to examine the dependencies of statements within a procedure context when applying the  $\delta_{move}$  transformation.

```

1 proc CheckAll(w, c)  $\equiv$ 
2   CheckDependenciesMove(w(resulttype), c);
3   CheckDependenciesMove(w(constantentity), c);
4   CheckDependenciesMove(w(typeentity), c);
5   CheckDependenciesMove(w(variableentity), c);
6   CheckDependenciesMove(w(parameterentity), c);
7   CheckStatements(w(body), c);
8   proc  $\leftarrow$  w(procedurecontext);
9   while proc  $\neq$  NIL do
10      CheckAll(proc, c);
11      proc  $\leftarrow$  proc(next)
12  od
13 end

```

Figure 46: Algorithm used to examine all the entities that may be affected when applying the  $\delta_{move}$  transformation to a procedure declaration.

## B.2 The Exchange Statement Transformation

The algorithm in Figure 47 is used to determine if a single entry and exit point exist. The algorithm examines the various statement types and recursively traverses compound structures such as **IF** and **WHILE** statements.

## B.3 The Create Procedure Transformation

The algorithm in Figure 48 is used by the  $\delta_{createprocedure}$  transformation to determine if the type dependencies of type and variable declarations will be satisfied after the new procedure is created.

```

1 proc SingleEntryExit(s)  $\equiv$ 
2 if s(type) = stmtAssign then return TRUE
3 elseif s(type)  $\in$  {stmtWhile, stmtRepeat,
4 stmtLoop, stmtFor} then
5     u  $\leftarrow$  s(body);
6     SEE  $\leftarrow$  TRUE;
7     while (u  $\neq$  NIL)  $\wedge$  SEE do
8         SEE  $\leftarrow$  SingleEntryExit(u);
9         u  $\leftarrow$  u(next)
10    od;
11    return SEE
12 elseif s(type) = stmtProcedureCall then
13     u  $\leftarrow$  s(procedurecall)(body);
14     SEE  $\leftarrow$  TRUE;
15     while (u  $\neq$  NIL)  $\wedge$  SEE do
16         SEE  $\leftarrow$  SingleEntryExit(u);
17         u  $\leftarrow$  u(next)
18    od;
19    return SEE
20 elseif s(type) = stmtIf then
21     u  $\leftarrow$  s(body);
22     SEE  $\leftarrow$  TRUE;
23     while (u  $\neq$  NIL)  $\wedge$  SEE do
24         SEE  $\leftarrow$  SingleEntryExit(u);
25         u  $\leftarrow$  u(next)
26    od;
27    if  $\neg$ SEE then return SEE fi;
28    w  $\leftarrow$  s(elsif);
29    while (w  $\neq$  NIL)  $\wedge$  SEE do
30         u  $\leftarrow$  w(body);
31         while (u  $\neq$  NIL)  $\wedge$  SEE do
32             SEE  $\leftarrow$  SingleEntryExit(u);
33             u  $\leftarrow$  u(next)
34        od;
35        w  $\leftarrow$  w(elsif)
36    od
37    if  $\neg$ SEE then return SEE fi;
38    u  $\leftarrow$  s(else);
39    SEE  $\leftarrow$  TRUE;
40    while (u  $\neq$  NIL)  $\wedge$  SEE do
41        SEE  $\leftarrow$  SingleEntryExit(u);
42        u  $\leftarrow$  u(next)
43    od;
44    return SEE
45 else return FALSE fi
46 end

```

Figure 47: Algorithm used to determine if a single entry and exit point exist.

```

1 proc CheckDependenciesProc(dep, p) ≡
2   while dep ≠ NIL do
3     d ← LastDesignator(d(dependency));
4     if d(type) ∈ {entType,
5                 entVariable} then
6       t ← TypeDependency(d(designator));
7       if ¬(t(designator)(context) ∈ p) ∧
8         t(designator)(context)(type) ≠ cntzPredefined then ERROR fi;
9     elseif d(type) = entConstant then
10      if ¬(d(designator)(context) ∈ p) ∧
11        d(designator)(context)(type) ≠ cntzPredefined then ERROR fi;
12    fi;
13    dep ← dep(next)
14  od
15 end

```

Figure 48: Algorithm used to examine type dependencies of type and variable declarations.

# Bibliography

- [1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.
- [2] Robert S. Arnold. Software Restructuring. *Proceedings of the IEEE*, 77(4):607–617, April 1989.
- [3] Darren C. Atkinson and William G. Griswold. Effective Whole-Program Analysis in the Presence of Pointers. 1998.
- [4] Ira D. Baxter. Scaling for the Design Maintenance System. In *Proceedings of the 1999 Conference on Software Transformation Systems*, pages 14–19, Los Angeles, California, May 1999.
- [5] Ira D. Baxter, A. Yahin, L. Moura, M. Sant’Anna, and L. Bier. Clone Detection Using Abstract Syntax Trees. In *Proceedings of the International Conference on Software Maintenance*, pages 368–377, 1998.
- [6] Robert William Bowdidge. *Supporting the Restructuring of Data Abstractions through Manipulation of a Program Visualization*. PhD thesis, Department of Computer Science and Engineering, University of California, San Diego, 1995.
- [7] Marc M. Brandis. *Optimizing Compilers for Structured Programming Languages*. PhD thesis, Swiss Federal Institute of Technology Zürich (ETH Zürich), 1995.
- [8] John G. Burch. *Systems Analysis, Design and Implementation*. boyd and fraser publishing company, 1992.
- [9] R.M. Burstall and John Darlington. A Transformation System for Developing Recursive Programs. *Journal of the ACM*, 24(1):44–67, January 1977.
- [10] Régis Crelier. OP2: A Portable Oberon-2 Compiler. In *2nd International Modula-2 Conference*, Loughborough, September 1991.

- [11] Régis B. J. Crelier. *Separate Compilation and Module Extension*. PhD thesis, Swiss Federal Institute of Technology Zürich (ETH Zürich), 1994.
- [12] Edsger W. Dijkstra. The Humble Programmer. *Communications of the ACM*, 15(10):859–866, October 1972.
- [13] Martin S. Feather. A System for Assisting Program Transformation. *ACM Transactions on Programming Languages and Systems*, 4(1):1–20, January 1982.
- [14] Martin S. Feather. Program Transformation - What We Didn't Know. In *Proceedings of the 1999 Conference on Software Transformation Systems*, pages 67–68, Los Angeles, California, May 1999. Panel Statement.
- [15] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.
- [16] André Fischer and Hannes Marais. *The Oberon Companion*. vdf Hochschulverlag AG an der ETH Zürich, 1998.
- [17] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Object Technology Series. Addison Wesley, 2000.
- [18] Keith Gallagher. Conditions to Assure Semantically Consistent Software Merges in Linear Time. In *Third Conference on Configuration Management*, 1991.
- [19] William G. Griswold. *Program Restructuring as an Aid to Software Maintenance*. PhD thesis, Department of Computer Science and Engineering, University of California, San Diego, 1991.
- [20] William G. Griswold and David Notkin. Semantic Manipulation of Program Source. Technical Report 91-08-03, Department of Computer Science & Engineering, University of Washington, August 1991.
- [21] William G. Griswold, Jimmy J. Yuan, and Yoshikiyo Kato. Exploiting the Map Metaphor in a Tool for Software Evolution. In *Proceedings of the 2001 International Conference on Software Engineering*, Toronto, March 2001.
- [22] Mary Jean Harrold and Brian Malloy. A Unified Interprocedural Program Representation for a Maintenance Environment. *IEEE Transactions on Software Engineering*, 19(6):584–593, June 1993.

- [23] Byung-Kyoo Kang and James M. Bieman. A Quantitative Framework for Software Restructuring. *Journal of Software Maintenance*, 11:245–284, 1999.
- [24] Walter F. Korman. Elbereth: Tool Support for Refactoring Java Programs. Master's thesis, Department of Computer Science and Engineering, University of California, San Diego, 1998.
- [25] Phillip W. Kutter and Alfonso Pierantonio. The Formal Specification of Oberon. *Journal of Universal Computer Science*, 3(5):443–503, May 1997.
- [26] David Alex Lamb and David Putman. SPRUCE: A Framework for Software Restructuring. Technical Report 1989-244, Department of Computing and Information Science, Queen's University, February 1989. Revised November 20, 1997.
- [27] Lawrence Markosian et al. Using an Enabling Technology to Reengineer Legacy Systems. *Communications of the ACM*, 37(5):58–70, May 1994.
- [28] Robert Morgan. *Building an Optimizing Compiler*. Butterworth-Heinemann, Woburn, MA, United States of America, 1998.
- [29] John David Morgenthaler. *Static Analysis for a Software Transformation Tool*. PhD thesis, Department of Computer Science and Engineering, University of California, San Diego, 1997.
- [30] John David Morgenthaler. Building an Efficient Software Manipulation Tool. Technical Report HKUST-CS98-02, Department of Computer Science, The Hong Kong University of Science & Technology, Clear Water Bay, Kowloon, Hong Kong, January 1998.
- [31] Hanspeter Mössenböck and Kai Koskimies. Active Text for Structuring and Understanding Source Code. Technical Report 3, Institut für Praktische Informatik (System Software), Johannes Kepler Universität Linz, Austria, August 1995.
- [32] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, San Francisco, California, United States of America, 1997.
- [33] Kurt Nørmark and Kasper Østerbye. Representing Programs as Hypertext. In *Proceedings of Nordic Workshop on Programming Environment Research*, pages 11–24, May 1994.
- [34] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, Graduate College of the University of Illinois at Urbana-Champaign, 1992.

- [35] Martin Reiser and Niklaus Wirth. *Programming in Oberon: Steps Beyond Pascal and Modula*. Addison-Wesley, 1992.
- [36] Don Roberts, John Brant, and Ralph Johnson. A Refactoring Tool for Smalltalk. *Theory and Practice of Object Systems*, 3(4), 1997.
- [37] Donald Bradley Roberts. *Practical Analysis for Refactoring*. PhD thesis, Graduate College of the University of Illinois at Urbana-Champaign, 1999.
- [38] George Sander. Graph Layout for Applications in Compiler Construction. Technical Report A/01/96, Universität des Saarlandes, February 1996.
- [39] Stephen R. Schach. *Classical and Object-Oriented Software Engineering*. McGraw-Hill, fourth edition, 1999.
- [40] Marc Shapiro and Susan Horwitz. The effects of the precision of pointer analysis. In *Proceedings of the 4th International Symposium on Static Analysis*, pages 16–34, January 1997.
- [41] Bjarne Steensgard. Points-to Analysis in Almost Linear Time. In *Symposium on Principles of Programming Languages*, pages 32–41, 1996.
- [42] Christoph Steindl. Program Slicing: Data Structures and Computation of Control Flow Information. Technical Report 11, Institut für Praktische Informatik (System Software), Johannes Kepler Universität Linz, Austria, March 1998.
- [43] M.G.J. van den Brand, P. Klint, and C. Verhoef. Term Rewriting for Sale. *Electronic Notes in Theoretical Computer Science*, 15, 1998. Elsevier Science B.V.
- [44] Reinhard Wilhelm and Dieter Maurer. *Compiler Design*. Addison-Wesley, 1995.
- [45] N. Wirth and J. Gutknecht. The Oberon System. *Software-Practice and Experience*, 19(9):857–893, September 1989.
- [46] Niklaus Wirth. *Compiler Construction*. Addison-Wesley, 1996.
- [47] Niklaus Wirth and J. Gutknecht. *Project Oberon – The Design and Implementation of an Operating System and Compiler*. Addison-Wesley, 1991.