Specifying Algorithms Using Evolving Algebra. Implementation of Functional Programming Languages.

by
Dag Diesen

March 1995

This report is a thesis for the dr. scient degree at
Department of Informatics
University of Oslo
Norway

Contents

I Modules in Evolving Algebra, an Extension **3**

1 The Art of Making a Specification **6**
- 1.1 A Specification . . . 6
  - 1.1.1 A Definition of Specification . . . 6
  - 1.1.2 Comments Regarding the Specification . . . 6
- 1.2 Specification as a Textual Description . . . 7
  - 1.2.1 A Simple Calculator . . . 7
  - 1.2.2 Using Textual Description in Specifications Today . . 8
- 1.3 Choosing an Abstraction Level . . . 9
  - 1.3.1 Operations . . . 9
  - 1.3.2 Algorithm . . . 9
  - 1.3.3 Computing Steps . . . 9
- 1.4 The Principle of Abstractions . . . 9
- 1.5 Using Resources . . . 10
- 1.6 Using the Evolving Algebra . . . 11
  - 1.6.1 Computing Medium and the Operations . . . 11
  - 1.6.2 The Computing Steps and Transition Steps . . . 11
  - 1.6.3 Using resources when executing Evolving Algebra specifications . . . 11
- 1.7 Understanding and Executing a Specification . . . 12
- 1.8 How to Maintain a Specification . . . 12
  - 1.8.1 The Need of Making Modules . . . 13
  - 1.8.2 The Need of Controlling and Understanding the Algorithm . . . 13
- 1.9 The Work with the Thesis . . . 13
  - 1.9.1 The Chronology of the Work . . . 13

2 Evolving Algebras **15**
- 2.1 A Specification Language, Requirement . . . 15
  - 2.1.1 Write an easily understood specification . . . 15
  - 2.1.2 Running the specification on a computer . . . 16
  - 2.1.3 Avoiding the encoding and decoding . . . 16
  - 2.1.4 Computing Directly on Abstract Data Structures . . 17
  - 2.1.5 Representing the time directly . . . 17
  - 2.1.6 Writing the specification at different abstraction levels 17
- 2.2 Core Evolving Algebra . . . 18
  - 2.2.1 Evolving Algebra and the Computing Medium . . . 19

2.2.2 Defining the Execution of Operations on the Computing Medium . . . 19
2.3 Specification of How to Compile and Evaluate a Functional Language . . . 20
2.4 Evolving Algebra, Non-determinism and Parallelism . . . 21
2.5 The Evolving Algebra Interpreter . . . 21
2.6 Gurevich Thesis . . . 21
2.6.1 Different Abstraction Levels . . . 21
2.6.2 Implementation . . . 23
2.7 The Turing Machine and Abstraction Levels . . . 24
2.7.1 Definition of a Turing Machine . . . 24
2.7.2 Express an Algorithm using a TM . . . 24
2.7.3 Express an Data Structure on a TM . . . 24
2.7.4 Express Time Units on a TM . . . 24
2.7.5 The Turing Theses . . . 25

3 Abstractions and Modules **26**
3.1 Computing Models . . . 26
3.1.1 Including the Use of Resources in the Specification . . 27
3.2 Abstraction Levels and Making Modules . . . 27
3.2.1 Abstraction Barrier . . . 27
3.2.2 Modules . . . 28
3.2.3 Co-routines . . . 28
3.2.4 Making Procedures . . . 28
3.3 Evolving Algebra and modules . . . 28
3.3.1 Embedding the Modules into Subroutines . . . 29
3.3.2 Dividing a Specification into Co-routines . . . 29
3.3.3 Naming Strategy . . . 29
3.3.4 Import and Export of Names . . . 30
3.3.5 Execution Control Strategy . . . 30
3.4 Specify the Control Strategy for Different Types of Modules 30
3.4.1 Making Co-operating Modules . . . 31
3.4.2 The Game Strategy Example . . . 31
3.4.3 Specify Call, Detach and Resume . . . 32
3.4.4 Making Subordinate Modules . . . 34
3.4.5 Making Instances of the Players . . . 35
3.4.6 Recursive Calls . . . 37
3.4.7 Name Spaces and Modules . . . 39
3.4.8 Components of a Module . . . 40
3.4.9 Function to be Used Within More than One Module Definition . . . 40
3.4.10 An Instance and a Revised Definition of the Evolving Algebra State . . . 41
3.4.11 A Definition of State in Extended Evolving Algebra . 42
3.4.12 Start of the Execution of Modules . . . 43
3.4.13 Components of The Extended Evolving Algebra Specification . . . 43
3.4.14 Threads . . . 43

3.5 Other Language Constructs 44

II Compilation and Evaluation of a Functional Language, a Specification 45

4 Maintaining Abstraction Levels 48
4.1 The Problem of Defining Abstraction Levels in a Specification 48
4.1.1 The Turing Machine as a Specification Language . . . 48
4.1.2 A Graph Machine 48
4.1.3 The Evolving Algebra 49
4.2 Other Examples Compared with of the Lambda Compiler and Interpreter Specification 49
4.2.1 The C-language Specification Example 49
4.2.2 The Prolog Specification Example 51
4.2.3 Specification of a Lambda Compiler and Lambda Evaluator 54
4.3 The Prolog Execution Tree Example Rewritten Using Modules 54
4.3.1 The Execution Tree 55
4.3.2 Move Down and Up in the Execution tree 56
4.3.3 Making a New Nodes in the Execution Tree 59
4.4 The Importance of Dividing Specification into Modules . . . 60
4.5 Implementation of a Functional Language 61
4.5.1 A supercombinator 61
4.5.2 Translate a Lambda Expression into a Supercombinator 61
4.5.3 Lazy Evaluation 62
4.5.4 Eager Evaluation 62
4.5.5 Eager and Lazy evaluation 62
4.5.6 Eager Evaluation and Weak Head Normal Form . . . 63
4.5.7 Using Environment or Graph Structure 64
4.5.8 Interpreting and Compiling an Expression 64
4.6 The Project 64
4.6.1 Specification of an Algorithm and the Use of Resources 65
4.7 The Main Steps of the Lambda Compiler and Lambda Evaluator 65
4.7.1 Compilation of Lambda Expression 65
4.7.2 Translating Lambda-expressions to Supercombinators 66
4.7.3 Compiling the Supercombinator Definitions 66
4.7.4 Evaluation of the Supercombinators 66
4.8 Dividing the Specification into Modules 66

5 Compilation of Supercombinators 69
5.1 Introduction 69
5.2 The Chosen Abstraction Level 69
5.3 Compile the Supercombinator Definitions 70
5.3.1 Compile All the Supercombinator Definitions 70
5.4 Compile a Supercombinator Definition 71
5.4.1 The Signature and Abbrevations 71
5.4.2 Start of the Compilation 73

5.4.3 Find a Supercombinator Definition 73
5.4.4 Compile the SC-definition Abstract Specification . . . 74
5.5 The Supercombinator Graph 74
5.5.1 Interior Nodes 74
5.5.2 Leaf Nodes 75
5.6 Making the Graph 75
5.6.1 The Supercombinator Definition Node 75
5.7 How to Compile of the Supercombinator Definition 76
5.7.1 The Recursive Definition Approach 76
5.7.2 Using the Core Evolving Algebra to Specify the Compilation 76
5.7.3 Specification of the Compilation of the SC-body . . . 77
5.7.4 An Example 79
5.7.5 A Brief Note About Efficiency 83
5.8 Select the Target Structure 83
5.8.1 Making a Tree 84
5.8.2 Making an Acyclic Graph 84
5.9 Promise to Make a Graph 85
5.9.1 Compilation into G-machine Code, the EA-specification 86
5.9.2 Make the Initial Instructions 87
5.9.3 Promise to Make the SC-definition 88
5.9.4 Promise to Make an Application Node 89
5.9.5 Promise to Make a Number Node 90
5.9.6 Promise to Make a Supercombinator Name Node . . 90
5.9.7 Promise to Make a Local Variable Name Node 91
5.9.8 Add Finished Code to the Sequence of Instructions . 91
5.9.9 Assign the Finished Code to the current SC-definition 92
5.9.10 Update of the Root of the Redex 92

6 Evaluation of the Graph 94
6.1 Introduction 94
6.2 The Reduction Process 94
6.2.1 The Signature 94
6.2.2 Specification of the Reduction 94
6.3 A Less Abstract Specification of a Reduction 95
6.3.1 Find the outermost left redex 95
6.3.2 Reduce the redex expression 96
6.3.3 Prepare for the next reduction step 96
6.4 The States of the Evaluator 96
6.4.1 The Structure of the Reduction Machine 96
6.4.2 The Components of the State 97
6.4.3 The Modes of the Evaluator 97
6.5 The Detailed Specification of a Reduction Machine 98
6.5.1 The Pieces of the Graph 98
6.5.2 Description of the Reduction Machine 99
6.5.3 The Signature 100
6.6 Set the Initial value of the State 101
6.6.1 The Evolving Algebra Specification 101

6.7 The Evolving Algebra Specification of the Unwinding and Making a Substitution List 102
6.7.1 Find the Redex of the Graph 102
6.7.2 Make the List of Substitutions 103
6.8 Specification of How to Build a New Instance 104
6.8.1 A Recursive Definition of the Building Process 104
6.8.2 The Build Process Described as Iterations 104
6.8.3 Common Abbreviations 105
6.8.4 Starts the Build of the New Instance 105
6.8.5 Replacing the Local Variable by the Substitution Value 107
6.8.6 Pop Finished Pointers of the Address Stack 108
6.8.7 Finish the Reduction Step 108
6.9 The Update 109
6.9.1 Specify the Update 109
6.10 Representation of the Supercombinator Definition 109
6.10.1 A Tree 109
6.11 Perform the Update 110
6.11.1 Make a Copy of the Result of the Reduction 110
6.11.2 Update the Root of the Redex 110
6.12 A Compiled sequence of instructions 111
6.12.1 Common Functions and Signature 112
6.12.2 Initialize the Mode to Evaluate Compiled Instructions 113
6.12.3 Change the Mode of Evaluation to Evaluate Compiled Instruction 113
6.12.4 The Pushglobal Instruction 113
6.12.5 The Pushint Instruction 114
6.12.6 The Push Instruction 114
6.12.7 The Mkap Instruction 114
6.12.8 The Slide Instruction 115
6.12.9 The Unwind Instruction 115
6.12.10 Push New Instruction Sequence on Instruction Stack 116
6.12.11 Change to graph mode 116
6.13 Update the Redex 116
6.13.1 The Update Instruction 116
6.13.2 The Pop Instruction 117
6.13.3 The Unwind Instruction 117

7 Strict and Lazy Arguments 118
7.1 Evaluate the Primitive Expression 119
7.1.1 The Recursive Process of evaluating the arguments . 119
7.2 Postpone the Evaluation of Strict Arguments 120
7.2.1 The process of evaluation 120
7.2.2 The Core Evolving Algebra Signature 122
7.2.3 Abbreviations 124
7.2.4 The Core Evolving Algebra Transitions 124
7.2.5 Restore the Dump 124
7.2.6 Apply the Primitive Operator 126
7.3 Handling Indirection Nodes 127

7.4 Too Lazy . . . 127
7.4.1 Which part of an expression should be treated as strict? 127
7.5 Strict Evaluation of Arguments Given to the Supercombinator Expression . . . 128
7.5.1 Abbreviations . . . 129
7.5.2 The Signature . . . 129
7.5.3 Make the Supercombinator Object . . . 129
7.5.4 Substitute Non Strict Argument . . . 130
7.5.5 Substitute Argument in WHNF form . . . 130
7.5.6 Supercombinator Parameters Annotated as Strict . . . 130
7.5.7 Restore the Dump . . . 131
7.5.8 Application node annotated as strict . . . 131
7.6 The G-machine Evaluator . . . 132
7.6.1 The Signature . . . 132
7.6.2 Push Boolean Data . . . 132
7.6.3 Push Boolean Data . . . 133
7.6.4 The Primitive Operators . . . 133
7.6.5 Executing the Eval Instruction . . . 134
7.6.6 Restore the Stack on the Dump . . . 135
7.7 Extensions Needed to Compile Primitives and Supercombinators . . . 135
7.7.1 Primitive Name Node . . . 135
7.7.2 Annotate Strict and Lazy arguments and Parameters 135
7.8 Extending the compiler to handle primitives . . . 136
7.8.1 The Signatures . . . 136
7.8.2 Abbreviation . . . 136
7.8.3 Changes needed in order to handle primitives . . . 137
7.8.4 Compiling Primitive Definitions . . . 137
7.8.5 Adding primitives to the globals . . . 137
7.8.6 Adding the primitive definition node to the graph . . . 138
7.9 Extend Supercombinator Definitions to Handle Primitives . . . 138
7.9.1 The Signature . . . 139
7.9.2 Abbreviation . . . 139
7.9.3 Making the a node holding the name of the primitive 139
7.9.4 Mark the Strict Parameter of a Supercombinator . . . 140
7.10 Add the Boolean Node . . . 140
7.11 Add strict annotation to the application node . . . 141
7.12 Generating Compiled Instruction from the Primitive Definitions 141
7.12.1 The Signature . . . 142
7.12.2 Start Compilation of the Primitive Definitions . . . 144
7.12.3 Prepare for Compilation of the a Primitive Definition 144
7.12.4 The Primitive Definition . . . 144
7.12.5 Make the List of Parameters for Primitives of Arity One 145
7.12.6 Make the List of Parameters for Primitives of Arity Two 146
7.12.7 Make the Parameter List for the If Primitive . . . 147
7.12.8 Make the G-machine Code for a Strict Argument Given to the Primitive . . . 148

7.12.9 Make the G-machine Code for a Non Strict Argument Given to the Primitive . . . 148
7.12.10 Make the G-machine Instruction for the Addition . . 148
7.12.11 Make the G-machine Instruction for the Negation . . 149
7.12.12 Make the G-machine Instruction for the If Primitive . 149
7.12.13 Traverse up . . . 150
7.12.14 End of the Primitive Definition . . . 150
7.12.15 Prepare for Executing the G-machine Code after Compiling All the Primitives and Supercombinators . . . 150
7.13 Extending the Supercombinator Definition . . . 151
7.13.1 Compiling the Boolean Expression . . . 151
7.13.2 Compiling the Data Expression . . . 151
7.13.3 Compiling the Primitive Name Expression . . . 151
7.13.4 Compiling the Variable Expression when the Corresponding Supercombinator Parameter is Marked as Strict . . . 152
7.13.5 Compiling the Variable Expression when the Corresponding Supercombinator Parameter is Marked as Non Strict . . . 153
7.14 Making the Primitive as Part of the Supercombinator Language Syntax . . . 153
7.15 Strictness Annotation of the Application Node . . . 154
7.15.1 Promise to Make an Application Node . . . 154

8 Using Exended Evolving Algebra in the Specification 156
8.1 Recursive Calls . . . 156
8.1.1 Specify the Compilation of of the Body of a Supercombinator as a Recursive Process . . . 156
8.1.2 Use of Recursive Modules in the Specification of Compilation and Evaluation . . . 160
8.2 Use of Co-routines and Parallel Programming . . . 160
8.2.1 Concurrent Specification . . . 160

III The Evolving Algebra Interpreter 163

9 The Evolving Algebra Interpreter 166
9.1 Introduction . . . 166
9.2 Evolving Algebra Definitions . . . 166
9.2.1 The Fixed Part of the Evolving Algebra . . . 166
9.2.2 The Transition . . . 167
9.2.3 Setting the Initial Values . . . 169
9.2.4 The Evolving Algebra Environment . . . 169
9.2.5 Commands . . . 170
9.2.6 System Commands . . . 171
9.3 How the Interpreter Works . . . 171
9.3.1 Parsing and Loading the Evolving Algebra Specification 171
9.3.2 Executing the Evolving Algebra Specification . . . 172

9.4 The Statistics . . . 174
9.4.1 The Statistical Data . . . 174
9.5 Comparision with Huggins EA-interpreter . . . 176
9.5.1 Implementation . . . 177
9.5.2 Function Definition . . . 177
9.5.3 The Initial Rules . . . 177
9.5.4 Transition Rules . . . 178
9.5.5 Expressions . . . 178
9.5.6 Guarded Rules . . . 178
9.5.7 Universe Extensions . . . 179
9.5.8 Universe Contractions . . . 179
9.5.9 Defining Functions and Universes . . . 179
9.5.10 Running the Interpreter . . . 180
9.5.11 Statistics . . . 180
9.6 Issue Regarding Implementation of Extended Evolving Algebra 181
9.6.1 Instances of a Module . . . 181
9.6.2 The Module . . . 182
9.6.3 Global Data . . . 182
9.6.4 Invoke and Invoke Return . . . 182
9.6.5 Telling If the Instance Can Be Invoked Again . . . 183

10 Running the Interpreter **184**
10.1 Supercombinators Used . . . 184
10.1.1 Church Numeral for one (pure version) . . . 184
10.1.2 Church Numeral for one (a simpler version) . . . 184
10.2 Weak Head Normal Form . . . 184
10.2.1 Reducing the Church Numerals to Weak Head Normal Form . . . 185
10.2.2 Extending the algorithm to take strict arguments and primitives . . . 185
10.3 Template instantiations without Defined Primitives . . . 186
10.3.1 Running the Church Numeral for one (pure version) . 186
10.3.2 Running the Church Numeral for one (simple version) 186
10.4 Simple G-machine without Defined Primitives . . . 186
10.4.1 Church Numeral for one (pure version) . . . 187
10.4.2 Church Numeral for one (a simpler version) . . . 187
10.5 Discussion of the Result . . . 187
10.5.1 Template Instantiations and G-machines for Supercombinators without Primitives . . . 187

IV Discussion and Conclusion **189**

11 Evolving Algebra as a Programming Language **192**
11.1 Introduction . . . 192
11.2 Evolving Algebra as a Specification and Programming Languages . . . 193
11.2.1 Specifying Data Structures . . . 193

11.2.2 Control Structures 193
11.3 Some Examples 194
11.3.1 Loops and Sequential Execution 194
11.3.2 Making the two sequences in parallel 198
11.3.3 Dividing into Cases 199
11.3.4 Define Common Operation on some Signatures 202
11.3.5 Modules and Subroutine Calls 203
11.4 The Dynamic Part of Evolving Algebra Described as an Algorithm 203
11.5 Some New Control Structures 205
11.5.1 Subroutines 205
11.5.2 Sequences 208
11.5.3 Case Construct 208
11.5.4 Polymorphic Operators 209
11.6 Abstraction 210
11.6.1 Abstraction and Making Modular Parts of the Specification 210
11.6.2 Levels of Abstraction in the Specification 210
11.6.3 Specification to be Read by Man 211
11.6.4 Specification as a Prototype Implementation 211
11.6.5 Combining Different Levels of Specifications 211
11.7 Understanding the Evolving Algebra Specification 211
11.8 Specifying Use of Resources 212

12 Evolving Algebra and Other Semantic Languages 213
12.1 Evolving Algebra and Operational Semantics 213
12.1.1 Operational Semantics Described as a Transitions System 213
12.1.2 Structural Operational Semantics 216
12.1.3 Natural Semantics 216
12.1.4 Use of Operational Semantics 217
12.1.5 Evolving Algebra 218
12.2 Evolving Algebra and Denotational Semantics 218
12.2.1 Evolving Algebra 218
12.2.2 Denotational Semantics 219
12.3 Make Comparison of Some Common Language Constructs . 220
12.3.1 Sequence of Operations 220
12.3.2 Fixed number of iterations 221
12.3.3 Unbound number of iterations 221
12.3.4 Recursive Definition 221
12.3.5 Non Determinism 221
12.4 How to describe properties in the algorithms or language . . 223
12.4.1 State Transition Systems 223
12.4.2 Use of Resources 223
12.4.3 Modify the Existing Specifications 224

13 Some Concluding Remarks Based on the Experience with Evolving Algebra **225**
13.1 The Round Trip 225
13.2 The Core Evolving Algebra 225
13.2.1 Evolving Algebra and the Use of Resources 226
13.3 Making Modules 226
13.3.1 Making Modules in the Evolving Algebra Specification 227
13.3.2 Why Adding New Features to Evolving Algebra . . . 227
13.3.3 Implementation of the Evolving Algebra Module Extension 228
13.4 Short Summary of other Evolving Algebra Implementations . 229
13.4.1 Evolving Algebra Interpreter Written in C 229
13.4.2 DASL Compiler Implemented in Prolog 229
13.5 Future Research 230

V Appendix **233**

A Evolving Algebra Specification Given to the Interpret **235**
A.1 The Template Instantiation Specification 235
A.1.1 Specification of the Compilation 235
A.1.2 Specification of the Reduction Process 242
A.2 The G-machine Specification 248
A.2.1 Specification of the Compilation 248
A.2.2 Specification of the Reduction Process 257
A.3 The Template Instantiation Specification extended to strict arguments and primitives 260
A.3.1 Specification of the Compilation 260
A.3.2 Specification of the Reduction Process 270
A.4 The G-machine Specification 282
A.4.1 Specification of the Compilation 282
A.4.2 Specification of the Reduction Process 300

B The Evolving Algebra Interpret **309**

List of Figures

5.1 The supercombinator definition 79
5.2 Step 1: Create a supercombinator definition node 79
5.3 Step 2: Create the first application node 80
5.4 Step 3: Create a local variable node 80
5.5 Step 4: Pop an element of the address stack 80
5.6 Step 5: Create the second application node 81
5.7 Step 6: Create a local variable node 81
5.8 Step 7: Pop an element of the address stack 81
5.9 Step 8: Create a supercombinator name node 82
5.10 Step 10: Two elements popped of the stack 82
5.11 Step 12: The last element popped of the address stack 82

7.1 After unwinding the primitive graph. 121
7.2 Just before computing the primitive expression. 121

Preface

This report is my thesis for the dr. scient. degree at the Department of informatics at the University of Oslo.

This report is divided into four main parts:

- The Evolving Algebra language and extension to the Evolving Algebra which in a better way handles modules.
- Specification of compilation and evaluation of a functional language.
- Description of the Evolving Algebra interpreter, written as part of the research leading to this report.
- Discussion and conclusion.

Each of the four parts of this report is preceded by an introduction. The readers are referred to the introductions preceding each part of the report for a more comprehensive description of the contents.

Here I will mention a little about how the work with the thesis proceeded.

The implementation of the Evolving Algebra interpreter was the first task to be done. The task of trying out the Evolving Algebra language and interpreter took place after the interpreter was implemented. The specification of how to compile and evaluate a functional language was written and run on the interpreter.

At an early stage the author was aware the lack of control structures in Evolving Language. Since the Evolving Algebra itself was intended to be used as a specification language, it was desirable to try out the language as it was defined, and see what could be done and what could not be done when making a large specification. The reason is that we do not want to make too many constructions in a meta-language which is used to specify other languages.

It turned out that the main problems with Evolving Algebra as a specification language, was the difficulties of dividing the specification into modules, if necessary. This gave rise to the first part of the report, where the module extension to Evolving Algebra is specified in chapter 3.

Thus, the presentation of the first three parts in this thesis comes in the opposite order compared with the time order of the correspondent research tasks.

Acknowledge

I would like to thank my supervisor, and my good friend, professor Herman Ruge Jervell for all support when I was working with the research leading to this report. This report could not be written without his stimulating discussions, necessary critiques, guidance, encouragement and patience. Especially in periods when the work with the thesis seemed to proceed too slowly, his patience and encouragement helped me much.

Thank to my internal supervisor at Department of Informatics, professor Olaf Owe for his support.

I would like to thank Helena Caseiro for clearing up a little problem which helped me understanding in detail how to compute Church Numerals.

Thanks to all members of the system administrators group for keeping the computer facilities healthy all the time. A special thank to Anders Ellefsrud for installing and making MIT Scheme interpreter to run to my satisfaction.

Many thanks goes to all of the employee at the Department of Informatics which in some ways helped me during my stay at the department.

I would also like to thank manager for the administration at the Department of Informatics, Elisabeth Hurlen for all her support during may stay at the Department of Informatics. She has become a good friend of mine, especially after years of co-operation in the period when I was member of the board of the Department of Informatics. In addition I will thank the members of the administration for running Department of Informatics in a way making it easy to do the research and other duties at the department.

Thanks to all my friends for encouragement during the work with this thesis. Especially, I would like to thank my good friends Arne Kristian Groven, Kjetil Karlsen and Olav Lysne for always be willing to talk with me, when something seemed to be difficult. I do not mention all of my other good friends here, but I have not forgotten none of you.

I would like to thank my sisters Anette Diesen, and Bodil Diesen for their encouragement. At last I would like to thank my parents, Anne-Marie Diesen and Knut Diesen for all support when writing this thesis and for believing I would be able to finish the work.

Part I

Modules in Evolving Algebra, an Extension

Introduction

This part of the report consists of the following:

- Discussion of the task of making semantic specifications.
- Presentation of Evolving Algebra as a semantic language.
- Description and presentation of the extension to Evolving Algebra which make it possible to making a modular Evolving Algebra specification.

When making a specification, we will want to make the specification which is correct, easy to understand, at the desired level of abstraction level, and possible to implement on a computer. It has also becomes a demand that we are able to specify in an abstract way the use of computing resources. The properties we will want a specification language to hold is the main contents in chapter 1.

The Evolving Algebra comes with the following theses [Gur93]:

- We can express the algorithm using Evolving Algebra.
- We can describe the data structure without use of code.
- We can represent the time directly.
- We do not need to change the signature in order to compute the specification.

In chapter 2 the formal definition of Evolving Algebra is stated. We will say something about the requirements of a specification language. We proceed with a discussion into which extent the Evolving Algebra meets those requirements stated. The possibilities of making a specification at the intended abstraction level with help of Evolving Algebra is treated. We close this chapter with a short review of the properties of the Turing Machine, and the lack of possibilities to make a specification at a chosen abstraction level with help of a Turing Machine.

In chapter 3 we define an extension to Evolving Algebra which makes it possible to divide an Evolving Algebra specification into modules. Mechanisms which permits us to make instances of a module specification, specify recursive calls on modules and specify co-routines is introduced.

Chapter 1

The Art of Making a Specification

1.1 A Specification

1.1.1 A Definition of Specification

There are many possible definitions of meaning of a specification. In [Mor90] we find:

> "The specification is the principal feature of abstract programs. Its *precondition* describes the initial states; its *postcondition* describes the final states; and its *frame* lists the variables whose values may change."

1.1.2 Comments Regarding the Specification

The definition above is especially suited for formal verification of a program. We describe condition before the program is to be executed, and the formal result of executing the program and in addition which variables is to be changed.

We do not say anything about how the implementation is going to perform its task, only what the implementation is supposed to do. The use of resources is not at all specified.

As a tool to write specifications, several formal specification languages exists in addition to the use of natural languages[1]. Some of the formal specification languages (or group of languages) are:

- Denotational semantics
- Operational semantics
- Formal languages from mathematical logic, such as predicate calculus

When we use denotational semantics or a formal languages as a specification language, we are going to say much of what we want to perform,

[1] Such as english, norwegian or chinese

but not very much of how we will perform an algorithm. And the definition above of what a specification means, tends to support this approach to the specification language.

An alternative approach to what we would expect by a specification language, is that we want to specify what we want to perform and to certain degree how we will perform an algorithm.

Operational semantics tends in some extent to support this approach.

With use of Evolving Algebra we go one step further in direction of telling how an algorithm is performed. Evolving Algebra permit us in an abstract way to specify the use of resources, such as the use of time and space when performing an algorithm.

So we will discuss the issue of making the semantics specification further, beginning with the problem of writing specification in a natural language.

1.2 Specification as a Textual Description

Since the use of natural language still is in common use, when writing a specification it is natural to begin the discussion with specifications written in a natural language.

1.2.1 A Simple Calculator

Consider the following specification of a stack. We may write a pure textual description, say:

> The stack operation we want is the following:
>
> 1. Push an element onto the stack.
> 2. Pop an element of an stack.
> 3. Get the value of the element on top of the stack.
> 4. Make the initial empty stack
> 5. Tests if the stack is empty or not
>
> The stack is going to be used in a simple calculator. The result of argument given to each operation is taken from the top of the stack, and the result of computation is stored as a new element on the top of the stack. The first version of the calculator is experimental, and therefore very simple. Only addition, subtractions, multiplications and division is supported. In addition we need to read and store the numbers to be computed.
>
> Since the experimental version of the calculator is going to be built on a general purpose computer equipped with hardware instructions such as Add, Subtract, Multiply and Divide and Store, we do not need to specify those operations further. But we will make a program to simulate the rest of the calculator, before we really make the calculator in hardware. So we need to

make a specification for that program, which we will give to the specialist of assembly programming on the NHDRM [2] computer.

The first time an electronic calculator was made, we could imagine a textual description like this one above *might* have been the first approach in order to specify the calculator. And the readers may be alerted of the lack of precision in such a textual description.

1.2.2 Using Textual Description in Specifications Today

Even to day, many important specification of programming languages, and important algorithms, is made as textual description with regards to the semantics of the languages or algorithm. We can take as two examples the following standards:

- Core protocol for the graphical X Window System, Version 11 [AN90]
- Revised 3 Report on the Algorithmic Language Scheme [RWC86].

The X Window System is an important industry standard for window systems. The X Window System define the communication between an a program which manage a display of graphical workstation or an advanced X-terminal (the server) and the application (the client) program. The syntax for the communication between the server and client is clearly defined in the Core protocol. But the semantics, in this case what the server is supposed to do when it receive the protocol message is mainly defined as a textual description. The exception from this, is that the replies which may be sent back to a client in response to certain request is more clearly defined.

Scheme is a Lisp dialect in widespread use. The syntax of the Scheme language is clearly defined. However, the semantics is mainly defined as a textual description. However, it is also a specification of Scheme using denotational semantics (See section 7.2 in [RWC86]).

It may be good reasons why the semantics of specifications in the example above is given as a textual description instead of as a specification using a formal language. The reasons for using textual description to specify the semantic may be the problems of understanding the model used in many specification languages, and the fact that many specification languages do not permit you to specify the use of resources.

Consider the specification in denotational semantics for Scheme (Section 7.2 in [RWC86]). It is not part of this specification that an implementation of Scheme is required to optimize the tail recursive call. In the textual description of the Scheme language this important property of the Scheme language is made very clear.

This point illustrate the shortcoming of many formal specification languages. Therefore we may want to discuss in some more depth the non-trivial task of making a specification. We will the start discussion with the informal description of the calculator above.

[2] The imaginary company with the name No Hope for this Difficult Register Machine

1.3 Choosing an Abstraction Level

We will want to make a certain level of abstraction, when we make an abstraction. We do not have any canonical level. So the level chosen is dependent of why we make the specification, and the intended use of the specification.

To define an abstraction level we need to define:

- A set of operations.
- Algorithms
- Computing steps

We may find some indication of the desired abstraction levels of the specification in the textual description of the NHDRM calculator above.

1.3.1 Operations

In the textual description above we have described the arithmetic operations of the calculator. In addition we have the stack operations, such as Pop, Push, Top and Is-empty. We can regard the instruction set as fixed at the chosen abstraction level. We can regard the operations as the computing medium at the chosen abstraction levels.

1.3.2 Algorithm

We have to define an algorithm which reflect the intended use of the calculator. We will want to use a language which is precise enough and still understandable for humans.

1.3.3 Computing Steps

We will also want to define computer steps at the chosen abstraction level. Then we can better limit ourselves to make the description at the desired abstraction level. In addition, we may want to count resources like the number of steps used to calculate an expression.

For the example of the calculator based on the NHDRM computer a computer step corresponding to performing one instruction will reflect the abstraction level indicated in the textual specification above.

1.4 The Principle of Abstractions

The abstraction levels as indicated above for the calculator specification, is not the only possible. We may choose a very abstract specification as:

> The calculator can perform the basic computer operations.

Or we can choose a much more detailed specification as:

- Telling in detail how to implement additions, subtractions, multiplication and division on the calculator. Telling how to implement the data-structure representing the the stack on the calculator.
- We may decide to use an assembly language to implement the prototype of the calculator. The specification may be the assembly program written.
- Or we may choose to make a Turing Machine description in order to make an hardware independent specification at a very detailed level.
- We may decide to build the calculator. And we describe how the calculator is to be build physically.
- It is possible to go further down, and specify all the different molecules used in the calculator.
- At last we can describe the calculator at quantum mechanical level.

The principle of abstraction can be explained in the following way:

> If we make a specification at a more detailed level, it can be seen seen as the same as to "code" the specification made for a more abstract level.

As an example we may "code" the multiplication of two numbers by adding the the first multiplicand as many times as specified by the second multiplicand.

So when we make a specification at a certain abstraction level, we do not want to "code" any part of the specification, since the coding force us to use a lower abstraction level than the intended level.

1.5 Using Resources

In many algorithms the use of resources, such as space and time is essential. In fact, the use of time and space is essential in computer science. We can not reason in a meaningful way about the computing of algorithms, if we do not consider the use of time and space, and how to achieve reasonable use of those computing resources. Despite this fact, many specification language do not take the use of resources into consideration.

A calculator which has very little space to store numbers, such that even a small expression is impossible to compute, has not much value for the user. So the user will want to know how many elements which can be stored on the stack. In a similar way, a calculator which is only able to compute very small number is not of much worth to the user. So the user may want to know the maximal and minimal value of the integers the calculator support.

The time used to perform calculation has much to say for a user. You will probably not want to wait the whole day, when the calculator perform the multiplication of two numbers, because the calculator use an extremely inefficient algorithm and may be an extremely inefficient representation of the numbers.

So the specification has to say something about the use of resources, such as time and space in an abstract way.

In the calculator example, the number of the basic computer operations (e.g. add) and the number of operations on the stack (e.g. pop) may be an abstract measure of the time the calculator will use.

The number of elements in the stack may be an abstract measure of space to be used by the calculator.

1.6 Using the Evolving Algebra

In section 2.2 we will give the definition of the (core) Evolving Algebra. Here we will discuss the use of Evolving Algebra as a specification language.

How do we use the principle as outlined above when we use Evolving Algebra as a specification language?

1.6.1 Computing Medium and the Operations

In Evolving Algebra we have the static part of the specification as the functions and the signature associated with the functions.

This functions given in the Evolving Algebra corresponds to the operations or computer medium at the chosen abstraction levels.

The algorithm in the specification is the set of transitions in an Evolving Algebra specification.

1.6.2 The Computing Steps and Transition Steps

A transition will change the computing medium in the following way:

- A constant or a function will be changed by a function update.
- New elements will be added to the universe(s) by the universe update.

What about the computing steps? We can think of each transition as a computer step. All updates within a transition is performed simultaneously. So the execution of a transition can be regarded as an atomic step, and hence as a computer step.

On the other hand, every transition may consists of many updates on several functions used in the Algebra. So we may instead consider a function updates or the adding of some new elements to a universe as a computing step. This steps is some sense the most basic step to be performed in an Evolving Algebra.

However, we do not want to take any decision here what a computing step should mean with regards to Evolving Algebra.

1.6.3 Using resources when executing Evolving Algebra specifications

The numbers of performed function updates or the number of performed transitions can be the abstract measure or the time used, when executing an Evolving Algebra specification.

The number of elements added to each of the universes can be regarded as an abstract measure of the use of space.

It is important that we measure the number of times for each transition updates which is performed. The reason is that in a real implementation some operations (which correspond to certain transitions or updates) may be more costly in terms of execution time than other operations.

In the same way we have to measure how many elements is added to each of the universes. In a real implementation adding an element to a certain universe may be more costly in terms of space than adding an element to another universe.

The implementation on calculator may require one more memory cell for each new element added to the stack. When adding a new element to the universe of numbers, we do not require more memory cells on the computer to store the integer which is increased as long as the integer do not reach a fixed value. Many implementation may define a maximal number a integer can have, instead of using more memory cells to store a big integer.

1.7 Understanding and Executing a Specification

We will need both to understand and perform a specification.

If the specification is at a high abstraction levels, the difficulties in understanding the specification may be to understand the abstract concepts used. A specification at a high abstraction levels may also be difficult or at least require much work to implement. That is because the abstract concepts will have to be written into some less abstract code when implemented.

A specification which is not very abstract may be difficult to read and understand because much details have to be specificed, so the specification may be very large and complicated.

The implementation of a less abstract specification should be easier, since we do not need to implement very abstract concepts. However, since the specification may be quite complicated, it may be difficult to ensure that the specification in fact is right and is implemented right.

1.8 How to Maintain a Specification

A specification at an abstraction levels suited for implementation, can be large and complicated. If we in addition take into account the use of (in abstract terms) resources when writing the specification, we will further increase the complexity of the specification.

Hence, we experience the same problems when writing such specification as computer programmers has experienced for years, when the programs to be made becomes big and complicated.

What should we do to solve such problems?

1.8.1 The Need of Making Modules

We can divide the specification into modules. The interaction between modules should be performed in some controlled way. The reason why we would want to partion the specification into modules is to divide a specification into smaller and more manageable parts.

1.8.2 The Need of Controlling and Understanding the Algorithm

In many cases we are able to logically separate the execution sequence, into more than one execution sequence. The control of the execution sequence can jump from one logical execution sequence [3] to another logical execution sequence in a controlled way.

We may use the principles for recursive procedures or co-routines to achieve such logical separation of the execution sequence of an algorithm and still control the interaction between the modules.

In this report we will describe how the Evolving Algebra language can be extended to permit an Evolving Algebra specification to be divided into logical modules.

This extension of Evolving Algebra language has mechanism which support the definition of modules which acts like co-routines or recursive procedures.

1.9 The Work with the Thesis

The work reported in this thesis is divided in three main parts:

1. Implementing an Evolving Algebra interpreter.
2. Make Evolving Algebra specification for compilation and evaluation of lambda expressions.
3. Implementing and testing this specification on the Evolving Algebra interpreter.

1.9.1 The Chronology of the Work

Here I will give a brief description of the chronology of the work with the thesis.

The basic work of implementing the Evolving Algebra interpreter took place in the period from the summer of 1991 to the summer of 1992.

The basic work with the specification case of compiling and evaluating lambda expressions was done in the period of from the summer of 1992 to the the end of 1993.

In the period from the beginning of 1994 to August of 1994, the integration of the specification and the prototype implementation of the specification at the Evolving Algebra interpreter took place. In the same period the last enhencement and test of the interpreter was done.

[3] Here we assume sequential execution.

In the autumn of 1994, some suggestion of extension to the Evolving Algebra language was found. The final work of writing this report was made in last part of the autumn of 1994 and in the first three months of 1995.

This mean that the work with the specification case was done in Evolving Algebra without the extensions suggested in 3, and in such a way that it could be executed at the interpreter without too much difficulties. However, the work with the specification case gave rise to the ideas of the extensions.

Since the Evolving Algebra interpreter was made before the specification was written, the final testing and refinement of the interpreter was made in parallel with the integration of the specification and prototype implementation of the specification case.

The specification of compiling and evaluation of lambda expression could be rewritten using the suggested extension to Evolving Algebra, and it should be done in the some time in the future. Due to the constraint on time and amount of work with the thesis, such rewrite is not done.

Chapter 2

Evolving Algebras

We will in this chapter describe the general requirement of a specification language and then introduce the specification language Evolving Algebra.

2.1 A Specification Language, Requirement

Here we want to discuss the requirements to a specification language.

In short we may state the requirement in the following points:

- The specification written in the specification language has to be easily understood by man.
- It should be possible to run the specification as a program on a computer.
- It should be possible to write the specification without the needs to use decoding and encoding.
- The specification language has to be defined in a way such that we can compute directly on abstract data structures.
- We require that the time can be represented directly.
- We need to write the specification at different abstraction levels. We will require that we can retain the specification at one abstraction level, when moving to the next level of abstraction.

2.1.1 Write an easily understood specification

A specification is intended to be read and understood by man.

We write a specification because because we want to define a language or an algorithm in precise way (more precise than only using a natural language to describe the algorithm or programming language).

Often the specification is intended to be precise enough to be implemented on a computer. And in addition the specification need to be understood both by the programmer and by the users.

Ideally we should be able to write the specification in advance, and then make the implementation of the language and algorithm without changing

the specification. When the implementation is done the specification define what the language or algorithm is supposed to do.

In some cases we may even want to formally prove that the implementation is correct according to its specification.

In all cases mentioned above we expect some persons to be able to understand the specification.

2.1.2 Running the specification on a computer

It is not enough to write a specification only intended to be read by humans. In many cases we need to run the specification on a computer.

At first glance it may seems that we do not recognize the distinction of a specification and an implementation of an algorithm. That is not the case, as we will see shortly.

Before making much effort in really implementing a system it might be desirable to make a prototype implementation in order to give the users some feeling on how the algorithm acts, and in this way give rise to further ideas and requirements of what the algorithm should do. We will want such a prototype implementation also to be a specification, which can be changed and used in the process of developing the algorithm further.

Often we need to meet requirements regarding the resources an algorithm will use. We may require that the algorithm do not use too much time or too much space during normal operation. If we can run the specification on a computer, we may be able to at least detect that the proposed implementation of the algorithm do not meet the requirements regarding time or space. The computation of the specification should give the possibilities to estimate the use of time and space when implemented.

The specification itself may only simulate the real use of time and space of the algorithm when it is run on a computer.

For example we may need to implement an application using assembly language to meet strict requirement regarding use of time in a real time implementation. The specification on the other hand may be implemented in a high level specification language, which only measure the time used in terms of assembly (or machine) instructions performed (where we know the time needed to perform the instructions on the computer to be used).

However, it would be nice if it was no need to make any distinction of implementation and a (detailed) specification of an algorithm. A computer program should ideally be easy to understand for a man, and at the same time possible to execute on the computer without using too much resources. So we can regard it as a goal to make a programming language to also be a specification language to some extent (at least when we are talking about a high level programming languages).

2.1.3 Avoiding the encoding and decoding

In a specification language we will avoid the use of decoding and encoding. If we need to encode and decode data (and may be part of the algorithm), it will clutter up the specification and make it much harder to read and

understand. The reasons why, is that encoding and decoding of data is the same as moving down at a lesser abstraction levels as explained in section 1.4. The same apply to the given operations at the chosen abstraction level.

Consider the stack used in the textual description of the simple calculator in subsection 1.2.1. For the abstraction level chosen, the abstract data structure seems to be the stack, and the abstract operations are such as Push and Pop. We will not want to encode and decode those operations, and we will not want to use some other ways of organize the the stack in our specification.

Still, encoding and decoding is quite common practice, when a specification of an algorithm is made.

Consider the use of a Turing Machine to make a specification. If the specification is intended to be made at an higher abstraction level not supported by a Turing Machine, coding will be needed in order to write the specification. We do not need to make a very large Turing Machine specification before we experience difficulties in reading and understanding the specification.

2.1.4 Computing Directly on Abstract Data Structures

So we want to compute directly on abstract data structures given the abstraction level we choose.

If we can not make a specification at the chosen abstraction level, we may be forced to code the data structure to fit the lower abstraction level permitted by the specification language. Hence, we may loose the opportunity to simulate or compute the use of space measured in units which fit the chosen abstraction level.

2.1.5 Representing the time directly

We require the time to be represented directly in a specification language in steps which also fit the chosen abstraction level.

So if we need to code the operations in order to use a certain specification languages, we may not be able to compute or simulate the use of time at the wanted abstraction level. Consider the stack again. One step of computation at the chosen abstraction level, should be a Push or Pop operation. So we would want to measure the time used in terms of such computation steps.

As an example of how critical the use of time may be, we can take a real time application which shows video on a computer screen. Each picture in the video has to be displayed at a certain fast rate. So a specification has to be made in way such that we can simulate how much it will cost to make each picture in (an abstract terms) of time.

2.1.6 Writing the specification at different abstraction levels

When we make specification for say an algorithm or a computer language, we are seldom able to write a clear and understandable specification and at the same time specify all details required for an implementation of the specification, using just one level of abstraction.

So we often need to use more than one level of abstraction in order to write a specification. It does not mean that we want to mix different levels of abstractions into one huge specification.

Instead we will want to write a specification at a certain abstraction level, and then write new and a more detailed specification at a lesser abstraction level covering the part of the specifications which needs to be detailed or covering the whole specification if necessary. This more detailed specification should be build upon the more abstract specification already written.

To write a specification for a complicated algorithm or language system, we may for example need to write the following specifications:

1. A specification at a very abstract level which gives an overview of what the system or algorithm is supposed to do.

2. A specification at the intermediate level which gives a human insight in how to perform the algorithm.

3. A specification giving some important details of how to implement the algorithm. This specification is supposed to be written such that it can be executed on a computer as a prototype implementation.

If the system is very complicated, we may even need to use more than three abstraction levels in order to complete the specification.

When writing the specification, we need to use the specification written at one abstraction level, when writing the specification at the next (lesser) abstraction level. So we will require that the specification at one abstraction level is retained, when writing a specification at the lesser abstraction level.

If we are forced to rewrite the whole specification, when moving from one abstraction level to the next lesser abstraction level much of the work done may be wasted.

As an example we have the formula:

$$b = cos(a) + sin(b)$$

A rewrite of the whole specification in order to tell how to compute an approximation value of cos(a) and sin(b) will probably be difficult to understand, and difficult to relate to the specification (formula) above.

Another example can be taken from proof theory. When making a proof we will often need to make lemmas, which in turn is used in the main derivation. In proof theory it is examples of how derivation without use of lemmas may be intractable, but if we permit the use of lemmas, the proof can be given using reasonable time and space.

So we will certainly require that the specification can be modularized in some way.

2.2 Core Evolving Algebra

The Evolving Algebra language follows the model of a given abstraction level, where an abstraction consists of:

- The computer medium which is a set of operations.
- The algorithm executed as computer steps using the computer medium.

The definitions of evolving algebra are taken from [BR91c] and from [Bör90a].

Definition 1 (Evolving algebra [BR91c]) *An evolving algebra of a given signature is a pair (A, T) consisting of a finite many sorted partial first order algebra A and a finite set T of transition rules of the signature.*

2.2.1 Evolving Algebra and the Computing Medium

The *signature* in the definition above is called the *static part* of the Evolving Algebra. The signature consists of a set of functions given in the finite, many sorted first order algebra. Those functions corresponds to the operations which constitute the computing medium at the chosen abstraction level.

2.2.2 Defining the Execution of Operations on the Computing Medium

The definitions of the transitions rule below will be called the *dynamic* part of Evolving Algebra. The set of Evolving Algebra transitions corresponds to the algorithm using the computer medium at the chosen abstraction level.

A transition corresponds to the change of the computing medium in the following way:

- A function update change an operation or data element.
- A universe update will in addition add new elements to be used in the algorithm.

Definition 2 (A transition rule [BR91c]) *A transition rule is an expression of the form*

If *condition*
then
$update_1$
$\vdots$
$update_k$

where condition is a boolean expression. If this expression is evaluated to "true" the updates belonging to the transition rule is executed.

There are two kinds of updates, the function update and the universe update.

To express redefinition of a function at one point we define the function update expression.

Definition 3 (Function update [BR91c]) *A function update is an expression of the form*

$$f(t_1, \ldots t_n) := t$$

If we need to add new elements to the universes and use the new elements in one or more function updates we define the universe update expression.

Definition 4 (Universe update) *A* Universe Update *(adapted from [Bör90c]) is an expression of the form:*

$$
\begin{array}{lll}
\mathit{EXTEND}\ U_1\ \mathit{by} & \mathit{temp}(U_1, 1) & \\
 & \cdots & \\
 & \mathit{temp}(U_1, n_{U_1}) & \\
\vdots & & \\
U_m\ \mathit{by} & \mathit{temp}(U_m, 1) & \\
 & \cdots & \\
 & \mathit{temp}(U_m, n_{U_m}) & \\
\mathit{WITH} & F_1 & \\
 & \vdots & \\
 & F_k & \\
\mathit{ENDEXTEND} & &
\end{array}
$$

where

$$U_1, \ldots, U_m$$

is universes to be extended, and

$$\mathit{temp}(U_i, 1), \ldots, \mathit{temp}(U_i, n_{U_i})$$

is temporary constants which holds the new elements to be added to the universe U_i. $F_1, \ldots, F_k$ is function updates within the universe updates.

Constants on the form $\mathit{temp}(U_i, j)$ may occur only within those function updates which is part of the universe update.

See [BR91c] (restated in appendix B) for the formal definition of universe update which is slightly more general. In this definition constants may be used to determine how many elements which are to be added to the universes, and simultaneous execution of function update instances for every new elements added, can be specified.

The interpreter described in appendix B implements simultaneous execution of instances of functions updates for every new element added to the universes.

However, in this report we will not need to perform more than one instance of the functions updates defined within a universe update, so the definition above suffice.

2.3 Specification of How to Compile and Evaluate a Functional Language

In this report we try to specify how to compile and execute the core of a functional language using Evolving Algebra. In essence the evaluation of a functional program is based on the reduction of lambda expressions.

So the plan for the specification will be as follows:

1. Convert the lambda expression to supercombinator expressions.

2. Compile the supercombinator expressions into some form of internal expressions. The form may be some sort of graph representation or some instructions on how to build a graph.

3. Reduce the supercombinator expressions.

The conversion of lambda expressions into supercombinators will only be specified at a *very* abstract level. We will give a much less abstract specification of the two last step, using Evolving Algebra.

2.4 Evolving Algebra, Non-determinism and Parallelism

Evolving Algebra is clearly designed to express non-determinism and parallel processing. So investigating the use of Evolving Algebra to specify non-deterministic or parallel systems should be interesting. However, in this report we do not treat specification of non-deterministic or parallel systems.

2.5 The Evolving Algebra Interpreter

In addition some of the specification is written in a way, such that it can be evaluated in the Evolving Algebra interpreter. This interpreter is written as part of the work with this dissertation. The Evolving Algebra interpreter measure in a abstract way the use of time and space when evaluating the Evolving Algebra specification.

2.6 Gurevich Thesis

The Yuri Gurevich theses [Gur93] about Evolving Algebra says that the Evolving Algebra has the properties as follows in the next paragraph.

For any algorithm defined at a certain abstraction level the following holds:

- We can express the algorithm using Evolving Algebra.
- We can describe the data structure without use of code.
- We can represent the time directly.
- We do not need to change the signature in order to compute the specification.

2.6.1 Different Abstraction Levels

How do the Gurevich theses apply at different abstraction levels? It is not so that the Evolving Algebra is to be used only at one abstraction level (say the most implementation specific level).

In Evolving Algebra we are free to define whichever function we need in our specification. Therefore it is quite natural to believe that Evolving Algebra should be used to specify the computation model at different level of abstraction.

Hence, Gurevich theses is to be applied at specification at arbitrary level of abstractions. So we need to say more about the computation model at different abstraction levels.

At each abstraction level we will have

- A description of an algorithm
- An abstract data structure
- A unit of time

The abstraction levels is divided by abstraction barriers.
Now, we restate Gurevich theses as follows:

- We can express any algorithm at an arbitrary level of abstraction, using Evolving Algebra.
- We can express any data structure at an arbitrary level of abstraction directly, without use of decoding and encoding in Evolving Algebra.
- We can express a time unit in the algorithm as a time unit (step) in Evolving Algebra at an arbitrary level of abstraction.
- We do not change the signature of Evolving Algebra when we compute the specification. That holds regardless of the abstraction levels of the specification.

We do not state any thesis about the *abstraction barriers* which divides the abstraction levels.

Algorithm

What should it mean that we can express any algorithm at any abstraction level. We may try to give a couple of examples.

Say, we want to specify operation on a stack. We should be able to specify operation on stack directly such as:

Top Get the top element from stack.

Pop Pop of the top element from stack.

Push Push a new element on the top of the stack.

Is-empty-stack Is the stack empty?

If we instead have to translate the stack operation to some other type of operation, say operations on arrays, we can not express the algorithm at the abstraction level where the stack concept is used.

We may want to specify operation on a graph. Similar to the stack example we may want to specify operations on directed acyclic graph:

Make-node Make a new node in the graph.

Make-edge Make an edge from one node to another node.

Get-node Get an node from the graph, which is pointed to by an edge.

Get-edges Get all edges from a specific node.

If we are forced to translate the operation on graphs to operation on lists or arrays, we can not use the intended level of abstraction, when specifying the algorithm.

Data structures

As we use abstraction levels when we specify operations, we use the abstraction levels when we specify data structure.

For the stack example we should be able to operate directly on a stack, and should *not* be forced to translate the data structure to something else, say an array representing the stack.

For the graph example we should operate directly on the graph, and not any other data structure, say lists or arrays.

The operations on the selected data structure should be simple and visible (nothing should be hidden).

Time Unit

Given a level of abstraction we want to use the appropriate time unit.

When we operate on stack, we want to count the operations on the stack.

If we instead has to count some other type of operations, e.g. operations on arrays which codes the stack operations, we are not able to represent the time directly.

The same applies on graph, where we should count the operations on graph and not something else.

Change of Signature

At the selected abstraction level the signature should remain unchanged during the computation of the specification. If there is some need to change the signature, the Evolving Algebra transition is specified in a wrong way.

The signature can be regarded as the way of specifying the data structure when using Evolving Algebra.

2.6.2 Implementation

As a consequence of the way Evolving Algebra is defined, we should in principle be able to implement an Evolving Algebra specification on a computer at the desired abstraction level.

If we want to implement a specification at a very abstract level, much of the implementation details will be hidden (left to the programmer).

If we choose to make an specification at an abstraction level where we specify important implementation details, not very much of implementation will be hidden.

2.7 The Turing Machine and Abstraction Levels

In this section we will discuss the Turing Machine related to the needs of making specification at different abstraction levels.

2.7.1 Definition of a Turing Machine

We take the definition of a Turing Machine from [FN84]:

Definition 5 (A Turing Machine) *Let* $\Sigma = \{s_0, \ldots, s_n\}$ *be a set called the* alphabet *and let* $Q = \{q_0, \ldots, q_m\}$ *be a set called the* states. *Let* Q_H *be an extra state and let* $Q_H = Q \cup \{q_H\}$.

A Turing Machine M *over* Σ, Q *consists of three functions* K, F *and* D *where* $K : Q \times \Sigma \rightarrow Q_H$, $F : Q \times \Sigma \rightarrow \Sigma$ *and* $D : Q \times \Sigma \rightarrow \{R, L\}$, *where* R *is Right and* L *is Left.*

2.7.2 Express an Algorithm using a TM

How can we express an algorithm when using a Turing Machine?

When a step is performed on a Turing Machine the following operations are performed:

- Enter a (possibly new) state.
- Write a symbol on the Turing Tape.
- Move the tape to the left or to the right.

From the above, it is clear that only a certain abstraction level for a TM is given (and this level is not very abstract).

2.7.3 Express an Data Structure on a TM

The data structure is (a possible infinite) tape, consisting of symbols. Again we have a given abstraction levels with regards to the data structure. To express operations on common data structures (such as graphs) we will need to code the data into symbols on the tape in order to represent the data structure.

2.7.4 Express Time Units on a TM

At *each step*, we enter a state, write a symbol on the Turing tape and move the tape to the left or to the right.

Again the time unit is fixed at the given abstraction levels. If we perform operations at higher abstraction levels, such as making a new node in a graph, we need to translate the operation to many steps in the Turing Machine. So we can not directly count operations on a higher abstraction level than the level given by the Turing Machine.

2.7.5 The Turing Theses

Consider the Turing Theses. Actually we have two Turing thesis:

Definition 6 (Turing Thesis I) *We can express any algorithm using a Turing Machine.*

The thesis above seems to capture our basic understanding of the power of expression which the Turing Machine has. We may one day find a fundamental new way of expressing an algorithm, which may invalidate Turing Thesis I. But, to day the Turing Machine can be regarded as one of the basic way of expressing an algorithm.

Definition 7 (Turing Thesis II) *We can express any algorithm using a Turing Machine* step by step *using a Turing Machine.*

If we add the words *step by step* to the Turing Thesis I, the thesis does not hold. We are simply not able to express algorithm which operates on more complex data structures than the one way tape (e.g. tree or graph), using the Turing Machine if we in addition require that the algorithm should work step by step. A graph has to be translated to some expression which can be stored on the one ways infinite tape, using the defined TM alphabet. The requirement of translation (or "coding") from a certain abstraction level to a lesser abstraction level, breaks the "step by step" assumption in the Turing Thesis II.

The strong Turing Thesis corresponds to the Gurevich Thesis (See section 2.6)

Chapter 3

Abstractions and Modules

3.1 Computing Models

When we write a specification we will use some models of computation.

When thinking about computation models, we may or may not consider use of resources, such as:

- Use of space (abstraction of computer memory).
- Use of time (abstraction of computer time).

We may want to get an abstraction, such that we do not consider the use of resources, or we may use a specification language which do not allow specification of resources to be used in the computing. If we use such a model, the computation model use:

1. Unlimited space
2. Unlimited time

When we refine the specification (or make an implementation to be run on a computer) we have to take into account the use of time and space. That is so, because in the real life we will never have an unlimited amount of resources, and we will want to know the use of those computing resources. Therefore it is important to specify the use of time and space, and in addition keep track of the use of time and space in a prototype implementation or a real implementation. If we are not able to do talk about the use of time and space, it is quite difficult to say very much which is meaningful about the process of computation.

If we use a specification language which does not permit use of the resources to be part of the specification, the refinement towards an implementation which uses specific amount of resources might be made in a more or less arbitrary way.

Even worse, we might not know if it is possible (or feasible) to implement the specification.

3.1.1 Including the Use of Resources in the Specification

We may choose to specify the use of resource, such as time and space at any abstraction levels of the specification.

Then we take into consideration the use of resources in the whole specification process from the highest abstraction level possible, toward the implementation of the specification.

A specification language which allow us to specify the use of resources should give us better precision compared with a specification language which does not allow us to specify use of resources.

3.2 Abstraction Levels and Making Modules

The Evolving Algebra specification language lets us specify use of resources such as use of time and space in an abstract way. We may also make quite detailed specification, or we may choose to make the specification at a more abstract level.

However, questions such as how to maintain different abstraction levels, and how to make gradually refinements of the specification arise.

In a specification language which does not permit us to talk about use of resources, or which is bound to a certain (low) abstraction levels, it will not be so natural to focus on those problems.

3.2.1 Abstraction Barrier

How can we make an abstraction barrier in Evolving Algebra? The way to make an abstraction barriers, is to specify appropriate functions, with appropriate signatures, and let the functions be the abstraction barrier for the chosen level of abstraction.

We can do it for a chosen abstractions level, and make the specification at this level.

The problems arise when we want to refine the specification from the given abstraction level. If we do not want to write the whole specification again, choosing another abstraction level, we have no clear way of how to proceed.

This type of problems do not come as a surprise. Similar problems had to be solved in programming languages suitable to implement large systems.

Making procedure definition and invoke the defined procedures is very common in higher level programming languages.

When implementing a larger system, we will have to make modules in order avoid an unmanageable size of complexity.

We may also want to divide the implementation in modules at the same abstraction level. Such modules should cooperate and exchange information. Co-routines can be seen as "procedures" which invoke each others.

What about a specification language? Should we be concerned about such problems, when making a specification? The specification language should be defined in a way such that we can tell exactly how a procedure call are to be implemented in a programming language. Do we not intermix

the specification language and the target programming language we want to specify?

The answer to the question posed is discussed below.

If the specification language just tell us what should be done at a certain (high) abstraction level and in addition we may not be able to specify use of resources, the need of dividing the specification may not arise. The problem of giving an understandable and precise enough specification will probably be the only problem which we try to solve, using such a specification language.

On the other hand, if the specification language permit us to talk about the resources and in addition permit us to choose the abstraction levels, we may be able to make a very detailed, large and complex specification. Then, we will get the same problems when making the specification as we experienced when using programming languages.

So when we try making abstractions at different abstraction levels and in addition we also specify the use of resources, we may make a too complex specification, or we may need to rewrite the whole specification when we choose a new level of abstraction. Therefore we may need to make modules in order to embed the specification at a lesser abstraction level into the specification at the higher abstraction level. We may also want to divide the specification at a certain abstraction levels into modules in order to avoid a too complex specification.

3.2.2 Modules

In this subsection we will say something more about specifying a system in parts.

3.2.3 Co-routines

We may also divide a system in modules at the same level and let the modules co-operate. To do so, we will specify co-routines. Co-routines may be regarded as routines which call each other and suspend its execution until the other routine has done its task, and then resume the execution.

3.2.4 Making Procedures

We may divide a system into a hierarchy, making a main program, or (hopefully) a main specification, and make subroutines to execute functions called by the main program (or specification). Subroutines may in turn call other subroutines.

3.3 Evolving Algebra and modules

The sequential Evolving Algebra has the following main property with regards to modules:

- An Evolving Algebra specification is entirely within only one *name-space*.

- The Evolving Algebra execution control consists of only one *sequence* of execution steps.

From the above we can see that no possibilities of dividing the specification into modules is build into the sequential Evolving Algebra. So we may want to discuss how to extend Evolving Algebra to support specification of different type of modules in a systematic way.

The extensions to the Evolving Algebra has to be made in a simple and understandable way, such that the specification language is not cluttered up with mechanisms difficult to understand and reason about.

3.3.1 Embedding the Modules into Subroutines

If we want to make modules to be invoked like procedure calls in conventional programming language we need to find a way to embed modules at abstraction level n into the module at abstraction level $n-1$ invoking the modules (abstraction level 1 is the most abstract level of a specification).

We have to make decision about the following issues:

- The name-space has to be defined
- The change of control when invoking a subroutine and returning from the subroutine.
- The exchange of data between the invoking environment and the subroutines.

We have to make specification of which names the subordinate module will import from the invocation environment, and which names can be exported to the invocation environment after the subroutine is finished.

We may specify where in the invoking sequence the control is transferred to the subroutine, and the point of return after the subroutine is finished.

3.3.2 Dividing a Specification into Co-routines

If we are going to make modules in a specification language which acts like co-routines in a specification language, we may require the following:

- The exchange of information between the modules have to be specified in a simple and clear way.
- The change of control between the invoking environment and the co-routines must be specified.
- The name-spaces has to be defined

3.3.3 Naming Strategy

We will often want to use new name space when executing a subroutine or co-routine definition. This name-space should be different from the calling environment and other instances of the co-routine or subroutines and from other routines which is to be executed.

In a module in the specification language it is natural to make a new name space for the module. We need to do the task in a simple and understandable way. One way to do so, could be to assign a fresh unique name to each instance of the defined modules and append the instance module name to all names used in the module specification.

3.3.4 Import and Export of Names

If we choose a new name space for a module, we want some of the names to be used strictly within the modules, when other names may be exported or imported by modules in order to exchange data between modules.

So we need a simple way of defining export from a name-space and import to a name space in the extension of Evolving Algebra.

3.3.5 Execution Control Strategy

When executing a subroutine, the sequence of execution in the invoking environment will be broken. The control is given to a new sequence of execution given in the subroutine. This sequence is then executed (possibly broken by invocation of other subroutines) until the end, and the execution resumes in the invoking environment at the point after invocation of the subroutine.

When executing a co-routine, the sequence of execution is given to the co-routine and the co-routine is executed until a point where a new co-routine or the co-routines invoking environment is given the control. The execution of the co-routine is then suspended. When the co-routine is invoked again the execution continue after the point where the co-routines gave up the control the last time.

Eventually the co-routines is finished and receive the status as inactive.

So, we have to extend the sequential Evolving Algebra, such that more than one execution sequence can be specified, and such that change of control from one execution sequence to another together with specification of return points can be specified.

3.4 Specify the Control Strategy for Different Types of Modules

In this section we will look closer into how to implement a control strategy for Modules in Evolving Algebra which is divided into modules.

Each co-routine or subroutine, and the main program can be seen as a logical unit. A routine is generating a logical executing sequence, during the routine is run. A jump to subroutine or to a new co-routine can be seen as a suspension of the invoking execution sequence. And this sequence is resumed when the control returns to the invoking routine.

So, we will look closer into the change of control when using subroutines or co-routines.

3.4.1 Making Co-operating Modules

In a programming language, we often want to specify co-routines in order to make parallel programming possible. That is not the reason why we could want to extend the specification languages to ease the specifications of co-routines in addition to procedures. The reason to make co-routines as part of a specification languages, is that some specification of computation problems may be difficult to express without use of modules which works together at the same level.

When making the specification of co-routines we have to specify the point where a suspended co-routine are to be invoked next time, and the return point when a detach is invoked from a co-routine. We may have a situation where one (instance of) a co-routine is invoked. This instance gives the control to another (instance of) co-routine, and the return point to the invoking routine for the first co-routine is inherited by the next co-routine. So if the next co-routine says detach, then the control of execution is transferred to the invoking routine.

3.4.2 The Game Strategy Example

To investigate a little further the use of co-routines, we will use game strategy as an example (See [Wan77]).

Suppose we want to try different strategies of a game. Two modules act as players, where the first player makes a move, and then gives the control to the other player which makes a move, and then gives the control to the first player. When one of the player wins the game, the control is given to the module which started the two players modules.

How could we express the move of control between the players using an extension to Evolving Algebra.

Our first approach could be to extend the Evolving Algebra by making Modules and let the co-routine statements be a part of Evolving Algebra.

Then, we would get the following Extended Evolving Algebra transitions for the modules:

```
Module: START THE GAME
if not(game-finished)
then
   Call(PLAYER-ONE)
   game-finished=True;
End Module
```

This transition issue the call statement to start the game.

```
Module: PLAYER-ONE
if game-won
then
   Detach;
```

The transition above finish the game if the player one module beats the player two module.

```
if not(game-won)
   game-status:=next-move-player-one;
   Resume(PLAYER-TWO);
End Module
```

Here the player one makes a move and gives the control of execution to player two.

```
Module: PLAYER-TWO
if game-won
then
   Detach;
```

The transition above finish the game if the player two module beats the player one module.

```
if not(game-won)
   game-status:=next-move-player-one;
   Resume(PLAYER-ONE);
End Module
```

Here the player two makes a move and gives the control of execution to player one.

3.4.3 Specify Call, Detach and Resume

What should we mean with such statements as Call, Detach and Resume? For people which know about object oriented programming, the meaning is quite clear and could be expressed as follows:

Call This statement starts the co-routine given as argument to the Call statement. The name of the routine which issue the Call statement (and eventually other necessary information) is transferred to the co-routine called as the return information.

Resume This statement gives the control of execution from one co-routine to another co-routine which is given as the argument to the Resume statement. The return information held by the co-routine issuing Resume is transferred to the co-routine which is given the control of the execution.

Detach Returns the control to the routine given in the return information (the Module which issue the Call statement).

Is it possible to find mechanisms to express more directly in a specification language the meaning of Call, Resume and Detach, with regards to the control of execution? Since we are working with the specification we want as simple and flexible mechanisms as possible to express transfer of controls to another execution sequence.

In object oriented programming, each execution sequence will maintain for itself, which statement to execute next[1]. So exchange address information, such as the next transition to be executed, between modules is not necessary. So we have to keep track only of the modules involved, when executing co-routines.

Then, we may translate the Call, Resume and Detach as follows:

```
Call(<co-routine>) ==> Invoke(<co-routine>,<it-self>)
Resume(<co-routine>) ==> Invoke(<co-routine>,
                            <inherited-return-module>)
Detach ==> Invoke-return(<inherited-return-module>)
```

The second argument given to the Invoke statement, is supposed to be the name of the module which gets the control when Invoke-return is called.

The Game Strategy Example

The Game Strategy Example where Evolving Algebra is extended by using Invoke and Invoke-return (instead of Call, Resume and Detach) will be as follows:

```
Module: START-THE-GAME
if not(game-finished)
then
   game-finished=True;
   Invoke(PLAYER-ONE,START-THE-GAME);
fi

if game-finished
then
   normal-termination:=True;
fi
End Module

Module: PLAYER-ONE
if game-won
then
   Invoke-return(START-THE-GAME);
fi

if not(game-won)
   game-status:=next-move-player-one;
   Invoke(PLAYER-TWO,START-THE-GAME);
fi
End Module

Module: PLAYER-TWO
if game-won
```

[1] Due to Ole Johan Dahl, who first discovered this fact.

```
then
   Invoke-return(START-THE-GAME);
fi

if not(game-won)
   game-status:=next-move-player-one;
   Invoke(PLAYER-ONE,START-THE-GAME)
fi
End Module
```

3.4.4 Making Subordinate Modules

We may want to make Evolving Algebra specification such that a module which calls a subordinate module. The subordinate module executes its task like a subroutine in a conventional programming language and returns to the calling module when the task is executed.

In the calling module we may think of an Evolving Algebra transition calling the subordinate module as follows:

```
Module: CALLING ROUTINE
if test
then
...
   Call-sub(SUBORDINATE-MODULE)
....
End Module

Module: SUBORDINATE-MODULE
....
If test-for-return
then
...
   Return
End Module
```

Also in this case we would want to simplify the Call-sub and Return statement.

So we translate this to statement as follows:

```
Call-sub(<co-routine>) ==> Invoke(<sub-routine>,<it-self>)
Detach ==> Invoke-return(<calling-module>)
```

So the calling and subordinate modules will be written as follows:

```
Module: CALLING ROUTINE
if test
then
...
   Invoke(SUBORDINATE-MODULE,CALLING-ROUTINE)
...
```

```
End Module

Module: SUBORDINATE-MODULE
....
If test-for-return
then
...
   Invoke-return(CALLING-ROUTINE)
End Module
```

3.4.5 Making Instances of the Players

In the preceding example we did not make use of explicit created instances of a module. Instead we defined different modules for each of the two players. In the Invoke and Invoke-return we could use the name of the modules as parameters, since we had only one instance of each module.

Now, we will want to create more than one instance of each module. Since we will not be able to refer to every instance directly, we will need some new mechanisms in the Extended Evolving Algebra to refer to the instances created:

Inherited-module A command which retrieves the module instance given as the second argument to the Invoke command.

Itself A command which retrieves the name of the module instance itself, when it is executing.

In addition we will need to extend Evolving Algebra with a mechanism to create new instances of a module. So we may think of instances of a module as elements added to special universes which has the same name as the name of the module definitions.

So we may define a special new instance update which create a new instance in the same way as a universe update. We will extend the Evolving Algebra with the following definition:

Definition 8 *A* instance update *is an expression of the form:*

$$
\begin{array}{lll}
\textit{MAKE INSTANCES of } M_1 \textit{ by} & \textit{instance}(M_1, 1) & \\
 & \cdots & \\
 & \textit{instance}(M_1, n_{M_1}) & \\
\vdots & & \\
 & M_m \textit{ by} & \textit{instance}(M_m, 1) \\
 & & \cdots \\
 & & \textit{instance}(M_m, n_{M_m}) \\
\textit{WITH} & F_1 & \\
 & \vdots & \\
 & F_k & \\
\textit{END MAKE INSTANCES} & &
\end{array}
$$

where

$$M_1, \ldots, M_m$$

is module definitions, and

$$instance(M_i, 1), \ldots, instance(M_i, n_{M_i})$$

is temporary constants which holds the new instances of the module M_i. $F_1, \ldots, F_k$ is function updates within the instance updates. Constants on the form $instance(M_i, j)$ may occur only within those function updates which is part of the instance update. The module name to be used in the instance update must be defined in the Evolving Algebra specification.

The Game Strategy Example Revisited

We can now restate the game example using two instances of the same module. The new mechanisms introduced above will be used.

The START-THE-GAME module exists in only one instance during a run, so it is not strictly necessary to use the Inherited-module command in this example. Instead we could chosen to refer to the instance by using the name of the module in the Invoke and Invoke-return call argument.

However, we will later need the Inherited-module command, when we are going to specify recursive calls of a module below, so we will use it in the game example as well.

```
Module: START-THE-GAME
if not(game-finished)
then
   MAKE INSTANCES of PLAYER by instance(PLAYER,1)
                               instance(PLAYER,2)
   WITH
        next-game-instance(instance(PLAYER,1)):=instance(PLAYER,2);
        next-game-instance(instance(PLAYER,2)):=instance(PLAYER,1);
        first-instance:=instance(PLAYER,1);
   END MAKE INSTANCES
   Invoke(first-instance,Itself)
   game-finished=True;
End Module

if game-finished
then
   normal-termination:=True
fi

Module: PLAYER
if game-won
then
   Invoke-return(Inherited-module);

if not(game-won)
```

```
   game-status:=next-move;
   Invoke(next-game-instance(Itself),Inherited-module);
End Module
```

An Instance

We need to approach the meaning of the term *instance* of a module.

An instance of a module should have the following components [2]:

- A pointer to the module definition.
- A unique instance identifier.
- The part of the Evolving Algebra which can be used to create the own state for the instance, When a new instance of a module is created, an initial state is set for the instance of the module.

A definition of an instance of the module is given later in this chapter.

Even if we do not explicit create instances, a definition of a module imply the creation of at least one instance of the module.

3.4.6 Recursive Calls

To be able to recursive invoke modules, we need to explicit create new instances of the module. So we will try to specify how we can specify recursive calls using the extension introduced so far.

Normal Recursion

```
Module DEMO-REC
....
if invoke-itself
then
   MAKE INSTANCES of DEMO-REC by instance(DEMO-REC,1)
   WITH
       Invoke(instance(DEMO-REC,1),Itself);
   END INSTANCES
fi

if return-from-me
then
   Invoke-return(Inherited-module)
End Module
fi
```

[2] A definition of an instance of the module is given later in this chapter.

Tail Recursive Call

To describe how to optimize the tail recursive call using extended Evolving Algebra, we replace Itself as the second argument to Invoke by Inherited-module. That means we will not return to the invoking instance after finished with the new instance of the module invoked by a recursive call. The calling instance can therefore be safely discarded, when issuing the tail recursive call. So we optimize the tail recursive call as we want to in Extended Evolving Algebra.

```
Module DEMO-TAIL-REC
....
if invoke-itself
then
   MAKE INSTANCES of DEMO-TAIL-REC by instance(DEMO-TAIL-REC,1)
   WITH
       Invoke(instance(DEMO-TAIL-REC,1),Inherited-module);
   END INSTANCES

if return-from-me
then
   Invoke-return(Inherited-module)
End Module
```

As en example of a tail recursive call which needs to be optimized, we specify the top level of an interpreter:

```
Module MY-INTERPRETER
if not(user_finished)
then
    result:=eval_command(command)
    MAKE INSTANCES of MY-INTERPRETER by instance(MY-INTERPRETER)
    WITH
        Invoke(instance(MY-INTERPRETER,1),Inherited-module)
    END INSTANCES
fi

if user_finished
then
   Invoke-return(Inherited-module)
fi
End Module
```

We can assume that the calling instances of the Interpreter which we will not return to will be discarded, such that not more than two instances[3] of

[3]The two instances are the calling instance, and the new instance to be invoked recursively. The initial start of the interpreter might also be regarded as a third instance of the interpreter module which we should return to just before we exits from the interpreter. However, we will define a start module to do the initializing and termination of the interpreter.

the interpreter will exist at the same time. So the user can safely use the interpreter specified above without fear of exhausting the memory on the computer.

As a general rule, we can assume that instance of a module which is not referenced by any other instances of a module is garbage instances which is to be discarded.

If a tail recursive call without optimizing had to be used, the maximum number of instances which exists at the same time would be equal to the number of commands executed by the interpreter. In this case the interpreter would soon or later run out of memory.

To make the initial start of the interpreter we may define a module as follows:

```
Module START-STOP-MY-INTERPRETER
if not(user-finished)
then
    state:=set-initial-state
    MAKE INSTANCES of MY-INTERPRETER by instance(MY-INTERPRETER)
    WITH
        Invoke(instance(MY-INTERPRETER,1),Itself)
    END INSTANCES
fi

if user-finished
then
   state:=set-finished-state
fi
End Module
```

Here we start the interpreter, telling that the interpreter should return to the START-STOP-MY-INTERPRETER when finished.

The specification above is an abstract description of an interpreter. We have not made any specification of what the interpreter is supposed to do.

3.4.7 Name Spaces and Modules

We will need to define the name space for an instance of a module, for all instances within a module definition and for more than one module definition.

Functions Created for an Instance

When defining a module we will want to let some of the function be used only in one of the instances of the module. Such functions are declared as private in the module. Other functions may be shared by more than one instances of a single module definition.

So we extend the definition of a module to include the signature of all functions which is private to an instance of a module:

```
BEGIN PRIVATE FUNCTIONS
....
example_get: DATA --> RESULT
....
example_start==Initial-value
....
END PRIVATE
```

The statements which initialize functions private to an instances is run once for each new instance.

Functions Created for All Instances Within a Module

In addition we may want to include functions to be shared among more than one instance of the module in the module definition:

```
BEGIN SHARED WITH ALL INSTANCES
....
get_shared_inst_data: DATA --> RESULT
...
example_start==Initial-value
...
END SHARED
```

The statements which initialize functions shared by all instances is run once when the running of Evolving Algebra starts.

3.4.8 Components of a Module

So an Evolving Algebra Module will consists of the following items:

- The name of the module definition.
- The signature for the private functions within an instance.
- Statements to set the initial state for a new instance of a module.
- The signature for the shared functions for all instances of the module.
- The statements to initialize the shared functions.
- The Evolving Algebra transitions using the Evolving Algebra extensions defined above.
- The end mark of the module.

3.4.9 Function to be Used Within More than One Module Definition

We will want some functions to be used within more than one module definition. To avoid any misunderstanding we will prefer to define such signatures outside any module definition. The definition may be as follows:

```
SHARE WITH MODULES: MY-INTERPRETER, START-STOP-MY-INTERPRETER
...
get_shared_with_modules: DATA --> RESULT
..
END SHARED
```

All functions shared between more than one global modules, are also shared among all instances of the modules. We have no way to explicit name the instances which is created dynamicly, so we do not need to consider the possibility of sharing functions with for example only one instance from module A and one instance from module B.

We may also want to define functions visible to all modules:

```
GLOBAL
...
get_global: DATA --> RESULT
...
END GLOBAL
```

Global functions is visible everywhere.

Set Initial values of Shared Functions

All functions shared between more than one instances that needs initial values, will receive the initial values when the Evolving Algebra specification starts to run.

3.4.10 An Instance and a Revised Definition of the Evolving Algebra State

We are now ready to define what an instance of an Evolving Algebra module means:

Definition 9 *An instance of an Evolving Algebra module consists of the following parts:*

- *Pointer to the part of the Evolving Algebra specification within the module.*
- *A unique identifier of the instance.*
- *The special defined constants which helps to control the jumps between instances at execution time.*
- *The part of the finite, first order and many sorted algebra defined within the private functions part of the module definition.*

The specification of the module itself is shared among all instances of the module.

The unique identifier of the instance may be constructed as pair consisting of the module name and a unique identifier for all instances of the

module, provided the module names are distinct within the Evolving Algebra specification.

The special constants to control the jumps between instances defined, is: *Itself* and *Inherited-module*[4].

When an instance is created a new copy of the private functions part of the module is made [5]

The private functions are initialized as defined in the module when a new instance is created.

The private part of the algebra will be modified during the run of the instance when the transitions defined in the module is running.

Discarding an Instance of a Module

We have given example of how we can optimize a tail recursive call by giving `Inherited` as the second parameter of the `Invoke` instead of `Itself`. This way of optimize the tail recursive call relays on the following assumption:

> Instances of modules where the system can not return to, is to be discarded.

So, which instances should be discarded? The answer is simple. All instances which is not referenced from other instances can and should be discarded.

An implementation could mark the instances of a module which is not to be discarded.

So an instance which is:

- Referred in non-local constants or functions should be marked as referenced before an Invoke statement is executed.
- Referred in the second parameter to Invoke should be marked as referenced before the Invoke statement is executed.

If needed, we may eventually introduce a construct in Extended Evolving Algebra to explicit discard an instance of a module. However, if we do so, we have to make sure to remove every reference to an explicit discarded instance of a module from the functions and constants, when we implement an Extended Evolving Algebra interpreter.

3.4.11 A Definition of State in Extended Evolving Algebra

The definition of the state has to take into account the fact that part of the Evolving Algebra specifications may be shared between instances and modules. So we need to restate the definition of an Evolving Algebra state as follows:

[4] The list of special constants can be extended, if needed.

[5] We do not require the private part of the algebra to be present when defining a new module. However, if more than one instance of a module is made, the absence of a private part do not make the instances what we expect it to be.

Definition 10 *A state (within an instance of the module) is the finite, first order many sorted algebra consisting of all functions visible when an instance of the module is executed.*

3.4.12 Start of the Execution of Modules

Since we can only use Invoke and Invoke-return to jump between modules, we need to define one module as the main module.

So we introduce a clause:

```
MAIN MODULE is STARTING-MODULE
```

The module `STARTING-MODULE` is the module which starts the execution of the Evolving Algebra definition.

3.4.13 Components of The Extended Evolving Algebra Specification

We can now write an sequential Evolving Algebra specification using modules. An Evolving Algebra specification with modules consists of the following components:

- Signature for one global set of functions.
- Signature for sets of functions which is to be shared by more than one module definition.
- Statements which sets the initial state of the shared and global functions.
- Clause telling the name of main module which initiates the execution of the Evolving Algebra definition.
- All module definitions.

We will not permit any transition to be defined outside a module.

3.4.14 Threads

We have separated the *jump of control* between the Evolving Algebra execution sequences defined as modules and the *name-spaces* which is to be used in modules.

One reason for this separations is to be able to easily express one of the type of multithreading in an Evolving Algebra specification.

Multithreaded programming is to set up more than one logical execution sequence (thread), and let all execution sequences be executed in almost the same name space. Some data has to be private even in multithreaded programming.

To apply the technique outlined above on threads, the threads need to be the type of threads where a thread itself issues the command which gives the control to another thread.

If an external process (or thread) is used to transfer the control between threads, we can (probably) not specify this type of threads using the extension to Evolving Algebra as outlined above.

3.5 Other Language Constructs

A person used to make program in higher order languages, will probably miss language constructs such as

- While loops
- Nested If-else constructs and more general Case constructs
- Lack of the possibilities to explicit create a set of assignments to be executed in sequence.

What about the lack of such usual language constructs in a specification language?

Provided we have solved the problems with regards to partion the specification into modules, the lack of language constructs listed above does not seem to be very serious.

As an example, we may take the iteration construct. If we find a way to extend the specification language to make subordinate modules, we may specify the invocation of a module in place of an iteration construct. The module will contain the iteration body, and the guard in Evolving Algebra can be made such that the guard is true as long the iteration step should be executed.

With regards to more general If, Else or Case constructs, we may simply extend Evolving Algebra specification language with such constructs, if needed.

If we prefer to let the Evolving Algebra language be as simple as possible (after introducing modules), it is not difficult to translate If, Else and Case constructs to the simple If statement.

We may also extend Evolving Algebra to execute certain function updates in a sequence, instead of executing those updates simultaneous.

The function updates to be executed in sequence can also be replaced by a composite function construct. As an alternative the set of transitions can be guarded by a status constant which determines the sequence of the updates to be executed.

Part II

Compilation and Evaluation of a Functional Language, a Specification

Introduction

This part will contain the specification regarding compilation and evaluation of a functional language. Two types of evaluations and compilations are specified:

- The template instantiation machine.
- The G-machine.

The template instantiation machine approach emphasize the aspect of evaluation, while the G-machine emphasize the aspect of compiling a functional language.

This part will consists of the following:

- Discussion of the how to make specification in Evolving Algebra at the desired abstraction level (See chapter 4).
- The specification for compilation of supercombinator definition. A supercombinator definition is part of a functional language (See chapter 5).
- Specification of how to evaluate a supercombinators in form of graph, or in form of instruction to make a graph (See chapter 6).
- Extending the specification to handle strict and lazy arguments. (See chapter 7)
- How we could use the module extension of Evolving Algebra to improve the specification made in the previous chapters (See chapter 8).

Chapter 4

Maintaining Abstraction Levels

In this chapter we will discuss the problem and the requirement of defining abstraction levels when we write a specification.

In addition we will begin the description of a specification for a lambda compiler and the lambda evaluator at a very abstract level.

Together with the specification given in the two subsequent chapter, we experience the problem which we might have, in finding suitable abstraction levels for the specification to be made.

The problem of finding a suitable abstraction level may be connected to the problem of dividing a specification into suitable modules.

4.1 The Problem of Defining Abstraction Levels in a Specification

It seems to be the case, that the problem of defining different abstractions levels is overlooked, when we try to make specification for algorithms or programming language. So we will take a look on this problem, before we jump to the Evolving Algebra specification of a Lambda Compiler and Evaluator.

4.1.1 The Turing Machine as a Specification Language

As explained in section 2.7 we can not use a Turing Machine to make a specification at an arbitrary abstraction level (unless we accept to encode and decode part or the whole of the specification). So, when we do not want to use the abstraction level that suits the Turing Machine, we face the problem of finding another suitable specification language.

4.1.2 A Graph Machine

A. Kolmogrov designed a machine similar to the Turing Machine, except that this machine operates on a graph, instead to be restricted to operate

on an infinite tape. So this graph machine permit a more abstract definition than a Turing Machine specification.

This machine gave rise to the Evolving Algebra as a specification language [Gur93].

4.1.3 The Evolving Algebra

The Evolving Algebra specification language generalize the ideas from Kolmogrov's Graph Machine. The static part of the Evolving Algebra is used to define a suitable data structure (e.g. a graph), and the dynamic part operates on the data structure defined.

In this way we open up the possibility of defining the (at least to some degree) abstraction level, by defining a suitable data structure corresponding to the abstraction level.

4.2 Other Examples Compared with of the Lambda Compiler and Interpreter Specification

In this section we will compare the specification of the Evolving Algebra specification of a lambda compiler and interpreter in this report with the following examples:

1. The Evolving Algebra specification of the C language.
2. The Evolving Algebra specification of the Prolog Compiler.

4.2.1 The C-language Specification Example

In [GK93] the semantic of the C language is specified.

The Use of Transitions

The transitions is written in a form of nested rules dividing the different cases using `if`, `elseif`, `else` and `endif`.

Only function updates are used within the transitions. No universe extension or contraction is used.

The use of Transitions in The Lambda Compiler and Interpreter Specification

The transitions use the simple `if` to state the precondition in the transitions. Universe extension and function updates are used within the transition.

The Type of Evolving Algebra Functions Used

The following types of function is used in the C-language example:

Dynamic function Updates is permitted on a dynamic function.

Static function Updates is not permitted on the static function.

External function The value of the function is permitted to vary each time the transition which use the function is performed. This type of function permits communication with the world outside the specification domain (e.g. input from the user will be specified using an external function).

It is worth noting that the use of external functions seems to replace the use of universe extension. In the specification the allocation of memory is specified with help of an external function.

It seems that the specification is based on the assumption that the information inherited from the syntax at compile time is build into some of the static functions. No Evolving Algebra transitions is specified for the while, do-while and for statement. The same is the case for labeled and compound statement.

The Type of Functions used in The Lambda Compiler and Interpreter Specification

Mainly dynamic and static function are used. The author was not aware of the possibility of using external functions when the main part of specification was written [1].

The C-language Specification

The specification is divided into four algebras which specify:

1. Handling C statements.
2. Evaluating expressions.
3. Allocate and initialize memory.
4. Handling function definitions

The first algebra which handles the semantics of C statements, specify the semantics for following types of statements:

> Expression, selection, iteration, jump, labels and compound statements.

The second algebra which covers expressions, specify the semantics of conditional, logical expression, general mathematical expression, assignment, increment, addressing expression, dereferencing, array reference, casting, function invocation, identifiers, and a sequence of expressions (where all expressions in the sequence is evaluated and the value of the last expression in the sequence is returned). In addition the semantics for the `sizeof` operator, casting expression and constant expression, `struct` and `union` references and bit fields references is defined.

[1] The use of external function is not according to the usual definition of a function, since an external function may return different values when invoked at different time.

The third algebra which handles the semantics of initializing and allocation of memory specify the semantics of declaration, automatic variables and non local jumps, identifiers and initializers.

The fourth algebra for function definitions specify the semantics for function invocation (for the caller and callee) and the semantics for global variables.

The Lambda Compiler and Interpreter specification

The specification is divided into the following main parts:

- Compilation and evaluation of the template instantiation machine.
- Compilation and evaluation of the G-machine.

Abstraction Levels Used in the Specification of the C language

The specification of the C programming language does not seem to have many levels of abstraction. We have the specification of expression which may be regarded to be at one level of abstraction. This specification is in turn embedded in the more general specification of a statement.

The algebras which specify how to allocate and initialize the memory and how to handle function definition, completes the specification at the same abstraction level as the specification of expressions. A stack is introduced in the specification of function definition in order to specify how to resume from a function call. The introduction of the stack does not change the level of the abstraction (at least according to the author opinion).

When necessary part of the C language is left unspecified by using oracles. Oracles is Evolving Algebra function which takes an unspecified value from the world outside the domain of specification. A value which an oracle gets, may vary every time an Evolving Algebra specification is performed.

Abstraction Levels Used in the Lambda Compiler and Interpreter specification

The lambda compiler and interpreter specification has two distinct abstraction levels (See subsection 4.2.3).

4.2.2 The Prolog Specification Example

A full specification of Prolog in Evolving Algebra can be found in [Bör90a], [Bör90b] and [Bör90c]. The core Prolog and built-in predicates for the control part of the Prolog is specified in [Bör90a], and the rest of the built in predicates is specified in [Bör90b] (for files, terms, arithmetic and input output) and [Bör90c] (predicates for database manipulations).

The Use of Transitions

A transition has form of a simple `if` statement. Function updates and universe extension is used in all three reports which together gives the specification.

Universe contraction is used in the first of the reports (See [Bör90a]) of the specification. So the reclaim of memory after computation of a goal, or backtracking after failure to compute a goal is made explicit with use of contraction in the specification.

The Use of Transitions in The Lambda Compiler and Interpreter Specification

In the specification used in this report, function updates and universe extension are used within a transition which has form of a simple `if` statement. Contraction is not used. The universe extension is defined such that elements to more than one universe can be added and referred to simultaneous within one universe extension.

The reason why not using contraction is that we can assume that the recycle of memory is done by a general garbage collector, which can be abstracted out from the specification.

The Prolog Specification

The specification is divided into two main part:

1. The Core Prolog and the built-in control predicates.
2. The Rest of the built-in predicates in Prolog.

Abbreviations of part of transition rules is used in order to ease the reading of the specification.

The Core Part of the Prolog Specification

The specification of the Core Prolog consists transitions for the following operations:

- Stop rule, which applies when all possible goals are tried and all possible solutions are computed.
- Success rule. This rule applies when a goal is computed with success.
- The failure rule. This rule applies if one of the subgoal can not be computed. Prolog backtracks since the goal can not be satisfied.
- The subgoal success rule. This rule applies when the current subgoal has been computed. Prolog then try to satisfy the next subgoal.
- Try next clause rule. This rule applies when the head of a clause in the program can not be unified with the head of the subgoal. Prolog tries the next clause in the Prolog program.
- The selection rule. This rule applies when the head of the clause in the program unifies with the head of the current subgoal. A new current goal is made. The tail of the clause becomes the new current subgoal, and the tail of the current subgoal and the tail of the current

goal becomes the remaining subgoals in the new goal. The current substitution is extended with the most general unifier.

- The cut rule. This rule applies when the special cut operator occur. Prolog starts working at the rest of the current subgoal, and the backtracking point is updated to be the backtrackings point for the current subgoal (The effect of this rule is that possible alternate goals are cut off).

The stop rule, the subgoal success rule [2], the failure rule [3], the selection rule and the cut rule is specified again in [Bör90b]. This specification is is done in a way such that there is no need to use contraction in the failure and cut rule.

The Built-in Predicates in the Prolog Specification

The transitions for the following built-in predicate in Prolog are given in the first report [Bör90a]:

true, repeat, fail, not, call, and, or.

In the second report [Bör90b] the transitions for the following database built-in predicates are given:

asserta, assertz, retract, clause.

In addition the semantics for the following built-in control predicates are specified again in [Bör90b]:

fail, call, conjunction, disjunction.

This specification is done in order to fit a slightly changed signature.

In the third report [Bör90c] the semantics are specified for the following predicates file manipulation predicates:

see, seeing, seen, tell, append, telling, told.

The semantics for the arithmetic, terms and input-output predicates are also specified in [Bör90c].

The Abstraction Levels in the Prolog Specification

The specification of the Prolog interpreter in Evolving Algebra seems to divide the abstraction into several abstraction levels. To some extent the specification can also be divided into modules.

In the reports, [Bör90a], [Bör90b] and [Bör90c], the execution tree of Prolog is represented as a stack structure. The abstraction levels of the specification obtained, therefore seems to be based on and follow this particular representation of Prolog.

[2] Renamed to the goal success rule in [Bör90b]

[3] Renamed to the backtrack rule in [Bör90b]

The division into abstraction levels and the chosen representation of Prolog's data structure seems to be a result of a careful analysis of the data structure used in the Prolog interpreter. However, no systematic general way of finding the different abstraction levels seems to appear, when reading the specification [4].

The following main abstraction levels (or modules) is obtained in the specification of Prolog:

1. Specification of the Core Prolog, including cuts.
2. Specification of the built-in control predicates.
3. Specification of the built-in database predicates.
4. Specification of the built-in manipulation predicates.
5. Specification of the built-in arithmetic, terms and input-output predicates.

The enumeration begins with the most abstract level and increase when the each level becomes less abstract. At each abstraction level, mainly distinct and different property of the Prolog language is specified.

4.2.3 Specification of a Lambda Compiler and Lambda Evaluator

When the author made the specification in the Lambda Calculus Compiler and Evaluator, the author was unable to find more than two clearly distinct abstraction levels. It might be the case that it is not possible to find a greater number of abstraction levels or some clever person may be able to find some other abstraction levels given the problem.

But the main question is not if there are more abstraction levels or not when making this particular specification. The problem is how to find the possible abstraction levels when writing a specification.

We need a more systematic way to find and define the abstraction levels and thus refine the specification from a very abstract one to a more detailed specification. Those issues still seems to remain unsolved as far as we are concerned with the specification languages, and therefore the problems needs to be addressed in future research with regards to specification and specification languages.

4.3 The Prolog Execution Tree Example Rewritten Using Modules

We will in this subsection take slightly different approach to the specification of Prolog. In [BR94] the execution of Prolog is described in terms of a search tree of possible solutions. Here we will see how we can specify the build of the Prolog search tree using recursive modules as described in subsection 3.4.6.

[4] In fact, the author of this report found it difficult both to understand exactly how the division into abstraction levels is done, and to see clearly all the abstraction levels used.

4.3.1 The Execution Tree

The Prolog execution tree has nodes which can be seen as state of the Prolog execution. Each node has branches where each branch represents a possible continuation of the Prolog execution. Each branch in the tree can represent the one of the following situations:

1. A possible computation which failed, due to the failure of finding a most general unifier with the first literal in the current goal and the head of the clause.

2. A computation which has been completed with success giving one possible answer to the question.

3. The branch representing a computation which is not ended. The current node will be at the end of this branch.

The Prolog search strategy can be described as a depth first search, in the sense that each branch in the graph is expanded as far as possible before the systems tries an alternative branch or backtracks.

The Prolog execution tree is build during the execution of a Prolog question. The build of the tree goes as follows:

1. The execution starts with the root node as the current node.

2. An ordered set of candidate clauses are taken from the Prolog program. As many new nodes as the number of candidate clauses are created as sons of the current node. Each of the candidate clauses are assigned to each son of the current node.

3. Then each of the clauses is tried in the given order until either a successful unification with the head of the clause and the first literal in the current goal is found, or no candidate clause is left. If a successful unification takes place, the node with the successful candidate clause becomes the new current node. If no candidate clauses are left the system bactracks, such that the father of the current node again becomes the current node.

4. If all the sequence of all goals are successfully computed the Prolog interpreter stops with success. The user is asked he or she wants to try another solution. If the user is not satisfied, the the system backtracs and a search for another solution starts.

5. If the goal is successfully computed the systems proceeds with the next goal.

Here we will adapt the Evolving Algebra specification from [BR94] for core Prolog using the module extension of Evolving Algebra. For a full specification of the Prolog interpreter we refer to [BR94].

4.3.2 Move Down and Up in the Execution tree

Moving down and up in the execution tree can be done in terms of recursive modules. When moving down in the tree, a recursive call is made. When we move upwards in the tree, because we need to backtrack, we just end the current instance of the module.

Each time we move down in the Prolog execution tree, we will need to make all sons to the node we visit for the first time.

```
goal==take_goal(fst_decgl(decglseq(currnode)))
act==fst_term(goal)
given_mgu==mgu(act,(rename(cl_head(clause(cll(fst_node(cands(currnode))))))
                          ,vi))
new_goal==cl_body(clause(cll(fst_node(cands(curr_node)))))
cutpt==take_node(decglseq(currnode))
cont==add_decgl(make_decgl(rest_terms(goal),cutpt),rest_decgls(decglseq))
new_deqglseg==apply_subst
                (given_mgu,(add_decgl(make_decgl(rename_goal(new_goal,vi)),
                                                 father(currnode)),
                            cont))
% Some abbreviations of signatures.
GOAL=TERM*
DECGOAL=GOAL x NODE
```

Here we state some abbreviation which we will use in the transitions below. We will as far as possible use ordinary Evolving Algebra functions.

```
MAIN MODULE is TOP-LEVEL

BEGIN GLOBALS
take_goal: DECGOL --> GOAL
take_node: DECGOAL --> NODE
fst_decgl: (DECGOAL + MARK)* --> DECGOAL + MARK
scnd_decgl: (DECGOAL + MARK)* --> DECGOAL + MARK
rest_decgl: (DECGOAL + MARK)* --> (DECGOAL + MARK)*
make_decgl: (GOAL x NODE) --> DECGOAL
add_decgl: DECGOAL x (DECGOAL + MARK)* --> (DECGOAL + MARK)*
type_of_decgl: {DECGOAL + MARK} --> {Decgoal,Mark}
fst_term: GOAL --> TERM
rest_terms: GOAL --> GOAL
rename: TERM x VIND --> TERM
rename_goal: GOAL x VIND --> GOAL
cl_body: CLAUSE --> LIT*
cl_head: CLAUSE --> LIT
fst_node: NODE* --> NODE
rest_nodes: NODE* --> NODE*
apply_subst: SUBST x (DECGOAL + MARK)* --> (DECGOAL + MARK)
comp_substs: SUBST x SUBST --> SUBST
addind: VIND --> VIND
```

```
...
END GLOBALS
```

For the definitions of other global functions, see [BR94].

```
BEGIN SHARED WITH MODULES: TOP-LEVEL,HANDLE-LEVELS
new_currnode: NODE
END SHARED

BEGIN SHARED WITH MODULES: HANDLE-LEVELS,MAKE-SONS
list_of_cand_clauses: CLAUSE*
temp_node: NODE
father: NODE --> NODE
cands: NODE --> NODE*
END SHARED
```

All functions shared between modules are also shared between instances of the modules listed.

```
Module TOP-LEVEL
BEGIN PRIVATE FUNCTIONS
root_node: NODE + {Empty}
% Initialize
root_node==Empty
END PRIVATE

if root_node=Empty
   EXTEND NODE by temp(NODE)
        new_currnode:=temp(NODE)
        root_node:=temp(NODE)
   ENDEXTEND
   MAKE INSTANCES of HANDLE-LEVELS by instance(HANDLE-LEVELS)
        Invoke(instance(HANDLE-LEVELS,Itself)
   END INSTANCES
fi

if root_node=/=Empty
then
    Stop:=Failure
fi
End Module
```

The module `TOP-LEVEL` describes the start of the Prolog computation. If the systems ever bactracks the whole way back to the top level node, there is no more possible computations to the initial Prolog question.

```
Module HANDLE-LEVELS

BEGIN PRIVATE FUNCTIONS
```

```
currnode: NODE
mode: {Call,Select}
END PRIVATE

% Initialize local variables
mode==Call
currnode==new_currnode
END PRIVATE

if is_user_defined(act)
   & mode=Call
then
     list-of-cand-clauses:=procdef(act,db)
     temp_node:=currnode
     MAKE INSTANCES of MAKE-SON by instance(MAKE-SONS)
          Invoke(instance(MAKE-SON),Itself)
     END INSTANCES
     mode:=Select
fi

if deqglseq(currnode)=Empty-seq
then
   stop:=Success
fi

if more_solution_wanted
   & stop:=Success
then
   stop:=0
   Invoke-return(Inherited-module)
fi

if goal=Empty-seq
then
   decglseq(currnode):=rest_decgls(decglseq(currnode)
fi


if   user_defined(act)
   & mode=Select
   & given_mgu=/=Nil
then
   currnode:=fst_node(cands(currnode))
   mode:=Call
   cands(currnode):=rest_nodes(cands(currnode))
   decglseq(fst_node(cands(currnode))):=new_deqglseq;
   s(fst_node(cands(currnode))):=compsubst(s(currnode),given_mgu)
   vi:=addind(vi,1)
```

```
   new_currnode:=fst_node(cands(currnode)
   MAKE INSTANCES of HANDLE-LEVELS by instance(HANDLE-LEVELS)
        Invoke(instance(HANDLE-LEVELS,Itself)
   END INSTANCES
fi

if   user_defined(act)
   & mode=Select
   & given_mgu=Nil
then
   cands(currnode)=rest_nodes(cands(currnode))
fi

if   user_defined(act)
   & mode=Select
   & cands(currnode)=Empty
then
   Invoke-return(Inherited-module)
fi
End Module
```

The module HANDLE-LEVELS describes the Prolog computation at one level. In fact the most of the core Prolog computation is described within this module.

4.3.3 Making a New Nodes in the Execution Tree

Here we will optimize the tail recursive invocation of modules to model iterative treatments making all sons of the current node in the execution tree.

```
Module MAKE-SONS
if list-of-cand-clause=Empty
then
   Invoke-return(Inherited-module)
fi

if list-of-cand-clause=/=Empty-List
then
   list-of-cand-clause:=tail_clauses(list-of-cand-clauses)
   EXTEND NODE by temp(NODE)
   WITH
       father(NODE):=temp_node
       cands(temp_node):=append-cands(temp(NODE,cands(temp_node)))
       cll(NODE):=first_clause(list-of-cand-clauses)
   ENDEXTEND
   MAKE INSTANCES of MAKE-SONS by instance(MAKE-SONS)
        Invoke(instance(MAKE-SONS),Inherited-module)
   END INSTANCES
```

```
...
End Module
```

Each of the candidate clause is associated to the sequence of sons in the pre-determined order.

At first glance it may be difficult to see what we obtain by optimize this tail recursion compared with the mechanism used in [BR94](p 11). But the mechanism used in [BR94] is not a general mechanism to express iterations. The tail recursive call of modules, provides a systematic way of expressing iterations.

It is easy to imagine a much more complicated situation, where we need to express many iterations, and in addition some may be nested within each other. The tail recursive calls of module instances can be used in such a situation. So the example above can be regarded as the most simple example of how an iteration may be specified.

However, if we really want to perform n operations *simultaneously* as is done in [BR94] we may need to make some further extensions to the module mechanisms.

4.4 The Importance of Dividing Specification into Modules

When making the specification of the lambda calculus interpreter and compiler, the specification in Core Evolving Algebra tends to be big and clumsy. A specification at a very detailed abstraction level, will need to be divided in some ways in order to manage the complexity caused by the huge amount of details in the specification. So when the author was writing the specification, the need to divide the specification into modules became clear.

We are able to write more detailed specification, using specification language which takes into account not only what an algorithm is supposed to do, but also how we will perform the algorithm and the amount of resources needed to perform the specification. Then, there is good reason to take ideas developed in the area of computer science with regards to defining abstraction levels and dividing the program into modules and try to apply the idea when writing a specification.

The problem of dividing the specification in different abstraction levels and the problem of dividing a specification at one abstraction levels into modules may be more coupled together than i appears at the first sight. When dividing a specification into modules, we may be able to refine each of the modules into a less abstract specification which is of manageable size. We may also be able to easier see the possibilities of making abstractions when we have divided the specifications into manageable tasks.

However, only further research can give answer about the relationship between the division of a specification into modules, and the use of abstraction levels.

In the the rest of this chapter and in the subsequent chapter the author

will point out some of his experience [5].

4.5 Implementation of a Functional Language

4.5.1 A supercombinator

We take the following definition of a supercombinator from [Jon87]

Definition 11 *A* supercombinator, *S, of arity n is a lambda expression of the form*

$$\lambda x_1.\lambda x_2 \ldots \lambda x_n.E$$

where E is not a lambda abstraction such that

1. *S has no free variables,*
2. *Any lambda abstraction in E is a supercombinator,*
3. *$n \geq 0$; there not needs to be lambda at alls.*

A supercombinator redex *consists of the application of a supercombinator to n arguments, where n is the arity of the supercombinator.*

A supercombinator reduction *replaces a supercombinator redex by an instance of the supercombinator body with the argument substituted for free occurrence of the corresponding formal parameters.*

We will want to use supercombinator instead of ordinary lambda expression in order to avoid the complicated substitution rules which will apply when reducing ordinary lambda expressions.

4.5.2 Translate a Lambda Expression into a Supercombinator

Consider the lambda expression:

$$(\lambda x.(\lambda.y \ * \ y\ x\ x)\ x)(+\ 3\ 3)$$

How should we translate the lambda abstractions in the expression into a to supercombinators? First we use the β-abstraction to make the free variable x in the innermost lambda abstraction into an extra parameter (See Chapter 13 in [Jon87] for a more comprehensive description of the translation process):

$$(\lambda x.(\lambda.x.\lambda.y \ * \ y\ x\ x)\ x\ x)$$

In order to distinguish the to two different variables with the name x, we rename the x to w in the innermost lambda abstraction:

$$(\lambda x.(\lambda.w.\lambda.y \ * \ y\ w\ w)\ x\ x)$$

[5] Where it was difficult to make the specification due to problems with finding more abstraction levels and dividing the specification into modules

In fact both the innermost and the outermost lambda abstraction is transformed into supercombinators, and we give the two lambda abstractions names as the following supercombinator definitions:

$$\begin{aligned} A\ w\ y &= *\ y\ w\ w \\ B\ x &= A\ x\ x \end{aligned}$$

The expression to be evaluated now becomes:

$$B\ (+\ 3\ 3)$$

The translation make use of the β-abstraction and variable renaming. To optimize this translation more complicated rules apply (See Chapter 13 in [Jon87] and Chapter 6 in [JL91]).

No Evolving Specification is given for this translation process in this report. In the subsequent chapters we will assume that all lambda expressions is given as supercombinators, and make the Evolving Algebra Specification according to this assumption.

4.5.3 Lazy Evaluation

If we are going to implement lazy evaluation, we want to postpone the evaluation of the argument given to a supercombinator as long as possible. In the example above we first reduce the expression as far as possible, before we compute the arithmetic primitives $*$ for multiplying and $+$ adding operations:

$$\begin{aligned} & B\ (+\ 3\ 3) \\ \rightarrow\ & A\ (+\ 3\ 3)(+\ 3\ 3) \\ \rightarrow\ & *\ (+\ 3\ 3)(+\ 3\ 3)(+\ 3\ 3) \\ \rightarrow\ & *\ 6\ 6\ 6 \\ \rightarrow\ & 216 \end{aligned}$$

4.5.4 Eager Evaluation

When using eager evaluation, the arguments given to the supercombinator is evaluated as soon as possible. So the example is reduced as follows:

$$\begin{aligned} & B\ (+\ 3\ 3) \\ \rightarrow\ & B\ 6 \\ \rightarrow\ & A\ 6\ 6 \\ \rightarrow\ & *\ 6\ 6\ 6 \\ \rightarrow\ & 216 \end{aligned}$$

If we know for certain that all occurrence of the arguments have to be evaluated, eager evaluation is the optimal evaluation strategy.

4.5.5 Eager and Lazy evaluation

Consider the supercombinator definition:

$$K\ x\ y\ =\ x$$

and the expression

$$K\ 5\ (/\ 3\ 0)$$

where / is the division operator. If we were using eager evaluation the evaluation fails because we try to compute 3 divided by zero, before evaluating the definition of the supercombinator K. If we instead use lazy evaluation, the following reduction will do:

$$\begin{array}{rl} & K\ 5\ (/\ 3\ 0) \\ \rightarrow & 5 \end{array}$$

giving the first argument as the result.

So in such case we will want to use lazy evaluation in order to avoid evaluating parts of the expression which do not need to be evaluated and which fail.

It may also be the case that it is possible to evaluate the part of the expression which in fact does not need to be evaluated [6]. If this argument is large we can avoid a lot of computation, if we use lazy evaluation.

So the best way to optimize the evaluation is to find out, when it is best to use eager evaluation and when it is necessary or best to use lazy evaluation. The analysis required to do evaluation in an optimal manner is quite difficult and not feasible to do in many cases.

4.5.6 Eager Evaluation and Weak Head Normal Form

The Weak Head Normal Form is defined in [Jon87] as follows:

Definition 12 *A lambda expression is in* Weak Head Normal Form (WHNF), *if and only if it is on the form:*

$$F\ E_1\ E_2 \ldots E_n$$

where $n \geq 0$ *and*

1. *either* F *is a variable or data object*
2. *or* F *is a lambda abstraction or built-in function and* $(F\ E_1\ E_2 \ldots E_m$ *is not a redex for any* $m \leq n$.

An expression has no top level redex *if and only if it is in weak head normal form.*

The Weak Head Normal Form differ from Normal Form, since an expression in Weak Head Normal form can contain inner redexes. An expression in normal form can not contain inner redexes.

The lazy evaluation process specified in chapter 6 will evaluate an expression into Weak Head Normal Form. When we introduce eager evaluation of supercombinator expressions, we can force the modified algorithm to evaluate all or some of the inner arguments by marking the arguments as strict and evaluate the arguments.

[6] In the example above we consider an expression given as the second argument to the K supercombinator.

4.5.7 Using Environment or Graph Structure

Functional languages interpreter or compiler will often use either environment structure or a graph structure.

An environment structure is a hierarchical structure, where variables and expressions is given a scoper where the are valid definitions. Most of the Lisp Interpreters are implemented using an environment structure.

A graph structure is a representation of expressions using a graph representation. The graph can be a tree or it can be an acyclic graph, or it can contain cycles, if desired.

A representation of the lambda expressions as an acyclic graph is used in the Evolving Algebra specification of the compilation and evaluation of the supercombinators below.

4.5.8 Interpreting and Compiling an Expression

We have a choice of the degree of compilation or interpretation of an supercombinator expression. If we choose to interpret the expression we can execute the reduction almost immediate and can save the compilation time. On the other hand it may not be possible to optimize the reduction process, so the interpretation of the expression may become a slowly process.

When interpreting a defined language, we are also able to make programs which generates new code in the same language, and then evaluate the code which was automaticly generated. We are not able to do so, if the program is compiled into some other target language.

If we use a compiler to make instructions in a target language which can be easily executed on a computer, we may be able to find many ways to optimize the compiled code. In this way we can speed up the process of evaluation. The price to pay is the cost of executing the compilation step, and the loss of flexibility since we can not make and evaluate expressions in the source language.

In this report we specify two ways to perform the reduction of supercombinator expressions:

- The first approach is to make a graph structure of the supercombinator definitions, and then make reduction on the graph structure beginning with the main supercombinator expression.

- The second approach is to compile each supercombinator definition into a sequence of instructions. When the compilation process is done, the instructions will make and perform the reduction process on a graph structure.

4.6 The Project

The main project reported in this thesis consisted of:

1. Writing an evolving algebra interpreter.

2. Making the specification of compiling and interpretation of a functional language.

3. Implementing, testing and running this specification of the interpreter.

The author does not know of any work, where a *large* Evolving Algebra specification is made together with the Evolving Algebra interpreter, and where both the large specification and the interpreter is implemented and tested within one project.

4.6.1 Specification of an Algorithm and the Use of Resources

The specification of the compilation and interpretation make substantial use of the ability of specifying use of resources in the Evolving Algebra specification language. Hence it is possible to measure in an abstract way the use of resources, when the specification is implemented and executed on an interpreter.

It should also be possible to compute (or reason) about the use of resources, even if a specification is made which is not implemented on any Evolving Algebra interpreter.

4.7 The Main Steps of the Lambda Compiler and Lambda Evaluator

Here we will explain the main steps in compiling lambda expressions into a graph or G-machine code and the evaluation of the lambda expressions. We will give some Evolving Algebra definitions at a very abstract level. In the subsequents chapters we will give a less abstract specification.

Note: When specifying the compilation and evaluation of lambda expressions, we will use the Evolving Algebra as presented in section 2.2. The extension of the Evolving Algebra discussed in chapter 3 will be treated to some extent. Alternative ways of making the specification using the extension will be discussed and examples will be given. We call the Evolving Algebra defined in section 2.2 the Core Evolving Algebra and the extension introduced in chapter 3 Extended Evolving Algebra, when we distinguish between the two versions of the specification language

4.7.1 Compilation of Lambda Expression

The compilation of the Lambda Expressions are performed in two main step.

- Translate the Lambda expressions into supercombinator definitions.
- Compile the supercombinator definitions.

Only the last step will be treated in subsequent chapters. The first step is a syntactically transformation. Therefore we will only give an very abstract definition of this translation process.

4.7.2 Translating Lambda-expressions to Supercombinators

```
if status=Lambda-lifting
then
   all_sc_defs := transform_to_supercomb_defs(lambda_src)
   status := Lambda-Compile
```

The way of translating lambda expressions to supercombinator expressions to translate lambda subexpressions with free variables into supercombinator definitions, where the free variables are added to the parameter-list of the supercombinator.

4.7.3 Compiling the Supercombinator Definitions

```
if status=Lambda-Compile
then
   target_representation := compile_sc_defs(all_sc_defs)
   status:=Evaluate
```

The next step is to compile the supercombinator definition to some target representation. The target representation may be a graph or may be sequences of instructions.

4.7.4 Evaluation of the Supercombinators

```
if status=Evaluate
then
   result := evaluate_supercombinators(target_representation)
   status := Done
```

The last step is evaluating the supercombinators. The supercombinators are reduced to weak head normal form.

4.8 Dividing the Specification into Modules

At this early stage in the specification it is possible to see some of the problems using the Core Evolving Algebra.

Since the Core Evolving Algebra use one name space and one execution sequence, we are forced to rewrite the specification when we move to a less abstract level.

What we really want to do, is to retain the specification made at this very abstract levels, and let it be the part of the total specification.

So let us see how this could be done using Extended Evolving Algebra.

We may take the compilation of a supercombinator as an example:

```
SHARE WITH MODULES: MAIN-MODULE, TRANSLATE-MODULE
initial_representation: SDEFS-EXPR
END SHARED MODULES
```

```
SHARE WITH MODULES: MAIN-MODULE, TRANSLATE-MODULE, COMPILE-MODULE
sc_representation: SDEFS-EXPR
END SHARED MODULES

SHARE WITH MODULES: MAIN-MODULE, COMPILE-MODULE, EVAL-MODULE
compiled_representation: GRAPH-STRUCTURE
END SHARED MODULES

SHARE WITH MODULES: MAIN-MODULE, EVAL-MODULE
result_representation: DATA
END SHARED MODULES
...
initial-representation:== ....
...
MAIN MODULE is START MODULE

Module: START-MODULE
BEGIN PRIVATE FUNCTIONS
status: STATUS
...
status:==Lambda-lifting
...
END PRIVATE FUNCTIONS
...
if status=Lambda-Compile
then
   INVOKE(COMPILE-SC-DEFS)
   status:=Evaluate
fi
...
End Module
```

The example above demonstrate how the specification can be divided into modules, like the division of a large program into subroutines.

```
..
Module: COMPILE-SC-DEFS
...
if finished
then
   compiled_representation:=result(...)
   INVOKE-RETURN(START-MODULE)
fi
End Module
```

Above we show how a module can return to the main module.

In the example above all shared constants expect `initial-representation` are set in the modules invoked from the main module.

The constant `initial-representation` gets its initial value when the Evolving Algebra starts the run.

Chapter 5

Compilation of Supercombinators

5.1 Introduction

We will in this chapter describe and specify the compiling of supercombinator definitions using evolving algebra. In subsequent chapters the description will be extended to cover handling of primitives operators and lazy arguments compared with strict arguments. For a more general description of compilation and reduction of supercombinators, see [JL91].

The Evolving Algebra specification in this chapter is based on Core Evolving Algebra. If we use Extended Evolving Algebra it is possible to write a specification using modules and to specify recursive calls directly. In chapter 8 the use of Extended Evolving Algebra will be discussed.

5.2 The Chosen Abstraction Level

Here we jump quite directly on a quite detailed abstraction level. Since we try to describe how to compute a supercombinator expression into a graph, we have to get into many details in order to give a specification of how to build the graph.

We have jumped from a very abstract level in 4.7 to a quite concrete abstraction level used the specifications below. It would be nice if we could find an intermediate level of abstraction, and thus refine this intermediate level to the detailed level as below.

However, the author was not able to find such an intermediate level of abstraction. Hence, the resulting specification becomes quite detailed, and it was not easy to manage all the details in the specifications in the subsequent chapters.

The specification was written in Core Evolving Algebra. When bound to the Core Evolving Algebra it is not easy to divide the specification into modules, since the algebra presuppose that all operations is performed in one logical sequence, and all functions belong to one name space.

5.3 Compile the Supercombinator Definitions

First we will describe the compilation of supercombinator definitions.

The assumption can be made, that all supercombinator definitions may be needed in the subsequent evaluation phase. Therefore we choose to compile all such definitions into some target representation.

The outcome of the compilation process are to be some target representation of all supercombinator definitions and in addition the initial state will be set.

The target representation of the supercombinator definition, which is to be used in this report is an acyclic graph or G-machine instructions.

Most of the details about how to compile a supercombinator representation will be postponed to subsequent sections in the report.

5.3.1 Compile All the Supercombinator Definitions

We are given a collection of supercombinator definitions. All of those definitions must be be compiled into a suitable target representation. The representation may for instance be a graph or a sequence of target instructions. In the first abstract approach we do not need to make any decision about the target representation.

Two main steps are needed before the evaluation can start:

- Compile all supercombinator definitions given.
- Set the initial state.

The signature

```
% The source:
sc_defs:                SCDEFS
empty_sc_defs:          SCDEFS --> BOOL
remove_next_sc_def:     SCDEFS --> SCDEFS
compile_next_sc_def:    SCDEFS --> TCODE

% The target:
all_pointers:           TADDRS
add_pointer:            TADDR x TADDRS --> TADDRS
sc_def_pointer:         TADDR --> TCODE

% Initial state:
current_state,initialize_state: STATE
```

The Transitions

The compilation of all supercombinator definitions to some target representation are specified below:

```
if not(empty_sc_defs(sc_defs))
then
```

```
EXTEND TADDR by temp(TADDR)
WITH
   all_pointers:=
       add_pointer(temp(TADDR),all_pointers)
   sc_def_pointer(temp(TADDR)):=
       compile_next_sc_def(sc_defs)
ENDEXTEND
src_sc_defs:=remove_next_sc_def(sc_defs)
```

We compile every supercombinator definition from the source program until all supercombinator definitions are compiled.

All compiled supercombinator definitions are compiled to some target representation. Every compiled supercombinator definition are given an unique pointer. The collection of target address are the value of constant `all_pointers`. We may at present think of the collection of address as a list of pointers to the target representation of supercombinator definitions.

```
if empty_sc_defs(src_sc_defs)
then
   current_state:=Reduce-begin
```

The transition below starts the reduction of the compiled definitions.

The specification above ends the top level description of the compiling process.

Note: This top level specification for compilation could be made in Extended EA as the main module for compilation. The rest of the specification could in turn be divided into modules, each invoked from the main module for compilation.

5.4 Compile a Supercombinator Definition

Here the main steps regarding the compilation of a supercombinator definition is described:

Three main steps will be needed:

- Find the next supercombinator definition to compile.
- Make a global association list which links the name of the SC-definition to a pointer (address) to the compiled definition.
- Compile the the definition into some suitable representation.

The three steps will be specified in the subsequent subsections.

5.4.1 The Signature and Abbrevations

Before we specify the transition, we will specify the signature and the abbreviations.

The signature

We will use the following signature

```
% Supercombinator expressions
main_sc_def_name:       NAME
all_sc_defs:            SCEXPR*
is_empty_sc_defs:       SCEXPR* --> BOOL
get_main_name:          SCEXPR* --> NAME
get_next_sc_def:        SCEXPR* --> SCEXPR
tail_sc_defs:           SCEXPR* --> SCEXPR*

empty_expr:             SCEXPR
expr_type:              SCEXPR --> SCTYPE
get_sc_def_name:        SCEXPR --> NAME
make_params:            SCDEF --> VNAME*
src_body:               SCDEF --> SCEXPR
first_app_expr:         SCEXPR --> SCEXPR
second_app_expr:        SCEXPR --> SCEXPR
make_num:               SCEXPR --> NUMBER
make_sc_name:           SCEXPR --> NAME
make_var_name:          SCEXPR --> VNAME

% Pointers
curr_sc_def_addr:       TADDR
get_name_from_globals:  TADDR --> NAME
value_of_addr:          TADDR --> SCEXPR

% nodes
node_type:              NODE --> TYPE
node_child:             NUMBER x NODE --> TADDR + {Empty}
node_params:            NODE --> NAME*
node_num:               NODE --> NUMBER
node_sc_name:           NODE --> NAME
node_var_name:          NODE --> VNAME
graph:                  TADDR --> NODE

% Stack operations
temp_stack:             TADDR*
empty_stack:            TADDR*
is_empty_stack:         TADDR* --> BOOL
pop_stack:              TADDR* --> TADDR*
top_addr:               TADDR* --> TADDR
push_stack:             TADDR x TADDR* --> TADDR*
push2_stack:            TADDR x TADDR  x TADDR* --> TADDR*

% status
status:                 STATUS
```

Abbreviations

We use the following abbreviations:

```
current_value==value_of_addr(top_addr(temp_c_stack))
current_node==graph(top_addr(temp_c_stack))
```

5.4.2 Start of the Compilation

```
if   status=Initial
   & not(is_empty_sc_defs(all_sc_defs))
then
   main_sc_def_name:=get_main_name(all_sc_defs)
   status:=Get-curr-sc-def
```

We define a transition which starts the compilation process. This transition gets the name of the main supercombinator definition and sets the status to "Get-curr-sc-def"

5.4.3 Find a Supercombinator Definition

```
if   status=Get-curr-sc-def
   & not(is_empty_sc_defs(all_sc_defs))
then
   EXTEND TADDR by temp(TADDR,1)
   WITH
      curr_sc_def_addr:=temp(TADDR,1)
      get_name_from_globals(temp(TADDR,1)):=
          get_sc_def_name(get_next_sc_def(all_sc_defs))
      get_addr_from_globals(
              get_sc_def_name(get_next_sc_def(all_sc_defs))):=
                temp(TADDR,1)
     value_of_addr(temp(TADDR,1)):=get_next_sc_def(all_sc_defs)
   ENDEXTEND
   all_sc_defs:=tail_sc_defs(all_sc_defs)
   status:=Compile-sc-def
```

Here we define the transition which gets the next supercombinator definition to be compiled. In addition a link from the supercombinator name to the pointer of the graph of the compiled supercombinator definition is made.

```
if   src_empty_sc_defs(src_sc_defs)
   & status=Find-sc-def
then
   status:=Perform-graph-reds
```

This transition finish the compilation, when all supercombinator definitions are compiled. The evaluation mode is set to perform graph reductions.

5.4.4 Compile the SC-definition Abstract Specification

```
get_target_sc_def:        TADDR --> TCODE
compile_sc_def:           SCDEF --> TCODE
```

The transition

```
if status=Compile-sc-def
then
    get_target_sc_def(curr_sc_def_addr):=compile_sc_def(src_curr_sc_def)
    status:=Get-curr-sc-def
```

This transition is an abstract specification of the compilation process. We compile the definition to some target code. The pointer to the supercombinator definition will now also points to the target code.

5.5 The Supercombinator Graph

The abstract representation of a supercombinator definition will be an annotated graph (a tree). The root node in this graph will be the Supercombinator Node which will be annotated with the list of parameters. The son of the supercombinator node will be the root node of the subgraph which represent the body of the supercombinator definition. The supercombinator body consists of application nodes as interior node, and leaf nodes which represent a number, a local variables or a name of a supercombinator.

5.5.1 Interior Nodes

Two types of interior nodes are needed.

The supercombinator definition node is the root node in the graph which represent the supercombinator definition. The node has one son, the root node of the subgraph describing the body of the supercombinator definition.

The only type of interior node which usually occur in the graph representing the body of the supercombinator is the application node. An application node has two sons. Each son represent an expression. The first son represents the expression to be applied. The second son represents the expression which the first expression is to be applied *upon*.

A Supercombinator Node

The name of the supercombinator is already stored in the global list which links supercombinator names to its definitions. In addition we need to store the parameters used in the supercombinator definitions. This parameters are stored as a list of parameter definitions (may be parameter names) along with the supercombinator node.

Application Nodes

Only pointers to the two children is stored with the application node.

5.5.2 Leaf Nodes

Here we describe the leaf nodes in the graph.

Numbers

A number is stored along with the number node.

Local Variable Names

The variable name is stored along with the variable node.

Supercombinator Names

The supercombinator name is stored along with supercombinator name node.

5.6 Making the Graph

This section contains the specification of how the supercombinator definitions are compiled to a pieces of graph.

5.6.1 The Supercombinator Definition Node

The transition which makes the supercombinator definition node is given below. This node will be the root node in the supercombinator definition graph.

```
if status=Compile-sc-def
   & expr_type(value_of_addr(curr_sc_def_addr))
then
   EXTEND TADDR by temp(TADDR)
          NODE by temp(NODE)
   WITH
      % Makes node
      graph(curr_sc_def_addr):=temp(NODE)
      node_type(temp(NODE)):=Supercomb
      node_params(temp(NODE)):=
         make_params(curr_sc_def_addr))
      node_child(1,temp(NODE)):=temp(TADDR)
      % Initialize the compile stack.
      value_of_addr(temp(TADDR)):=src_body(value_of_addr(curr_sc_def_addr)
      temp_stack:=push_stack(temp(TADDR),empty_stack)
   ENDEXTEND
   status:=Compile-the-expression
```

This transition specify how we compile the list of parameters from the left hand side of the supercombinator definition.

A supercombinator definition node is made.

5.7 How to Compile of the Supercombinator Definition

5.7.1 The Recursive Definition Approach

The process of building the graph of the body of a supercombinator could be specified as a recursive process:

```
Procedure make_graph(expression)
if not(application(expression))
then
   <make a leaf node of appropriate type>
else if application(expression)
then
   <make an application node>
   son1:-make_graph(firstapp(expression))
   son2:-make_graph(secondapp(expression))
fi
```

Note: This recursive definition can naturally be written using Extended Evolving Algebra (See chapter 8). Hence, the clumsy specification in section 5.7.3 can be avoided.

5.7.2 Using the Core Evolving Algebra to Specify the Compilation

If we are forced to use Core Evolving Algebra, this recursive definition has to be translated to an iterations, and explicit use of stack. Such translation breaks the structure of the specification, making it difficult to read and understand.

In this section we will explain how a graph which represents the supercombinator can be build using Core Evolving Algebra. This graph can in some sense be seen as an abstract specification of the target code.

At this stage we are not concerned about making efficient target code. The task of making efficient representation (with regard to space and time consumption) will be addressed in the subsequent sections.

The compile stack

The specification given below describes the compilation process as iterations. It is necessary to use a compilation stack to keep track of the parts of graph not yet builded.

The invariant is that a pointer to a node on the top of the compile stack represents a subgraph which is build. This pointer can therefore be discarded from the compile stack. The way the graph is build ensure that the invariant holds.

When an application node is made, pointers to its (unfinished) children are put above the pointers the application node made. As the result the

subgraphs of the (two) children of the application node are build and discarded before the pointer to the application node will appear on the top of the compile stack.

When an leaf node is made its pointer is on the top of the stack. This node represent a finished subgraph and can immediately be discarded.

5.7.3 Specification of the Compilation of the SC-body

Here we give the specification for the process of compiling the body of a supercombinator definition.

Compilation of an Application Expression

```
if status=Compile-the-expression
   & (not(is_empty_stack(temp-c_stack)))
   & expr_type(current_value)=APexpr
then
   EXTEND POINTER by temp(POINTER,1),temp(POINTER,2)
          NODE by temp(NODE)
          % Makes the node.
          current_node:=temp(NODE)
          current_value:=empty_expr
          node_child(1,temp(NODE)):=temp(POINTER,1)
          node_child(2,temp(NODE)):=temp(POINTER,2)
          node_type(temp(NODE)):=APnode
          % Makes elements to the compile stack.
          value_of_addr(temp(POINTER,1)):=
                first_app_expr(current_value)
          value_of_addr(temp(POINTER,2)):=
                second_app_expr(current_value)
          temp_stack:=push2_stack
                (temp(POINTER,2),temp(POINTER,1),temp_stack)
   ENDEXTEND
```

The transition below specify how to make the application node. Pointers to the two subexpressions are kept on the compile stack *above* the pointer to the application node.

Compilation of a Number Expression

```
if status=Compile-the-expression
   not(is_empty_stack(temp-c_stack))
   & expr_type(current_value)=Numexpr
then
   EXTEND NODE by temp(NODE)
          current_node:=temp(NODE)
          current_value:=empty_expr
          node_type(temp(NODE)):=Num
          node_num(temp(NODE)):=
```

```
            make_num(current_value)
   ENDEXTEND
```

This transition specify how to make a number node.

Compilation of a Supercombinator Name Expression

```
if status=Compile-the-expression
   not(is_empty_stack(temp-c_stack))
   & expr_type(current_value)=SCname
then
   EXTEND NODE by temp(NODE)
          current_node:=temp(NODE)
          current_value:=empty_expr
          node_type(temp(NODE)):=SCName
          node_sc_name(temp(NODE)):=
              make_sc_name(current_value)
   ENDEXTEND
```

This transition specify how to make a supercombinator name node.

Compilation of a Local Variable Name Expression

```
if status=Compile-the-expression
   not(is_empty_stack(temp_stack))
   & expr_type(current_value)=VARname
then
   EXTEND NODE by temp(NODE)
          current_value:=empty_expr
          current_node:=temp(NODE)
          node_type(temp(NODE)):=LVar
          node_var_name(temp(NODE)):=
              make_var_name(current_value)
   ENDEXTEND
```

This transition specify how to make a local variable name node.

Traverse Up One Step in the Graph

```
if status=Compile-the-expression
   & not(is_empty_stack(temp_stack))
   & expr_type(current_value)=EMPty
then
   temp-stack=pop_stack(temp_stack)
```

Here we specify to traverse up on step in the graph by popping of an element from the compile stack. We assume that all nodes which is popped of the compile stack represents part of the graph which is build.

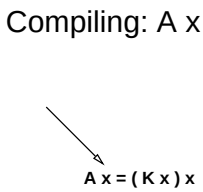

Figure 5.1: The supercombinator definition

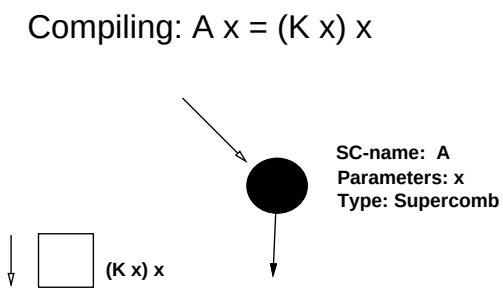

Figure 5.2: Step 1: Create a supercombinator definition node

Empty Adress Stack

```
if   status=Compile-the-expression
   & is_empty_stack(temp_stack)
then
   status:=Get-curr-sc-def
```

Here we are finished with compilation of a supercombinator definition.

5.7.4 An Example

Here we will show an example of how a supercombinator expression are compiled.

The supercombinator expression to compile are the expression:

```
A x = (K x) x
```

We will show the process of compilation step by step.

Before the compilation of a supercombinator definition starts, we have the following situation shown in figure 5.1.

The first step is to make the node for the supercombinator definition. See the figure 5.2.

In the second step an application node is created 5.3. Pointers to its two sons are made. The application expression is divided into two subexpressions, where the subexpressions are associated to the two pointers at top of the address stack.

The third step makes an local variable name node. The pointers at top of the address stack is set to point to the new node. See figure 5.4.

The fourth step pops the top pointer of the address stack, prepearing for the next application expression to be compiled. See figure 5.5.

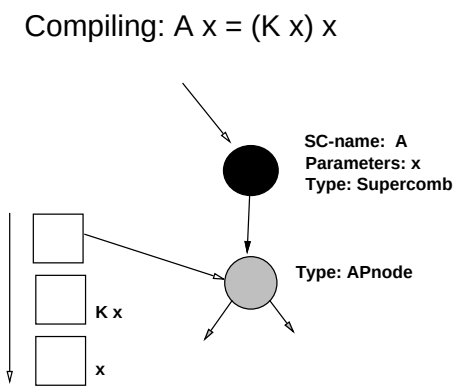

Figure 5.3: Step 2: Create the first application node

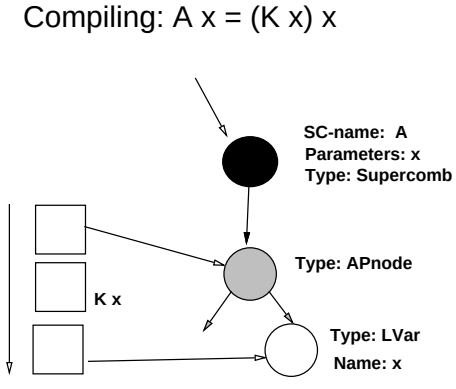

Figure 5.4: Step 3: Create a local variable node

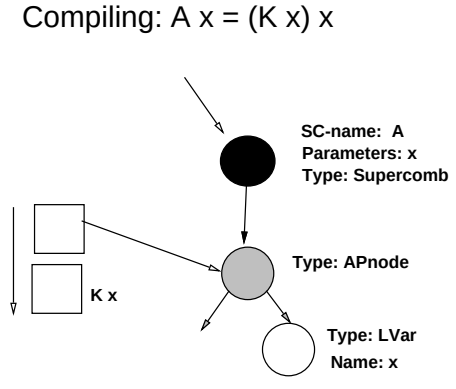

Figure 5.5: Step 4: Pop an element of the address stack

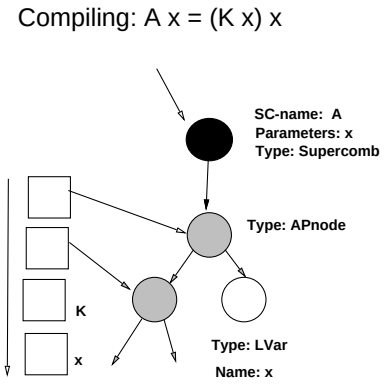

Figure 5.6: Step 5: Create the second application node

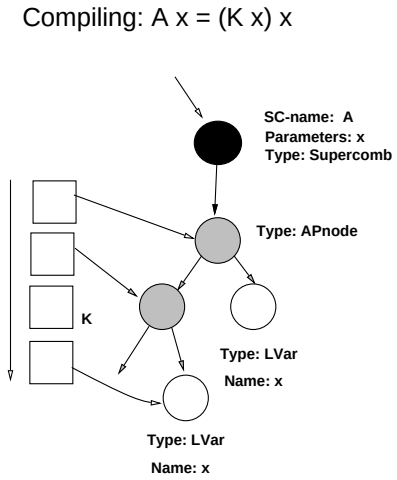

Figure 5.7: Step 6: Create a local variable node

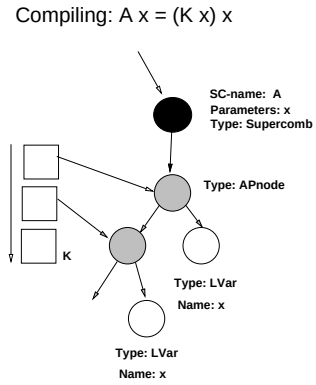

Figure 5.8: Step 7: Pop an element of the address stack

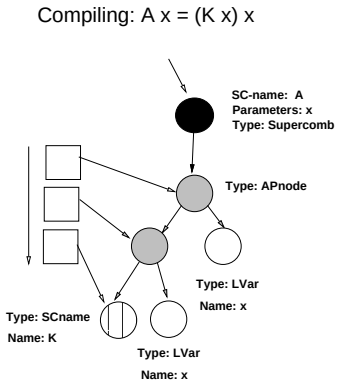

Figure 5.9: Step 8: Create a supercombinator name node

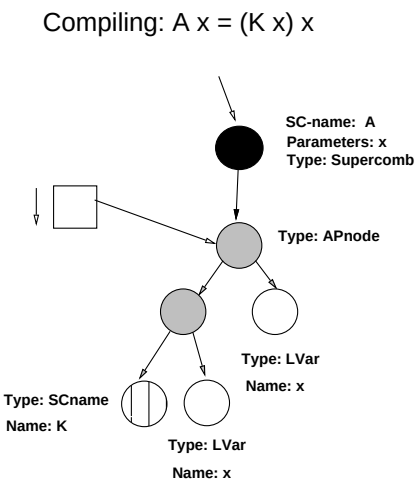

Figure 5.10: Step 10: Two elements popped of the stack

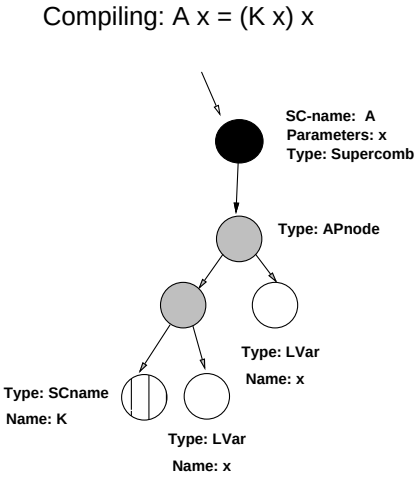

Figure 5.11: Step 12: The last element popped of the address stack

The fifth and the sixth step makes a new application node and a new local variable node. See figure 5.6 and 5.7.

In the seventh step the top pointer is popped of the address stack. Then a supercombinator name node is made. See figure 5.8 and 5.9.

The graph representing the supercombinator definition is made.

The only steps to be done is top pop off all pointers from the address stack until the stack is empty. See figure 5.10 and 5.11.

5.7.5 A Brief Note About Efficiency

At this stage in the specification process we could choose to finish our specification.

We could leave to the programmers to discover where to optimize in the system and that way let the performance depends on the actual implementation of the system. In this way we would follow the traditional way of excluding any requirements concerning efficiency from the specification.

This approach may not be what we want. To understand why, we can use the Scheme language as an example. In the Scheme which is a dialect of Lisp, the specification of Scheme require the implementation to use constant space when executing tail recursive procedures. An implementation of Scheme which did not optimize the tail recursive call to use constant space, would force the programmer to use special constructs to do iterations instead of simply use tail recursive procedure definition.

5.8 Select the Target Structure

In the abstract specification of compilation into the target structure of the supercombinator definition, the target structure is described as a tree.

This target structure capture the information needed to perform the reductions of the supercombinators, and therefore this abstract specification of the compilation process is sufficient if we do not care about making an efficient representation of the target structure.

Examples of issues affecting the efficiency of the reduction process are:

- We may want to specify sharing of identical local variables when building the supercombinator body. Sharing such variables will prevent making unnecessary copies of expressions which is instantiated for the variables during the reduction process.

- There is no need to actually build a graph when the supercombinator definitions are compiled. We can instead generate instructions to make the graph at evaluation time.

We will consider the following choice of the target representation:

1. Making a tree.

2. Making an acyclic graph.

3. Making instructions to be executed later.

None of the improvements in the target representation to be discussed require major change of the abstract evolving algebra specification.

5.8.1 Making a Tree

The abstract specification of the target representation specify a tree as the target structure of the compilation (See Section 5.5).

5.8.2 Making an Acyclic Graph

A graph is not always considered to be the most efficient representation of the target structure with regards to use of time. Traversing a graph at evaluation time may take more time than executing an instruction sequence which makes the graph.

Despite of that, we may want to implement a system which evaluates a graph structure. If we want to let a functional program make some code which in turn is invoked by the evaluator, it may be desirable to evaluate The reason is that the internal data structure used by the evaluator is some sort of acyclic graph. We may also implement an evaluator able to evaluate both target instruction sequences and target graph structure.

In this subsection we show how we easily can make the change in the specification needed to build an acyclic graph instead of a tree.

Sharing Distinct Variables

The improvement we will discuss is the sharing of distinct local variable nodes.

If the same variable appears more than once in the body of the supercombinator definition, unnecessary duplication of the argument will occur when all occurrence of the same variable are instantiated with the same argument expression. This duplication can be avoided by sharing the local variable node.

The only transition which needs change is the transition which makes the local variable leaf node. In addition we assume that the parameter list in the current supercombinator definition node are initialized such that the predicate `is_made_var_node` gives the value "False" for every distinct variable found in the parameter list.

The Signature

The following additional signature is used:

```
existing_var_node:    NAME --> NODE
is_made_var_node:     NAME x NAME* --> BOOL
```

Abbreviations

We will use the following abbreviations for value found on the top of the compile stack and the node for the definition of the supercombinator.

```
current_value==value_of_addr(top_addr(temp_stack))
current_node==graph(top_addr(temp_stack))
current_var_name==make_var_name(value_of_addr(top_addr(temp_stack)))
current_param_list==node_params(graph(curr_sc_def_addr))
```

The Transitions

```
if status=Compile-the-expression
   & not(is_empty_stack(temp_stack))
   & expr_type(current_value)=VARname
   % Change: tests if the variable node is made.
   & is_made_var_node(current_var_name,current_param_list)
then
   % Change: Let the pointer points to the node already made.
   current_node:=existing_var_node(current_var_name)
```

Here an association list of variables and nodes are used. This list is associated with the current supercombinator definition node, This list is used to keep track of which node is already made.

The first transition apply if the local variable node is made. This transition sets the pointer to the node associated with the variable.

```
if status=Compile-the-expression
   & not(is_empty_stack(temp_stack))
   & expr_type(current_value)=VARname
   % Change: Tests if not the variable node is made.
   & not(is_made_var_node(curr_param_list)
then
   EXTEND NODE by temp(NODE)
          current_node:=temp(NODE)
          current_value:=empty_expr
          node_type:=LVar
          node_loc_var_name(temp(NODE)):=
              make_loc_var_name(current_value)
          % Change: Assocate the variable node to the variable name.
          existing_var_node(current_var_name):=temp(NODE)
          is_made_var_node(current_var_name,curr_param_list):=True
   ENDEXTEND
```

The second transition apply if the local variable is not made. The new node is made and a list the local variable is associated with the new node.

5.9 Promise to Make a Graph

The basic step in the evaluation process of supercombinators is to reduce an supercombinator expression. We call the expression to be reduced a redex. The redex is replaced by an instance of the body of an supercombinator definition. The local variables which occurs in the body of the supercombinator definitions are substituted by the argument found in the redex.

Since we may create more than one instance of the same supercombinator definition we have to make a new instance for every reduction.

Instead of making the graph of the supercombinator definition at compile time, we make *instructions* which can be used to make the graph at evaluation time. In this way we do not need to traverse the body of supercombinator definition graph at evaluation time when making a new instance to replace the redex.

The instruction set we use is called G-machine code. This instruction set and a set of recursive equations which specify the compilation scheme can be found in [Jon87].

The evolving algebra specification of compilation into G-machine code follows the same pattern as the (abstract) specification of compilation into the graph representation. Hence we can make a new evolving algebra specification which is quite similar to the (abstract) evolving algebra specification which makes the graph (See 5.5).

5.9.1 Compilation into G-machine Code, the EA-specification

In the subsections below we give the specification of the compilation process into G-machine code.

The signature of the functions used in the transitions below:

The signature

```
% Expression
expr_type:                  SCEXPR --> SCTYPE
src_body:                   SCEXPR --> SCEXPR
number_of_params:           SCEXPR* --> NUMBER
make_params:                SCEXPR --> SCEXPR*
first_app_expr:             SCEXPR --> SCEXPR
second_app_expr:            SCEXPR --> SCEXPR
num_expr:                   SCEXPR --> NUMBER
sc_name_expr:               SCEXPR --> NAME
make_var_name:              SCEXPR --> NAME
get_position:               PARAMPOS* x NAME --> NUMBER

% Stack operations.
temp_stack                  TADDR*
initial_stack:              TADDR*
pop_stack:                  TADDR* --> TADDR*
top_addr:                   TADDR* --> TADDR
push_stack:                 TADDR x TADDR* --> TADDR*
push2_stack:                TADDR x TADDR  x TADDR* --> TADDR*
is_empty_stack:             TADDR* --> BOOL

% Functions which acts on pointers.
finished_code:              TADDR --> INSTR*
curr_sc_def_addr:           TADDR
```

```
has_code:                   TADDR --> BOOL
value_of_addr:              TADDR --> SCEXPR
instructions:               TADDR --> INSTR*
hascode:                    TADDR --> BOOL
get_param_list:             TADDR --> PARAMPOS*
increment_pos_list:         NUMBER x PARAMPOS* --> PARAMPOS*

% Functions which acts on instructions.
get_operator:               INSTR --> OPERATOR
get_operand:                NUMBER x INSTR --> OPERAND
make_g_code:                INSTR --> INSTR*
make_g_code2:               INSTR x INSTR --> INSTR*

% Making a sequence of instructions.
code_list,empty_code:       INSTR*
concatenate_code:               INSTR* x INSTR* --> INSTR*

% Operations on the nodes
graph:                      TADDR --> NODE
node_type:                  NODE --> TYPE
def_arity:                  NODE --> NUMBER

% Add numbers
add:                        NUMBER x NUMBER --> NUMBER
```

Abbreviation

We will use the following abbreviation for value found on the top of the compile stack. This value may be an expression to compile or a node in the part of the graph which is already build. assigned to the address at top of the compile stack:

```
current_address==top_addr(temp_stack)
current_value==value_of_addr(top_addr(temp_stack))
```

5.9.2 Make the Initial Instructions

```
if   status=Get-curr-sc-def
   & is_empty_scdefs(all_sc_defs)
then
   EXTEND INSTR by temp(INSTR,1),temp(INSTR,2)
   WITH
      get_operator(temp(INSTR,1)):=Pushglobal
      get_operand(1,temp(INSTR,1)):=main_sc_def_name
      get_operator(temp(INSTR,2)):=Unwind
      instr_stack:=make_g_code_two(temp(INSTR,1),temp(INSTR,2));
   ENDEXTEND
   status:=Exec-code
   leftbranch:=1
```

```
   rightbranch:=2
```

The initial sequence of instructions:

```
Pushglobal <main-sc-name>
Unwind
```

is made and stored on the instruction stack.

5.9.3 Promise to Make the SC-definition

```
if   status=Compile-sc-def
   & expr_type(current_value)=SDEFexpr
then
   EXTEND TADDR by temp(TADDR,1),temp(TADDR,2)
          INSTR by temp(INSTR,1),temp(INSTR,2)
          NODE by temp(NODE)
   WITH
      graph(curr_sc_def_addr):=temp(NODE)
      node_type(temp(NODE)):=Global
      def_arity(temp(NODE)):=number_of_params(make_params
                             (value_of_addr(curr_sc_def_addr)
      % Pass the body of the sc-definition.
      value_of_addr(temp(TADDR,2)):=
           src_body(value_of_addr(curr_sc_def_addr))
      % Pass the position-list of parameters.
      get_param_list(temp(TADDR,2)):=
           make_param_pos_list(value_of_addr(curr_sc_def_addr))
      has_code(temp(TADDR,2)):=False
      %   Make the instruction: Slide d+1
      get_operator(temp(INSTR,1)):=Slide
      get_operand(1,temp(INSTR,1)):=
         add(1,number_of_params(make_params
                 (value_of_addr(curr_sc_def_addr))))
      %   Make the instruction: Unwind
      get_operator(temp(INSTR,2)):=Unwind
      instructions(temp(TADDR,1)):=
          make_g_code_two(temp(INSTR,1),temp(INSTR,2))
      has_code(temp(TADDR,1)):=True
      value_of_addr(temp(TADDR,1)):=valueofaddr(currscdefaddr)
      temp_stack:=push_stack(temp(TADDR,2),
             pushstack(temp(TADDR,1),initial_stack))
   ENDEXTEND
   status:=Compile-the-body
```

Here we give the specification for how to make the G-machine code which prepare for the next reduction step.

The G-machine code sequence which is made are:

```
Slide d+1, Unwind
```

The variable d is the number of parameters given to the supercombinator definition which is used as to build the new instance to replace the redex.

The parameter list which is used throughout the compilation process is pairs of positions and local variable names. The positions are initially the position in the parameter list in the supercombinator definition.

The positions may later change during the compilation phase in order to give the right relative position of the local variables on the address stack used in the evaluation of the G-machine code.

5.9.4 Promise to Make an Application Node

```
if status=Compile-the-body
   & not(is_empty_stack(temp-c_stack))
   & expr_type(current_value)=APexpr
   & has_code(current_address)=False
then
   % Make the list of finished code empty
   code_list:=empty_code_list
   EXTEND TADDR by temp(TADDR,1),temp(TADDR,2)
          INSTR by temp(INSTR)
      % Makes elements to the compile stack.
      value_of_addr(temp(TADDR,1)):=
          first_app_expr(current_value)
      has_code(temp(TADDR,1)):=False
      get_param_list(temp(TADDR,1)):=
          increment_pos_list(1,get_param_list(current_address))
      value_of_addr(temp(TADDR,2)):=
          second_app_expr(current_value)
      has_code(temp(TADDR,2)):=False
      get_param_list(temp(TADDR,2)):=
          get_param_list(current_address)
      %   Make the instruction: MKap
      get_operator(temp(INSTR)):=MKap
      %   Attach code to the current address
      instructions(current_address):=make_g_code(temp(INSTR))
      has_code(current_address):=True
      temp_stack:=push_stack(temp(TADDR,2),
          push_stack(temp(TADDR,1),temp__stack))
   ENDEXTEND
```

Here we give the specification of the compilation of the G-machine code which makes the application node.

The G-machine code sequence which is made is:

```
Mkap
```

The parameter list is passed to the subexpressions. For the first application expression the positions in the parameter list is increased by 1.

On the temporary compilation stack the second subexpression to be applied upon are stored on the top. The first subexpression to apply is

stored immediately below and the `Mkap` instruction is store immediately below the two subexpressions.

5.9.5 Promise to Make a Number Node

```
if status=Compile-the-expression
   not(is_empty_stack(temp_stack))
   & expr_type(current_value)=Numexpr
   & has_code(current_address)=False
then
   EXTEND INSTR by temp(INSTR)
   WITH
      %   Make the instruction: Pushint i
      get_operator(temp(INSTR):=Pushint
      get_operand(1,temp(INSTR):=make_num(current_value)
      %   Attach code to the current address
      instructions(current_address):=
          make_g_code(temp(INSTR))
      has_code(current_address):=True
   ENDEXTEND
```

Here we give the specification of the compilation of the G-machine code which makes the number node.

The G-machine code sequence which is made is:

```
Pushint i
```

The integer i is the number taken from the supercombinator expression,

5.9.6 Promise to Make a Supercombinator Name Node

```
if status=Compile-the-expression
   not(is_empty_stack(temp-c_stack))
   & expr_type(current_value)=SCname
   & has_code(current_address)=False
then
   EXTEND INSTR by temp(INSTR)
   WITH
      %   Make the instruction: Pusglobal f
      get_operator(temp(INSTR):=Pushglobal
      get_operand(1,temp(INSTR):=make_sc_name(current_value)
      %   Attach code to the current address
      instructions(current_address):=make_g_code(temp(INSTR))
      has_code(current_address):=True
   ENDEXTEND
```

Here we give the specification of the compilation of the G-machine code which makes the supercombinator name node.

The G-machine code sequence which is made is:

```
Pushglobal f
```

The name f is the supercombinator name taken from the supercombinator expression,

5.9.7 Promise to Make a Local Variable Name Node

```
if status=Compile-the-expression
   not(is_empty_stack(temp-c_stack))
   & expr_type(current_value)=VARname
   & has_code(current_address)=False
then
   EXTEND INSTR by temp(INSTR)
   WITH
      % Make the instruction: Push parampos(x)
      get_operator(temp(INSTR):=Push
      get_operand(1,temp(INSTR):=
          get_position(get_param_list(current_address),
          make_var_name(current_value))
      % Attach code to the current address
      instructions(current_address):=make_g_code(temp(INSTR))
      has_code(current_address):=True
   ENDEXTEND
```

Here we give the specification of the compilation of the G-machine code which makes the local variable name node.

The G-machine code sequence which is made is:

```
Push pos
```

The integer *pos* is the position taken from pair (*pos* x) in the parameter list where x is the local variable found in the supercombinator expression.

The parameter list is passed down and updated during the compilation process. At evaluation time the integer *pos* will index the corresponding argument for the local variable on the address stack.

5.9.8 Add Finished Code to the Sequence of Instructions

```
if status=Compile-the-expression
   & not(is_empty_stack(temp_stack))
   & has_code(current_address)=True
then
   % Move instruction on top of the stack to
   % the end of the permanent instruction sequence
   code_list:=concatenate_code(code_list,
           instructions(current_address))
   temp-stack=pop_stack(temp-stack)
```

Here we move finished code from the top of compile stack to the sequence of instructions for the supercombinator definition. The element at top of the compile stack is popped of. This step can bee seen as traversing a step up in the abstract graph of nodes.

5.9.9 Assign the Finished Code to the current SC-definition

```
if status=Compile-the-expression
   & is_empty_stack(temp_stack))
then
   % Attach the finished code to the address of sc_def.
   finished_code(graph(curr_sc_def_address)):=code_list
   code_list:=empty_code_list
   status:=Get-curr-sc-def
```

Here we make a binding to the address of the current supercombinator definition and the finished sequence of instructions.

5.9.10 Update of the Root of the Redex

The instructions generated as described above fails to update the root of the redex at the end of the reduction step, Hence shared subgraphs may be evaluated many times.

The problem is the Slide $d + 1$ instruction. This instruction simply removes all pointers to the redex and lets the remaining pointer point to the instance of the supercombinator body which replaces the redex. Instead we need an instruction sequence which overwrites the root of the redex in the graph.

So we need to change the specification which generates the code finishing the current reduction and prepearing for the next reduction.

```
if   status=Compile-sc-def
   & expr_type(current_value)=SDEFexpr
then
   EXTEND TADDR by temp(TADDR,1),temp(TADDR,2)
          INSTR by temp(INSTR,1),temp(INSTR,2),temp(INSTR,3)
          NODE by temp(NODE)
   WITH
      graph(curr_sc_def_addr):=temp(NODE)
      node_type(temp(NODE)):=Global
      def_arity(temp(NODE)):=number_of_params(make_params
                             (value_of_addr(curr_sc_def_addr)
      % Pass the body of the sc-definition.
      value_of_addr(temp(TADDR,2)):=
           src_body(value_of_addr(curr_sc_def_addr))
      % Pass the position-list of parameters.
      get_param_list(temp(TADDR,2)):=
           make_param_pos_list(value_of_addr(curr_sc_def_addr))
      has_code(temp(TADDR,2)):=False
      %   Make the instruction: Update d
      get_operator(temp(INSTR,1)):=Update
      get_operand(1,temp(INSTR,1):=
          number_of_params(make_params
                (value_of_addr(curr_sc_def_addr)
```

```
    %   Make the instruction Pop d
    get_operator(temp(INSTR,2)):=Pop
    get_operand(1,temp(INSTR,2):=
        number_of_params(make_params
              (value_of_addr(curr_sc_def_addr)
    %   Make the instruction: Unwind
    get_operator(temp(INSTR,3):=Unwind
    instructions(temp(TADDR,1)):=
        make_g_code_three(temp(INSTR,1),temp(INSTR,2),temp(INSTR,3))
    has_code(temp(TADDR,1)):=True
    value_of_addr(temp(TADDR,1)):=valueofaddr(curr_scdef_addr)
    temp_stack:=push_stack(temp(TADDR,2),
           pushstack(temp(TADDR,1),initial_stack))
ENDEXTEND
status:=Compile-the-body
```

The improved G-machine instruction sequence to perform the update is shown below (See p 97 in [Jon87]):

```
Update d, Pop d, Unwind
```

This small changes should give a significant improvement with regards to efficiency to the G-machine evaluation.

Chapter 6

Evaluation of the Graph

6.1 Introduction

This chapter describe and specify the *reductions* of pure supercombinator expressions using Evolving Algebra (See also [JL91]). Primitives, strict and lazy arguments are treated in chapter 7.

This chapter cover the reduction process for:

- The template instantiation machine
- The G-machine

6.2 The Reduction Process

Below we give a very abstract specification of the reduction process.

6.2.1 The Signature

```
some_redex_left:      SCGRAPH --> BOOL
current_graph:        SCGRAPH
reduce:               SCGRAPH --> SCGRAPH
```

6.2.2 Specification of the Reduction

The transition is given below. This transition may loop forever.

```
if  some_redex_left(current_graph)
then
   curr_graph:=reduce(current_graph)
```

The reduction process can be seen as repeating reduction steps until no redex are can be found.

No details about how to perform a reduction step is given at this abstract level.

6.3 A Less Abstract Specification of a Reduction

The reduction step can be divided into three main tasks.

1. Find the next redex.
2. Reduce the redex.
3. Prepare for the next reduction step.

The Signature

```
curr_graph,def_graph:  GRAPH
scdef:                 SCOBJ
redex:                 SCOBJ
status:                STATUS
some_redex_left:       GRAPH --> BOOL
find_next_scdef:       GRAPH x GRAPH --> SCOBJ
find_next_redex:       GRAPH --> SCOBJ
status:                STATUS
result:                SCOBJ
substlist:             SUBSTS
redex:                 SCOBJ
args:                  SCOBJ*
params:                NAME*
instantiate:           SCOBJ x SUBSTS --> SCOBJ
get_scbody:            SCOBJ --> SCOBJ
make_substlist:        SCOBJ* x NAME* --> SUBSTS
get_params:            SCOBJ --> NAME*
make_args:             SCOBJ --> SCOBJ*
update_graph:          SCOBJ --> GRAPH
```

6.3.1 Find the outermost left redex

```
if some_redex_left(curr_graph)
then
   redex:=find_next_redex(curr_graph)
   scdef:=find_next_scdef(def_graph,curr_graph)
   status:=Instantiate
```

This evolving algebra transition simply finds the supercombinator definitions to be used later and the redex. A copy of the body of the `scdef` will be instantiated by the arguments made from the redex.

Here we search the graph for the outermost left redex. We also need the supercombinator definition named by the root symbol in the redex. However, we do not use the evolving algebra to specify that we want the outermost left redex as the next redex and how to find the root symbol of the redex at this level of specification.

6.3.2 Reduce the redex expression

```
if status=Instantiate
then
   result:=instantiate(get_scbody(scdef),substlist)
      WHERE substlist=make_substlist(args,params)
        WHERE params=get_params(scdef)
              args=make_args(redex)
   status:=Update
```

A copy of the body of the supercombinator definition is instantiated by the arguments made from the redex expression. We need make a list of substitution where the parameters are taken from the supercombinator definition and the arguments are made from the redex.

The instance created is called the result of the reduction.

6.3.3 Prepare for the next reduction step

```
if status=Update
then
    curr_graph:=update_graph(result)
```

The last step updates the graph with the result of reduction. We do not say how the update is performed. The result of the reduction may either be copied into the graph without replacing the redex or the result may replaces the redex in the graph.

6.4 The States of the Evaluator

In the following sections we will specify an interpreter able to evaluate both a graph and sequences of G-machine instructions.

In this section we will describe the structure of the machine which evaluates the supercombinators. In addition we will specify those part of the initial state which is not made in the compilation process.

We will also briefly mention the initial values of the components of the state.

6.4.1 The Structure of the Reduction Machine

The evaluator can bee seen as a state transitions system. The state consists of the following components:

A stack of instructions.

An address stack.

A dump stack.

A Graph.

The globals.

6.4.2 The Components of the State

In this subsection we will describe the components of the state and the initial state. The setting of initial value which is not performed at compile time will be described in section 6.6.

The Graph

The graph is represented by the function `graph`, which given an address returns a node.

An interior node have pointer to its childs. A pointer to a child is given as an address to the child. Attributes such as types and values of a node is associated with each node through functions.

The initial graph is made by the compiler.

In addition a simple piece of graph consisting of a supercombinator name node to the main supercombinator has to be set.

The Globals

The globals consists of all supercombinator names, each name associated to the address of the supercombinator in the graph. The globals are given by two functions `get_addr_from_globals` and `get_name_from_globals`. The function `get_addr_from_globals` returns an address to a node in the graph, if a name of a supercombinator is given. The function `get_name_from_globals` returns the name of the supercombinator if an address to a supercombinator is given.

The globals are made by the compiler.

The Dump

The dump is a stack of address stacks. The initial value of the dump is set to the empty stack.

The Instruction Stack

The instruction stack is set to the the instruction "Egraph" as its initial value. This instructions sets the evaluator to perform the reduction on a graph.

The Address Stack

The address stack is set to the address of initial supercombinator name node on the graph.

6.4.3 The Modes of the Evaluator

Since our evaluator are able to evaluate both compiled instructions and the graph on the heap, the evaluator operates in two modes.

1. Evaluate the graph.

2. Evaluate a sequence of instructions which builds the graph.

If the instruction at the top of the instruction stack is "Egraph" the evaluator is in the graph mode. If some other instruction appear on the top of the instruction stack, the evaluator turns to the mode of executing instructions.

We do use the special instructions "Egraph" to mark the graph mode instead of using the empty code stack for the following reasons:

1. If we use the empty code stack as a condition for changing from instruction mode evaluation mode, then we have to make the requirement that the "Unwind" instruction always is the last instruction at the bottom of the stack.

2. We do not want to exclude the use the empty instruction stack as one of the termination condition of the evaluation process.

6.5 The Detailed Specification of a Reduction Machine

We will in the subsequent sections give a detailed specification of the reduction machine. The specification will describe an graph reduction machine.

6.5.1 The Pieces of the Graph

All pieces of the graph which represent compiled definition of supercombinators are part of the graph state. None of those pieces are allowed to change during the evaluation. Copies of those pieces is made when needed.

But we may permit new definition to be added to the graph state, if we want to implement an extensions to the evaluator such that new definition may be constructed and run at evaluation time.

The root of each piece of the graph which represents a supercombinator definition is a supercombinator definition node. This node will have an application node as its only son. This application node is the root of the subgraph which represents the body of the supercombinator definition.

The piece of graph which the evaluator is allowed to change, is the instance constructed in a previous reduction step. The redex to reduce may be part of this graph or may be the entire old instance. The redex will be replaced by a new instance of the supercombinator definition named by the root symbol of the redex. The main task of the evaluator is to create such new instances to replace a redex.

This piece of graph representing an instance will consists of application nodes as interior node and number nodes, local variable nodes and supercombinator name nodes as leaf nodes.

An application node will have pointers to exactly two sons. The first son points to a subgraph representing the "function" expression to be applied. The second son represent the "argument" expression to be applied upon.

The leaf nodes has the property as described below:

- A number node has a number.
- A supercombinator name node has a name of a supercombinator definition.

In the supercombinator definition pieces of graph we will find a third type of leaf node, a local variable name node. This node contains a name of a local variable given as one of the parameter in the supercombinator definition. When building an instance all local variables nodes are replaced by its correspondent arguments found in the redex.

6.5.2 Description of the Reduction Machine

The specification which follows can be seen as a detailed abstract specification. We do not try to optimize the this "abstract" machine.

Set the Initial Values

The initial values of the address stack, the dumpstack and the instruction stack is set.

Find the Symbol of the Root of the Redex

The first step is to find the root symbol of the redex. The root symbol of the redex is the name of the supercombinator definition which will be used in creating a instance to replace the redex. The root symbol of the redex will also be used to describe a primitive, when we describe how to deal with primitives in a subsequent chapter.

In order to find the root symbol we traverse down the leftmost chain of the graph until we find a node which is not an application node.

Find the Root of the Redex

I the leaf node found has a name of a supercombinator we look up the numbers of parameters, n, in the supercombinator definition node. Then we traverse up to application number n from the leaf node. This application node holds the subgraph which represents the redex.

On the way up we use the opportunity to make an substitution array, where we store the parameter variables in the supercombinator definition and the correspondent arguments found in the subgraph representing the redex.

Creating a New Instance

Next a new instance is made by copying the piece of the graph representing the supercombinator definition and substitute every occurrence of a local variable node by the subgraph representing the correspondent arguments from the redex.

The Update

After a new instance is build we arrange the address stack such that the address on the top of the stack points to the newly created instance replacing the redex.

This update may also change the address of the redex to point to the new instance, but needs not do so.

6.5.3 The Signature

```
% Address stack
root_of_the_redex:     TADDR
root_of_the_instance:  TADDR
curr_sc_def_addr:      TADDR
main_sc_def_name:      NAME
curr_params:           NAME*
curr_arity:            NUMBER
addr_stack:            TADDR*
empty_addr_stack:      TADDR*
length_addr_stack:     TADDR* --> NUMBER
top_addr:              TADDR* --> TADDR
push_addr:             TADDR x TADDR* --> TADDR*
push2_addrs:           TADDR x TADDR x TADDR* --> TADDR*
pop_addr:              TADDR* --> TADDR*
pointer_to_def:        TADDR --> TADDR
finished:              TADDR --> BOOL
% The graph
graph:                 TADDR --> NODE + SCOBJ
node_child:            NUMBER x NODE --> TADDR
node_type:             NODE --> TYPE
node_sc_name:          NODE --> NAME
node_num:              NODE --> NUM
node_loc_var_name:     NODE --> NAME
node_params:           NODE --> NAME*
number_of_params:      NODE --> NUMBER
left_branch:           NUMBER
right_branch:          NUMBER
% Making the substitution list
counter:               NUMBER
get_subst_arg_addr:    NAME --> ADDR
get_param_var:         NUMBER x NAME* --> NAME
% The globals
get_addr_from_globals: NAME --> TADDR
% Get the substitution value
get_subst_value:       ADDR --> SCOBJ
copy_subst_value:      ADDR --> SCOBJ
% Perform the Update
set_update:            TADDR x TADDR* --> TADDR*
```

```
% Instructions
egraph_instr:           INSTR
instruction_stack:      INSTR*
make_g_code:            INSTR --> INSTR*
get_operator:           INSTR --> OPERATOR
% The dump stack
dump_stack,
  empty_dump_stack:     DUMP*
% The status
status:                STATUS
```

The Abbreviations

```
sc_def_addr==
     get_addr_from_globals(node_sc_name(graph(top_addr(addr_stack))))
current_left_child_addr==
     node_child(left_branch,graph(top_addr(addr_stack)))
current_argument==
       node_child(right_branch,graph(top_addr(pop_addr(addr_stack)))
current_node==graph(top_addr(addr_stack))
curr_def_node==graph(pointer_to_def(top_addr(addr_stack))
subst_val_address==get_subst_value_addr(node_loc_var_name(curr_def_node))
```

6.6 Set the Initial value of the State

6.6.1 The Evolving Algebra Specification

```
if status=Initial
then
   % Set the initial value of the dump stack.
   dump_stack:=empty_dump_stack
   % Sets the value of some zero-arity functions.
   left_branch=1
   right_branch=2
   status=Unwind
   EXTEND INSTR by temp(INSTR)
          TADDR by temp(TADDR)
          NODE by temp(NODE)
       % Make the special Egraph instruction
       get_operator(temp(INSTR)):=Egraph
       egraph_instr:=temp(INSTR)
       % Set the initial value of the instruction stack
       instruction_stack:=make_g_code(temp(INSTR))
       % Make the initial redex.
       graph(temp(TADDR):=temp(NODE)
       node_type(temp(NODE):=SCName
       node_sc_name(temp(NODE):=main_sc_def_name
```

```
        % Make the initial addr_stack
        addr_stack:=push_addr(temp(TADDR),empty_addr_stack)
    ENDEXTEND
    status=Unwind
```

Here we specify in details how to set the initial values of the components of the state which is not set during the compilation process.

The initial values of the address stack, the dumpstack and the instruction stack is set in the specification above.

We provide initial values for the following components of the stack:

The instruction stack The initial instruction is set to single instruction "Egraph", which sets the initial mode to evaluate the graph. The supercombinator definition is supposed to be in form of a graph.

the address stack The initial address stack is set to the address of the to the node containing the main supercombinator definition.

The dump stack The dump stack is set to be empty.

The name of the main supercombinator definition are set at compile time. The address of the supercombinator definition node can be found using the name of the supercombinator definition.

6.7 The Evolving Algebra Specification of the Unwinding and Making a Substitution List

6.7.1 Find the Redex of the Graph

```
if node_type(graph(top_addr(addr_stack)))=APnode
   & status=Unwind
then
   addr_stack:=push_addr(current_left_child_addr,addr_stack)

if   status=Unwind
   & node_type(graph(top_addr(addr_stack)=Num
then
   result:=nodenum(graph(topaddr(addrstack)))
   status:=Normal-form
```

Here we specify the traversal down the left branch on the graph. When we hit a node which is not an application node, the end of the left branch is reached.

If the leaf node is a supercombinator name node we have hit the name of a supercombinator definition. We look up the supercombinator definition node.

The supercombinator definition node may be of the type Supercomb or of the type Global. If the supercombinator definition node is of type Supercomb the node is the root of a supercombinator definition graph. If the type of the

supercombinator definition node is of type Global the node has an compiled instruction sequence used to build a supercombinator definition graph.

Here we give the specification for dealing with a compiled supercombinator definition graph. The specifications for using compiled instructions sequences is given below (See 6.12).

```
if   node_type(graph(top_addr(addr_stack)=SCName
   & node_type(graph(sc_def_addr))=Supercomb
   & gt(length_addr_stack(addr_stack),node_num_params(graph(sc_def_addr)))
   & status=Unwind
then
   % Find the address to the node sc-def
   curr_sc_def_addr:=sc_def_addr
   % Find the parameter list
   curr_params:=node_params(graph(sc_def_addr))
   % Find the arity of the sc-def
   curr_arity:=number_of_params(graph(sc_def_addr))
   status:=Make-substs-init
```

If the supercombinator definition node is the root of a supercombinator definition graph, we get the address, parameter list and the arity of the supercombinator definition.

6.7.2 Make the List of Substitutions

```
if   status=Make-subst-init
then
   counter:=0
   status:=Make-subst

if   status=Make-subst
   & lt(counter,curr_arity)
then
   get_subst_value_addr(get_param_var(add(counter,1),curr_params)):=
                  current_argument
   addr_stack:=pop_addr(addr_stack)
   counter:=add(counter,1)
```

Here we specify the traversal up the left branch of the graph the number of times given by the arity of the supercombinator. The application node we get at the end of the traversal is the root of the redex.

When traversing up the graph, the substitution list is made. The substitution list constists of pairs of local variable and address to each of the arguments found in the redex.

```
if status:=Make-subst
   & counter=curr_arity
then
   root_of_redex_addr:=top_addr(addr_stack)
   status:=Init-instance
```

When the traversal up the graph is finished, we know the application node which is the root of the redex. The pointer to the root of the redex is saved.

6.8 Specification of How to Build a New Instance

In this section we specify how to build an instance of the supercombinator definition. Since the definition may be used to build more than one instance we always make a new copy of the body of the definition.

All occurrence of the local variables in the copy of the body of the supercombinator definition are substituted by the correspondent arguments from the redex.

The build process starts with the root node of the body of the supercombinator and proceeds in a top down style until the graph representing the body of representation is build.

6.8.1 A Recursive Definition of the Building Process

Below we give a recursive definition of the build process.

```
build-instance(node)
  if application(node)
  then
     new-left-child:=build-instance(left-branch(node)
     new-right-child:=build-instance(right-branch(node))
     new-node:=mkap-node(new-left-address,new-right-address)
  else
  if local-variable(node)
     new-node:=substitute(argument,local-variable,node)
  else
     new-node:=build-leaf-node
  fi
```

Note: Here it is also possible to use Extended Evolving Algebra to express the recursive build process (See Chapter 8).

6.8.2 The Build Process Described as Iterations

The process of building is described as an iterations using evolving algebra. We are building an graph which represents the new instance. The body in the current supercombinator definition are copied. All local variables correspondent to the parameters in the supercombinator definition are substituted for the correspondent arguments in the redex.

So the basic operation we have to perform is as following:

- If an application node is found, build an copy of this node with pointers to its two sons.
- If a number node is hit, build a copy of this node.

- If a supercombinator name node is found, build a copy of this node.
- If a local variable node is found, then substitute the correspondent redex argument expression for the local variable.

In addition it is necessary to use a stack of pointers to keep track of the build process. If the graph is copied from the super definition graph the build process will be "top down".

We will use compiled instructions in 6.12 and 6.13 to build instances. In this case it the build is performed "bottom up".

The specifications below gives quite many details. It would have been possible to add some less detailed evolving algebra specifications. The author consider writing such specification in this particular case, more or less as a duplication of the recursive description given above.

6.8.3 Common Abbreviations

The `current_node` is the current node in the instance we are building. The `curr_def_node` is the correspondent node in the supercombinator definition graph.

6.8.4 Starts the Build of the New Instance

Set the Start Values

```
if status=Init-instance
then
   EXTEND TADDR by temp(TADDR)
   WITH
      pointer_to_def(temp(TADDR)):=
          node_child(1,graph(curr_sc_def_addr))
      addr_stack:=push_addr(temp(TADDR),addr_stack)
      root_of_instance:=temp(TADDR)
      finished(temp(ADDR)):=False
   ENDEXTEND
   status=Build-instance
```

This evolving algebra specification sets the start values for the build of the new piece of graph. We do the following steps:

1. Make an address which is going to point to the root of the instance we are creating.
2. Find the root of the body of the supercombinator definition. The root of the body is the only son of the supercombinator definition node.
3. Make an link from the pointer of the instance to the root of the body of the supercombinator definition.
4. Push the address of the new instance on the top of the address stack.
5. Make a global pointer to the address of the new instance.
6. Update the status.

Creating an Application Node

```
if   status=Build-instance
   & not(finished(top_addr(addr_stack))
   & node_type(curr_def_node)=APnode
then
   finished(top_addr(addr_stack)):=True
   EXTEND TADDR by temp(TADDR,1),temp(TADDR,2)
          NODE by temp(NODE)
   WITH
     % Make the application node.
     current_node:=temp(NODE)
     node_type(temp(NODE)):=node_type(curr_def_node)
     % Make pointers to the two sons of the application node.
     node_child(left_branch,temp(NODE)):=temp(TADDR,1)
     node_child(right_branch,temp(NODE)):=temp(TADDR,2)
     % Mark the pointers.
     finished(temp(ADDR,1):=False
     finished(temp(ADDR,2):=False
     % Create links to the correspondent supercombinator definition
     % pointers.
     pointer_to_def(temp(TADDR,1)):=
        node_child(left_branch,curr_def_node)
     pointer_to_def(temp(TADDR,2)):=
        node_child(right_branch,curr_def_node)
     % Push the pointers on the address stack.
     addr_stack:=
        push2_addrs(temp(TADDR,2),temp(TADDR,1),addr_stack)
   ENDEXTEND
```

Here we specify how a copy of an application node is made:

- Creates an application node.
- Creates two pointers to the two sons of the application node.
- Mark the pointers as not finished.
- Create links from the pointers of the application node to the correspondent pointers in the supercombinator definition piece of the graph.
- Push the pointers to the two sons of the application node on the address stack.
- Mark the pointers to the application node as finished.

A Number Node

```
if status=Build-instance
   & not(finished(top_addr(addr_stack))
   & node_type(curr_def_node)=Num
```

```
then
   finished(top_addr(addr_stack)):=True
   EXTEND NODE by temp(NODE)
      current_node:=temp(NODE)
      node_type(temp(NODE)):=node_type(curr_def_node)
      node_num(temp(NODE)):=node_num(curr_def_node)
   ENDEXTEND
```

Here we specify how a number node is created:

- The number node is made.
- The number on the correspondent number node in the supercombinator definition piece of graph is set on the number node.
- The pointer to the number node is marked as finished.

A Supercombinator Name Node

```
if status=Build-instance
   & not(finished(top_addr(addr_stack))
   & node_type(curr_def_node)=SCname
then
   finished(top_addr(addr_stack)):=True
   EXTEND NODE by temp(NODE)
      current_node:=temp(NODE)
      node_type(temp(NODE)):=node_type(curr_def_node)
      node_sc_name(temp(NODE)):=node_sc_name(curr_def_node)
   ENDEXTEND
```

Here we specify how a supercombinator name node is made:

- The supercombinator name node is made.
- The name on the correspondent supercombinator name node in the supercombinator definition piece of graph is set on the supercombinator name node.
- The pointer to the number node is marked as finished.

6.8.5 Replacing the Local Variable by the Substitution Value

The Transition

```
if status=Build-instance
   & not(finished(top_addr(addr_stack))
   & node_type(curr_def_node)=LVar
then
% Substituting.
   finished(top_addr(addr_stack)):=True
   graph(top_addr(addr_stack)):=
           get_subst_value(subst_val_addr)
```

The following steps are performed when replacing a local variable name node by the piece of graph representing the correspondent arguments found in the redex:

- The correspondent local variable node on the supercombinator definition piece of the graph is found.
- The argument from the redex correspondent to the local variable is taken from the list of substitutions.
- The pointer from the application node is set to the substitution value found.
- The pointer to the substitution is marked as finished.

In the abstract specification above we do not specify the details of how to make the substitution value in the graph. It is possible to make a copy of the argument to substitute for the local variable, or we can for every distinct variable simply set a pointer to the subgraph representing graph to be substituted for the variable.

The abbreviation `subst_val_addr` is specify the address to the subgraph to be substituted for the variable. This address is found in the substitution list.

The function `get_subst_value` is the abstraction for getting the peace of graph to be substituted for the variable.

6.8.6 Pop Finished Pointers of the Address Stack

```
if   status=Build-instance
   & finished(top_addr(addr_stack))
   & ne(top_addr(addr_stack),root_of_instance)
then
   addr_stack:=pop_addr(addr_stack)
```

When a pointer on the address stack is on the top of the address stack and marked as finished, it is popped of the address stack. The process of making an instance is finished when the root of instance appear at top of the stack.

6.8.7 Finish the Reduction Step

```
if   status=Build-instance
   & finished(top_addr(addr_stack))
   & top_addr(addr_stack)=root_of_instance
then
   status:=Update
```

When the instance is made the status is changed to Update.

6.9 The Update

The update may be performed by updating the root of the redex or simply by setting the address on top of the stack to points to the new instance.

6.9.1 Specify the Update

```
if status=Update
then
   addr_stack:=set_update(root_of_instance,addr_stack)
   status:=Unwind
```

We do not want to specify how we treat the pointer to the redex. Therefore all details about making the top elements of the address stack are hidden by use of the function `set_update` in this abstract specification.

6.10 Representation of the Supercombinator Definition

Here we discuss some type of graph which represents the instance.

6.10.1 A Tree

We can not share occurrence of local variables or substitution values in a tree structure. So use of a tree structure implies that we copy a lot of arguments to be substituted for the local variables when an instance is created.

```
if status=Build-instance
   & not(finished(top_addr(addr_stack)))
   & node_type(curr_def_node)=LVar
then
   finished(top_addr(addr_stack)):=True
   graph(top_addr(addr_stack)):=
           copy_subst_value(subst_val_addr)
```

Here we describe the substitution as making a copy of the argument to substitute for the local variable. We do not want to specify details of how to build this part of the subgraph. Hence we use the function `copy_subst_value` to make the copy of the graph.

An Acyclic Graph

When we use an acyclic graph we can share occurrences of a distinct variable and the correspondent substitution value.

The Transition

```
if status=Build-instance
   & not(finished(top_addr(addr_stack))
   & node_type(curr_def_node)=LVar
then
   % Set the pointe to the substitution value.
   finished(top_addr(addr_stack)):=True
   graph(top_addr(addr_stack)):=graph(subst_val_addr)
```

The task of substituting the correspondent argument for the local variable is simply to set the pointer to point to the subgraph which represents the argument.

6.11 Perform the Update

We can perform the update in to ways. We can leave the result of the reduction as a piece of graph without updating the pointer of the redex.

The other way is updating the pointer of the redex, and so discarding the redex. Updating the pointer of the redex prevents the same reduction to be performed more than one time.

6.11.1 Make a Copy of the Result of the Reduction

```
if status=Update
then
   addr_stack:=
       push_addr(root_of_instance,pop_addr(pop_addr(addr_stack)))
   status:=Unwind
```

Here we set the top of the address stack to the newly created instance. The pointer to the redex is pushed of the address stack, but the root of the redex may be pointed to by another pointer.

6.11.2 Update the Root of the Redex

```
if status=Update
then
   EXTEND NODE by temp(NODE)
   WITH
   % Update the root of redex by the new instance.
   node_type(temp(NODE)):=Ind
   node_child(1,temp(NODE)):=root_of_instance
   graph(root_of_redex_addr):=temp(NODE)
   addr_stack:=
       push_addr(root_of_redex,pop_addr(pop_addr(addr_stack)))
   status=Unwind
```

An indirection node which points to the new instance is created. The pointer to the root of redex is set to this new indirection node. In this way we overwrite the root of the redex and prevents this reduction to take place anymore.

Find the Root Symbol of the Redex

When we traverse the leftmost branch of the graph in order to find the root symbol of the redex, we have to deal with the indirection node introduced above.

The Transitions

```
if node_type(graph(top_addr(addr_stack)))=Ind
then
   addr_stack:=push_addr(1,pop_addr(addr_stack))
```

The transition specified below permits the traversal through the indirection node.

6.12 A Compiled sequence of instructions

A sequence of instructions are executed and the new instance of the supercombinator definition to replace the redex is build.

The instruction sequences are compiled such that the build of the graph representing the new instance is performed in a bottom up style, starting with the leaf node to the left in the graph. The root node of the body of supercombinator definition is the last node made.

The transitions below show how each of the G-code instructions is evaluated.

The following G-code instructions are treated below:

- The Pushglobal instruction, which makes a name node.
- The Pushint instruction, which makes a number node.
- The Push instruction, which push a new address on top of the stack.
- The Mkap instruction, which makes an application node.
- The Slide instruction, pops addresses (below the top address) of from the stack.
- The Unwind instruction, which makes an address stack to the graph, which is to be evaluated.
- The Update instruction, which performs update of the redex.
- The Pop instruction, which pops addresses from the address stack.

6.12.1 Common Functions and Signature

Here we list common functions, signature and abbreviations.

The Signature

```
% Address stack
addr_stack:            TADDR*
top_addr:              TADDR* --> TADDR
get_nth_addr:          NUMBER x TADDR* --> TADDR
push_addr:             TADDR x TADDR* --> TADDR*
curr_glob_def_addr:    TADDR
curr_sc_def_addr:      TADDR
pop_addr:              TADDR* -->TADDR*
pop_n_addrs:           NUMBER x TADDR* --> TADDR*
length_addr_stack:     TADDR x NUMBER
% The graph
graph:                 TADDR --> NODE + SCOBJ
node_child:            NUMBER x NODE --> TADDR
node_type:             NODE --> TYPE
node_sc_name:          NODE --> NAME
node_num:              NODE --> NUMBER
def_arity:             NODE --> NUMBER
curr_params:           NAME*
curr_arity:            NUMBER
left_branch:           NUMBER
right_branch:          NUMBER
% The code stack
instr_stack,
     finished_code:   INSTR*
top_instr:            INSTR* --> INSTR
pop_instr:            INSTR* --> INSTR*
concat_code:          INSTR* x INSTR* --> INSTR*
get_operator:         INSTR --> OPERATOR
get_operand:          INSTR --> NAME + TADDR
% The globals
get_addr_from_globals: NAME --> TADDR
% The status
status:                STATUS
% Add two numbers
add:                   NUMBER x NUMBER --> NUMBER
```

Abbreviation

```
sc_def_addr==
     get_addr_from_globals(node_sc_name(graph(top_addr(addr_stack))))
arg_addr==
   node_child(right_branch,graph(get_nth_addr
               (nth_arg_app_node,addr_stack)))
```

```
nth_arg_app_node==add(get_operand(1,top_instr(instr_stack)),2)
slide-number==add(get_operand(1,top_instr(instr_stack)),1)
current_node=graph(top_addr(addr_stack))
update-number==add(get_operand(1,top_instr(instr_stack)),1)
pop-number==get_operand(1,top_instr(instr_stack))
ind_addr==node_child(1,graph(top_addr(addr_stack)))
```

6.12.2 Initialize the Mode to Evaluate Compiled Instructions

If the mode initially is set to execute compiled instructions, the instruction stack has to be set to the instruction sequence:

```
Pushglobal <main-supercombinator-name>
Unwind
```

and the constant `status` has to be set to `Exec-code`. This mode may be set at the end of compilation of the supercombinator definitions.

6.12.3 Change the Mode of Evaluation to Evaluate Compiled Instruction

```
if   node_type(graph(top_addr(addr_stack)))=SCName
   & node_type(graph(curr_sc_def_addr))=Global
   & gt(length_addr_stack(addr_stack),def_arity(graph(curr_sc_def_addr)))
   & status=Unwind
then
   % Get the instruction list
   instr_stack:=concat_code(finished_code(graph(curr_sc_def_addr)),
           instr_stack)
   status:=Exec-code
```

Assume we are in a mode of evaluating supercombinator definitions which is stored as graphs. If the supercombinator definition to be evaluated next consists of compiled instructions, the supercombinator definition node will be of type Global. Then we change the mode of evaluation to execute the sequence of compiled instructions. This instruction sequence attacked to the node makes a new instance of the supercombinator body when executed.

6.12.4 The Pushglobal Instruction

```
if   status=Exec-code
   & get_operator(top_instr(instr_stack))=Pushglobal
then
   EXTEND TADDR by temp(TADDR)
          NODE by temp(NODE)
   WITH
     % Make the supercombinator name node.
     graph(temp(ADDR)):=temp(NODE)
```

```
    node_type(temp(NODE)):=SCname
    node_sc_name(temp(NODE)):=get_operand(1,top_instr(instr_stack))
    addr_stack:=push_addr(temp(TADDR),addr_stack)
  ENDEXTEND
  instr_stack:=pop_instr(instr_stack)
```

The execution of the pushglobal instruction makes a name node in the graph. This name may be the name of a supercombinator definition node or the name of a global node to be used later in the next reduction step.

A supercombinator definition node is the root of a piece of graph which contains a supercombinator definition. A global node contains an instruction sequence which can be used to build an instance of the supercombinator definition body. The global nodes and the supercombinator definition nodes are stored in a table where the name of the node is the key to the pointer of the node.

6.12.5 The Pushint Instruction

```
if   status=Exec-code
   & get_operator(top_instr(instr_stack))=Pushint
then
   EXTEND TADDR by temp(TADDR)
          NODE by temp(NODE)
   WITH
     % Make the number node.
     graph(temp(ADDR)):=temp(NODE)
     node_type(temp(NODE)):=Num
     node_num(temp(NODE)):=get_operand(1,top_instr(instr_stack))
     addr_stack:=push_addr(temp(TADDR),addr_stack)
   ENDEXTEND
   instr_stack:=pop_instr(instr_stack)
```

This evolving algebra transition makes a number node.

6.12.6 The Push Instruction

```
if   status=Exec-code
   & get_operator(top_instr(instr_stack))=Push
then
   addr_stack:=push_addr(arg_addr,addr_stack)
   instr_stack:=pop_instr(instr_stack)
```

The address to the argument n of the redex is pushed on top of the address stack.

6.12.7 The Mkap Instruction

```
if   status=Exec-code
   & get_operator(top_instr(instr_stack))=Mkap
```

```
then
   EXTEND TADDR by temp(TADDR)
          NODE by temp(NODE)
   WITH
     % Make the application node.
     graph(temp(ADDR)):=temp(NODE)
     node_type(temp(NODE)):=APnode
     % Make pointers to the two sons of the application node.
     node_child(left_branch,temp(NODE)):=get_nth_addr(1,addr_stack)
     node_child(right_branch,temp(NODE)):=get_nth_addr(2,addr_stack)
     % Update the address_stack
     addr_stack:=push_addr(temp(TADDR),pop_n_addrs(2,addr_stack))
     instr_stack:=pop_instr(instr_stack)
   ENDEXTEND
```

Here an application node is made.

6.12.8 The Slide Instruction

```
if   status=Exec-code
   & get_operator(top_instr(instr_stack))=Slide
then
   addr_stack:=push_addr(top_addr(addr_stack),
                         pop_n_addrs(pop-number,addr_stack))
   instr_stack:=pop_instr(instr_stack)
```

This transition pops n elements below the top address from the address stack. The top address is retained.

6.12.9 The Unwind Instruction

The transitions below unwinds the stack until a supercombinator name or a number node is hit. If a number node is hit, then the normal form is reached, and the evaluation halts.

```
if   status=Exec-code
   & get_operator(top_instr(instr_stack))=Unwind
   & node_type(current_node)=APnode
then
   addr_stack:=
       push_addr(node_child(left_branch,current_node),addr_stack)
```

This evolving algebra transition unwinds an application node.

```
if   status=Exec-code
   & get_operator(top_instr(instr_stack))=Unwind
   & node_type(current_node)=Num
then
   status:=Normal-form
   instr_stack:=pop_instr(instr_stack)
   result:=node_num(current_node)
```

If we hit a number node when executing the Unwind instruction, then we have reached the normal form. We get the value attacked to the number node.

```
if   status=Exec-code
   & get_operator(top_instr(instr_stack))=Unwind
   & node_type(graph(current_node)=SCname
then
   curr_glob_def_addr:=get_addr_from_globals(node_sc_name(current_node))
   instr_stack:=pop_instr(instr_stack)
   status=Exec-SCdef
```

A supercombinator name node is found. The next part of the instruction may be to retrieve to instruction sequence for the next compiled supercombinator definition stored in the global node, or to change the mode to evaluate a supercombinator definition stored as a piece of graph.

6.12.10 Push New Instruction Sequence on Instruction Stack

```
if   status-Exec-SCdef
   & node_type(graph(curr_glob_def_addr))=Global
   & gt(length_addr_stack(addr_stack),def_arity(graph
             (curr_glob_def_addr)))
then
   instr_stack:=
      concat_code(finished_code(graph(curr_glob_def_addr)),instr_stack)
```

Here a new instruction sequence from the global node is put on the address stack. This instruction sequence make a new instance of the supercombinator body, when executed.

6.12.11 Change to graph mode

```
if   status-Exec-SCdef
   & node_type(graph(curr_glob_def_addr))=Supercomb
   & gt(length_addr_stack(addr_stack),def_arity(graph
               (curr_glob_def_addr)))
then
   curr_sc_def-addr:=curr_glob_def_addr
   curr_params:=node_params(graph(curr_glob_def_addr))
   curr_arity:=def_arity(graph(curr_glob_def_addr))
   status:=Make-subst-init
```

Here a supercombinator definition node is found and the graph mode is changed to perform reduction on the graph.

6.13 Update the Redex

6.13.1 The Update Instruction

```
if   status=Exec-code
```

```
   & get_operator(top_instr(instr_stack))=Update
then
   EXTEND NODE by temp(NODE)
   WITH
     graph(get_nth_addr(update-number,addr_stack)):=temp(NODE)
     node_type(temp(NODE)):=Ind
     node_child(1,temp(NODE):=top_addr(addr_stack))
   ENDEXTEND
   addr_stack:=pop_addr(addr_stack)
   instr_stack:=pop_instr(instr_stack)
```

This transition performs the update of the redex. An indirection node to the result of the reduction replaces the redex in the graph.

6.13.2 The Pop Instruction

```
if   status=Exec-code
   & get_operator(top_instr(instr_stack))=Pop
then
   addr_stack:=pop_n_addrs(pop-number,addr_stack)
   instr_stack:=pop_instr(instr_stack)
```

Here we pop off n address from the address stack.

6.13.3 The Unwind Instruction

```
if   status:=Exec-code
   & get_operator(top_instr(instr_stack))=Unwind
   & node_type(graph(top_addr(addr_stack)))=Ind
then
   addr_stack:=push_addr(ind_addr,pop_addr(addr_stack))))
```

The Unwind instruction pass through the indirection node.

Chapter 7

Strict and Lazy Arguments

In this chapter we will extend the evaluator to handle primitive expressions. That means that we introduce the notation of strict arguments and parameters. Some arguments given to the supercombinator or primitive may be required to be evaluated before the primitive or supercombinator is applied.

The strategy when evaluating expressions consisting only of supercombinators turned out to be quiet simple. We were able to treat all arguments as lazy and thus postpone the evaluation of the argument expression. That means the supercombinator expression was always instantiated before any of its arguments was evaluated. So the normal order reduction to weak head normal form could be used. This way of evaluation is described in chapter 6 above.

A primitive expressions consists of a operator which may have some parameters. To be able to apply certain primitives some of its arguments which corresponds to the parameters may be required to be in weak head normal form. For instance the "add" primitive require all its arguments to be numbers.

Some other of the arguments given to a primitive may be required to be lazy.

As an example of arguments required to be lazy consider the "if" (or "cond") primitive. We do not want all arguments given to "if" primitive to be evaluated before the primitive is applied. If the second and third arguments given to the "if" primitive were treated as strict arguments, then it would be impossible use the "if" primitive to control the evaluation of an expression. We would not be able to use the "if" primitive to stop a recursive call because all arguments would be evaluated before the "if" primitive could be applied.

Another example where lazy evaluation may be required is the expression below given as a non-strict argument to a supercombinator or a primitive:

$$((a + b)/c)$$

This expression computes to a number provided that the variable c does not get zero as its value.

Since the division will fail in case the c becomes zero it may be necessary to postpone the evaluation of the expression as long as possible. Hence a

primitive or supercombinator expression which do not evaluate the expression above if c is zero, will work if the lazy evaluation strategy is employed.

Some of the arguments given to the primitive may be strict. If such arguments are given to a supercombinator and is passed as strict arguments to a primitive in the supercombinators body, it makes no sense to postpone the evaluation. Then we can optimize the evaluation and evaluate such arguments to weak head normal form before evaluating the supercombinator which takes the the arguments.

Thus the notation of strict and lazy arguments apply both to the supercombinators and primitives.

So introducing primitives changes the evaluation strategy. We have to evaluate some arguments before applying the primitive and postpone the evaluation of the non-strict arguments.

7.1 Evaluate the Primitive Expression

A primitive expressions consists of a built-in operator and arbitrary numbers of arguments. Some or all arguments given to the primitive needs to be evaluated before the built-in operators are applied. For instance the multiply operator needs both of its arguments evaluated to numbers.

Thus the following steps are required to evaluate a primitive:

- Build the (graph) representation for the primitive expression.
- Evaluate all strict arguments.
- Apply the primitive operator.

7.1.1 The Recursive Process of evaluating the arguments

The process of evaluating the primitive operator can be expressed as the following recursive procedure:

```
Evaluate(expression)
if whnf(expression)
then
   return expression
else-if primitive(expression)
   n:=0
   for n < arity_of_primitive
   do
   if strict(argument(n))
       Evaluate(argument(n))
       n:=n+1
   od
   compute_primitive(expression)
else-if supercombinator(expression)
   n:=0
   for n < arity_of_primitive
```

```
    do
    if strict(argument(n))
        Evaluate(argument(n)
        n:=n+1
        substitute(argument(n),parameter(n))
    od
    reduce-sc(expression)
fi
```

Note: If we use Extended Evolving Algebra we could express the recursive process of evaluating strict arguments directly (See chapter 8 for a discussion).

7.2 Postpone the Evaluation of Strict Arguments

Before applying a primitive some or all arguments given to the primitive has to be evaluated before the primitive can be applied.

We may choose to evaluate strict arguments only when required, and *postpone* the evaluation of the strict argument as long as possible. We do not perform any analysis to decide which subexpressions is going to be evaluated later.

So we only require information about the arguments needed to be evaluated. Hence the only "analysis" performed is to find strict parameters of the primitive and evaluate the strict arguments which corresponds to its parameters.

7.2.1 The process of evaluation

The process of evaluation consists of the following steps:

- Unvind the Spine
- If a primitive symbol is found, then look up the required data about the primitive.
- Get the arguments given to the primitive.
- Evaluate all arguments which corresponds to the primitives strict parameters.
- Apply the primitive by computing the function associated with the primitive.
- Make the node holding the result of the computation.

The Spine

The unwinding of the spine (the leftmost branch in the graph) is performed in the same way as for supercombinators. When we reach the leaf node of the spine, we check if the leaf node contains a primitive symbol or a supercombinator symbol. If a primitive symbol is found, we prepare for applying the primitive.

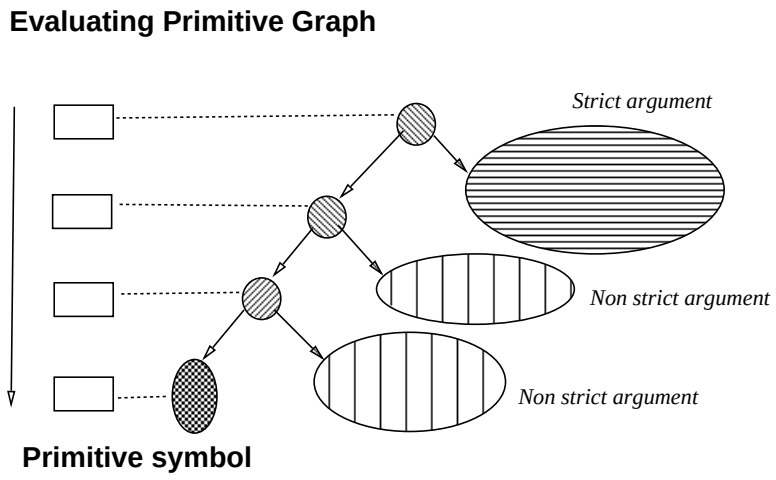

Figure 7.1: After unwinding the primitive graph.

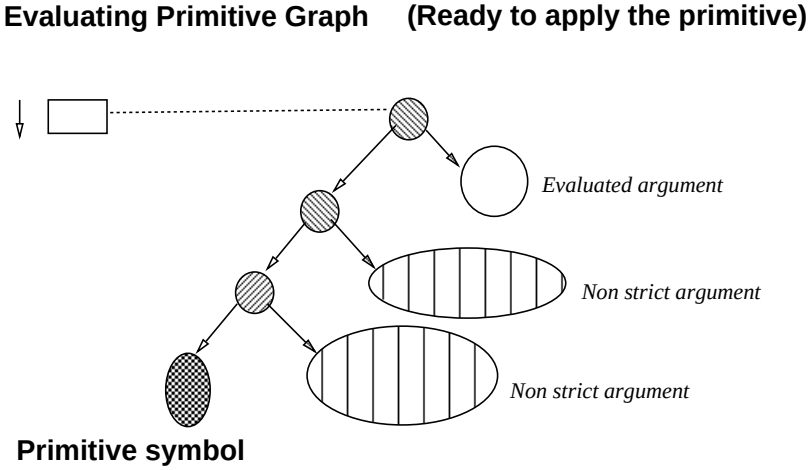

Figure 7.2: Just before computing the primitive expression.

The Primitive Symbol at End of the Spine

From now on, we assume that a primitive symbol is attached to the leaf node at end of the spine. That means that the top address on the address stack points to the primitive node. The address stack will have pointers to the root node of the primitive expression and all application nodes on the spine. See figure 7.1.

The Data About the Primitive

All arguments given to the primitive is associated with the spine. We have associated the following data about the each primitive:

- The arity of the primitive operator.
- Which of the primitives arguments is strict.
- The function associated with the primitive.

Get the Lazy and Strict Arguments

Now, we need to look up all strict arguments and evaluate each of the arguments before the primitive operator can be applied. The resulting graph is shown in figure 7.2.

Since the arguments may be complicated expressions, evaluation of the arguments is a recursive process.

Apply the Primitive

After the evaluation of all arguments which corresponds to strict parameters is performed, we apply the primitive. That means to compute the function associated with the primitive, giving the arguments of the primitive to this function.

The Result

The result of applying the primitive is stored in the node of right type. We may also want to update the root of the redex with the result.

7.2.2 The Core Evolving Algebra Signature

The Evolving Algebra Code consists of the transitions needed to handle primitives and evaluation of strict arguments.

The Signature

```
Defining the RESULT set.
RESULT=NUMBER+(CHAR*)+BOOL

% Functions on primitives.
primobj:              PRIMOBJ
```

```
prim_name:           PRIMOBJ --> NAME
prim_operator:       PRIMOBJ --> FUNCTION
prim_arity:          PRIMOBJ --> NUMBER
prim_counter:        PRIMOBJ --> NUMBER
prim_arg:            PRIMOBJ x NUMBER --> NODE
prim_arg_list:       PRIMOBJ --> NODE*
strict_params:       PRIMOBJ --> BOOL*
required_types:      PRIMOBJ --> TYPE*

% Apply the primitive.
apply-primitive:     FUNCTION x NUMBER x NODE* --> RESULT
result_type:         FUNCTION --> TYPE

% Get information about the parameters of the primitive.
is_strict_param:         BOOL* x NUMBER --> BOOL
has_required_type:       TYPE x TYPE* x NUMBER --> BOOL

% Functions on nodes.
is_strict_app_node:      NODE --> BOOL
node_primitive_name:     NODE --> NAME
node_primitive_function: NODE --> FUNCTION
node_type:               NODE --> TYPE
node_result:             NODE --> RESULT
node_child:              NUMBER x NODE --> TADDR
node_arity:              NODE --> NUMBER
is_in_whnf:              NODE --> BOOL
strict_params_info:      NODE --> BOOL*
required_types_info:     NODE --> TYPE*

% Functions on the address stack.
addr_stack:          TADDR*
graph:               TADDR --> NODE
top_addr:            TADDR* --> TADDR
pop_addr:            TADDR* --> TADDR*
make_addr_stack:     TADDR --> TADDR*
length_as:           TADDR* --> NUMBER

% Functions on dumps
dump_stack:          DUMP*
empty_dump_stack:    DUMP* --> BOOL
addr_stack_dump:     DUMP --> TADDR*
primobj_dump:        DUMP --> PRIMOBJ
push_dump:           DUMP x DUMP* --> DUMP*
pop_dump:            DUMP* --> DUMP*
top_dump:            DUMP* --> DUMP

% Computing numbers
decr1:               NUMBER --> NUMBER
```

7.2.3 Abbreviations

```
current_node:=graph(top_addr(addr_stack))
current_arg_node=
    graph(node_child(right_branch,graph(top_addr(addr_stack))))
current_prim_arg_node=
    graph(node_child(right_branch,graph(top_addr(pop_addr(addr_stack)))))
current_arg_pointer=
    node_child(right_branch,graph(top_addr(addr_stack)))
primitive_definition_node=
   graph(get_addr_from_globals(node_primitive_name(current_node)))
```

7.2.4 The Core Evolving Algebra Transitions

Reach weak head normal form

```
if status=Unwind &
   in__in_whnf(current_node)
   & dump_stack = empty_dump_stack
then
   final_result:=node_value(curren_node)
   status:=Weak-head-normal-form
fi
```

Here we reach the final weak head normal form since the dumpstack is empty.

7.2.5 Restore the Dump

```
if status=Unwind &
   is_in_whnf(current_node) &
   dump_stack =/= empty_dump_stack &
   type_of_dump_el(top_dump(dump_stack)) = Primobj
then
    addr_stack:=addr_stack_dump(top_dump(dump_stack))
    prim_obj:=prim_obj_dump(top_dump(dump_stack))
    node_child(right_branch,graph(top_addr(pop_stack
              addr_stack_dump(top_dump(dump_stack))))):=
              top_stack(addr_stack)
    dump_stack:=pop_dump(dump_stack)
fi
```

If we reach weak head normal form and find dump stack non-empty we restore the top element of dump stack and resume the process of the arguments which was suspended.

We insert the node evaluated to weak head normal form, into the graph after restoring the dump, so we can get it right, in the case where we do not apply the update of the graph.

Get the Primitive Operator

```
if
   status=Unwind &
   node_type(current_node)=PrimName &
   length_as(addr_stack) > node_arity(current_node)
then
   EXTEND PRIMOBJ by temp(PRIMOBJ)
   WITH
   obj_prim_name(temp(PRIMOBJ)):=node_primitive_name(current_node)
   obj_prim_def_node(temp(PRIMOBJ)):=primdefnode;
   % Initialize the counter
   obj_prim_counter(temp(PRIMOBJ)):=1
   obj_prim_arg_list(temp(PRIMOBJ)):=empty_prim_arg_list;
   prim_obj:=temp(PRIMOBJ)
   status:=Get-prim-args
   ENDEXTEND
fi
```

After unwinding of the graph we may find a supercombinator name or a primitive name in the leaf node of the spine. If the leaf node is a primitive we prepare for evaluating the primitive.

A primitive object is made which consists of the primitive name, a primitive node, a counter and (a placeholder for) the arguments.

Get the Ready Primitive Argument

```
if status=Get-prim-args &
  obj_prim_counter(prim_obj) <=
              node_arity(obj_prim_def_node(prim_obj)) &
  is_strict_param(param_type(
                             obj_prim_counter(prim_obj),
                             node_param_info(obj_prim_def_node
                                             (prim_obj)))) &
  is_in_whnf(current_prim_arg_node) &
  has_required_type(node_type(current_prim_arg_node),
                    (param_type(obj_prim_counter(prim_obj),
                                node_param_info(obj_prim_def_node
                                                 (priobj)))));
then
  obj_prim_arg_list(prim_obj):=
         pusharg(obj_prim_arg_list(prim_obj),
                 current_prim_arg_node)
  obj_prim_counter(prim_obj):=incr1(obj_prim_counter(prim_obj))
  addr_stack:=pop_addr(addr_stack)
fi
```

If the argument is in weak head normal form and has a type required by the primitive, the node is given as one of the arguments to the primitive.

Get the Non Strict Primitive Argument

```
if status=Get-prim-args &
  obj_prim_counter(prim_obj) <= node_arity(obj_prim_def_node(prim_obj)) &
  not(is_strict_param(param_type(obj_prim_counter(prim_obj))),
                      node_param_info(obj_prim_def_node(prim_obj)))
then
  obj_prim_arg_list(prim_obj,prim_counter(prim_obj)):=
            pusharg(obj_prim_arg_list(prim_arg_list),
                    current_prim_arg_node)
  obj_prim_counter(prim_obj):=incr1(obj_prim_counter(prim_obj))
  addr_stack:=pop_addr(addr_stack)
fi
```

If the argument is not required to be strict the node representing the argument is placed in the primitive object.

Force the Evaluation of the Primitive Strict Argument

```
if status=Get-prim-args &
  obj_prim_counter(prim_obj) <= node_arity(obj_prim_def_node(prim_obj)) &
  & is_strict_param(param_type(obj_prim_counter(prim_obj))),
                    node_param_info(obj_prim_def_node(prim_obj))
  is_in_in_whnf(current_prim_arg_node))
then
  EXTEND DUMP by temp(DUMP)
     addr_stack_dump(temp(DUMP)):=addr_stack
     prim_obj_dump(temp(DUMP)):=prim_obj
     type_of_dump_el(temp(DUMP)):=PrimObj
     dump_stack:=push_dump(dump_stack,temp(DUMP))
     addr_stack:=make_addr_stack(current_prim_arg_pointer)
  ENDEXTEND
fi
```

If the argument is in a non strict form and a strict form is required by the primitive, then the argument is evaluated. The address stack and primitive object is placed on the top of the dump stack before the evaluation take place.

7.2.6 Apply the Primitive Operator

```
if status=Get-prim-args &
  obj_prim_counter(prim_obj) > node_arity(obj_prim_def_node(prim_obj))
then
    prim_result:=
          apply_primitive(obj_prim_name(prim_obj),
                          (reverse(obj_prim_arg_list(prim_obj)))
    status:=Prim-result
fi
```

After traversing up the spine to the root node of the primitive, the primitive is applied.

```
if status=Prim-result &
   type_of_result(node_result)=Pass-node
then
    current_node:=result_value(prim_result)
    status:=Unwind
fi
```

The result may be some part of the graph. If this is the case, we just let the rootnode of the subgraph be the current node.

```
if status=Prim-result &
   type_of_result(node_result)=Make-node
then
    EXTEND NODE by temp(NODE)
    WITH
        node_value(temp(NODE)):=result_value(prim_result)
        node_type(temp(NODE)):=type_of_node(prim_result)
        current_node:=temp(NODE)
    ENDEXTEND
    status:=Unwind
fi
```

The result may be a computed number or data in some other form. Then we make the node, and we set the current node to this node.

7.3 Handling Indirection Nodes

```
if prim_counter(prim_obj) > 0
   & is_in_whnf(current_arg_node)
   & is_ind_node(current_arg_node)
then
   current_arg_pointer:=
      node_child(1,current_arg_node)
fi
```

If an indirection node occurs it is discarded.

7.4 Too Lazy

7.4.1 Which part of an expression should be treated as strict?

We have seen that all strict arguments given to a primitive have to be evaluated before the primitive can be applied.

What about non strict arguments? For instance we may look at the "cond" primitive. It takes three arguments. The first arguments tests if the second or third arguments are to be applied.

In order to apply the "cond" primitive the first argument has to be evaluated to a boolean value and is certainly a strict argument. Depending on the value of the first argument either the second or the third argument is going to be evaluated.

Since neither the second argument nor the third arguments needs to be evaluated in order to apply the "cond" primitive both the second and third arguments can be considered as lazy arguments.

That means that we may postpone the evaluation of the argument selected when applying the "cond" operator.

On the other hand, after applying "cond" we know if the second or the third argument is going to be evaluated. Hence we may choose for efficience reason not to postpone the evaluation of the selected argument given to the "cond" operator. The second or third arguments may be huge expressions which can be reduced to a simple expression (e.g. numbers).

The analysis to decide which part of an expression should be evaluated as strict can be very complicated. Hence we do not specify how to perform such analysis. We simply assume that the those parts of the expression which should be evaluated is is annotated as strict.

So we simply add an evolving algebra transition to force annotated strict arguments to be evaluated.

The evolving algebra transition for strict arguments given to the primitive is shown above.

7.5 Strict Evaluation of Arguments Given to the Supercombinator Expression

When we introduce primitives we also get the possibility that some parameters of a supercombinator or some arguments given to the supercombinators may be strict.

In case of strict parameters, the body of the supercombinator may use primitives in such ways that some some of the expression substituted for parameters will always be evaluated.

We may also annotate some of the application argument as strict. In such case the definition of the supercombinator body does not imply strictness of the parameters. Instead an particular application of the supercombinator may cause some of the application arguments to be treated as strict.

In both case we assume that an strictness analysis is performed. Here we will refrain from describing the strictness analysis and simply assume that such analysis is done.

We choose to evaluate strict arguments just before substituting the the formal parameter variable by the argument expression. So in addition to keep the address stack on the dump we need to keep the state of the substitution on the dump. We will use the `scobj` as a placeholder to store the state of the substitution process when strict argument is evaluated.

7.5.1 Abbreviations

In addition to the abbreviations in 7.2.3 we define the following abbreviation:

```
sc_definition_node=
   graph(get_addr_from_globals(node_sc_name(current_node)))
current_sc_arg_node=
    graph(node_child(right_branch,graph(top_addr(popstack(addr_stack)))))
current_sc_arg_pointer=
    node_child(right_branch,graph(top_addr(popstack(addr_stack))))
```

7.5.2 The Signature

In addition to the signature defined in 7.2.2 we define the following addition to the signature:

```
% Node
node_sc_name: NODE --> NAME

% Primitive
scobj:            SCOBJ
sc_name:          SCOBJ --> NAME
strict_params_sc: SCOBJ --> BOOL*
sc_counter:       SCOBJ --> INTEGER
sc_param_names:   SCOBJ --> NAME

% Parameters and arguments:
get_subst_arg_addr: SCOBJ x NAME --> TADDR
get_param_name:     NAME* x NUMBER --> NAME

% The dump element
sc_obj_dump:        DUMP --> SCOBJ
```

7.5.3 Make the Supercombinator Object

```
if status=Unwind &
   node_type(current_node)=SCName &
   length_as(addr_stack) > node_arity(sc_definition_node))
then
   EXTEND SCOBJ by temp(SCOBJ)
   WITH
      obj_scdef_node(temp(SCOBJ)):=sc_definition_node
      curr_arity(temp(SCOBJ)):=node_arity(sc_definition_node)
      curr_counter(temp(SCOBJ)):=0
      curr_parmams(temp(SCOBJ)):=node_params(sc_definition_node)
      sc_obj:=temp(SCOBJ)
   ENDEXTEND
   status:=Make-substs
fi
```

We prepare for evaluating of the supercombinator. A supercombinator object is created.

7.5.4 Substitute Non Strict Argument

```
if status=Make-substs &
   curr_counter(sc_obj) < curr_arity(sc_obj) &
   get_sc_param_info(incr1(curr_counter(sc_obj)),
                     curr_params) = Nonstrict
then
   get_subst_value_addr
              (get_param_var(incr1(curr_counter(sc_obj)),
                 curr_params(sc_obj))):=current_sc_arg_pointer
   curr_counter(sc_obj):=incr1(curr_counter(sc_obj)
   addr_stack:=pop_addr(addr_stack)
fi
```

A non-strict argument is added to the list of substitutions. We do not need to evaluate the argument (represented as a subgraph) to weak head normal form.

7.5.5 Substitute Argument in WHNF form

```
if status=Make-substs &
   curr_counter(sc_obj) < curr_arity(sc_obj) &
   get_sc_param_info(incr1(curr_counter(sc_obj)),
                     curr_params) = Strict  &
   is_in_whnf(node_type(current_arg_node)
then
   get_subst_value_addr
              (get_param_var(incr1(curr_counter(sc_obj)),
                 curr_params(sc_obj))):=current_sc_arg_pointer
   curr_counter(sc_obj):=incr1(curr_counter(sc_obj)
   addr_stack:=pop_addr(addr_stack)
fi
```

A strict argument which is in weak head normal form is added to the list of substitutions.

7.5.6 Supercombinator Parameters Annotated as Strict

```
if status=Make-substs &
   curr_counter(sc_obj) < curr_arity(sc_obj) &
   get_sc_param_info(incr1(curr_counter(sc_obj)),
                     curr_params) = Strict  &
   not(is_in_whnf(node_type(current_arg_node))
then
   EXTEND DUMP by temp(DUMP)
   WITH
```

```
      addr_stack_dump(temp(DUMP)):=addr_stack
      scobj_dump(temp(DUMP)):=sc_obj
      type_of_dump_el(temp(DUMP)):=Sc-obj
      dump_stack:=push_dump(dump_stack,temp(DUMP))
      addr_stack:=make_addr_stack(current_sc_arg_pointer)
   ENDEXTEND
   status=Unwind
fi
```

If a parameter of a supercombinator is annotated as strict, then the argument which replace the variable is evaluated before the substitutions is performed.

7.5.7 Restore the Dump

```
if status=Unwind &
   is_in_whnf(current_node) &
   dump_stack =/= empty_dump_stack &
   type_of_dump_el(top_dump(dump_stack)) = Scobj
then
    addr_stack:=addr_stack_dump(top_dump(dump_stack))
    sc_obj:=sc_obj_dump(top_dump(dump_stack))
    node_child(right_branch,graph(top_addr(pop_stack
             addr_stack_dump(top_dump(dump_stack))))):=
             top_stack(addr_stack)
    dump_stack:=pop_dump(dump_stack)
    status=Make-substs
fi
```

The top element of the dump is restored if the expression is in weak head normal form and the dump stack is non-empty.

7.5.8 Application node annotated as strict

```
if status=Make-instance &
   not(is_in_whnf(node_type(current_sc_arg_node)) &
   is_strict_app_node(current_node)
then
   EXTEND DUMP by temp(DUMP)
   WITH
      addr_stack_dump(temp(DUMP)):=addr_stack
      scobj_dump(temp(DUMP)):=sc_obj
      type_of_dump_el(temp(DUMP)):=Sc-obj
      dump_stack:=push_dump(dump_stack,temp(DUMP))
      addr_stack:=make_addr_stack(current_sc_arg_pointer)
   ENDEXTEND
   status=Unwind
fi
```

If the argument given to the application node which is annotated as strict, this subgraph is evaluated before the evaluation of the application continues. When the subgraph is evaluated, a redex on weak head normal form will cause the dump to be restored. Note also the possibility of recursive evaluation of strict arguments.

7.6 The G-machine Evaluator

A compiled instruction sequence needs to have instructions for saving a the address stack before evaluating the strict argument and restore the address stack after the evaluating is finished.

The "eval" instruction cause the G-machine to save the address stack on the dump stack. In addition we need to extend the Unwind instruction to restore the address stack when an argument is evaluated to weak head normal form.

7.6.1 The Signature

```
% Instructions
instr_stack: INSTR*
top_instr:   INSTR* --> INSTR
pop_instr:   INSTR* --> INSTR*
make_g_code: INSTR --> INSTR*

% Instruction stack
instr_stack_dump: DUMP --> INSTR*

get_operator: INSTR --> OPERAND
```

We will use the following abbreviations:

```
current_node:=graph(top_addr(addr_stack))
primitive_definition_node=
   graph(get_addr_from_globals(node_primitive_name(current_node)))
primitive_definition_addr=
   get_addr_from_globals(node_primitive_name(current_node)))
```

7.6.2 Push Boolean Data

```
if status=Exec-code &
   get_operator(top_instr(instr_stack))=Pushprimglobal
then
   EXTEND NODE by temp(NODE)
   EXTEND TADDR by temp(TADDR)
   WITH
       graph(temp(TADDR):=temp(NODE)
       node_type(temp(NODE):=PRIMName
       node_value(temp(NODE):=get_operand(1,top_instr(instr_stack))
       addr_stack:=push_stack(addr_stack,temp(TADDR))
```

```
fi
```

The specification above tells how a node holding a name of a primitive is made.

7.6.3 Push Boolean Data

```
if status=Exec-code &
   get_operator(top_instr(instr_stack))=Pushbool
then
   EXTEND NODE by temp(NODE)
   EXTEND TADDR by temp(TADDR)
   WITH
       graph(temp(TADDR):=temp(NODE)
       node_type(temp(NODE):=Bool
       node_value(temp(NODE):=get_operand(1,top_instr(instr_stack))
       addr_stack:=push_stack(addr_stack,temp(TADDR))
fi
```

How to make a node holding a boolean value is specified above.

The node holding unspecified data is made in a similar way.

7.6.4 The Primitive Operators

```
if status=Exec-code &
   get_operator(top_instr(instr_stack))=Add
then
   resul_tvalue:=get_result_value
        apply_primitive(``plus'',
           make_num_arg_list(
           node_value(graph(top_addr(addr_stack)))
           node_value(graph(top_addr(pop_stack(addr_stack)))) ))
   status:=Make-num-node
fi
```

Above is the specification for the Add instruction.

```
if status=Make-num-node
then
   EXTEND NODE by temp(NODE)
   EXTEND TADDR by temp(TADDR)
   WITH
       graph(temp(TADDR):=temp(NODE)
       node_type(temp(NODE):=Num
       node_value(temp(NODE):=result_value
       addr_stack:=push_stack(pop_stack(pop_stack(addr_stack)),
           temp(TADDR))
fi
```

We need also to make a new node in the graph holding the result of the addition.

```
if status=Exec-code &
   get_operator(top_instr(instr_stack))=Cond
then
   instr_stack:=concat_cond_code(
        get_result_value(apply_primitive(``if'',
           make_cond_arg_list(
           node_value(graph(top_addr(addr_stack)))
           get_operand(1,top_instr(instr_stack)),
           get_operand(2,top_instr(instr_stack)) ))),
        pop_instr(addr_stack))
    addr_stack:=pop_stack(addr_stack)
fi
```

Above is the specification for the Cond instruction.

7.6.5 Executing the Eval Instruction

```
if status=Exec-code &
   get_operator(top_instr(instr_stack))=Eval &
   not(is_in_whnf(graph(top_addr(addr_stack))))
then
   EXTEND DUMP by temp(DUMP)
          INSTR by temp(INSTR)
        addr_stack_dump(temp(DUMP)):=pop_stack(addr_stack)
        instr_stack_dump(temp(DUMP)):=pop_instr(instr_stack)
        dump_stack:=push_dump(temp(DUMP),dump_stack)
        % Set new initial values
        addr_stack:=push_stack(empty_stack,top_addr(addr_stack))
        get_operator(temp(INSTR)):=Unwind
        instr_stack:=make_g_code(temp(INSTR))
   ENDEXTEND
fi
```

In the specification above the Eval instruction causes a dump to be made and the strict argument is evaluated into Weak Head Normal Form.

```
if status=Exec-code &
   get_operator(top_instr(instr_stack))=Eval &
   is_in_whnf(graph(top_addr(addr_stack)))
then
  instr_stack:=pop_instr(instr_stack)
fi
```

The argument is already in Weak Head Normal Form, so the "Eval" instruction is simply popped of the stack.

7.6.6 Restore the Stack on the Dump

```
if status=Exec-code &
   get_operator(top_instr(instr_stack))=Unwind &
   is_in_whnf(current_node) &
   dump_stack =/= empty_dump_stack
then
   addr_stack:=push_stack(addr_stack_dump(top_dump(dump_stack)),
               top_addr(addr_stack))
   instr_stack:=instr_stack_dump(top_dump(dump_stack))
   dump_stack:=pop_dump(dump_stack)
fi
```

When the graph is evaluated to weak head normal form, we tests if there are elements on the dump stack. If so the top element on the dump stack restores the address stack and the instruction stack.

```
if status=Exec-code &
   get_operator(top_instr(instr_stack))=Unwind &
   nodetype(graph(top_addr(addr_stack)))=PrimName
then
   curr_glob_def_addr:=primitive_definition_addr
   instr_stack:=pop_instr_stack(instr_stack)
   status:=Exec-sc-def

if status=Exec-code &
   node_type(graph(curr_glob_def_addr)=Global &
   length_as(addr_stack) > def_arity(graph(curr_glob_def_addr)
then
   instr_stack:=concat_code(finished_code(graph(curr_glob_def_addr)),
                  instr_stack)
   status:=Exec-code
```

7.7 Extensions Needed to Compile Primitives and Supercombinators

7.7.1 Primitive Name Node

We need a new type of node holding the primitive name. The primitive name refers to the primitive definition node and is used in the process of evaluating supercombinator definitions where the body define a primitive expression.

7.7.2 Annotate Strict and Lazy arguments and Parameters

Since we can have both strict and lazy arguments we want to mark which arguments are strict and which arguments are lazy in the compilation process.

We may assume that some strictness analysis is performed. Here we do not specify how we perform such analysis.

We may annotate the strict expressions in two ways:

- If we know that the some of arguments given to a supercombinator definition or a primitive definition always will be evaluated, the arguments which corresponds to the strict parameters may be annotated as strict.

- If an expression which is argument to an application is known to be strict in this particular case, then we mark the application argument as strict, meaning that the argument is to be evaluated before the supercombinator or primitive expression.

7.8 Extending the compiler to handle primitives

7.8.1 The Signatures

```
% Definition of primitive definitions source.
status:             STATUS
src_prim_defs,
   empty_prim_defs: PRIMDEF*
current_primitive:  PRIMDEF
get_next_primitive: PRIMDEF* --> PRIMDEF
tail_prim_defs:     PRIMDEF* --> PRIMDEF*

% Adding primitive to linked list.
curr_prim_addr:          TADDR
curr_prim_name:          NAME
prim_name:               PRIMDEF --> NAME
get_addr_from_globals:   NAME --> TADDR
get_name_from_globals:   TADDR --> NAME

% Adding properties to the primitive nodes
get_prim_func:            PRIMDEF --> FUNCTION
get_prim_arity:           PRIMDEF --> NUMBER
get_prim_strict_params:   PRIMDEF --> BOOL*
get_prim_required_types: PRIMDEF --> TYPE*
```

7.8.2 Abbreviation

To get the value from the top of the compile stack we use the abbreviation as listed below.

```
current_value==value_of_addr(top_addr(temp_c_stack))
current_node==graph(top_addr(temp_c_stack))
```

7.8.3 Changes needed in order to handle primitives

We need to extend the target graph to hold information about the primitives. The primitive can be considered as built-in functions or external defined functions.

At least we need to introduce a new type of leaf node holding the primitive names in the body of a supercombinator definition.

In addition we extend the global association list to hold the name of the primitives in addition to the name of the supercombinators. The graph is extended to keep single nodes for every primitive defined. The primitive definition nodes will hold information necessary to evaluate the arguments of the primitives and to apply the primitives. This way we can deal with primitives in much the same way as supercombinator definitions.

7.8.4 Compiling Primitive Definitions

```
if status:=Find-sc-def &
   empty_sc_defs(all_prim_defs)
then
   status:=Find-prim-def
fi

if status:=Find-prim-def &
   not(empty_prim_defs(all_prim_defs)
then
   current_primitive:=get_next_primitive(src_prim_defs)
   src_prim_defs:=tail_prim_defs(src_prim_defs)
   status:=Make-glob-primitive
fi

if status=Get=curr-prim-def &
   empty_prim_defs(all_prim_defs)
then
   status:=Perform-graph-reds
```

If we assume that we can obtain some informations about the primitive functions, we let the specification above iterate through all primitive definitions to get information about the primitives.

All primitive definitions are processed after processing of the supercombinator definitions.

7.8.5 Adding primitives to the globals

```
if status=Make-glob-primitive
then
   EXTEND TADDR by temp(TADDR)
   WITH
   curr_prim_addr:=temp(TADDR)
   curr_prim_name:=prim_name(current_primitive)
```

```
   get_addr_from_globals(prim_name(current_primitive)):=temp(TADDR)
   get_name_from_globals(temp(TADDR)):=prim_name(current_primitive)
   ENDEXTEND
   status:=Make-prim-node
```

We handle primitives as much as possible in the same ways as supercombinator definitions. So we add primitive names to the global association list. The addresses on the association list will points to a node holding the primitive definition. All properties about the primitive will be associated with the primitive definition node.

7.8.6 Adding the primitive definition node to the graph

```
if status:=Make-prim-node
then
   EXTEND NODE by temp(NODE)
   WITH
      % Makes node
      graph(curr_prim_addr):=temp(NODE)
      node_type(temp(NODE)):=Primitive
      % --- Add the annotation of strict parameters to the node
      strict_params_info(temp(NODE)):=
             get_prim_strict_params(current_primitive)
      % --- Add the information about the arity node.
      node_arity(temp(NODE)):=
            get_prim_arity(current_primitive)
      % --- Add information of the expected type of the arguments
      % --- which corresponds to the parameters.
      required_types_info(temp(NODE)):=
             get_prim_required_types(current_primitive)
   ENDEXTEND
   status:=Compile-the-expression
```

The definitions in the graph is extended with all defined primitives. A primitive node will always be a single leaf node in the definition part of the graph. The primitive nodes is used to hold the information about the primitive. The information associated with the node are arity of the primitive, which arguments are strict, required types of strict arguments and some representation of the operator of the primitive.

7.9 Extend Supercombinator Definitions to Handle Primitives

We modify the specification to add information about strict supercombinator parameters.

In addition we define the transitions needed to compile the definition of primitive name nodes.

7.9.1 The Signature

```
% Address stack
initial_stack,
      temp_c_stack:          TADDR*
is_empty_stack:              TADDR --> BOOL
push2_stack:                 TADDR x TADDR x TADDR* --> TADDR*
value_of_addr:               TADDR --> SCEXPR

% Supercombinator definition
src_curr_sc_def:             SCDEF
make_params:                 SCDEF --> NAME*
src_body:                    SCDEF --> SCEXPR
get_sc_arity:                SCDEF --> NUMBER

% Supercombinator expressions
first_app_expr:              SCEXPR --> SCEXPR
second_app_expr:             SCEXPR --> SCEXPR
expr_type:                   SCEXPR --> SCTYPE


% Node properties
node_params:                 NODE --> NAME*
node_child:                  NUMBER x NODE --> TADDR

% Oracles
get_strict_params:           BOOL*
get_strict_app_arg:          BOOL
```

7.9.2 Abbreviation

```
current_value==value_of_addr(top_addr(temp_c_stack))
current_node==graph(top_addr(temp_c_stack))
```

7.9.3 Making the a node holding the name of the primitive

```
if status=Compile-the-expression
   & not(is_empty_stack(temp_c_stack))
   & expr_type(current_value)=PrimName
then
   EXTEND NODE by temp(NODE)
        current_node:=temp(NODE)
        current_value:=empty_expr
        node_type(temp(NODE)):=PrimName
        node_primitive_name(temp(NODE)):=
              make_prim_name(current_value)
   ENDEXTEND
```

The primitive name node is analog to the supercombinator name node.

This node will occur in the graph of the supercombinator body.

We use the primitive name to look up the primitive definition node.

7.9.4 Mark the Strict Parameter of a Supercombinator

```
if status=Compile-sc-def
then
   EXTEND TADDR by temp(TADDR)
          NODE by temp(NODE)
   WITH
      % Makes node
      graph(curr_sc_def_addr):=temp(NODE)
      node_type(temp(NODE)):=Supercomb
      node_params(temp(NODE)):=
         make_params(value_of_addr(curr_sc_def_addr))
      % --- Add the annotation of strict or lazy parameters
      % --- to the node.
      strict_params_info(temp(NODE))
             :=make_strict_info(value_of_addr(curr_sc_def_addr))
      % --- Add the information about the arity
      node_arity(temp(NODE)):=
            get_sc_arity(value_of_addr(curr_sc_def_addr))
      node_child(1,temp(NODE)):=temp(TADDR)
      % Initialize the compile stack.
      value_of_addr(temp(TADDR)):=src_body(src_curr_sc_def)
      temp_c_stack:=push_stack(temp(TADDR),initial_stack)
   ENDEXTEND
   status:=Compile-the-expression
```

Here specify how to make a supercombinator node. This node will be the root node of the supercombinator definition graph.

Parameters of the supercombinator may be annotated as strict or non-strict.

The null-ary function `get_strict_params` is not a used as a constant. Instead this function can be considered as an oracle because we do not want to specify how we perform the strictness analysis.

7.10 Add the Boolean Node

```
if status=Compile-the-expression
   not(is_empty_stack(temp-c_stack))
   & expr_type(current_value)=Numexpr
then
   EXTEND NODE by temp(NODE)
          current_node:=temp(NODE)
          current_value:=empty_expr
          node_type(temp(NODE)):=Num
          node_value(temp(NODE)):=
```

```
            make_boolean(current_value)
   ENDEXTEND
```

This transition specify how to make a node containing boolean data.

We make similar Evolving Algebra transition, to specify how to make a node containing numeric or some other unspecified data. The function `node_value` is used to store the numeric, boolean or unspecified data.

7.11 Add strict annotation to the application node

```
if status=Compile-the-expression
   not(is_empty_stack(temp-c_stack))
   & expr_type(current_value)=APexpr
then
   EXTEND TADDR by temp(TADDR,1),temp(TADDR,2)
          NODE by temp(NODE)
          % Makes the node.
          current_node:=temp(NODE)
          current_value:=empty_expr
          node_child(1,temp(NODE)):=temp(TADDR,1)
          node_child(2,temp(NODE)):=temp(TADDR,2)
          node_type(temp(NODE)):=APnode
          % --- State if the argument to the application
          % --- node is strict or non-strict.
          is_strict_app_node(temp(NODE)):=get_strict_app_arg
          % Makes elements to the compile stack.
          value_of_addr(temp(TADDR,1)):=
                first_app_expr(current_value)
          value_of_addr(temp(TADDR,2)):=
                second_app_expr(current_value)
          temp_c_stack:=push2_stack
                (temp(TADDR,2),temp(TADDR,1),temp_c_stack)
   ENDEXTEND
```

The transition below specify how to make an application node. We annotate the application node as strict or non-strict. If the application is annotated as strict, the argument expression of the application will be evaluated before the function expression.

As for supercombinators we do not specify how we obtain the strict annotation. So the function `get_strict_app_arg` is to be used as an oracle.

7.12 Generating Compiled Instruction from the Primitive Definitions

When we introduce primitives we know that some arguments are strict. At least we know the strict arguments given to the primitives. For instance all arguments given to the addition primitives are strict. So we can make

a compile scheme, where we add the "eval" instruction after pushing every strict argument.

7.12.1 The Signature

```
% Signatures
signature status : STATUS
signature add: ((NUMBER x NUMBER) --> NUMBER)

signature tempstack: (TADDR *)
signature emptystack: (TADDR *)
signature initialstack: (TADDR *)
signature isemptystack: ((TADDR *) --> BOOL)
signature topaddr: ((TADDR *) --> TADDR)
signature pushstack: (((TADDR *) x TADDR) --> (TADDR *))
signature popstack: ((TADDR *) --> (TADDR *))

signature currscdefaddr: TADDR
signature valueofaddr: (TADDR --> [SCEXPR + INSTR])
signature graph: (TADDR --> NODE)

signature mainscdefname: SCNAME
signature getmainname: ((SCEXPR *) --> SCNAME)
signature allscdefs: (SCEXPR *)
signature isemptyscdefs: ((SCEXPR *) --> BOOL)
signature getnextscdef: ((SCEXPR*) --> SCEXPR)
signature tailscdefs: ((SCEXPR*) --> (SCEXPR*))

signature currprimdefaddr: TADDR
signature allprimdefs: (PRIMEXPR *)
signature isemptyprimdefs: (PRIMEXPR --> BOOL)
signature getnextprimdef: ((PRIMEXPR *) --> PRIMEXPR)
signature tailprimdefs: ((PRIMEXPR *) --> (PRIMEXPR*))

% Primitive expressions
signature getprimdefname: (PRIMEXPR --> PRIMNAME)
signature makeprimparamposinfolist: (PRIMEXPR --> (PPOSINFO *))
signature makeprimarity: (PRIMEXPR --> NUMBER)

% Primitive informations
signature primexprtype: (TADDR --> PRIMEXPRTYPE)
signature nameofprimitive: (TADDR --> PRIMNAME)
signature getprimposparaminfolist: (TADDR --> (PPOSINFO *))
signature getprimposition: (((PPOSINFO *) x NUMBER) --> NUMBER)
signature primposition: (TADDR --> NUMBER)
signature getpriminfoparam: (((PPOSINFO *) x NUMBER) --> PPINFO)
signature priminfoparam: (TADDR --> PPINFO)
signature secondprimpos: (TADDR --> NUMBER)
```

```
signature thirdprimpos: (TADDR --> NUMBER)
signature defprimarity: (TADDR --> NUMBER)

signature makeprimname: (SCEXPR --> PRIMNAME)

signature getnamefromglobals: (TADDR --> SCNAME)
signature getaddrfromglobals: (SCNAME --> TADDR)

signature emptyexpr: SCEXPR
signature currscdef: SCEXPR
signature exprtype: (SCEXPR --> SCTYPE)
signature makeparams: (SCEXPR --> (SCEXPR *))
signature getscdefname: (SCEXPR --> SCNAME)
signature srcbody: (SCEXPR --> SCEXPR)
signature firstappexpr: (SCEXPR --> SCEXPR)
signature secondappexpr: (SCEXPR --> SCEXPR)
signature makenum: (SCEXPR --> NUMBER)
signature makebool: (SCEXPR --> BOOLDATA)
signature makedata: (SCEXPR --> DATA)
signature makescname: (SCEXPR --> SCNAME)
signature makevarname: (SCEXPR --> VARNAME)

signature numberofparams: ((SCEXPR *) --> NUMBER)
signature getparamlist: (TADDR --> (PARAMPOS *))
signature makeparamposlist: (SCEXPR --> (PARAMPOS *))
signature incrementposlist: ((NUMBER x (PARAMPOS *)) --> (PARAMPOS *))
signature getposition: (((PARAMPOS *) x VARNAME) --> NUMBER)

% Info about which sc parameters is strict
signature getparaminfo: (((PARAMPOS *) x VARNAME) --> SCPINFO)

signature getoperator: (INSTR --> OPERATOR)
signature getoperand: ((NUMBER x INSTR) --> OPERAND)

signature codelist: (INSTR *)
signature instrstack: (INSTR *)
signature emptycodelist: (INSTR *)
signature makegcode: (INSTR --> (INSTR *))
signature makegcodetwo: ((INSTR x INSTR) --> (INSTR *))
signature concatcode: (((INSTR *) x (INSTR *)) --> (INSTR *))

signature instructions: (TADDR --> (INSTR *))
signature hascode: (TADDR --> BOOL)

signature nodetype: (NODE --> NTYPE)
signature defarity: (NODE --> NUMBER)
signature finishedcode: (NODE --> (INSTR *))
signature leftbranch: NUMBER
```

```
signature rightbranch: NUMBER
```

7.12.2 Start Compilation of the Primitive Definitions

```
if  status=Get-curr-sc-def &
     isemptyscdefs(allscdefs)
then
   status:=Get-curr-prim-def;
endupdates
```

This transitions initiate the compilation of primitive definitions.

7.12.3 Prepare for Compilation of the a Primitive Definition

```
if    status=Get-curr-prim-def &
      not(isemptyprimdefs(allprimdefs));
then
   EXTEND TADDR by temp(TADDR)
       currprimdefaddr:=temp(TADDR,1);
       getnamefromglobals(temp(TADDR,1)):=
           getprimdefname(getnextprimdef(allprimdefs));
       getaddrfromglobals(getprimdefname
          (getnextprimdef(allprimdefs))):=temp(TADDR,1);
       valueofaddr(temp(TADDR,1)):=getnextprimdef(allprimdefs);
   ENDEXTEND
    allprimdefs:=tailprimdefs(allprimdefs);
    status:=Compile-prim-def;
endupdates
```

The next primitive definition is taken from the list of primitive definitions. A new address is given to the primitive definition.

7.12.4 The Primitive Definition

```
if = (status, Compile-prim-def);
then
   EXTEND TADDR by temp(TADDR,1),temp(TADDR,2)
          NODE by temp(NODE)
          INSTR by temp(INSTR,1),temp(INSTR,2)
       defarity(temp(NODE)):=
                 makeprimarity(valueofaddr(currprimdefaddr))
       graph(currprimdefaddr):=temp(NODE);
       nodetype(temp(NODE)):=Global;
       primexprtype(temp(TADDR,2)):=NameParams;
       nameofprimitive(temp(TADDR,2)):=
                 getnamefromglobals(currprimdefaddr));
       getprimposparaminfolist(temp(TADDR,2)):=
                    makeprimparamposinfolist(valueofaddr(currprimdefaddr));
       defprimarity(temp(TADDR,2)):=
```

```
                    makeprimarity(valueofaddr(currprimdefaddr));
       hascode(temp(TADDR,2)):=False;
% Slide n + 1, Unwind
       getoperator(temp(INSTR,1)):=Slide;
       getoperand(1,temp(INSTR,1)):=
                 add(1,makeprimarity(valueofaddr(currprimdefaddr))));
       getoperator(temp(INSTR,2)):=Unwind;
       instructions(temp(TADDR,1)):=
                 makegcodetwo(temp(INSTR,1),temp(INSTR,2));
       hascode(temp(TADDR,1)):=True;
       tempstack:=pushstack(
                     pushstack(emptystack,temp(TADDR,1)),
                     temp(TADDR,2));
   ENDEXTEND
   status:=Compile-part-of-primdef;
```

The first step in compiling a primitive definition is performed. A temporary stack of addresses is set up. This stack will hold the state of the compilation process.

The two last G-machine instructions in the sequence of instructions for the primitive is made:

```
Slide <n + 1>
Unwind
```

The number n is the arity of the primitive.

Instead of generating the slide instruction , we could have generated instructions to update the root of the redex. In this way it is possible to avoid repeatedly generation of the same pieces of graph representing the supercombinator body.

7.12.5 Make the List of Parameters for Primitives of Arity One

```
if   status=Compile-part-of-primdef &
     not(isemptystack(tempstack)) &
     hascode(topaddr(tempstack))=False &
     primexprtype(topaddr(tempstack))=NameParams &
     defprimarity(topaddr(tempstack))=1
then
   EXTEND TADDR by temp(TADDR,1),temp(TADDR,2)
      % First parameter
       primexprtype(temp(TADDR,2)):=Param;
       primposition(temp(TADDR,2)):=
                 getprimposition(getprimposparaminfolist
                         (topaddr(tempstack)),1)
       priminfoparam(temp(TADDR,2)):=
                 getpriminfoparam(getprimposparaminfolist
                         (topaddr(tempstack)),1)
```

```
      hascode(temp(TADDR,2)):=False;
     % The built-in function
      primexprtype(temp(TADDR,1)):=Priminstr;
      nameofprimitive(temp(TADDR,1)):=
                   nameofprimitive(topaddr(tempstack));
      hascode(temp(TADDR,1)):=False;
     % Put addresses on the address stack
      tempstack:=pushstack(pushstack
                (popstack(tempstack),temp(TADDR,1)),
                                    temp(TADDR,2));
   ENDEXTEND
```

The transition above prepare for compilation of parameters for one-ary primitives.

7.12.6 Make the List of Parameters for Primitives of Arity Two

```
if   status=Compile-part-of-primdef &
     (not(isemptystack(tempstack)) &
     hascode(topaddr(tempstack))=False &
     primexprtype(topaddr(tempstack))=NameParams &
     defprimarity(topaddr(tempstack))=2
then
   EXTEND TADDR by temp(TADDR,2),temp(TADDR,2),temp(TADDR,3);
      % Second parameter
      primexprtype(temp(TADDR,3)):=Param;
      primposition(temp(TADDR,3)):=
                getprimposition(getprimposparaminfolist
                        (topaddr(tempstack)),2)
      priminfoparam(temp(TADDR,3)):=
                getpriminfoparam(getprimposparaminfolist
                        (topaddr(tempstack)),2)
      hascode(temp(TADDR,3)):=False;
     % First parameter
      primexprtype(temp(TADDR,2)):=Param;
      primposition(temp(TADDR,2)):=
                add(getprimposition(getprimposparaminfolist
                          (topaddr(tempstack)),1),1);
      priminfoparam(temp(TADDR,2)):=
                getpriminfoparam(getprimposparaminfolist
                            (topaddr(tempstack)),1)
      hascode(temp(TADDR,2)):=False;
     % The built-in function
      primexprtype(temp(TADDR,1)):=Priminstr
      nameofprimitive(temp(TADDR,1)):=
                   nameofprimitive(topaddr(tempstack));
      hascode(temp(TADDR,1)):=False;
```

```
      % Put addresses on the address stack
       tempstack:=pushstack(pushstack(
                    pushstack(popstack(tempstack),temp(TADDR,1)),
                         temp(TADDR,2)),temp(TADDR,3));
   ENDEXTEND
```

The transition above prepare for compilation of parameters for two-ary primitives.

7.12.7 Make the Parameter List for the If Primitive

```
if   status=Compile-part-of-primdef &
     not(isemptystack(tempstack)) &
     hascode(topaddr(tempstack))=False &
     primexprtype(topaddr(tempstack))=NameParams &
     defprimarity(topaddr(tempstack))=3 &
     nameofprimitive(topaddr(tempstack))=If
then
   EXTEND TADDR by temp(TADDR,2),temp(TADDR,2);
      % First parameter
       primexprtype(temp(TADDR,2)):=Param;
       primposition(temp(TADDR,2)):=
                 getprimposition(getprimposparaminfolist
                         (topaddr(tempstack)),1)
       priminfoparam(temp(TADDR,2)):=
                 getpriminfoparam(getprimposparaminfolist
                         (topaddr(tempstack)),1)
       hascode(temp(TADDR,2)):=False;
      % The built-in function
       primexprtype(temp(TADDR,1)):=Priminstr
       nameofprimitive(temp(TADDR,1)):=
                     nameofprimitive(topaddr(tempstack))
       secondprimpos(temp(TADDR,1)):=
                 getprimposition(getprimposparaminfolist
                         (topaddr(tempstack)),2)
       thirdprimpos(temp(TADDR,1)):=
                 getprimposition(getprimposparaminfolist
                         (topaddr(tempstack)),3)
       hascode(temp(TADDR,1)):=False;
      % Put addresses on the address stack
       tempstack:=pushstack(pushstack
                    (popstack(tempstack),temp(TADDR,1)),
                                    temp(TADDR,2));
   ENDEXTEND
```

The transition above prepare for compilation of parameters for the three-ary `if` primitive.

7.12.8 Make the G-machine Code for a Strict Argument Given to the Primitive

```
if   status=Compile-part-of-primdef &
     not(isemptystack(tempstack)) &
     hascode(topaddr(tempstack))=False &
     primexprtype(topaddr(tempstack))=Param &
     priminfoparam(topaddr(tempstack))=Strict
then
   EXTEND INSTR by temp(INSTR,1),temp(INSTR,2);
       getoperator(temp(INSTR,1)):=Push;
       getoperand(1,temp(INSTR,1)):=
                     primposition(topaddr(tempstack));
       getoperator(temp(INSTR,2)):=Eval;
       instructions(topaddr(tempstack)):=
                makegcodetwo(temp(INSTR,1),temp(INSTR,2));
       hascode(topaddr(tempstack)):=True;
   ENDEXTEND
```

This transition makes the G-machine code for handling strict arguments given to the primitive.

7.12.9 Make the G-machine Code for a Non Strict Argument Given to the Primitive

```
if   status=Compile-part-of-primdef) &
     not(isemptystack(tempstack)) &
     hascode(topaddr(tempstack))=False &
     primexprtype(topaddr(tempstack))=Param &
     priminfoparam(topaddr(tempstack))=Nonstrict
then
   EXTEND INSTR by temp(INSTR);
       getoperator(temp(INSTR)):=Push;
       getoperand(1,temp(INSTR)):=
                     primposition(topaddr(tempstack));
       instructions(topaddr(tempstack)):=
                makegcode(temp(INSTR));
       hascode(topaddr(tempstack)):=True;
   ENDEXTEND
```

This transition makes the G-machine code for handling non strict arguments given to the primitive.

7.12.10 Make the G-machine Instruction for the Addition

```
if   status=Compile-part-of-primdef &
     not(isemptystack(tempstack)) &
     hascode(topaddr(tempstack))=False &
     primexprtype(topaddr(tempstack))=Priminstr &
```

```
     nameofprimitive(topaddr(tempstack))=Plus
then
   EXTEND INSTR by temp(INSTR);
       getoperator(temp(INSTR)):=Add;
       instructions(topaddr(tempstack)):=
                 makegcode(temp(INSTR));
       hascode(topaddr(tempstack)):=True;
   ENDEXTEND
```

This transition makes the G-machine code for the addition operation.

7.12.11 Make the G-machine Instruction for the Negation

```
if   status=Compile-part-of-primdef &
     not(isemptystack(tempstack)) &
     hascode(topaddr(tempstack))=False &
     primexprtype(topaddr(tempstack))=Priminstr &
     nameofprimitive(topaddr(tempstack))=Negate
then
   EXTEND INSTR by temp(INSTR);
       getoperator(temp(INSTR)):=Neg;
       instructions(topaddr(tempstack)):=
                 makegcode(temp(INSTR));
       hascode(topaddr(tempstack)):=True;
   ENDEXTEND
```

This transition makes the G-machine code for the unary negation operation.

7.12.12 Make the G-machine Instruction for the If Primitive

```
if   status=Compile-part-of-primdef &
     not(isemptystack(tempstack)) &
     hascode(topaddr(tempstack))=False &
     primexprtype(topaddr(tempstack))=Priminstr &
     nameofprimitive(topaddr(tempstack))=If
then
   EXTEND INSTR by temp(INSTR,1),temp(INSTR,2),temp(INSTR,3)
       getoperator(temp(INSTR,1)):=Push;
       getoperand(1,temp(INSTR,1)):=
                   secondprimpos(topaddr(tempstack));
       getoperator(temp(INSTR,2)):=Push;
       getoperand(1,temp(INSTR,2)):=
                   thirdprimpos(topaddr(tempstack));
       getoperator(temp(INSTR,3)):=Cond;
       getoperand(1,temp(INSTR,3)):=
                     makegcode(temp(INSTR,1));
       getoperand(2,temp(INSTR,3)):=
                     makegcode(temp(INSTR,2));
       instructions(topaddr(tempstack)):=
```

```
                makegcode(temp(INSTR,3));
        hascode(topaddr(tempstack)):=True;
   ENDEXTEND
```

This transition makes the G-machine code for the condition operation.

7.12.13 Traverse up

```
if   status=Compile-part-of-primdef &
     not(isemptystack(tempstack)) &
     hascode(topaddr(tempstack))=True
then
   % The instruction is always appendended at the end of
   % the instruction list.
    codelist:=concatcode(codelist,
                              instructions(topaddr(tempstack)));
    tempstack:=popstack(tempstack);
```

This transition adds the ready G-machine code to the sequence of G-machine instructions for the primitive.

7.12.14 End of the Primitive Definition

```
if   status=Compile-part-of-primdef &
     isemptystack(tempstack)
then
    status:=Get-curr-prim-def;
    finishedcode(graph(currprimdefaddr)):=codelist;
    codelist:=emptycodelist;
```

The finished sequence of G-machine instructions is saved in the node for the primitive definition.

7.12.15 Prepare for Executing the G-machine Code after Compiling All the Primitives and Supercombinators

```
if   status=Get-curr-prim-def
     isemptyprimdefs(allprimdefs)
then
   EXTEND INSTR by temp(INSTR,1),temp(INSTR,2)
          getoperator(temp(INSTR,1)):=Pushglobal;
          getoperand(1,temp(INSTR,1)):=mainscdefname;
          getoperator(temp(INSTR,2)):=Unwind;
          instrstack:=
                 makegcodetwo(temp(INSTR,1),temp(INSTR,2));
   ENDEXTEND
    status:=Exec-code;
    leftbranch:=1
    rightbranch:=2
```

After the compilation of all the primitives instruction stack is initialized with the G-machine instructions sequence which starts the execution of G-machine instructions.

The initial sequence of G-machine instructions is:

```
Pusglobal <main-sc-definition-name>
Unwind
```

7.13 Extending the Supercombinator Definition

7.13.1 Compiling the Boolean Expression

```
if   status=Compile-the-body &
     not(isemptystack(tempstack)) &
     exprtype(valueofaddr(topaddr(tempstack)))=BOOLexpr &
     hascode(topaddr(tempstack))=False
then
   EXTEND INSTR by temp(INSTR)
       getoperator(temp(INSTR)):=Pushbool;
       getoperand(1,temp(INSTR)):=
                 makebool(valueofaddr(topaddr(tempstack)));
       instructions(topaddr(tempstack)):=
                 makegcode(temp(INSTR));
       hascode(topaddr(tempstack)):=True;
   ENDEXTEND
```

Instruction for making a boolean node is made.

7.13.2 Compiling the Data Expression

```
if   status=Compile-the-body &
     not(isemptystack(tempstack)) &
     exprtype(valueofaddr(topaddr(tempstack)))=DATAexpr &
     hascode(topaddr(tempstack))=False
then
   EXTEND INSTR by temp(INSTR)
       getoperator(temp(INSTR)):=Pushdata;
       getoperand(1,temp(INSTR)):=
                 makedata(valueofaddr(topaddr(tempstack)));
       instructions(topaddr(tempstack)):=
                 makegcode(temp(INSTR));
       hascode(topaddr(tempstack)):=True;
   ENDEXTEND
```

Instruction for making a data node is made.

7.13.3 Compiling the Primitive Name Expression

```
if   status=Compile-the-body &
```

```
     not(isemptystack(tempstack)) &
     exprtype(valueofaddr(topaddr(tempstack)))=PrimName &
     hascode(topaddr(tempstack))=False
then
   EXTEND INSTR by temp(INSTR);
   withupdates
       getoperator(temp(INSTR)):=Pushprimglobal;
       getoperand(1,temp(INSTR)):=
                 makeprimname(valueofaddr(topaddr(tempstack)));
       instructions(topaddr(tempstack)):=
                 makegcode(temp(INSTR));
       hascode(topaddr(tempstack)):=True;
   ENDEXTEND
```

Instruction for making a primitive node is made.

7.13.4 Compiling the Variable Expression when the Corresponding Supercombinator Parameter is Marked as Strict

```
if   status=Compile-the-body &
     not(isemptystack(tempstack)) &
     exprtype(valueofaddr(topaddr(tempstack)))=VARname &
     getparaminfo(getparamlist(topaddr(tempstack)),
        makevarname(valueofaddr(topaddr(tempstack))))=Strict &
     hascode(topaddr(tempstack))=False
then
   EXTEND INSTR by temp(INSTR,1),temp(INSTR,2);
       getoperator(temp(INSTR,1)):=Push;
       getoperand(1,temp(INSTR,1)):=
             getposition(getparamlist(topaddr(tempstack)),
                   makevarname(valueofaddr(topaddr(tempstack))));
       getoperator(temp(INSTR,2)):=Eval;
       instructions(topaddr(tempstack)):=
                 makegcodetwo(temp(INSTR,1),temp(INSTR,2));
       hascode(topaddr(tempstack)):=True;
   ENDEXTEND
```

Here the code for handling strict supercombinator parameters is made.

The fact that the supercombinator parameter is strict is discovered when an occurrence of the supercombinator variable which corresponds to the strict supercombinator parameters is compiled in the body of the supercombinator definition.

The compiled Eval G-machine instruction is defined to make a dump stack, only when the graph representing the argument is not in weak head normal form. This way the overhead in case of many occurrences of the same supercombinator variable is reduced.

It would be possible to generate a G-machine instruction sequence which would evaluate arguments given to strict supercombinator parameters be-

fore making the body. This G-machine sequence could be prepended the sequence for making the body of the supercombinator definition graph, and would be similar to the sequence used to handle arguments given to the primitives. This way of reducing strict parts of the graph eliminates any overhead if the same variable occur more than once in the body of the definition, since all the Eval instruction is made only one time for every strict parameter in the supercombinator definition.

7.13.5 Compiling the Variable Expression when the Corresponding Supercombinator Parameter is Marked as Non Strict

```
if   status=Compile-the-body &
     not(isemptystack(tempstack)) &
     exprtype(valueofaddr(topaddr(tempstack)))=VARname &
     = (getparaminfo(getparamlist(topaddr(tempstack)),
           makevarname(valueofaddr(topaddr(tempstack))))=Nonstrict &
     = (hascode(topaddr(tempstack)),False
then
   EXTEND INSTR by temp(INSTR);
       getoperator(temp(INSTR)):=Push;
       getoperand(1,temp(INSTR)):=
          getposition(getparamlist(topaddr(tempstack)),
                      makevarname(valueofaddr(topaddr(tempstack))));
       instructions(topaddr(tempstack)):=
                 makegcode(temp(INSTR));
       hascode(topaddr(tempstack)):=True;
   ENDEXTEND
```

Here the code for handling non strict supercombinator parameters is made.

7.14 Making the Primitive as Part of the Supercombinator Language Syntax

We can do it more efficient. In [JL91] the strict context is introduced. That permits some code to be performed as inline computation without first making a graph.

If some some graph has to be made within the strict context, an Eval instruction is appended to the code that makes the graph.

That means that this part of the graph is evaluated before result is used in the computing of the expression. The strict context can be seen as a result of some sort of strictness analysis.

In order to compile the expression using strict context, we have to make the primitives as part of the supercombinator syntax. The expressions given to the primitives are compiled as part of the primitives.

It is not difficult to specify the generation of the more efficient G-machine code in Evolving Algebra. This specification will a little more complicated

than the specification above. However, since the Evolving Algebra specification will use much of the same mechanism as the specification for compiling primitives to G-machine code, we refrain from giving the specification here.

Although the code produced when using strict context would be more efficient, there are also some drawbacks. Since the compilation of primitives and supercombinators are mixed, the G-machine compilator may be more difficult to modify if new primitives are to be introduced, or if some modification has to be made to the specification of what the primitives are supposed to do.

7.15 Strictness Annotation of the Application Node

Here we may want to abstract from a complicated strictness analysis and simple assume that the compilers know some ways to annotate application expressions as strict. As an example we may assume that we know when an argument given to the application node can be treated as a strict.

If an application expression are strict, we append an "Eval" instruction to the G-machine instruction. The evaluator of the G-machine instructions will evaluate the expression as strict, such that the argument given to the application is evaluated first.

The Evolving Algebra Specification for this abstraction is given below.

7.15.1 Promise to Make an Application Node

The Transition

```
if status=Compile-the-expression
   & not(is_empty_stack(temp-c_stack))
   & expr_type(current_value)=APexpr
   & strict_application(current_value)
then
   % Make the list of finished code empty
   code_list:=empty_code_list
   EXTEND TADDR by temp(TADDR,1),temp(TADDR,2),temp(TADDR,3)
          INSTR by temp(INSTR,1),temp(INSTR,2)
      % Makes elements to the compile stack.
      value_of_addr(temp(TADDR,1)):=
          first_app_expr(current_value)
      has_code(temp(TADDR,1)):=False
      get_param_list(temp(TADDR,1)):=
          get_param_list(current_value)
      value_of_addr(temp(TADDR,2)):=
          second_app_expr(current_value)
      has_code(temp(TADDR,2)):=False
      get_param_list(temp(TADDR,2)):=
          incr_pos(1,get_param_list(current_value))
      %   Make the instruction: MKap
      operator(temp(INSTR):=MKap
```

```
    %   Attach code to the current address
    value_of_addr(TADDR,3):=make_g_code(temp(INSTR,1))
    has_code(TADDR,3):=True
    % ---- Make the Eval Instruction
    %   Make the instruction: Eval
    operator(temp(INSTR,2):=Eval
    %   Attach code to the current address
    current_value:=make_g_code(temp(INSTR,2))
    has_code(current_address):=True
    temp_c_stack:=push3_stack(temp(TADDR,3),temp(TADDR,2),
        temp(TADDR,1),temp_c_stack)
ENDEXTEND
```

This specification will compile an instruction sequence for evaluation of an application.

The G-machine code sequence which is made is:

```
Mkap, Eval
```

provided the application in some ways is determined to be strict.

Chapter 8

Using Exended Evolving Algebra in the Specification

In the three previous chapters we have used the Core Evolving Algebra to make specifications for compilation and evaluation of supercombinators. The reader will notice that the Evolving Algebra specification is not easy to read. So we will discuss how it can be possible to improve the specification using Extended Evolving Algebra. We will also in short discuss possible use of co-routines and concurrent specification.

8.1 Recursive Calls

During compilation and evaluation of supercombinator expression we may want to use recursive calls on modules. Besides the fact that it is easier to understand recursive programming, we may be able to specify how to optimize tail recursive calls in a clean way.

8.1.1 Specify the Compilation of of the Body of a Supercombinator as a Recursive Process

As an example we can take the compilation of the body of the supercombinator into an acyclic graph. The specification written in Core Evolving Algebra in section 5.7.3 make use of a stack to describe the recursive process of compilation. This specification is difficult to read. In addition this specification do not say anything of how to optimize the last tail recursive invocation. To extend the specification to cover how to optimize the last tail recursive call in Core Evolving algebra, would make the specification in 5.7.3 harder to understand.

So we want to write the specification directly with recursive calls, since this is the most natural way to express the this part of the compilation algorithm. Hence we will use the Extended Evolving Algebra to make a new specification of the compilation process.

Below, we will show how we can use Extended Evolving Algebra to write the recursive definition. We will only show the transitions describing the recursive process.

The Global and Local Functions

Below we will define functions which is global in the compilation process, and functions which is local to each instance made in the recursive process.

```
GLOBAL
   current_pointer: POINTER
   pointer_value: POINTER --> NODE
   is_empty_pointer: POINTER --> BOOL

   first_son: NODE --> POINTER
   second_son: NODE --> POINTER
   node_type: NODE --> NODETYPE
   node_value: NODE --> NODEVALUE

   current_expression: EXPR
   first_app_expr: EXPR --> EXPR
   second_app_expr: EXPR --> EXPR
   is_leaf_expr: EXPR --> BOOL
   is_app_expr: EXPR --> BOOL
   get_expr_value: EXPR --> NODEVALUE
END GLOBAL
```

All global functions used to build the graph is defined above.

The Make Graph Module

Below we will specify the module which make the graph of supercombinator instances. Instances of this module is made during the recursive process.

```
Module: MAKE-GRAPH
BEGIN PRIVATE FUNCTION
  local_new_node: NODE
  local_expression: EXPR
  local_status: STATUS

% Set initial value for each instance.
  local_status==Initial
END PRIVATE FUNCTION
```

Here we define the constants local to each instance of the module.

The Bottom of the Recursion Process

```
if   local_status:=Initial
   & is_leaf_expr(current_expression)
then
   EXTEND NODE by temp(NODE,1)
   WITH
       node_type(temp(NODE,1)):=Leaf-node
```

```
        pointer_value(current_pointer):=temp(NODE,1);
        node_value(temp(NODE,1)):=get_expr_value(current_expression);
   ENDEXTEND
   Invoke-return(Inherited-module)
fi
```

If the current expression is not an application, we make the leaf node of the graph. This transition represents the bottom of one recursion branch in recursion tree of the process.

Making the Application Node

```
if local_status:=Initial
   & is_app_expr(current_expression)
then
   EXTEND NODE by temp(NODE,1)
          POINTER by temp(POINTER,1),
                     temp(POINTER,2)
          WITH
          first_son(temp(NODE,1)):=temp(POINTER,1);
          second_son(temp(NODE,1)):=temp(POINTER,2);
          node_type(temp(NODE,1)):=App-node;
          pointer_value(current_pointer):=temp(NODE,1);
          local_new_node:=temp(NODE,1);
   ENDEXTEND
   local_expression:=current_expression;
   local_status:=Proceed;
fi
```

If the current expression is an application, a new application node is made.

Performing the First Recursive Call

```
if   local_status:=Proceed
   & node_type(local_new_node)=App-node
   & is_empty_pointer(first_son(local_new_node))
   & is_empty_pointer(second_son(local_new_node))
then
   MAKE INSTANCES of MAKE-GRAPH by instance(MAKE-GRAPH,1),
   WITH
       current_pointer:=first_son(local_new_node);
       current_expression:=first_app_expr(local_expression);
       Invoke(instance(MAKE-GRAPH,1),Itself);
   END INSTANCES
fi
```

Here the graph is extended by performing the first recursive call to the module. This recursive call will construct the first of the two subtrees branching from the application node.

Performing the Second Tail Recursive Call

```
if   local_status:=Proceed
   & node_type(local_new_node)=App-node
   & not(is_empty_pointer(first_son(local_new_node)))
   & is_empty_pointer(second_son(local_new_node))
then
   MAKE INSTANCES of MAKE-GRAPH by instance(MAKE-GRAPH,1),
   WITH
       current_pointer:=second_son(local_new_node);
       current_expression:=second_app_expr(local_expression);
       Invoke(instance(MAKE-GRAPH,1),Itself);
   END INSTANCES
fi
```

The transition above performs the last recursive call to the module. This recursive call will construct the second of the two subtrees branching from the application node.

Performing the Return to the Invoking Instance

```
if   local_status:=Proceed
   & node_type(local_new_node)=App-node
   & not(is_empty_pointer(first_son(local_new_node)))
   & not(is_empty_pointer(second_son(local_new_node)))
then
     Invoke-return(Inherited-module);
fi
End Module
```

Her we specify the return to the the invoking instance. This transition will be performed if we do not optimize the last tail recursive call as explained below.

Optimize the last tail recursive call

Since the recursive call in 8.1.1 is to be performed as the last operation, the call is tail recursive. We can optimize this tail-recursive call replacing the `Invoke` statement above with the following statement:

```
Invoke(instance(MAKE-GRAPH,1),Inherited-module);
```

The predefined constant `Itself` is replaced by the predefined constant `Inherited-module`. Since the recursive process will not return to this instance after the recursive call, this calling instance can be deleted when the recursive call is performed.

An implementation of the Extended Algebra interpret is supposed to delete the calling instance, if the recursive process is not going to return to the calling instance of the module. In this way we optimize the tail recursive call.

Using only Core Evolving Algebra we are not able to express in a natural way how to optimize this last tail recursive call.

8.1.2 Use of Recursive Modules in the Specification of Compilation and Evaluation

In this section we will point out where Extended Algebra could be used to specify recursive processes, and where it is possible to optimize tail recursive calls of modules.

In section 6.8.1 we give the recursive definition of how to make a new instance of a supercombinator to be used in the reduction process. The recursive specification can be written directly in Extended Evolving Algebra in the same way as in 8.1, and the specification of how to optimize the tail recursive process can be done in the same way.

In section 7.1.1 we define a recursive procedure which evaluate strict arguments. We can again use Extended Evolving Algebra to specify this recursive definition in a direct way. Such recursive definition will apply to evaluation of strict arguments, whether they are given to a primitive definition or supercombinator definition. We do not specify the recursive definition in Extended Evolving Algebra.

8.2 Use of Co-routines and Parallel Programming

It can be naturally to make some comments about eventually use of co-routines in the specification made so far. We could consider to make instances of co-routines for all strict argument to be evaluated.

However, since each strict argument can consists of expressions consisting of definitions which again takes strict arguments, specification of a recursive process will be the more appropriate way of making the specification.

8.2.1 Concurrent Specification

If the graph representing supercombinators is made as a tree, it should be possible to specify that each of the child can be evaluated in parallel. However, if we choose to share some of the subgraphs between more than one parent node, using an acyclic graph, only those part in the subgraph which is not shared, can be performed in parallel. So in order to specify parallel execution of modules, we have to extend the Evolving Algebra, such that e. g. locks can be specified when needed.

So, it can be of interest to extend Evolving Algebra to permit specification of parallel or distributed processing. Such extension should use proper mechanism to express parallel processing, such that the specification can be made as simple and accurate as possible.

In the thesis we limit the scope to cover sequential processing, only.

But we could wonder if co-routines could be used to specify some quasi-parallel processing of strict arguments in an acyclic graph. If, only one process was active at any time, the shared part of the graph could be evaluated in the right way from whichever of its parent. At once the shared graph had been evaluated into a normal form, it would not be evaluated any more. That would be the case, either we specify directly a recursive process, or we used the form of co-routines. So, we do not obtain any more using the

approach of making co-routines compared to specify the recursive process directly.

Part III

The Evolving Algebra Interpreter

Introduction

This part consists of the following:

- Chapter 9 has a description of the Evolving Algebra interpreter implemented by the author. This interpreter is compared with the interpreter made by [Hug94]. We also discuss the possibility of implementing an interpreter for the Extended Evolving Algebra (as defined in chapter 3).
- Results from some runs of the interpret. We have in fact been able to measure the use of resources of the runs (See chapter 10).

The Evolving Interpreter written by the author is written in the Scheme language. The interpreter runs specification written in the Core Evolving Algebra. The interpreter is written such that any function defined in Evolving Algebra can be associated with a Scheme procedure. In this way functions at any abstraction level can (in principle) be defined and run on the interpreter.

The interpreter has the possibilities to count the use of resources when a specification is run on the interpreter. Reports which shows the use of resources can be printed after a run.

In appendix B the Evolving Algebra interpreter is described in more details than in chapter 9.

Chapter 9

The Evolving Algebra Interpreter

9.1 Introduction

This section is divided into three main parts:

1. Description of the interpreter implemented in Scheme by the author.
2. Comparision with an Evolving interpreter written in the C language implemented [Hug94].
3. Discussion of implementation of Extended Evolving Algebra (as introduced in chapter 3).

9.2 Evolving Algebra Definitions

9.2.1 The Fixed Part of the Evolving Algebra

The definition of a named algebra and signature can be considered as the fixed part of an evolving algebra definition. The system checks the syntax and make the internal data structure when the definition and signature is parsed.

Algebra Definition List

The statement which define a named algebra is used to give a name of the evolving algebra definition paired with a list which contains a list of sets and lists of function used. At present time this type of statement is not used in the system and may be omitted.

We give the following example:

```
algebra stack : (LOCATION, VALUE,{0,1};top,bottom;pop,push;empty)
```

The Signature

The signature part of the fixed evolving algebra definition associate a signature definition with each function symbols used in the evolving algebra

definition. The system checks the syntax and make the internal data structure. The internal data structure generated will be used by the dynamic part of the system, so this part of the definition may not be omitted.

We give the following example of a signature definition:

```
signature push : ((STACK x STACKEL) --> STACK)
```

9.2.2 The Transition

The dynamic part of evolving algebra consists of a one or more transitions. A transition consists of two main parts, the predicate and the updates. The updates can be of two types the function update and the universe update.

When running the evolving algebra system, the predicates in all transitions are tested. If one of the predicates evaluates to "true" this transition is selected to be performed. If more than one of the predicates evaluate to "true", one of the transitions which has its predicate evaluated to "true" is chosen. This choice is made in a non determinate way.

The Predicate

A predicate is an expression which computes to "true" or "false".

In addition to usual functions relational operators such as *equal*, *lesser than* and *not* may be given.

An example of a predicate is:

```
if ( = (halt,0) &
     (! emptycmds(cmds)) &
     = (first(cmds) , "Store") &
     isnumber(first(next(cmds))) );
```

The Updates

The updates are the part of transition which makes the algebra definitions to change. The system can take two types of updates, function updates and universe updates.

An function updates assigns a new value to a point in the value space.

a universe update adds new elements to one or more universe and use this new elements as values in function updates which is part of the universe update.

All updates within a transition are executed simultaneously.

A third form of change to the algebra, where elements is removed from the universe, is not implemented.

Example of a function update is given below:

```
funcupdate array(current) := currval(now)
```

The transition ends when the

```
endupdates
```

statement is parsed.

The Universe Updates

A universe update consists of two parts. The first part are one or more universe extensions. Each universe extension tells which universe is get new elements and how many elements to be added.

The second part has one or more function updates. The function updates use the new elements when assigning new value to the function.

The universe extension

a universe extension expression specify how many elements to add to one of the universes.

Examples of universe extension expressions:

```
extend
   extenduniverse U
   extenduniverse V # newelems
   extenduniverse W # 5 counter
```

In the example `extend` tells that one or more universe extension expressions follows. One elements is added to the universe `U`. In the third line in the example above the number of elements which is the value of `newelems` is added to the universe `V`. In the fourth line five elements is added to the universe `W` and in addition the constant `counter` is used to generate five instances of the function updates, where the value substituted for the constant `counter` goes from 1 to 5.

The function updates within a universe update

The function updates within a universe update can be specified in two ways, telling if instances for all elements added to the universe(s) are to be generated or not.

If the the word "Every" occur in one of the sub-expression which specify the number of the new element to be used, one instance for for every new element added to the universe is generated. In this case we require the name of an generic constant (we can also call the constant a counter) in the universe extension expression for the universe. For all new elements added to the universe the generic constants gets the number of one of the new elements. This constant is used in one of the expressions computing the number of one of the new elements to be used in the generated function update instance.

If no "Every" occurs in the subexpression which computes the number of the new element to be used, no more than one instance is made. In this case we do not need any generic variable in the universe extension to compute this the update.

Some examples follows below of function update within an universe extension follows below:

```
withupdates
   funcupdate tree(temp(V,2)) := initvalue;
```

```
   funcupdate previous(temp(W,Every(add(counter,1)))) :=
              stack(temp(W,Every(counter)))
endextend
```

The second function update in the example above, gives 5 instances where the constant `counter` vary from 1 to 5. However, the fifth instance of the function update is discarded, because the instance

```
previous(temp(W,6)) := stack(temp(W,5))
```

gives no meaning.

9.2.3 Setting the Initial Values

The transitions in evolving algebra makes the evolving algebra to a state transition system. Hence we need to set the initial state of the evolving algebra.

The evolving algebra interpreter gives the possibility to set initial values of functions and to initialize the universes with a set of elements.

Example of initialize a function value:

```
initial f(3) :== 2;
```

Example of the command to initialize a universe:

```
initialset U :== {True,False}
```

At present it is possible to omit to initialize the universe(s) without affecting the evaluation of the transitions.

9.2.4 The Evolving Algebra Environment

The Evolving Algebra interpreter does not define a number of fixed Scheme procedure to be used for the functions defined in the evolving algebra transitions. Instead, the user of the system has to provide the procedures to be used. The Evolving Algebra Interpreter has commands to load the procedures to be used and assign the procedures to the function names and universe names.

Some standard procedures are written and and can be considered as part of the system. The procedures reside in a standard library of the evolving algebra system.

Load Scheme Procedures into the Evolving Algebra Environment

The procedures provided by the user has to be loaded into the evolving algebra system using a "loadproc" command. One or more files containing the procedures are loaded into the a special Scheme environment used in the Evolving Algebra interpreter.

Binding Function Names to Procedures

To bind scheme functions to procedure names we use the "assignfunc" command. This command binds the function symbol to the procedure names as follows:

- The name of a lookup procedure used to lookup the value of the function expression is given.
- The name of an update procedure must be given. The procedure is used to give a new value to a function in a function update.
- In addition a symbol which may be used in the scheme procedures as user supplied parameter must be given.

Binding Universe Names to Procedures

In order to perform universe extension we need to bind Scheme procedures to universe names. The task of such procedure is to generate the new elements to be added to a universe when performing a universe extension. In addition this procedure may initialize and update a data structure which represents the set of elements in the universe.

To bind a universe name to a Scheme procedure we use "assignunverse" command. The command binds the universe name to a Scheme procedure as follows:

- The name of the extension procedure has to be given.
- A symbol which may be used in the scheme procedure as a user supplied parameter has to be given.

9.2.5 Commands

In this subsection we describe other commands which may be given to the Evolving Algebra interpreter. The user has the following options:

- Perform the transition as specified in the evolving algebra given to the interpreter.
- Print the definition and performing statistic.
- Read the evolving algebra specification from file.
- Execute a Scheme expression in the Evolving Algebra environment.
- Reset the interpreter so a new specification can be loaded.
- Fetch values assigned to a universe name.
- Fetch values assigned to a function name.

9.2.6 System Commands

The following commands is useful for those who whish to make changes to the Evolving Algebra interpreter.

- Load a new version of a file of procedures belonging to the system.
- Display Scheme bindings made in the Evolving Algebra environment.

9.3 How the Interpreter Works

In this section we will say some more about how the Evolving Algebra interpreter works. From the user point of view the interpreter performs two main step when given an Evolving Algebra specification. The first step is to parse the specification. The second step is to execute the specification which has been parsed.

9.3.1 Parsing and Loading the Evolving Algebra Specification

When starting, the interpreter initialize its internal data structure and enters the main loop. The interpreter is now ready to accept input from the user. We call the main loop the main level of the interpreter. If the interpreter enters some other loop we say the interpreter enters a new level.

The Levels of the Interpreter

Here we will describe the levels the interpreter may enter.

The main level At the main level the algebra definition lists, the signatures, the predicate part of the transition, the initial values of the specification and all the commands can be given. To exit the main loop is the same as exit the interpreter.

The update level The update level is entered when the predicate part of the transition is parsed with success. At the update level the function update statement and the universe update statement can be given. If a universe update statement is given the interpreter first enter the universe extension level and then the special function update level within a universe extension. This level is finished when an "endupdates" statement is given.

The universe extension level The universe extension level is invoked giving the "extenduniverse" statement in the update level. Only universe extension statements can be given in this loop. This level is finished when entering the special function update level.

The special function update level This level is entered when the "withupdates" statement is entered at the universe extension level. Only function update with an extended syntax can be given in this loop. This level is finished when an "endextend" statement is given. The interpreter then returns to the update level.

The loading of the specification

The Evolving Algebra specification is stored in an internal data structure for executing. When the interpreter is in the main level this data structure can be reset at any time.

Reading the Evolving Algebra Specification from File

The definition of the Evolving Algebra specification can be read from file. The interpreter will parse the specification and return the result of the parsing to the user. The command for reading from file can only be invoked at the main level.

The Output

The result of parsing the statements and expressions in the evolving algebra specification is returned to the user along with a prompt. If the Evolving Algebra specification is read from a file it is possible to write the result of parsing to a file.

Errors

If some errors are found, an error message is given.

9.3.2 Executing the Evolving Algebra Specification

Executing the Transitions

When the user gives the command run the specification, a main loop which test the predicate for all transitions is started. If one or more predicates evaluates to "true" one of the transitions which has a "true" predicate value is chosen. The loops continue until the predicates for all transitions evaluate to "false".

So we can write the following skeleton of a procedure:

```
loop
   1. Evaluate every predicate in the list all defined transitions
      in the specification.
   2. If none of the predicates are evaluated to ``true''
      then quit.
   3. Choose one of the transitions which has its predicate
      evaluated to true and perform all updates within
      this transition.
endloop
```

Executing the Updates

All updates within a transition are to be executed simultaneously. This implementation compute values for all updates in the first pass, and then perform all the assignment for functions updates in a second pass.

When a universe update is given a numbers of new elements are added to one of the universe. Then some function updates using the new elements are given as part of the universe update.

Making Instances of the Function Updates within the Universe Update

A function update expression within a universe update can refer to all the new elements added to the universes. So instances of the these function updates are to be made for all new elements added to all universe.

The following algorithm could be used to generate the instances of the function updates in the universe update:

```
Let U1 ... Um be all universe to be extended in this updates.
for i=1 ... #U1
do
   ....
       for k=1 ... #Um
       do
         Make instance (i, ...,l) of all functions updates.
       od
   ....
od
```

Here instances are made for every combination of new elements taken from distinct universes. However, since this algorithm has exponential complexity we need to optimize the algorithm above.

This optimization can be done in the following ways:

1. For each function update, check if new elements are referred in a generic way in the expression or the functions update are to be computed for just one new element from each universe. Do not make equal instances if only one of the new elements added to each universe are to be used.

2. If instances are to be made for an function update, make loops only for the universes which are referred in a generic way in expressions in the function update.

After the optimization the algorithm is still of exponential complexity, but we do not make many equal instances of the function updates.

Output the Result of Executing the Specification

The output which appear when executing the specification contains records of all dynamic transitions of the Evolving Algebra. All updates and extensions to the universe are recorded along with all predicates evaluated to "true" for every execution of a transition.

9.4 The Statistics

9.4.1 The Statistical Data

Here we describe the statistical data produced by the system.

The Definitions Statistic

The definition statistic consists of totals for the Evolving Algebra specification, the sums for each transition and numbers of universe extensions and function updates for each universe extensions defined.

For the Evolving Algebra specification we give the following totals:

- The number of defined function names.
- The number of defined universe names.
- The number of defined transitions.
- The number o function updates (which is not part of a universe update) defined.
- The number of universe updates defined.
- The number of universe extensions defined.
- The number of functions updates defined which is part of a universe update.

For each transition defined we print the following numbers:

- The number of defined function updates.
- The number of defined universe updates which is not part of a universe update.
- The number of universe extensions defined.
- The number of functions update which is part of a universe update.

For each universe update defined we print the following numbers:

- The number of defined universe extensions.
- The number of defined functions updates as part of the universe update.

The Runtime Statistics

The runtime statistics consists of the totals for the Evolving Algebra specification performed, the sums for each transition, the sums for each universe updates, the number of performed and discarded instances of each function update within a universe update and the number of elements added to the universe for each universe extension.

Our main interest is to show the use of the resources when simulating the run of the dynamic transitions of the evolving algebra specification.

Two types of resources are to be measured, the use of room and the use of time. When performing Evolving Algebra transitions we count the numbers of function updates and the number of elements added to the defined universes.

The number of elements added to a universe corresponds to the use of room (i.e. memory in a computer). The number of function updates performed corresponds to the use of time (i.e. execution time in a computer).

If we specify an indeterminate system we are also able to measure the degree of indeterminism in the specification. If the number of predicates evaluated to "true" are equal to the number of performed transitions, we have specified a determinate system. If the result of dividing the number of predicates by the numbers of performed transitions is greater than 1.0 we have introduced some degree of indeterminism which is measured by this ratio.

The Numbers Printed in the Runtime Statistics

The following totals will be printed:

- The total number of time a transition is performed.
- The total number of predicates which is evaluated to "true".
- The total number of predicate which is evaluated to "true" divided by the total number of transitions performed. A number greater than 1 means that some degree of indetermination is a property of the Evolving Algebra specification, since the choice of more than one transition which has "true" as value of its predicate is an indeterminate choice.
- The total numbers of function updates (which is not part of a universe update) performed.
- The total numbers of universe updates performed.
- The total numbers of universe extensions performed.
- The total numbers of elements added to a universe.
- The total numbers of function updates (which is part of a universe update) performed.
- The total numbers of function updates (which is part of a universe update) discarded.
- The total numbers of function updates performed.

The following sums will be printed for each defined transition:

- The number of times this transition is performed.

- The number of function updates (which is not part of a universe update) performed.
- The number of universe extensions performed.
- The number of universe updates performed.
- The number of elements added to a universe.
- The number of times a function updates (which is part of a universe update) is performed.
- The number of times a function updates (which is part of a universe update) is discarded.
- The number of elements added a universe divided by number of time this transition is performed.
- The number of times a universe update is performed divided by the number of times this transition is performed.
- The sums of all function updates performed.

The following sums will be printed for each defined universe update:

- The number of elements added to all universes extended in this universe update.
- The number of times the functions updates defined as part this universe update is performed.
- The number of times the function updates defined as part of this universe update is discarded.

The following sum will be printed for each defined universe extension:

- The number of new element added to the universe defined in this extension.

The following sums will be printed for each function updates which is part of a universe update:

- The number of instances of this function update made and performed.
- The number of instances of this function update made and discarded.

9.5 Comparision with Huggins EA-interpreter

In this section I will call the Evolving Algebra interpreter implemented by the author the Scheme EA-Interpreter and the Evolving Interpreter implemented by James K. Huggins the C EA-Interpreter [Hug94].

9.5.1 Implementation

The Evolving Algebra Interpreter made by James K. Huggins is implemented in the programming language C, therefore it is called the C EA-interpreter here.

The EA-interpreter described in this report is made in the Scheme Lisp (MIT Scheme 7.1), hence it is called the Scheme EA-interpreter.

9.5.2 Function Definition

The C EA-interpreter

Each function definition consists of:

- A function name.
- The arity of the function.
- Flags telling how the function are to be used.
- Default expression.
- Function tuples. The value at certain locations is pre-initialized.

The arity of the function is checked against the number of arguments given to the function in expressions.

The Scheme EA-interpreter

Each function definition consists of:

- A function name
- Full signature of the function.
- A Scheme procedure which defines the "semantic" of the computation of the function value.
- A Scheme procedure which defines how the update of the functions eventually should be performed.
- A flag which state some properties of the function. The meaning of the flag is determined by the procedures associated with the function.

The existence of the signature is checked against other part of the Evolving Algebra specification. The system gives warning if more than one signature of the same function is loaded. The syntax of the signature definition is checked.

9.5.3 The Initial Rules

Both the C Evolving Algebra Interpreter and the Scheme Evolving Algebra interpreter has the rules for giving initial values to be set before the specification is to be run.

9.5.4 Transition Rules

9.5.5 Expressions

The C EA-interpreter

An expression may conist of:

- An integer
- A real number
- A quoted string
- A function name for a zero arity function
- A function name together with the arguments given to the function
- A LOAD reference which refer to values which can not be given a direct reference.

The Scheme EA-interpreter

An expression may consist of:

- An integer
- A quoted string and other ways of providing values
- A function name for a zero arity function
- A function name together with the arguments given to the function

Most value including the empty list can be given to the Scheme Evolving Algebra interpreter, so no LOAD reference seems to be necessary.

Function Assignments

The function assignments is similar in the C Evolving Algebra interpreter and the Scheme Evolving interpreter.

9.5.6 Guarded Rules

The C EA-interpreter

In the C Evolving Algebra interpreter a guarded rule may have the form of simple *if* rule, or *elseif* may be used for to do one selection of many possible. An *else* rule may optionally be given to state the default selection.

The transition rules may be nested in the C Evolving Algebra interpreter.

The Scheme EA-interpreter

In the Scheme Evolving Algebra interpreter the guarded rule consists only of the simple *if* rule.

The transition rules may not be nested.

9.5.7 Universe Extensions

The C EA-interpreter

The universe extension is implemented permitting one or more universes to be extended. The new elements added to each universe can be temporarily referred with rules inside the universe extension.

The Scheme EA-interpreter

Also in the Scheme EA-interpreter one or more universes can be extended by new anonymous elements. The new elements inside each universe be referred with function assignments inside the universe extension.

In addition it is possible to refer to the new elements added to the universes in a generic way, causing all function assignments to be generated for all possible combinations of new elements.

9.5.8 Universe Contractions

Universe contraction is implemented in the C Evolving Algebra interpreter.

It is not implemented in the Scheme Evolving Algebra interpreter.

9.5.9 Defining Functions and Universes

Standard Universes

The C Evolving Algebra has defined 8 standard universes. Other universes may be defined implicit by using the extension rule.

Dynamically defined universes

In the Scheme Evolving Algebra all universes are implicit defined. A universe is implicit defined by using the universe name in in the expression which gives the signature for a function.

If an extension is to be used for a universe, a Scheme procedure which makes the new elements to be added to the universe has to be defined and associated with the universe name.

Standard Functions

In the C Evolving Algebra interpreter a set of standard functions and universes is implemented. The set consists of 30 functions which act as predefined primitives in the C Evolving Algebra interpreter.

Other functions may be defined. The "semantics" of the functions i defined by:

- The functions default expression. For example the a one-ary Successor functions may be defined in terms of the standard two-ary function Add.

- By values given in the tuple part of the function definition, values given in the initial rules of the Evolving Algebra specification and values given during the run of Evolving Algebra rules.

If more than the 30 standard functions is needed, the C Evolving Algebra interpreter has to be extended.

Dynamically defined Functions

The Scheme Evolving Algebra interpreter has no minimal set of standard functions defined. All functions defined has to be associated with a Scheme procedure, and the Scheme procedures used to define the "semantics" of the functions is also loaded into the system. Both the association of functions to Scheme procedures and the loading of the Scheme procedures from a file, can be done dynamically at any time before the Evolving Algebra specification is executed.

The only exception to this scheme of dynamically associated functions with Scheme procedures, is that some predicate functions, such as equal, lesser than, greater than, not equal and not are predefined (for convenience reason).

9.5.10 Running the Interpreter

The C EA-interpreter

The Evolving Algebra specification has to be loaded at the time the interpreter is started. Then runtime commands can be given. Some of the commands which may be given is: run the given specification, run until an expression has value false, run until an expression has value true, reset the interpreter, reload the interpreter, run one step or run n steps, eval an expression, set trace, set no trace or exit.

The Scheme EA-interpreter

The Evolving Algebra specification is a full interpreter. The Evolving Algebra specification or part of the Evolving Algebra specification may be loaded from file any time, or the user may give the specification or part of the specification directly into the interpreter.

The Evolving Algebra specification may be run in one run or a given number of n steps may be executed each time, until the execution of the Evolving Algebra specification (eventually) halts.

It is possible to evaluate the state of the execution by running Scheme expressions in the environment set up by the Evolving Algebra interpreter.

9.5.11 Statistics

The Scheme EA-interpreter

The Scheme EA-interpreter saves statistics about each Evolving Algebra specification and the run of the Evolving Algebra specification. In this way

it is possible to count the use of resources, such as use of "time" and "space" when running the Evolving Algebra specification.

The number of elements added to the universes can be seen as an abstract count of the use of space in a computer, and the number of functions assignment can be seen as an abstract count of the use of time in a computer.

The C EA-interpreter

The C Evolving Algebra interpreter does not count any statistics.

9.6 Issue Regarding Implementation of Extended Evolving Algebra

In this section we will briefly discuss some issue with regards to the future implementation of the Extended Evolving Algebra. We will consider data which needs to be associated with modules and with instances of modules, and with the global data-structure. We will tell what Invoke and Invoke-return commands should do, and briefly discuss cases when we safely can discard instances of a module. The possibility of discarding instances of modules is essential if we want to optimize tail-recursive call of modules.

9.6.1 Instances of a Module

The purpose of dividing the Evolving Algebra specification into modules is to provide a possibility of making more than one logical execution sequence of transitions. So within each instance of module we have to store the state of locally defined functions each time another module or instance of a module is invoked. The exception from this rule, is if we will never return to the instance, which is the case when we specify to optimize tail recursive calls.

Some of the EA-functions is not intended to be updated. Such functions may point to algorithms to be performed, where the algorithms remains fixed during the execution of the EA-specification. To optimize the use of space, such function should be marked as fixed.

An implementation should record the following data about each instance of a module:

- The identifier of the instance.
- The name of the module associated with the instance.
- The contents of the Inherited value. The contents is the instance identifier given as the second parameter, the last time the instance was invoked with the Invoke statement.
- The contents of the Itself value. The contents is always the identifier of the instance itself.
- The state of all local defined functions for the instance (except those functions which never change).
- A flag telling if this instance can be invoked again or not.

9.6.2 The Module

Some data are shared for all instances of a module. An implementation should record the following data about each module:

- The name of the module.
- The definition of the module.
- The state of all non-local functions defined to be visible only within the module.
- Pointers to all functions visible outside the module.

9.6.3 Global Data

An implementation should record the following global data:

- The module which begins the execution of the Evolving Algebra specification.
- All functions defined outside the module.
- The state of each of the functions defined outside the module.
- For each defined function, information about the modules which are allowed to access the function.
- The instance which is active.
- A flag which tells if the Evolving Algebra has finished the execution or not.

9.6.4 Invoke and Invoke Return

When the Invoke statement is performed the following happens:

- The instance of the module given in the first parameter is invoked.
- The Inherited constant of the instance invoked is updated with the second argument given to the Invoke statement.
- The execution of the instance continues according to the state of the instance.

When the Invoke Return statement is performed the following happens:

- The instance of the module given in the parameter is invoked.
- The Inherited constant keeps its value.
- The execution of the instance continue according to the state of the instance.

9.6.5 Telling If the Instance Can Be Invoked Again

If the identifier is stored in a non-local function, or is given as the second argument to the Invoke statement, the instance should be marked to be accessible from another instance.

If the identifier of an instance is never stored in a non-local functions, and its identifier is not given to the Invoke function, the instance should be marked an non-accessible when another instance is invoked with the Invoke statement. The same is the case if the identifier is never stored in a non-local function and and another instance is invoked with Invoke-return.

A non accessible instance can be garbage collected when it becomes not active.

Chapter 10

Running the Interpreter

10.1 Supercombinators Used

10.1.1 Church Numeral for one (pure version)

The compilation and execution of the Evolving Algebra Definition of simple template instantiations was run on the supercombinator expressions shown below. The expression is the Church Numeral for the number one defined under the assumption that only S, K and I exists as real combinators (See p 48 [HS86] for the definition of the Church Numerals). All other combinators are combinations of S, K and I.

```
One = (S B) (K I) f x
B =  S (K S) K
S x y = x z (y z)
K x y = x
I z = z
```

10.1.2 Church Numeral for one (a simpler version)

This is the Church numeral for one assuming that B, S, K and I exists as real combinators. This definition is simpler to compute than the previous one.

```
One = (S B) (K I) f x
B x y z = x (y z)
S x y = x z (y z)
K x y = x
I z = z
```

10.2 Weak Head Normal Form

The weak head normal form is defined in [Jon87] as follows:

Definition 13 *A lambda expression is in* weak head normal form (WHNF) *if and only if it is of the form*

$$FE_1E_2 \ldots E_n$$

where $n \geq 0$*;*
and *either* F *is a variable or data object*
or F *is a lambda abstraction or built-in function*
$(FE_1E_2 \ldots E_n)$ *is not a redex for* $m \leq n$*.*

An expression has no top level redex *if and only if it is in weak head normal form.*

It is important to realize that an expression in weak head normal form may contain inner redexes, such that the expression in weak head normal form is *not* necessarily in normal form.

10.2.1 Reducing the Church Numerals to Weak Head Normal Form

Both the template instantiations and the G-machine definitions of the supercombinators reduce the expressions into *weak head normal form* provided that we do not introduce primitives or permit evaluation of strict arguments.

The Church Numeral constructed as defined above will reach weak normal form just after the first number is computed. The reason why, is that the next redex to be chosen is not at the top level. The reduction process stops since we have reached the weak head normal form. But the redex for the Church Numeral is *not* in normal form.

For example the Church Numeral for two reduces as follows f and x is here regarded as data objects, since we do not permit free variables in supercombinator expressions:

```
Two = (S B (S B (K I))) f x
```

which in the first step is reduced to

```
B f ((S B) (K I) f) x
```

which is thereafter reduced to:

```
f ((((S B) (K I)) f) x)
```

The last expression is in weak head normal form but not in normal form since it contains an inner redex.

10.2.2 Extending the algorithm to take strict arguments and primitives

If we extend the algorithm for Template Instantiation and G-machine to permit strict arguments we may force the inner redex to be evaluated. In this this way can compute the Church Numeral into normal form, if desired, since we can decide the time the arguments are to be evaluated.

10.3 Template instantiations without Defined Primitives

The results below is for Evolving Algebra Code defining simple template instantiations.

The Evolving Algebra definition consists of:

- 26 defined transitions.
- 93 defined assignments (function updates and function updates within universe updates).
- 16 defined universe extensions.
- 60 defined function names.
- 14 defined universe names.

The Evolving Algebra Definition was run with two versions of Church Numerals.

10.3.1 Running the Church Numeral for one (pure version)

The result of the run of the complicated version of Church numeral for one was as follows:

- The number of times a transitions is executed is 224.
- The number of assignments performed is 709.
- The number of new elements added to all the universe is 135

10.3.2 Running the Church Numeral for one (simple version)

The result of the run was as follows:

- The number of times a transitions is executed is 146.
- The number of assignments performed is 485.
- The number of new elements added to all the universe is 102.

10.4 Simple G-machine without Defined Primitives

The results below is for Evolving Algebra Code defining G-machine compilation and execution.

The Evolving Algebra definition consists of:

- 19 defined transitions.
- 85 defined assignments (function updates and function updates within universe updates).

- 16 defined universe extensions.
- 64 defined function names.
- 14 defined universe names.

10.4.1 Church Numeral for one (pure version)

The result of the run of the complicated version of Church numeral for one was as follows:

- The number of times a transitions is executed is 154.
- The number of assignments performed is 613.
- The number of new elements added to all the universe is 137.

10.4.2 Church Numeral for one (a simpler version)

The result of the run was as follows:

- The number of times a transitions is executed is 113.
- The number of assignments performed is 484.
- The number of new elements added to all the universe is 111.

10.5 Discussion of the Result

10.5.1 Template Instantiations and G-machines for Supercombinators without Primitives

When comparing the result of running the Evolving Algebra for template instantiation with the result of running the Evolving Algebra for G-machine, we see that the G-machine perform less number of assignments and transitions compared with the Template instantiation. On the other hand the G-machine makes some more new elements to be added to the the universe.

For the simple version of Church Numeral for one, 9 more elements was added to the universe using the G-machine, and for the more complicated pure version of the Church Numeral only 2 more elements was added. The reduction of number of assignments for the G-machine is 109, and the reduction of number of performed transitions is 70 for the complicated pure version of the Church Numeral for one (7 and 33 for the simple version of Church Numeral for one).

Part IV

Discussion and Conclusion

Introduction

This part consists of the following:

- Discussion of Evolving Algebra as a programming language.
- Discussion of other semantic languages and Evolving Algebra.
- Some concluding remarks.

Evolving Algebra is a specification language which is more flexible with regards to abstractions, and can be used to specify the use of computation resources in an abstract way. So we are able to make the specification as detailed and complex as we want. Then, it makes sense also to regard Evolving Algebra as some sort of prototyping programming language. So we discuss the property of Core Evolving Algebra as a programming language in chapter 11. The lack of control structures and the need to making module as part of the abstractions is discussed.

In chapter 12 Evolving Algebra, Operational Semantic and Denotational Semantic is discussed. To some extent a comparison with Evolving Algebra and Denotational Semantic is made.

In chapter 13 we have made some concluding remarks about the work done, main features of Evolving Algebra, and why extensions to the Evolving Algebra is needed. A short summary of other (besides the authors) implementations of Evolving Algebra interpreters is written. The chapter (and the reports) closes with a short notes of possible further research in the field of Evolving Algebra.

Chapter 11

Evolving Algebra as a Programming Language

11.1 Introduction

When implementing the specification of the interpretation and compilation of the functional language on the Evolving Algebra interpreter, the Evolving Algebra specification language was used like a programming language. It can be well worth to evaluate the Evolving Algebra specification language as a programming language. The reason why we would try such an evaluations can be given as follows:

- When writing detailed specification, the task of writing specification becomes very similar to the task of writing a computer program.
- The possibility that exists to specify use of resources such as time and space, makes a specification in Evolving Algebra more similar to a computer program, than a specification written in a some more traditional specification language (e. g. denotational semantics).

Besides of evaluating the Evolving Algebra as a programming language, some new construction defining control structures which can be used to extend the Evolving Algebra language are introduced in section 11.5.

It does not mean that the Evolving Algebra specification language should be extended with all those new constructions. It is merely intended to shed light to the problem of writing, understanding and implementing specification written in Evolving Algebra, especially at a detailed level of abstraction [1]. The reason why we do not want to define too many constructs, is the potential problem and complication which lies in the needs of defining the semantic of the specification language itself.

The author have skill and training as a computer programmer. Many of the construction suggested in section 11.5 is similar to well known construction in programming language. So, the suggested construction is assumed to be easy to use and understand for persons with programming skills. But

[1] And may be other specification languages with the same or more flexibility, which may be discovered in the future.

it can be questioned if even well known program language construction in general is easy to understand from a semantic point of view and to use when writing a program or specification.

11.2 Evolving Algebra as a Specification and Programming Languages

11.2.1 Specifying Data Structures

Some of the strength of Evolving Algebra is the possibility to represent whichever data structure you want. In the Evolving Algebra specification which specify the graph reduction we are able to represent trees, acyclic graphs, stacks and sequence of instructions.

11.2.2 Control Structures

A language suitable for defining computing tasks or to be used as a specification language will need some control-structures. In addition to telling what to do, we will need (to some degree) to tell *how* to do what we want. If a specification is going to be implemented, it is necessary to tell how to perform what we want to achieve. That is true even if we only want to make a prototype of an real implementation.

So which control structures is provided in the Core Evolving Algebra?

Test on a condition A transition with updates are performed if the predicate belonging to the transition evaluates to "true" and this transition is chosen among all transitions which predicates evaluates to "true".

One main loop After execution of a transition all transitions are tested to see if some of the predicates becomes "true". The loop stops if none of the transitions evaluates to "true".

Simultaneous assignment All updates within a transition is performed simultaneously.

Dynamic adding of new elements to a universe One ore more of the universe may be extended by adding one or more new elements to the universe.

As we can see it is not many control structures in the Core Evolving Algebra Language. So we want to discuss what we loose and what we gain by the lack of control structures.

When we try to specify a languages (or an algorithm) it is important express the meaning of certain constructs (e.g optimized tail recursion). If the specification language itself contains few constructs we are forced to define the meaning of more powerful language constructs using only few and hopefully well understood constructs in the specifications language.

On the other hand lack of control structures may cause some problems. Some of the problems are listed below:

- Difficult to express what we want to express. It is not a natural way of expressing many constructions. Say we want to express an iteration. If we do not introduce new constructions we have to test if a predicate for a transition satisfies some invariant, and execute the updates in the transition as many times the invariant is satisfied. So we will use the same construction in Evolving Algebra to express an iteration and a "case" or "if" statement in Evolving Algebra.

- Problems to divide the specification into modules. Since we are forced to use one simple main loop, all transitions in a specification can be dependent on each others, such that changes in one transition may affect any of the other transitions in the specifications.

- Not easy to change the specification. A change in a specification may cause the whole specification to be rewritten. That is so since we can not divide the specification into modules.

- The specification may be difficult to read. That is especially true if many details have to be specified.

11.3 Some Examples

To illustrate some of the points above, we may use some examples. We will first express the examples using some well known language constructs (which should be self explanatory) and using some simple block structure. Then we specify the same examples using Core Evolving Algebra.

11.3.1 Loops and Sequential Execution

The examples below will illustrate how we specify sequential processing and loops in Evolving Algebra.

Making Two Sequences of Numbers

Say we want to express an algorithm which is supposed to do the following:

1. Generate an ascending sequence of numbers from 1 to n.

2. Generate an descending sequence of numbers from n to 1.

The ascending sequence has to be generated before the descending sequence of numbers.

We will use two loops to generate the two sequences of numbers.

Using a block structured programming languages, we would express the algorithm like the follows:

```
BEGIN_SEQUENCE
  FOR a:=1 TO n
  DO
    BEGIN_SEQUENCE
      a:=a+1;
```

```
      asc_seq:=concat(seq,a);
    END_SEQUENCE
  OD;

  FOR b:=n TO 1
  OD
    BEGIN_SEQUENCE
       b:=b-1;
       desc_seq:=concat(seq,b);
    END_SEQUENCE
  OD;
END_SEQUENCE
```

The two main step in the algorithm is the generation of the ascending and the descending sequence.

In Evolving Algebra we will express the two loops using the following transitions (with the initial values given below):

```
% The initial values
a==0
b==n (where n is a number)
make_descending_sequence==false
```

The Evolving Algebra transitions follows below:

```
if a < n &
   make_ascending_sequence
then
   a:=a+1;
   asc_seq:=concat(asc_seq,a)
fi

if a=n &
   not(make_descending_sequence)
then
   make_descending_sequence:=true;
fi

if b > 1 &
   make_descending_sequence
then
   b:=b-1;
   desc_seq:=concat(desc_seq,a);
fi
```

We are able to get the job done in Evolving Algebra. However, the price we have to pay is that we obscure the fact that the two main steps in the algorithms is the two loops making the ascending and the descending sequence of numbers.

We might be tempted to believe that the first and last transitions above resembles the two loops. However, they do not, since both transitions perform just one round in the loop each time they are performed.

We will give another illustration of this in the next examples.

Making one sequence

The next algorithm makes a sequence in one loop.

Here we want to express the generation of the following sequence:

$$1, n, 2, n-1, \ldots, n-1, 2, n, 1$$

In a block structured programming language we might use the following statements to express the generation of the sequence above:

```
FOR i=1 TO n
DO
  BEGIN_SEQUENCE
     a:=i;
     seq:=concat(seq,a);
     b:=n-i;
     seq:=concat(seq,b);
  END_SEQUENCE
OD
```

The main step is the loop which generates the sequence.

The tasks to be performed each time the loop is performed is to generate the ascending number followed by the descending number.

In Evolving Algebra we would express the generation of the sequence above in the following way (the initial values is given first followed by the Evolving Algebra transitions):

```
a==0
b==n % (where n is a number)
make_ascending_number==true
make_descending_number==false
```

The Evolving Algebra Transitions follows below:

```
if a < n &
   make_ascending_number
then
   a:=a+1;
   seq:=concat(seq,a);
   make_ascending_number:=false
   make_descending_number:=true
fi

if b > 1 &
   make_descending_number
then
```

```
   b:=b-1;
   seq:=concat(seq,b);
   make_ascending_number:=true;
   make_descending_number:=false;
fi
```

In the example above the first transition add the first ascending number within the loop to the sequence of numbers, and the last transition adds the second descending number within the loop to the sequence of numbers. The loop is finished when none of the transitions above gets "true" in the *if* part of the transition.

Here the Evolving Algebra transitions describe the minor steps within the loop. The first transition adds the next ascending element to the sequence, the second transition adds the next descending number to the sequence. Again we see that the main step, which is the loop generating the sequence is obscured in the Evolving Algebra specification.

Give a more Abstract Specification

We may try to abstract the algorithm, so we do not specify the iterations. So in a conventional language we express the algorithm which generates the ascending and descending sequences of numbers at the more abstract level:

```
BEGIN_SEQUENCE
   make_ascending_seq_of_numbers;
   make_descending_seq_of_numbers;
END_SEQUENCE
```

This algorithm can be translated to Evolving Algebra in the following way (the initial values is given first, followed by the transitions):

```
descending_sequence==false
ascending_sequence==true
```

The transitions follows below:

```
if ascending_sequence
then
   asc_seq:=make_ascending_seq_of_numbers;
   ascending_sequence:=false;
   descending_sequence:=true;
fi

if descending_sequence
then
   desc_seq:=make_descending_seq_of_numbers;
   descending_sequence:=false;
fi
```

Note that algorithm still has to meet the requirement that the ascending sequence has to be made before the descending sequence.

Here we are able to express the two main steps to be performed. The cost is that we can not specify anything about the loops to be performed in order to make the two sequences. If we want to make a more detailed specification which describe how the two sequences of numbers are generated, we need to rewrite the whole specification.

Some Comments

The two examples above show us that it is possible to specify algorithms which is using loop constructs and sequential execution using Evolving Algebra. However, if we want to express such algorithm *step by step* (where we regard a loop as a main step), we are not able to to so in Evolving Algebra (unless we abstract out the specification of the loops).

We are simply missing a suitable general construct to define a loop, and in addition we do not have any construct expressing that some statements are to be performed in a sequence.

11.3.2 Making the two sequences in parallel

We may relax on the requirement on which order the the two sequents of numbers are to be made.

An Abstract Specification

First we use a block structured language to express the algorithm:

```
DO_IN_PARALLEL
   make_ascending_seq_of_number;
   make_descending_seq_of_number;
END_DO_IN_PARALLEL
```

No we can give the following abstract Evolving Algebra version of the algorithm:

```
if make_sequence
then
   asc_seq:=make_ascending_seq_of_numbers;
   desc_seq:=make_descending_seq_of_numbers;
fi
```

At this level we can clearly express the two main steps to be performed in parallel.

A Less Abstract Specification

Each transition adds a descending or an ascending element to the descending or ascending sequence of numbers. The two transitions may be interleaved in arbitrary order, the only requirement is that each number within each sequence is performed in right order,

The less abstract version of the algorithm will now be as follows:

```
DO_IN_PARALLEL
  FOR a:=1 TO n
  DO
    BEGIN_SEQUENCE
      a:=a+1;
      asc_seq:=concat(seq,a);
    END_SEQUENCE
  OD;

  FOR b:=n TO 1
  OD
     BEGIN_SEQUENCE
       b:=b-1;
       desc_seq:=concat(seq,b);
     END_SEQUENCE
  OD;
END_DO_IN_PARALLEL
```

In Evolving Algebra we will use the following transitions:

```
% The initial values
a==0
b==n (where n is a number)

if a < n
then
   a:=a+1;
   asc_seq:=concat(asc_seq,a)
fi

if b > 1
then
   b:=b-1;
   desc_seq:=concat(desc_seq,a);
fi
```

Again we see that the transitions express the minor steps to be performed within each loop. The main tasks of making two sequences in parallel are not expressed in any explicit way.

In the block structured language, it is easy to follow both the minor and major steps in the algorithm.

It seems that the core of the problems lies in the lack of suitable control structures in Evolving Algebra. So we will investigate this point further giving more examples.

11.3.3 Dividing into Cases

Test if the an integer is positive, negative or zero

This very simple algorithm will tell if an integer is positive, negative or zero.

Using a block structured languages we may express the simple test as follows:

```
CASE number OF
     negative(number):
        result:=Negative
     zero(number):
        result:=Zero
     positive(number):
        result:=Positive
END_CASE
```

This simple test will translate to the following Evolving Algebra transitions:

```
if negative(number)
then
   result:=Negative
fi

if zero(number)
then
   result:=Zero
fi

if positive(number)
then
   result:=Zero
fi
```

The three Evolving Algebra transitions describe the three possible outcome of the test on the number. If we regard the test as one step, we are able in this particular case to describe the step using the three Evolving Algebra transition shown above.

However, we have in the examples so far, used the guard (the *if* part) of the transition to specify the following different language constructs:

- Sequential execution
- A loop
- Selection

A more Complex Test

No we extend the example to test if the number is an integer, a real number or a complex number. If it is an integer number we test if it is positive negative or a zero number. The major step is the test if the number is integer, a real or a complex number. The minor step is to test if the integer is positive, negative or zero.

Using a Case construction in a block structured language, we define the test as follows:

```
CASE number OF
     integer(number):
       CASE number OF
          negative(number):
             result:=Integer
          zero(number):
             result:=Zero
          positive(number):
             result:=Positive
       END_CASE
     real(number):
       result:=Real
     complex(number):
       result:=Complex
END_CASE
```

The translation to Evolving Algebra will be as follows:

```
if integer(number) &
   negative(number)
then
   result:=Negative
fi

if integer(number) &
    zero(number)
then
   result:=Zero
fi

if integer(number) &
   positive(number)
then
   result:=Zero
fi

if real(number)
then
   result:=Real
fi

if complex(number)
then
   result:=Complex
fi
```

Again we see that we are able to get the work done in Core Evolving Algebra. But the test as stated in the block structured language, describe the test in two steps, the first step is to find out if the number is an integer,

a real number or complex number. If the number is an integer the second step is to test if the number is positive, negative or zero. This is obscured in the Evolving Transition above, by combining the two tests into one test for integers.

We may ensure that the two tests are performed strictly in sequence by adding a new Evolving Algebra transition:

```
if integer(number)
   then
   result:=Integer
fi
```

In addition we change the tests for positive, negative or zero integer in the following way (only the Evolving Algebra transition for the positive integer is shown):

```
if result = Integer &
   positive(number)
then
   result:=Positive
fi
```

Even this specification is not very clear. We are forced to use six transition to describe the tests. For the reader of the specification it is not very clear that we perform the tests into two sequential steps. In addition which step is the major step and which step is the minor step is not obvious.

11.3.4 Define Common Operation on some Signatures

Say, we want to manipulate a sequence of elements. We may want to take the first element or the nth' element from a sequence. In addition we want to take the sequence with the first element removed and test if the sequence is empty.

In the example we have a sequence of letters and a sequence of numbers. The signature is given below:

```
numbers: NUMBER*
letters: LETTER*
```

The functions for the sequence of numbers will be as follows:

```
first-number:          NUMBER* --> NUMBER
tail-number:           NUMBER* --> NUMBER*
nth-number:            POSITION x NUMBER* --> NUMBER
is_empty_numbers:      NUMBER* --> BOOL
```

The functions for the sequence of letters is shown below:

```
first-letter:          LETTER* --> LETTER
tail-letter:           LETTER* --> LETTER*
nth-letter:            POSITION x LETTER* --> LETTER
is_empty_letters:      LETTER* --> BOOL
```

As we can see, we have to invent different names for common operators, depending on the signature the operators is applied upon.

11.3.5 Modules and Subroutine Calls

The extension suggested in chapter 3 covers the problem of dividing a specification into a modules and subroutines. So, from the programmers point of view, it is not more to add.

11.4 The Dynamic Part of Evolving Algebra Described as an Algorithm

We may think that the tedious and inconvenient way to define the Evolving Algebra specification is due to some very fundamental basic elements in the language. Since the static part (definition of the signatures is based on first order logic, we may regard this part as a basic and well understood part of Evolving Algebra.

The question is if the dynamic part can be considered to be a similar basic well understood part of the specification language.

Let us try to describe the process of executing a Evolving Algebra specification using some block structured language.

```
WHILE <some if-predicates is evaluated to true>
DO
  SELECT ARBITRARY ONE FROM
     <all transition where the
     if-predicates is evaluated to true>
  WITH THE SELECTED <transition>
      DO IN PARALLEL
         FOR EVERY <specified universe extension> DO
            DO IN SEQUENCE
               FOR EVERY <specified universe> DO
                   MAKE <new elements and add to the universe>;
               OD (for every);
               FOR EVERY <specified function update within the
                   universe extension> DO
                   FOR EVERY <possible combination of new elements
                        within an function update> DO
                        MAKE <an instances of the function update> AND
                        PERFORM FUNCTION UPDATE <instance>
                   OD (for every)
               OD (for every);
            OD (in sequence)
         OD (for every):
         FOR EVERY <specified function update at outer level> DO
             PERFORM FUNCTION UPDATE <as specified>
         OD
      OD (do in parallel)
  END <selected transition>
OD (while loop)
```

This description should not be considered as a new definition of Evolving Algebra (The definition can be found in [BR91c] and [Bör90a]). It is an illustration of how dynamic part of the Evolving Algebra specification language itself can be described as a quite complex algorithm. This description must be understood in some way.

This somewhat abstract description of an Evolving Algebra interpret reveals that the description of the dynamic part of Evolving Algebra is not simple nor is build using well understood primitive building stones.

We have construction such as SELECT ARBITRARY ONE FROM and DO IN PARALLEL ... : ... OD and DO IN SEQUENCE ... ; ... OD which itself needs a semantic specification. We need to specify exactly what PERFORM FUNCTION UPDATE means. In addition we need to define the semantic for the construction FOR EVERY ... DO ... OD.

We give an informal explanation of the constructs used above:

WHILE ... DO ... OD As long as the stated conditions is true do what is specified in the inner block. This loop makes a sequence of operations.

SELECT ARBITRARY ONE FROM ... Choose arbitrary one element from the finite set specified.

WITH THE SELECTED ... END With the chosen element do what is specified in the inner block.

DO IN PARALLEL ... : ... OD Do the specified operations in parallel

DO IN SEQUENCE ... ; ... OD Do the specified operations in sequence.

FOR EVERY ... DO ... OD For every element in the set do the specified operation once. The set is assumed to be finite. The computation may be performed in any order or in parallel.

PERFORM FUNCTION UPDATE Compute the functions update. The arguments on the left hand sides specify in which point the function gets a new value, and the expression at the right hand side gives the new value. The assignments is done such that the function updates may be performed in parallel.

MAKE Make what the text in the specification says.

It is possible to get an algorithmic understanding of the abstract description, since we always handle a finite many transitions, finite many universes, finite many elements to be added to the universes, finite many function updates and universe update within a transition.

From the description above we can see that Evolving Algebra to some extent favor specification suited for parallel programming and more or less indeterminate choices.

The examples above (See section 11.3) illustrate that specification for determinate programming, where many operations are to be performed in some order is poorly supported. Simple language constructs as for and while

loops is tedious to specify in Evolving Algebra. And it is not much support to modularize the specification.

We could argue that some degree of indeterminate specification and parallel constructions provides the necessary abstraction.

However, it is up to the person who has to make the specification to decide the level of abstraction at every part of the specification. So the specification language itself should not enforce particular types of specifications.

A basic specification language should therefore support specification of usual language construct at any degree of determination and independent of the possibilities to execute some part of the specification in parallel.

11.5 Some New Control Structures

Here we introduce some new control structures. The signature is assumed to be fixed, so only the dynamic part (transitions) of Evolving Algebra are extended. The control structure presented is common language constructs well known in common programming languages.

We will for rest of this section speak about pure Evolving Algebra as Evolving Algebra without any of the control structures introduced below.

11.5.1 Subroutines

We need some tools to support modules. One of the simplest way to support module is to introduce a subroutine construct. *In chapter 3 we discuss much more general construction for modules.*

We introduce the PERFORM command for a subroutine call:

```
if <predicate>
then
     ....
     PERFORM <module-name>
     ....
fi
```

The transitions belonging to a subroutine module are encapsulated as follows:

```
SUBROUTINE MODULE <module-name>
% First transitions listed in the module
if <predicate-1>
then
  <updates-1>
fi
...
SUBROUTINE MODULE END
```

Translation to Pure Evolving Algebra

We will first show the translation of a subroutine call to pure Evolving Algebra.

```
if <predicate>
then
   ...
   % Translation of PERFORM <module-name>
   EXTEND SUBRELEM by temp(SUBRELEM)
     subroutine_name(temp(SUBRELEM)):=<module-name>
     subroutine_stack:=push_subroutine_stack
                         (temp(SUBRELEM),subroutine_stack)
   ENDEXTEND
   ...
fi
```

Next the translation of the subroutines module follows below:

```
% Translation of the first transitions
% listed in the module.
%
if subroutine_name(top_subroutine_element
                      (subroutine_stack))=<module-name>
   & <predicate-1>
then
   <updates-1>
fi
% Other transitions in the module comes below.
...

% A special transitions which returns to the calling level of transitions.
%
if subroutine_name(top_subroutine_element
                      (subroutine_stack))=<module-name>
   & not(<predicate-1>)
   %
   % Predicates for other transitions in the module comes
   % below in the test
   ...
then
   subroutine_stack:=pop_subroutine_stack(subroutine_stack)
fi
```

Informal Description of the Subroutines

The following are executed when a subroutine call are performed:

- All transitions within the subroutine module are tested.

- If some of the predicates has the value "true" one of the transitions are performed.
- When none of the transitions are "true" the control is returned to the calling level (which may be another module).

Recursive calls

We may use the constructs above to specify recursive calls. If we do not care about how the recursive calls are performed the translation above suffice. However, if we do care (e. g. we want to implement the optimized tail recursion as defined in the Scheme Standard) about how recursive calls are to be implemented, it is better to use the general mechanism introduced in chapter 3.

Iteration as an explicit construct

A special case of a recursive control structure is known as an iteration. We do not use any stack to hold any information about the level of the recursion when we use an iteration. Therefor many programming language use this form of control structure to explicitly specify an tail recursive call which is not using any stack to store the levels of recursion calls (See also chapter 3 which specify construction for optimized tail recursive call).

We introduce the iteration as follows:

```
if <predicate>
then
   WHILE <invariant-true> DO
   <set-of-updates-inv>
   OD
   <set-of-updates-after>
fi
```

We do not introduce similar constructs known as repeat loop, for loop or a combination of the repeat and the while loop.

The while loop above could easily be translated to pure Evolving Algebra as follows:

```
if <predicate>
   & <invariant-true>
then
   <set-of-updates-inv>
fi
```

The first transition above is the transitions in the while loop.

```
if <predicate>
   & not(<invariant-true>)
then
   <set-of-updates-after>
fi
```

The last transition is the set of updates to be performed after the while loop.

11.5.2 Sequences

Sometimes it may be convenient to state that the set of updates have to be ordered in some sequences.

```
if <predicate>
then
  BEGIN SEQUENCE
  <set-of-updates-0>
  NEXT-STEP:
  ....
  END SEQUENCE
fi
```

Translation to Pure Evolving Algebra

The translation to pure Evolving Algebra is shown below:

```
% Start of the sequence.
if <predicate>
then
   next_step_<trans-id>:=0
fi

% The first set of updates in the sequence
if next_step_<trans-id>=0
then
   next_step_<trans-id>:=add(1,next_step_<trans-id>)
   <set-of-updates-0>
fi

% Reset at the end of the sequence
if next_step_<trans-id> > no_of_set_of_updates
then
   next_step_<trans-id>:=0
fi
```

11.5.3 Case Construct

At last we introduce the case command as an slightly generalization of the if construct. We may give the case command the following form:

```
if <predicate>
then
   CASE element OF
        <val-1>:
        <set-of-updates-1>
        <val-2>:
        NOUPDATES
        ......
```

```
        <val-n>:
        <set-of-updates-n>
        OTHERWISE
        <set-of-updates-n+1>
   ESAC
   <other-updates>
fi
```

We translates the case command to the following pure Evolving Algebra:

```
% One branch
if <predicate>
   & <element> = <val-1>
then
   <set-of-updates-1>
   <other-updates>
fi

% Branch with no updates
if <predicate>
   & <element> = <val-2>
then
   <other-updates>

% The default branch
if <predicate>
   & <element> =/= <val-1>
   & <element> =/= <val-2>
   ...
   & <element> =/= <val-n>
then
   <set-of-updates-n+1>
   <other-updates>
fi
```

11.5.4 Polymorphic Operators

Every functions in Evolving Algebra is connected to a signature definition. So in order to specify common operation (e. g. operations on a list) we need to specify a new function for every signature the operation applies to. We do not always want to invent new name to specify common functions, so we may want to introduce polymorphic operations. As an example we introduce some polymorphic operations on list structurers:

```
NTH_ELEMENT(list-of-instructions)
FIRST_ELEMENT(list-of-instructions)
TAIL_LIST(list-of-instructions)
NULL(list-of-instructions)
```

We assume that it is possible to translate the polymorphic definitions to pure Evolving Algebra for all signatures (or types) we want to apply the operators to.

The example above may be translated to functions applied to a list of instructions as shown below. The operators applied on other data structures are to be translated to similar Evolving Algebra functions.

```
nth_element_prog_stack(list_of_instructions)
first_element_prog_stack(list_of_instructions)
tail_list_prog_stack(list_of_instructions)
null_list_prog_stack(list_of_instructions)
```

11.6 Abstraction

We do not always want to specify a language constructs or an algorithm in great details. So we will want to make abstractions in our specification. How easy is it to make make abstractions in Evolving Algebra?

We do not need to think of the specification at the level of assembly programming (or at the level of defining Turing Machine). The Evolving Algebra gives the flexibility to make more abstract specifications.

When making a specification on how to make a graph, we can think on the levels of nodes, and archs in the graph. It is simple to make updates which corresponds naturally to the execution steps.

11.6.1 Abstraction and Making Modular Parts of the Specification

If we for some reasons do not want to specify some mechanism, we simply invent some functions (names), and maybe we will tell (informally) what this function is supposed to do. In this way we can do some abstraction by simply hiding some mechanisms.

We can also specify functions which does not taking any argument, using those functions as "oracles". We can then try to make independent specifications of what those functions are supposed to do.

On the other hand, we are left with quite primitive control structures in pure Evolving Algebra. This lack of control structures cause problems, simply because we are not able to modularize the specification, and then hide details in some of the modules. So we are forced to totally rewrite the Evolving Algebra specification for every level of abstraction.

If we extend the control structures of Evolving Algebra it may be easier to make the necessary abstraction within some part of the specification, without affecting other part of the specification.

11.6.2 Levels of Abstraction in the Specification

It may be desirable to make the specification in more than one level of abstraction. At least we would want to make a specification on a level intended to be read only by man, and in addition make a specification which could be interpreted by an Evolving Algebra interpreter.

11.6.3 Specification to be Read by Man

A specification at this level needs only contain enough to be understood by a human reader. All unimportant mechanisms can be hidden, such that the specification only deals with what is considered to be important. To ease the reading we want to extend the number of control structures in pure Evolving Algebra.

11.6.4 Specification as a Prototype Implementation

A specification intended to be evaluated by an Evolving Algebra interpreter, needs to be specified in a such way that it can be executed on a computer. If the interpret to be used is using only pure Evolving Algebra, then the specification itself has to be written at a quite low level, or some (maybe large procedures) has to be written for functions which has to be abstracted. In addition the specification may be difficult to read for humans.

11.6.5 Combining Different Levels of Specifications

It seems to be difficult to combine different levels of abstractions in an Evolving Algebra specification. Lack of the possibilities to modularize an Evolving Algebra specification, makes it difficult to combine different levels of specifications.

We may do such combinations easier by introducing new the control structures or new mechanisms. As an example the introduction of subroutine makes it easier to abstract parts of the specification.

11.7 Understanding the Evolving Algebra Specification

A specification is intended to be read by humans. So is it possible to use Evolving Algebra to write specification which can be read and understood? It seems to be possible even when using pure Evolving Algebra. We have the following possibilities:

- We can write a specification at a level intended most to be read by humans. We do not try to implement any prototype of such a specification.
- It may be desirable to write a specification which can be interpreted by an Evolving Algebra interpret. If we want such a specification to be read and understood by humans, it may be necessary to slightly rewrite the specification using some abbreviations of composite functions.

As an example of an abbreviation we may write:

```
first_item_from_source:=first(pointer_to(get_source(input_list)))
```

where functions on the right hand side is supposed to be small primitives easy to implement (as procedures) in an Evolving Algebra evaluator.

11.8 Specifying Use of Resources

A usual approach when making semantic specification is to specify what to do and nothing else. Use of resources such as processing time and memory space is considered not to be part of a semantic specification. Use of time and space depends on the particular implementation of the specified algorithm or language.

It can be good reason not to limit the semantic specification to only describe what to do. Someone who write a program or algorithm to be computed will be concerned about the time and space used to perform the computation. Much effort has to be done to find solution which makes reasonable use of resources. So why should use of resources not be part of the semantic specification?

As an example we can take the the optimalization of the tail recursive call in Lisp. An tail recursive procedure, is an recursive procedure where the recursive call is performed as the last sentence in a sequence. In the case of tail recursive procedures it is possible to skip the operation of saving the procedure instance on a stack for later execution when performing the recursive call.

If we optimize the tail recursive call as described above, a tail recursive call will not allocate space dynamicly to procedure instances. A tail recursive procedure call will be equivalent to perform iterations in a imperative language (like Algol) with regards to use of space. So there is no need to to introduce special construction like `do` loops to perform iterations, since a tail recursive call can be used.

In Scheme the optimization of tail recursive procedure call is specified to be the part of the Scheme language.

In contrast the Common Lisp do not specify such optimization. To avoid wasting space we have to use special construction similar to `do` loop to perform iterations instead of using tail recursive call.

We can see the needs of specifying the use of resources in an abstract way as part of the semantic.

Evolving Algebra is well suited to specify use of resources in an abstract way (independent of implementation on a specific machine, operating system or programming language).

In addition we are able to measure the resources used, when implementing the specification at an Evolving Algebra interpret.

Chapter 12

Evolving Algebra and Other Semantic Languages

In this section we will describe other semantic languages and to some extent compare the other languages with the Evolving Algebra semantic language.

The author is aware the difficulties of comparing different semantic languages. Different semantic languages may be useful to express different aspects and way of thinking about the programming language or algorithm. So no comparison can be really justified, if the purpose is to find out which is the best of the semantic languages. So the purpose here is mainly to shed light on how Evolving Algebra is as a semantic language in relation to other semantic language.

12.1 Evolving Algebra and Operational Semantics

We will briefly compare Evolving Algebra and Operational Semantics. In this section we will discuss the following main flavors of operational semantics:

- Operational semantics as a transitions system
- Structural Operational Semantics
- Natural Semantics

Then, we will treat the aspects of Evolving Algebra language which is of special interest with regards to the operational semantics.

12.1.1 Operational Semantics Described as a Transitions System

General Framework

Operational semantics can be described as a general transition system (See [Plo81]). Each step can be seen as a transition from one *configuration* to the next configuration, and is called a transition relation:

$$\gamma \rightarrow \gamma$$

A terminal transition system is a transition system, where it exists a set of final configuration T such that no transition relation from a final configuration to other configurations holds. This set T of configurations is intended to denote normal termination of the system.

A named transition system is a transition system where each transition relation is given a distinct name.

A stuck configuration is a configuration in a terminal transition system, say δ, such that no relation to other transition relation holds and $\delta \notin T$. A stuck configuration is intended to denote an error termination of the system.

Transition Systems and Automata

The first type of systems specified using the transition systems outlined above, is different types of automata.

The three types of automata described in [Plo81] is

- Finite automata
- Petri Nets
- SMC machine

We will below give a brief summary of embedding of the Finite Automata and the SMC machine into the transition system.

The finite automata is embedded into a terminal transition systems by defining each configuration to be a pair of a (control) state and a sequence in the given alphabet (data). A set of final configuration T is given such that any final configuration has a control component in the final state F of the finite automata. We also assume that an initial control component q_0 is given.

The Stack Memory Control (SMC) machine is embedded into a terminal transition system bye defining each configuration to consist of a triple of:

- Value Stack
- Memory Cell
- Control Stack

The L-language which is used as an example language can compute simple arithmetic and boolean expressions, and perform assignment, sequencing of operations, if tests, while loop and operation of null arity. The SMC machine is the tool used to define the semantic to the L-language.

Operational Semantics for Simple Expressions and Commands

In chapter 2 in [Plo81] the SMC machine approach to semantic language is generalized. The SMC machine tends to make too many transition steps, because the use of memory is treated in great detail. So the very detailed specification regarding the use of memory is omitted. The transition steps becomes more like a sequence of reductions in a term rewrite system. In fact the possible transitions is specified like a formal deduction system.

In the last three chapters in [Plo81] the operational semantics is described in form of derivation rules. Each transition now becomes one step in the term rewriting system (or one step in a formal deduction system which specify the properties of the language).

In this way semantics for the following command language constructs are developed:

Simple Expression Expression such as addition, multiplication, subtraction, division, and boolean expressions.

Boolean expression The usual operators are treated. Rules for sequential evaluation and parallel evaluation is specified.

Simple Commands Commands such as assignment, sequencing, **if, then, else** and **while**.

Dynamic Errors Type checking mechanism for catching dynamic errors, which may occur when a language construct is evaluated.

Static Errors Type checking mechanism for catching static errors, which can be detected after a syntactic check of the program construction.

Examples of how the principle of structural induction may be applied to the semantics rules for simple expressions and commands mentioned above are given.

Definition and Declarations

The operational semantics for definitions and declarations is given in form of formal deduction rules. The following language constructs is treated in chapter 3 in [Plo81]:

Local Definitions Local definitions in a simple applicative language.

Compound Definitions Sequential definitions, simultaneous definitions and block definitions in an applicative language.

Declarations Declarations in an imperative language. Sequential, simultaneous and block definitions is treated. Values, stores and names becomes the three main elements in the semantic description.

Functions, Procedures and Classes

In chapter 4 in [Plo81] a semantic for functions, procedures and classes are specified. The semantics for the following parameter mechanisms are specified:

- Call by value
- Call by name
- Call by reference

12.1.2 Structural Operational Semantics

Structural operational semantics try to describe the individual steps in the execution of a program (See Section 2.1 in [NN92]).

The express the semantic we use a transition relation of the form

$$(S, s) \Rightarrow \gamma$$

where S is statement(s) to be executed and s is the initial state. The γ may be one of two

- A new configuration (S_1, s_1) meaning that the execution of S is not finished and the new configuration which says what remains to do.
- A state s_t meaning that the execution of S is finished.

From this form and the usual notation of derivational rules the semantic is given by semantic rules and and axioms, which form sequences of derivations.

The derivation sequence may be:

- Finite, where the sequence ends in a defined state.
- Infinite
- Finite, not ending in any defined state.

The last case is obtained by introducing a statement `abort` and not defining any relation for `abort`. The derivation sequence will thus end in an intermediate configuration, meaning that the execution of `abort` does not gives any defined state.

In structural operational semantics we may express usual language constructions such as assignment, the empty statement, composition, the `if` statement and the `while` statement. In addition we are able to express a non-deterministic or statement, and interleaving in parallel a statements. We may also express the semantic for a block structured language (See p 52 in [NN92]).

It seems to be difficult to express operation on arbitrary data structures using Structured Operational Semantics.

12.1.3 Natural Semantics

Natural semantics try to describe the initial and final state after execution of a statement in a program. See Section 2.2 in [NN92]. To describe a transition we use the relation of the form:

$$(S, s) \Rightarrow s_1$$

The meaning of this relation is that the execution of the statement S from the state s will terminate in the state s_1.

From this form we can define semantics using axioms an rules. The derivations has the form of derivation tree, since we may use more than one premise in the rules.

The derivation tree may be:

- Finite, where the sequence ends in a defined state.
- Infinite derivation tree.

An derivation tree will be infinite if the execution can not reach a defined for one of the following reasons:

- The execution loops.
- No defined state can be reached because the execution aborts.

As in structural operational semantics we may express usual language constructions as assignment, the empty statement, composition, the `if` statement and the `while` statement in natural semantics.

When concerning nondeterministic or statement and interleaving in parallel statements, we have the following situation:

Non determinism Can not be exactly specified. The looping is suppressed by non-determinism.

Execution in parallel The interleaving can not be expressed since all execution of statements is atomic.

We can without to much difficulties express the semantics for languages using block structures and procedures.

As for structured operational semantics it seems difficult to express operations on arbitrary data structures.

12.1.4 Use of Operational Semantics

When using Operational Semantics we define some operations to be performed on some fixed data structures. In this way we hope to explain the semantics of the language constructs by telling which operations the constructs are supposed to perform on an abstract computer. However, operation on a fixed data-structure sets a limit of what we can express.

Use of resources

When generating finite derivation sequences or derivation trees in structural operational semantics as described in [Plo81] and in [NN92] we may count the number of steps used to describe the execution tree for (an instance) of a program. However, it may be quite difficult (if possible at all) to relate the number of derivation steps to some abstract measure of resources used (e.g. times of execution or use of space).

In natural semantics it is even worser. We are not able to distinguish a loop which lasts forever, and an execution which terminates in some undefined state after a finite number of steps (See p 44 in [NN92]).

So, we can safely conclude that the ability to specify use of resources as the part of the semantic is not possible using operational semantics.

12.1.5 Evolving Algebra

Pure Evolving Algebra may seem very similar to operational semantics. However, the main difference is that we are not bound to any particular data structure. Instead we are able to specify operations on any data structure we want. And since we are free to define whichever functions we want, we are free to choose any level of abstraction.

In the pure Evolving Algebra we are limited to use very few operations in order to define a semantic:

- An assignment
- The `if` test in the first part of a transition
- Generation of new elements to add to some universe.
- One loop which lasts as long there are transitions to perform.

The benefit of using so few operations is the possibility of counting the use of those operations. In this way we get easily an abstract measure of the use of resources (number of performed function updates and number of new elements added to the universes), which may correspond to the use of time and space on a computer.

However, we may abstract out some parts of the semantic by defining function which is said to perform some parts of the execution. So two assignments (often called function updates in this report) may not use the same amount of resources, since we may hide the use of resources behind the function defined and used. In such case it may make sense to measure the use of each defined function update and each new elements created for every universe used in the Evolving Algebra specification.

12.2 Evolving Algebra and Denotational Semantics

In section we will compare core evolving algebra and denotational semantics. The description of the denotational semantics is mainly based on the text [Kir91].

The comparison is divided into the following main sections:

1. An introduction section about Evolving Algebra and Denotational Semantics.
2. Comparison of language constructs.
3. How to describe properties in the semantic languages.

12.2.1 Evolving Algebra

Evolving algebra consists of a many-sorted, finite, partial, first order algebra and a set of transition rules.

The transition rules change the value of functions and set in the algebra, so we can say that a semantic specification written in evolving algebra is dynamic.

We can change the algebra in the following way using the transition rules:

- Function Updates.
- Adding a new element to a set.
- Delete an element of a set.

An evolving algebra specification has a set of transition rules. Every transition rule consists of a test and lot of updates. All changes are performed step by step. An operational description of the changes in the algebra could be described in the following way:

```
while <some transitions apply> do
     1. Perform a non deterministic selection among the transitions
        which meets the condition for execution.
     2. Perform all updates within a transition simultaneous.
od
If none of the transitions apply then Stop.
```

If not more than one transition apply at every step we are making a model of a determinate system. In the other case we are making model of a system which is indeterminate.

12.2.2 Denotational Semantics

The semantic is given by specifying mappings from a syntactical to a semantic domain.

Recursive Equations

We need to define functions as recursive equations. An example of a recursive equation follows:

$$f(x) \ = if \ b(x) \ then \ f(g(x)) \ else \ x)$$

To ensure that a recursive equation has at least one solution, we must show that the recursive defined function f has at at least one fix-points. In this way we ensure that the recursive equation is at least partial defined.

Assign the following signature for the function f:

$$f : A \rightarrow A$$

If the equation

$$a = f(a)$$

holds for at least one element a in A, then we say f has a fix-points.

If f has one ore more fix-points we define the semantic function for the equation

$$a = f(a)$$

to be the least fix-points of f.

Such fix-points exists if and only if the function f is continuous. A function which maps a complete partial ordering to a complete partial ordering is continuous if and only if it is monotone and preserve the least upper bound.

Models in Denotational Semantics

denotational semantics and the models associated with the specifications gives precise mathematical definitions. The price we have to pay is the use of complicated and in some sense a model which is not manageable.

If we use total higher order functions we may at least expect a combinatorial explosion of possible values. In many cases the models of the semantic definitions will be non-algorithmic.

In such models we get objects of infinite size and uncountable sets. As a consequence we get non-algorithmic structures.

In order to be sure that we can give some meaning to a definition in denotational semantics, we have to prove the consistence of the definition.

It turns out that such proofs often are complicated for common program language construction. As an example it is enough to mention the proof of consistence for the *while* loop. To do such proof we have to prove that we can solve the fix-points equation for the *while* loop. (See [Kir91]).

To show the consistence of a model is not equal to understand the model. We will need an algorithmic understanding of the language or algorithm where we define the model. Such algorithmic understanding i impossible if the model is non-algorithmic.

12.3 Make Comparison of Some Common Language Constructs

12.3.1 Sequence of Operations

Denotational Semantics

In denotational semantics we compose the functions which define the sequence of operations. If we need to use `goto` or `exit` we have to use continuations and reverse the compositions of functions. Continuations can be best understood as a specification of the remaining computations in the program.

Evolving Algebra

A sequence of operation can be defined by transitions which are performed in sequence or we may prefer to use composition of the functions.

12.3.2 Fixed number of iterations

When the program know in advance how many times it will iterate, we have a fixed number of iterations. In this case we can be sure that the iteration loop will terminate.

Denotational Semantics

In denotational semantics we express the iterations as composing the same function n times.

Evolving Algebra

We express the iterations as a transition which is performed as long as the counter do not exceed n.

12.3.3 Unbound number of iterations

Here the number of iterations is not fixed in advance. The termination of the loop depends on some conditions. The condition for termination may not be set, such that we have the possibility of looping forever.

Denotational Semantics

In denotational semantics we express the iterations in form of a recursive equation. So we have to find the least fix-points to this equation.

Evolving Algebra

In evolving algebra we simply specify the transition which execute one iteration step as long the condition for termination does not hold.

12.3.4 Recursive Definition

Denotational Semantics

In denotational semantics we state a recursive definition as a recursive equation. We have to prove that the function defined is total and unique in order to ensure a consistent definition.

Evolving Algebra

A recursive definition must be rewritten to an iterative construct. We have to ensure the correctness of the rewrite.

12.3.5 Non Determinism

A language we may want to specify is a nondeterministic languages where we use a set of guarded `if` and `do` sentences. A guarded sentence is a pair where the first element is a predicate and the second element is a sentence to be computed if the predicate evaluates to "true". If more than one of

the predicates in the set of guarded sentences evaluate to "true" we have a non deterministic choice among those sentences. See [Kir91] for a complete description of the language. The description of the `do` sentence in the non deterministic language (See p 245 in [Kir91]) is very similar to the way we define the operational behavior of a set of transitions in evolving algebra.

Evolving algebra

Non-deterministic specification in evolving algebra is obtained if more than one of the predicates evaluates to "true" at the same time. A non deterministic selection is made of one of those transitions which have the predicate value "true".

Denotational Semantics

If we specify a non deterministic language construct we get a transition of a state to a set of states, where each state may be a possible result of the non deterministic operation.

The signature of a function defining a non deterministic construct could be a set of states, S, mapped to the power set of S:

$$f : S \rightarrow \mathcal{P}(S)$$

All possible sets of states which can be the result of the execution of the indeterminate construction is contained in the range of the function f.

To be able to specify an iteration in a indeterminate language, we have to define a fix-points for the function used to define the iteration. That means we have to prove that f is a continuous function mapping a complete partial order to itself, i. e.:

$$f : S \rightarrow \mathcal{P}[S]$$

So we have to find a suitable ordering of the family of set defining the range of an indeterminate construct.

We will also have to restrict the family of sets of states to finitely generated subsets of the power set. All sets which may be the result of computing the indeterminate language has to be in the new family of set.

So the we have to define a complete partial order called the power-domain where the following demands has to be satisfied (See also p 249 [Kir91]:

- The carrier of the power-domain are a subset of the power set.
- Every singleton set made from an element in the set of states is in the power-domain.
- The power-domain is closed under unions.
- For every strict function f which maps the complete partial order of states to the power-domain there exists an extension f^+ which maps the power-domain to itself.

The extension f^+ is defined at page 249 in [Kir91].

Since the subset ordering does not apply the Egli-Milner ordering can be chosen (See pp 251:253 in [Kir91]).

To find a suitable set for the power-domain, we make execution trees for the imperative. The nodes in the execution represents the state, and each arc represents a possible execution step. The leaf nodes in the tree will represent the terminating states.

We assume that we have only a finite number of indeterminate choices. Then the execution trees has a finite number of branches, although a branch may have an infinite length.

So we choose all finite nonempty sets and in addition all infinite sets containing the undefined element. An infinite set can be seen as a result of executing an infinite loop.

We then have to prove that we have found a complete partial order which maps imperatives in the indeterminate languages. (See [Kir91]).

12.4 How to describe properties in the algorithms or language

12.4.1 State Transition Systems

Denotational Semantics

State transitions are specified through suitable semantic functions.

Evolving Algebra

State transition are described by updates within a transition. A state is described as a first order algebra upon finite sets.

12.4.2 Use of Resources

Denotational Semantics

When using denotational semantics we specify the relationship between input and output data. Given some input we specify what output we get. We do not specify which operations to be performed, the sequence of operations or how we perform the operations.

So it is difficult to specify the use of time and space.

Evolving Algebra

All steps to be performed can be specified in evolving algebra. The specification can be understood in an algorithmic way. Since we are free to define and use any first order logic function we want, we may give specifications at level similar to high level programming languages.

So we can specify use of resources. Resources may be the use of time or use of space. If we want to get a measure for use of time we may count the number of updates performed, and we can get the use of space by counting

the number of elements in the (finite) universes [1]. Thus the specification can take account of the use of resources like time and space.

12.4.3 Modify the Existing Specifications

Denotational Semantics

Introduction of new features in a language may very well lead to a rewrite of the specification. One example is the introduction of the `go to` statement. When `go to` is introduced we need to use continuations and reverse the compositions of functions.

Evolving Algebra

In evolving algebra change in the specification can be done by changing, adding or removing a transition. Sometimes it may be necessary to change some of the signatures, so transitions which is not going to be changed may be affected. A group of transitions may also need to get new tests added as a consequence of change in one of the transition.

[1] If we use a table to represent the values of a function which takes one or more arguments, the space may also be mesaured by the number of points defined for the function (the number of elements in the table for the function).

Chapter 13

Some Concluding Remarks Based on the Experience with Evolving Algebra

13.1 The Round Trip

The work done in this thesis is divided into three main parts:

1. Implementation of the Evolving Algebra interpreter.
2. Using the Evolving Algebra language to write specification of the interpretation and compilation of a functional language based on graph reductions.
3. Implement and running the specification on the Evolving Algebra interpret.

As far as the author know, this is the first time such a round trip is done.

It is possible to write a large specification, implement an interpreter and run a large specification using Evolving Algebra.

In addition some extension to Evolving Algebra is is added. The extensions make it possible to divide an Evolving Algebra specification into modules.

13.2 The Core Evolving Algebra

The specification of the interpretation and compilation part of functional language is done using Core Evolving Algebra.

The Evolving Algebra permits detailed specification to be written. When a specification becomes detailed, the specification also tends to be large.

The core Evolving Algebra permits the use of one execution sequence, and one name space. So the author's experience with the Core Evolving Algebra specification language can be described as such:

- It is possible to give a quite detailed specification of an algorithm.

- A detailed specifications tends to be large.
- Because the Core Evolving Algebra has only one name space and one execution sequence, a (detailed) specification may soon be so complicated, such that it is difficult to both to manage the specification and to avoid faults in the specification.
- It is possible to make the specification using exactly the Core Evolving Algebra language, and no use of abbreviation or extraneous notation. This property is essential if we want to run the specification on an Evolving Algebra interpreter, [1].
- There is no problem of specifying use of abstract resources such as time and space in Core Evolving Algebra. On the interpreter it is possible to measure the use of resources.

13.2.1 Evolving Algebra and the Use of Resources

The possibilities of specifying resources in Evolving Algebra is of interest when making a specification.

The traditional approach is to leave the use of resources, such as use of time and space out of the specification. A traditional specification tells us only what is going to be done. Gradually, when the use of semantics has evolved, it has becomes more and more needed to include the use of resources in the semantic specification.

Often, the use of resources is an essential part of the specification. As an example, it makes a lot of difference if an optimized tail recursive call is used in a language, or if new elements always are added to the recursion stack.

In Evolving Algebra we are able to specify something about how we want to compute an algorithm. And we are able to specify the behavior of the algorithm to such extent that it possible in an abstract way to specify the use resources, such as computing time and computing space. In addition we are able to measure the use of time and space (in an abstract way) when the Evolving Algebra specification is executed on an Evolving Algebra interpreter as a computer program.

We are not able to express such use of abstract resources neither in operational semantics nor in denotational semantics.

13.3 Making Modules

In computer science it is important take the abstraction of program and algorithms serious. One important tools used, when making such abstraction is simply to divide the detailed specification of the algorithm [2] into modules. So good specification (or programming) practice within computer science is

[1] Supposed that the interpreter is not augmented to run any extension to the Core Evolving Algebra.

[2] Such detailed specification intended to be run on a computer is known as a program within the computer science community.

to divide detailed specification into modules, and try to handle the modules as much as possible as whole entities.

In logic the needs to divide specifications into modules does not seems to be taken as serious as it should be. In part it has been possible to limit the size of the specification, formula or algorithm in order to keep the specification readable.

When it has not been possible to limit the size of the specification new name has been invented when necessary to re-name parts of the specification not in focus, so the specification should still be readable. This strategy only works when we do not consider automatic generation of new distinct functions [3].

13.3.1 Making Modules in the Evolving Algebra Specification

The extension to Evolving Algebra described in chapter 3 permits the specification to be divided into modules. One or more instance of an Evolving Module can be defined.

The extension to the Evolving Algebra permits following constructions to be easily specified:

- Modules which act like procedures.
- Recursive procedure calls.
- Tail recursive procedure calls.
- Executing of co-routines.

The extension provide *mechanisms* which can be used to define modules and to specify the use of the modules. Since the Evolving Algebra extension is based on mechanisms, and not directly on a specific concept (such as recursive procedure calls), new features or mechanisms can be added to the extension in the future, if necessary.

13.3.2 Why Adding New Features to Evolving Algebra

The author felt a strong need to divide the specification written in Core Evolving Algebra. The reason for this lies in the experience of writing large specification in Evolving Algebra.

When writing large and detailed specification the author found the following shortcomings with Core Evolving Algebra:

- Difficult to maintain the specification. Since all functions are global, and we are specifying just one execution sequence, it is difficult to keep track of all details about how the specification is presumed to operate.

[3] Which is necessary when maintaining more than one instance of a procedure during execution of recursive calls.

- If a new abstraction level is desired, the whole specification needs to be rewritten. In this way a top down approach to the task of writing a specification becomes difficult.

- As a consequence of the two points above it may be difficult to ensure that the specification in fact is "correctly" written.

13.3.3 Implementation of the Evolving Algebra Module Extension

The module extension of Evolving Algebra is not implemented on the Evolving Algebra interpreter (See chapter 9). So the question is if this extension could be implemented reasonable well on an Evolving Interpreter.

The author do not see any reason why the module extension could not be implemented on an Evolving Algebra interpreter.

The following point is important with regards to the implementation:

- We may think of the Evolving Algebra module as a one execution context. A finite set of procedures, both global functions and functions shared between certain modules and instances of modules is visible within one context.

  An execution context can be seen as one sequence of execution steps performed within an environment of visible functions. Each execution step can for the purpose of implementation be defined to be an Evolving Algebra transition. The execution sequence in the context can be halted and possibly be resumed later.

- Only one execution context can be the *current* execution context at any time.

- The data structure in the Evolving Algebra interpreter has to reflect the module specification. No Evolving Algebra function should be available (visible) for execution outside what is defined for the current execution context.

- The Evolving Algebra interpreter must be able to switch from one execution context to another execution context according to the extended Evolving Algebra specifications.

- More than one instance of a module may be created as an execution context. The instances and all functions local to one of the instances has to be given distinct internal names within the interpreter.

- All communication between the execution context take places through global or shared functions. However, some special constants used to control the jumps between different contexts has to be maintained.

None of the requirements above is difficult to implement when implementing an interpreter for Evolving Algebra. However, care has to be taken with regards to efficiency of the implementation.

For efficiency reason it can be well worth to investigate the possibility of combining the use of a high level programming language like Scheme with a medium level programming language like C, when implementing an extended Evolving Algebra interpreter.

If we think that Evolving Algebra should be extended to cover distributed and parallel computing, such approach may in practice be necessary. In such case the the language used to implement the interpreter should be able to call subroutines written in C. In addition it should be possible to call routines written in the high level language (such as Scheme) from a C routine. The reason why, is that most of the libraries and operating system calls has an interface which is supposed to be used from the C language.

On the other hand the author judge the implementation of an Evolving Algebra interpreter solely in the C language (or C++ language), to bee far too complicated to be feasible within reasonable time and amount of work. In this respect the author consider the implementation of an Evolving Algebra interpreter which has all properties as described in chapter 9 [4] and in addition implements the extensions proposed in this report.

13.4 Short Summary of other Evolving Algebra Implementations

13.4.1 Evolving Algebra Interpreter Written in C

An implementation of an Evolving Algebra interpreter in C which the author know about (See [Hug94]) is based on a fixed number of functions, which may be combined in order to define the implementation of new functions. The author believe that this way to implement an Evolving Algebra interpreter is too restrictive with regards to the possibility to execute Evolving Algebra specification at different chosen abstraction levels. The reason is that the whole specification is dependent on those (may bee too few) basic functions (See section 9.5 for a comparison between the C interpreter and the Scheme interpreter).

This interpreter (See [Hug94]) is the only interpreter besides the author's interpreter the author has inspected.

13.4.2 DASL Compiler Implemented in Prolog

In addition to the interpreter mentioned in subsection13.4.1 above, the author is recently become aware of a report by [Kap93]. This report describes a prototype of a compiler for the language *DASL* which is an extension to the Evolving Algebra specification language. In addition to pure Evolving Algebra the language includes polymorphic types and equational specifications which constitutes a confluent rewrite system.

This compiler is implemented in Prolog.

[4] e.g. links user defined procedures which define the semantics of Evolving Algebra functions.

13.5 Future Research

This thesis does not cover Evolving Algebra specifications for parallel systems and distributed systems.

The extension of Evolving Algebra for making modules described in chapter 3 can to some extent be used to describe processes on parallel or distributed systems. However, those tools has to be extended in order to cover communication and synchronization of processes in the parallel and distributed environment.

This is left to the future research task to make specification of parallel and distributed systems, and extend the Evolving Algebra to cover such systems, and divide such systems into modules.

Bibliography

[AN90] editor Adrian Nye. *X Protocol Reference Manual.* O'Reilly, Sebastopol, 2nd edition, 1990.

[Bör90a] Egon Börger. A logical operational semantics of full prolog. Report 111, Wissenschaftliches Zentrum, Institut für Wissensbasierte Systeme, 1990.

[Bör90b] Egon Börger. A logical operational semantics of full prolog part II. Built-in predicates for database manipulations. In B. Rovan, editor, *MFCS'90 Mathematical Foundations of Computer Science*, volume 452 of *LNCS*, pages 1–14. Springer, 1990.

[Bör90c] Egon Börger. A logical operational semantics of full prolog part III: Built-in predicates for files, terms, arithmetic and input-output. Report 117, Wissenschaftliches Zentrum Institut für Wissensbasierte Systeme, 1990.

[BR91a] Egon Börger and Elvinia Riccobene. A formal specification of PARLOG. Unpublished, 1991.

[BR91b] Egon Börger and Dean Rosenzweig. A formal analysis of prolog database views and their uniform implementation. To appear as: [Tech. Rep. EECS Univ. of Michigan], 1991.

[BR91c] Egon Börger and Dean Rosenzweig. From prolog algebras towards WAM — a mathematical study of implementation. To appear in: CSL'90. 4th Workshop on Computer Science Logic, Springer LNCS, 1991.

[BR94] Egon Börger and Dean Rosenzweig. A mathematical definition of full prolog. Report, EECS Departement, University of Michigan, 1994. URL: ftp://ftp.eecs.umich.edu /groups/Ealgebras/prolog.ps.

[FN84] Jens Erik Fenstad and Dag Normann. Algorithms and logic. Unpublished, 1984.

[GK93] Yuri Gurevich and James K.Huggins. The semantics for the c programming language. Report, EECS Departement, University of Michigan, 1993. URL: ftp://ftp.eecs.umich.edu /groups/Ealgebras/calgebra.ps.

[Gur93] Yuri Gurevich, 1993. Personal communication.

[HS86] J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinators and Lambda-calculus*. Cambridge University Press, 1986.

[Hug94] James K. Huggins. An evolving algebra interpreter. Unpublished, 1994.

[JL91] Simon L. Peyton Jones and David Lester. *Implementing Functional Languages. A tutorial*. Prentic Hall, 1991.

[Jon87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentic Hall, 1987.

[Kap93] Angelica Maria Kappel. Executable specification based on dynamic algebra. In A. Voronkov, editor, *Logic Programming and Automated Reasoning*, volume 698 of *LNAI*, pages 578–592. Springer, 1993. URL: ftp://ftp.eecs.umich.edu /groups/Ealgebras/prolint.abst.ps.

[Kir91] Bjørn Kirkerud. The semantics of programming languages. Unpublished, 1991.

[Mor90] Carrol Morgan. *Programming from specifications*. Prentic Hall, 1990.

[NN92] Hanne Riis Nielson and Flemming Nielson. *Semantics with applications. A formal Introduction*. John Wiley, 1992.

[Plo81] Gordon D. Plotkin. A structural approach to operational semantic. Report, Computer Science Department, Aarhus University, rhus, 1981. Reprinted 1991.

[RWC86] Jonathan Rees and editors William Clinger. Revised 3 report on the algorithmic language scheme. *ACM SIGPLAN Notices*, (21(12)), 1986.

[Wan77] Arne Wang. Kvasiparallelle rutiner - ko-rutiner. Kompendium 3, Institutt for Informatikk, Universitetet i Oslo, 1977.

Part V

Appendix

Appendix A

Evolving Algebra Specification Given to the Interpret

Her we list the Evolving Algebra specification as given to the interpret.

A.1 The Template Instantiation Specification

A.1.1 Specification of the Compilation

The specification to compile the supercombinator definitions into a graph representation is shown below:

```
% resets before loading
reset

% Signatures
signature status : STATUS

signature tempstack: (TADDR *)
signature emptystack: (TADDR *)
signature initialstack: (TADDR *)
signature isemptystack: ((TADDR *) --> BOOL)
signature topaddr: ((TADDR *) --> TADDR)
signature pushstack: (((TADDR *) x TADDR) --> (TADDR *))
signature popstack: ((TADDR *) --> (TADDR *))

signature currscdefaddr: TADDR
signature valueofaddr: (TADDR --> [SCEXPR + INSTR])
signature graph: (TADDR --> NODE)

signature mainscdefname: SCNAME
signature getmainname: ((SCEXPR *) --> SCNAME)
signature allscdefs: (SCEXPR *)
signature isemptyscdefs: (SCEXPR --> BOOL)
```

```
signature getnextscdef: ((SCEXPR*) --> SCEXPR)
signature tailscdefs: ((SCEXPR*) --> (SCEXPR*))

signature getnamefromglobals: (TADDR --> SCNAME)
signature getaddrfromglobals: (SCNAME --> TADDR)

signature emptyexpr: SCEXPR
signature currscdef: SCEXPR
signature exprtype: (SCEXPR --> SCTYPE)
signature makeparams: (SCEXPR --> (SCEXPR *))
signature getscdefname: (SCEXPR --> SCNAME)
signature srcbody: (SCEXPR --> SCEXPR)
signature firstappexpr: (SCEXPR --> SCEXPR)
signature secondappexpr: (SCEXPR --> SCEXPR)
signature makenum: (SCEXPR --> NUMBER)
signature makescname: (SCEXPR --> SCNAME)
signature makevarname: (SCEXPR --> VARNAME)

signature nodechild: ((NUMBER x NODE) --> [TADDR + {Empty}])
signature nodetype: (NODE --> NTYPE)
signature nodeparams: (NODE --> (VARNAME *))
signature nodenum: (NODE --> NUMBER)
signature nodescname: (NODE --> SCNAME)
signature nodevarname: (NODE --> VARNAME)

% Load procedure files
% Load standard user environment procedures from file!
%
loadproc "Ea-system-lib/ea-std-user-extension.scm";
loadproc "Ea-system-lib/ea-std-user-update.scm";
loadproc "Ea-system-lib/ea-std-user-lookup.scm";

% Load procedures maintaining lists
loadproc "Ea-system-lib/ea-std-user-list.scm";

% Load arithmetic procedures
loadproc "Ea-system-lib/ea-std-user-arithmetic.scm";

% Load graphical reduction procedure.
loadproc "Graf-reduksjon-lib/ea-graf-red-app-expr.scm";
loadproc "Graf-reduksjon-lib/ea-graf-red-find-type.scm";
loadproc "Graf-reduksjon-lib/ea-graf-red-number.scm";
loadproc "Graf-reduksjon-lib/ea-graf-red-scdef.scm";
loadproc "Graf-reduksjon-lib/ea-graf-red-scname.scm";
loadproc "Graf-reduksjon-lib/ea-graf-red-variable.scm";

% Assignments
% assign <func-symb>, <lookup-proc>, <upd-proc>, <fmess-symb>;
```

```
% Status
assignfunc status, constant-std-lookup-data,
                    user-update-constant, std-const-dta;
% Pointer to the graph
assignfunc graph, table-std-lookup,
                  user-update-function, std-table;
assignfunc valueofaddr, table-std-lookup,
                  user-update-function, std-table;
% The address stack
assignfunc currscdefaddr, constant-std-lookup-data,
                    user-update-constant, std-const-dta;
assignfunc tempstack, constant-std-lookup-data,
                    user-update-constant, std-const-dta;
assignfunc emptystack, constant-std-lookup-data,
                    user-update-constant, std-const-dta;
assignfunc initialstack, constant-std-lookup-data,
                    user-update-constant, std-const-dta;
assignfunc isemptystack, empty-list, dummy-func, upd-not-perm;
assignfunc topaddr, first-from-list, dummy-func, upd-not-perm;
assignfunc pushstack, add-to-list, dummy-func, upd-not-perm;
assignfunc popstack, tail-from-list, dummy-func, upd-not-perm;

% The sc definitions
assignfunc mainscdefname, constant-std-lookup-data,
                     user-update-constant, std-const-dta;
assignfunc getmainname, get-scdef-main-name,
                     dummy-func, upd-not-perm;
assignfunc allscdefs, constant-std-lookup-data,
                     user-update-constant, std-const-dta;
assignfunc isemptyscdefs, empty-list, dummy-func, upd-not-perm;
assignfunc getnextscdef, first-from-list, dummy-func, upd-not-perm;
assignfunc tailscdefs, tail-from-list, dummy-func, upd-not-perm;

% The globals
assignfunc getnamefromglobals, table-std-lookup,
             user-update-function, std-table;

assignfunc getaddrfromglobals, table-std-lookup,
             user-update-function, std-table;

% The sc expression
assignfunc emptyexpr, constant-std-lookup-data,
                    user-update-constant, std-const-dta;
assignfunc currscdef, constant-std-lookup-data,
                    user-update-constant, std-const-dta;
assignfunc getscdefname, get-scdef-name, dummy-func, upd-not-perm;
assignfunc exprtype, find-type-gen, dummy-func, upd-not-perm;
```

```
assignfunc makeparams, get-params, dummy-func, upd-not-perm;
assignfunc srcbody, get-scbody-expr, dummy-func, upd-not-perm;
assignfunc firstappexpr, get-first-app-expr,
                    dummy-func, upd-not-perm;
assignfunc secondappexpr, get-second-app-expr,
                    dummy-func, upd-not-perm;
assignfunc makenum, get-sc-number, dummy-func, upd-not-perm;
assignfunc makescname, get-sc-name, dummy-func, upd-not-perm;
assignfunc makevarname, get-sc-var, dummy-func, upd-not-perm;

% The graph
assignfunc nodechild, table-std-lookup,
                     user-update-function, std-table;
assignfunc nodetype, table-std-lookup,
                    user-update-function, std-table;
assignfunc nodeparams, table-std-lookup,
                       user-update-function, std-table;
assignfunc nodenum, table-std-lookup,
                    user-update-function, std-table;
assignfunc nodescname, table-std-lookup,
                    user-update-function, std-table;
assignfunc nodevarname, table-std-lookup,
                       user-update-function, std-table;

% Assign to universe
% assign <universe-symbol> <universe-ext-proc> <umess-symbol>;
assignuniverse NODE, std-ext-collection, number;
assignuniverse TADDR, std-ext-collection, newsymbol;

% Initial values
loadalg "Graf-red-source/SC/church-partiell-two.src";
%initial allscdefs :==
%    [((= ((sc "Main") ()) (((sc "K") (num 1)) (num 2)))
%      (= ((sc "K") ((var "x") (var "y"))) ((sc "I") (var "x")))
%      (= ((sc "I") ((var "z"))) (var "z")) )]
initial emptyexpr :== [ (empty) ]

%initial status :== "Get-curr-sc-def"
initial status :== "Initial"
initial emptystack :== [()]
initial currscdefaddr :== [start]

%initial tempstack :== [()]


% Start the compilation
if    ( = (status, "Initial") &
        (! isemptyscdefs(allscdefs)) );
```

```
%then
   funcupdate mainscdefname:=getmainname(allscdefs)
   funcupdate status:="Get-curr-sc-def"
endupdates

% Prepare for compilation of the supercombinator definition
if    ( = (status, "Get-curr-sc-def") &
        (! isemptyscdefs(allscdefs)) );
%then
   extend
      extenduniverse TADDR;
   withupdates
      funcupdate currscdefaddr:=temp(TADDR,1);
      funcupdate getnamefromglobals(temp(TADDR,1)):=
           getscdefname(getnextscdef(allscdefs));
      funcupdate getaddrfromglobals(getscdefname(getnextscdef(allscdefs))):=
           temp(TADDR,1);
      funcupdate valueofaddr(temp(TADDR,1)):=getnextscdef(allscdefs);
   endextend
   funcupdate allscdefs:=tailscdefs(allscdefs);
   funcupdate status:="Compile-sc-def";
endupdates

% Prepare for performing graph reductions
if ( isemptyscdefs(allscdefs) &
     = (status, "Get-curr-sc-def") );
%then
   funcupdate status:="Perform-graph-reds"
endupdates

% Supercombinator definition
if ( = (status, "Compile-sc-def") &
     = (exprtype(valueofaddr(currscdefaddr)),
         "SDEFexpr"));
%then
   extend
      extenduniverse TADDR;
      extenduniverse NODE;
   withupdates
      funcupdate graph(currscdefaddr):=temp(NODE,1);
      funcupdate nodetype(temp(NODE,1)):="Supercomb";
      funcupdate nodeparams(temp(NODE,1)):=
                    makeparams(valueofaddr(currscdefaddr));
      funcupdate nodechild(1,temp(NODE,1)):=temp(TADDR,1);
      funcupdate valueofaddr(temp(TADDR,1)):=
                 srcbody(valueofaddr(currscdefaddr));
      funcupdate tempstack:=pushstack(emptystack,temp(TADDR,1));
   endextend
```

```
   funcupdate status:="Compile-the-body";
endupdates

% Application
if ( = (status, "Compile-the-body") &
     (! isemptystack(tempstack)) &
     = (exprtype(valueofaddr(topaddr(tempstack))),
        "APexpr"));
%then
   extend
      extenduniverse TADDR # 2;
      extenduniverse NODE;
   withupdates
      funcupdate graph(topaddr(tempstack)):=temp(NODE,1);
      funcupdate valueofaddr(topaddr(tempstack)):=emptyexpr;
      funcupdate nodechild(1,temp(NODE,1)):=temp(TADDR,1);
      funcupdate nodechild(2,temp(NODE,1)):=temp(TADDR,2);
      funcupdate nodetype(temp(NODE,1)):="APnode";
      funcupdate valueofaddr(temp(TADDR,1)):=
         firstappexpr(valueofaddr(topaddr(tempstack)));
      funcupdate valueofaddr(temp(TADDR,2)):=
         secondappexpr(valueofaddr(topaddr(tempstack)));
      funcupdate tempstack:=pushstack(
                    pushstack(tempstack,temp(TADDR,1)),
                    temp(TADDR,2));

   endextend
endupdates

% Number expression
if ( = (status, "Compile-the-body") &
     (! isemptystack(tempstack)) &
     = (exprtype(valueofaddr(topaddr(tempstack))),
        "NUMexpr"));
%then
   extend
      extenduniverse NODE;
   withupdates
      funcupdate graph(topaddr(tempstack)):=
         temp(NODE,1);
      funcupdate valueofaddr(topaddr(tempstack)):=
         emptyexpr;
      funcupdate nodetype(temp(NODE,1)):="Num";
      funcupdate nodenum(temp(NODE,1)):=
         makenum(valueofaddr(topaddr(tempstack)));
   endextend
endupdates
```

```
% Name expression
if ( = (status, "Compile-the-body") &
     (! isemptystack(tempstack)) &
     = (exprtype(valueofaddr(topaddr(tempstack))),
        "SCname"));
%then
   extend
      extenduniverse NODE;
   withupdates
      funcupdate graph(topaddr(tempstack)):=
         temp(NODE,1);
      funcupdate valueofaddr(topaddr(tempstack)):=
         emptyexpr;
      funcupdate nodetype(temp(NODE,1)):="SCName";
      funcupdate nodescname(temp(NODE,1)):=
         makescname(valueofaddr(topaddr(tempstack)));
   endextend
endupdates

% Local variable name expression
if ( = (status, "Compile-the-body") &
     (! isemptystack(tempstack)) &
     = (exprtype(valueofaddr(topaddr(tempstack))),
        "VARname"));
%then
   extend
      extenduniverse NODE;
   withupdates
      funcupdate graph(topaddr(tempstack)):=
         temp(NODE,1);
      funcupdate valueofaddr(topaddr(tempstack)):=
         emptyexpr;
      funcupdate nodetype(temp(NODE,1)):="LVar";
      funcupdate nodevarname(temp(NODE,1)):=
         makevarname(valueofaddr(topaddr(tempstack)));
   endextend
endupdates

% Traverse up
if ( = (status, "Compile-the-body") &
     (! isemptystack(tempstack)) &
     = (exprtype(valueofaddr(topaddr(tempstack))),
        "EMPty"))
%then
   funcupdate tempstack:=popstack(tempstack);
endupdates;

% End of on supecombinator definition
```

```
if ( = (status, "Compile-the-body") &
     isemptystack(tempstack));
%then
   funcupdate status:="Get-curr-sc-def";
endupdates;

% Loads part two
loadalg "Graf-red-source/supercomb-red.src";
```

A.1.2 Specification of the Reduction Process

The specification to reduce the compiled graph is shown here:

```
% To be loaded into the evolving algebra interpret
% after supercomb-comp.src

% More signatures.
%
signature getoperator: (INSTR --> OPERATOR)
signature egraphinstr: INSTR
signature instrstack: (INSTR *)
signature makegcode: (INSTR --> (INSTR *))
signature addrstack: (TADDR *)
signature scdefaddr: TADDR
signature dumpstack: (DUMP *)
signature emptydumpstack: (DUMP *)
signature leftbranch: NUMBER
signature rightbranch: NUMBER
signature currparams: (VARNAME *)
signature currarity: NUMBER
signature lengthas: ((TADDR *) --> NUMBER)
signature numberofparams: ((VARNAME *) --> NUMBER)
signature rootofredex: TADDR
signature currcounter: NUMBER
signature getsubstvalueaddr: (NAME --> TADDR)
signature getparamvar: ((NUMBER x (NAME *)) --> NAME)
signature add: ((NUMBER x NUMBER) --> NUMBER)
signature pointertodef: (TADDR --> TADDR)
signature finished:  (TADDR --> BOOL)
signature rootofinstance: TADDR
signature sameaddr: TADDR x TADDR --> BOOL
signature result: NUMBER


% Procedures to be loaded
loadproc "Ea-system-lib/ea-std-user-misc.scm";

% More assignments
assignfunc egraphinstr, constant-std-lookup-data,
```

```
                        user-update-constant, std-const-dta;
assignfunc instrstack, constant-std-lookup-data,
                             user-update-constant, std-const-dta;
assignfunc addrstack, constant-std-lookup-data,
                      user-update-constant, std-const-dta;
assignfunc scdefaddr, constant-std-lookup-data,
                      user-update-constant, std-const-dta;
assignfunc dumpstack, constant-std-lookup-data,
                     user-update-constant, std-const-dta;
assignfunc emptydumpstack, constant-std-lookup-data,
                     user-update-constant, std-const-dta;
assignfunc leftbranch, constant-std-lookup-data,
                       user-update-constant, std-const-dta;
assignfunc rightbranch, constant-std-lookup-data,
                       user-update-constant, std-const-dta;
assignfunc currparams, constant-std-lookup-data,
                        user-update-constant, std-const-dta;
assignfunc currarity, constant-std-lookup-data,
                        user-update-constant, std-const-dta;
assignfunc getoperator, table-std-lookup,
                        user-update-function, std-table;
assignfunc makegcode, make-list, dummy-func, upd-not-perm;
assignfunc lengthas, length-of-the-list, dummy-func, upd-not-perm;
assignfunc numberofparams, length-of-the-list, dummy-func, upd-not-perm;
assignfunc makegcode, make-list, dummy-func, upd-not-perm;
assignfunc rootofredex, constant-std-lookup-data,
                        user-update-constant, std-const-dta;
assignfunc rootofinstance, constant-std-lookup-data,
                        user-update-constant, std-const-dta;
assignfunc result, constant-std-lookup-data,
                        user-update-constant, std-const-dta;
assignfunc currcounter, constant-std-lookup-data,
                        user-update-constant, std-const-dta;
assignfunc getsubstvalueaddr, table-std-lookup,
                        user-update-function, std-table;
assignfunc pointertodef, table-std-lookup,
                        user-update-function, std-table;
assignfunc finished, table-std-lookup,
                        user-update-function, std-table;
assignfunc getparamvar, get-param-name, dummy-func, upd-not-perm;
assignfunc add, add-numbers, dummy-func, upd-not-perm;
assignfunc sameaddr, compare-two-vars, dummy-func, upd-not-perm;


% Assign to the universe.
assignuniverse INSTR, std-ext-collection, newsymbol;

% Initial values
```

```
initial emptydumpstack:== [()]
initial addrstack:== [()]

if = (status, "Perform-graph-reds");
%then
   funcupdate dumpstack:=emptydumpstack;
   funcupdate leftbranch:=1;
   funcupdate rightbranch:=2;
   extend
      extenduniverse INSTR;
      extenduniverse TADDR;
      extenduniverse NODE;
   withupdates
      funcupdate getoperator(temp(INSTR,1)):="Egraph";
      funcupdate egraphinstr:=temp(INSTR,1);
      funcupdate instrstack:=makegcode(temp(INSTR,1));
      funcupdate graph(temp(TADDR,1)):=temp(NODE,1);
      funcupdate nodetype(temp(NODE,1)):="SCName";
      funcupdate nodescname(temp(NODE,1)):=mainscdefname;
      funcupdate addrstack:=pushstack(emptystack,temp(TADDR,1));
   endextend
   funcupdate status:="Unwind";
endupdates;

if ( = (status, "Unwind") &
     = (nodetype(graph(topaddr(addrstack))), "APnode"));
%then
   funcupdate addrstack:=pushstack(addrstack,nodechild(leftbranch,
                            graph(topaddr(addrstack))));
endupdates

if ( = (status, "Unwind") &
     = (nodetype(graph(topaddr(addrstack))), "Num"));
%then
   funcupdate result:=nodenum(graph(topaddr(addrstack)))
   funcupdate status:="Normal-form";
endupdates

if ( = (status, "Unwind") &
     = (nodetype(graph(topaddr(addrstack))), "SCName"));
%then
   funcupdate scdefaddr:=
         getaddrfromglobals(nodescname(graph(topaddr(addrstack))));
   funcupdate status:="Unwind-scname";
endupdates

if ( = (status, "Unwind-scname") &
     = (nodetype(graph(scdefaddr)), "Supercomb") &
```

```
     > (lengthas(addrstack),numberofparams(nodeparams(graph(scdefaddr)))) )
%then
   funcupdate currparams:=nodeparams(graph(scdefaddr));
   funcupdate currarity:=numberofparams(nodeparams(graph(scdefaddr)));
   funcupdate status:="Make-substs-init";
endupdates

% Initialize the substitutions
if = (status, "Make-substs-init");
%then
   funcupdate currcounter:=0;
   funcupdate status:="Make-substs";
endupdates

% Extend the list of substitution pair.
if ( = (status, "Make-substs") &
     < (currcounter,currarity));
%then
   funcupdate getsubstvalueaddr(getparamvar
               (add(currcounter,1),currparams)):=
                    nodechild(rightbranch,graph(topaddr
                    (popstack(addrstack))));
   funcupdate currcounter:=add(currcounter,1);
   funcupdate addrstack:=popstack(addrstack);
endupdates

% Finished with the substitutions
% Retain the last element of address stack until the root of redex constant
% is set.
if ( = (status, "Make-substs") &
     = (currcounter,currarity));
%then
   funcupdate rootofredex:=topaddr(addrstack);
   funcupdate status:="Init-instance";
endupdates

if = (status, "Init-instance");
%then
   extend
     extenduniverse TADDR;
   withupdates
     funcupdate pointertodef(temp(TADDR,1)):=
                  nodechild(1,graph(scdefaddr));
     funcupdate addrstack:=pushstack(addrstack,temp(TADDR,1));
     funcupdate rootofinstance:=temp(TADDR,1)
     funcupdate finished(temp(TADDR,1)):="False";
   endextend
   funcupdate status:="Build-instance";
```

```
endupdates

if ( = (status, "Build-instance") &
     =/= (finished(topaddr(addrstack)),"True") &
     = (nodetype(graph(pointertodef(topaddr(addrstack)))),"APnode"));
%then
   funcupdate finished(topaddr(addrstack)):="True";
   extend
      extenduniverse TADDR # 2;
      extenduniverse NODE;
   withupdates
      funcupdate graph(topaddr(addrstack)):=temp(NODE,1);
      funcupdate nodetype(temp(NODE,1)):=
         nodetype(graph(pointertodef(topaddr(addrstack))));
      funcupdate nodechild(leftbranch,temp(NODE,1)):=temp(TADDR,1);
      funcupdate nodechild(rightbranch,temp(NODE,1)):=temp(TADDR,2);
      funcupdate finished(temp(TADDR,1)):="False";
      funcupdate finished(temp(TADDR,2)):="False";
      funcupdate pointertodef(temp(TADDR,1)):=
         nodechild(leftbranch,graph(pointertodef(topaddr(addrstack))));
      funcupdate pointertodef(temp(TADDR,2)):=
         nodechild(rightbranch,graph(pointertodef(topaddr(addrstack))));
      funcupdate addrstack:=pushstack(
                     pushstack(addrstack,temp(TADDR,1)),temp(TADDR,2));
   endextend
endupdates

if ( = (status, "Build-instance") &
     =/= (finished(topaddr(addrstack)),"True") &
     = (nodetype(graph(pointertodef(topaddr(addrstack)))),"Num"));
%then
   funcupdate finished(topaddr(addrstack)):="True";
   extend
      extenduniverse NODE;
   withupdates
      funcupdate graph(topaddr(addrstack)):=temp(NODE,1);
      funcupdate nodetype(temp(NODE,1)):=
         nodetype(graph(pointertodef(topaddr(addrstack))));
      funcupdate nodenum(temp(NODE,1)):=
         nodenum(graph(pointertodef(topaddr(addrstack))));
   endextend
endupdates


if ( = (status, "Build-instance") &
     =/= (finished(topaddr(addrstack)),"True") &
     = (nodetype(graph(pointertodef(topaddr(addrstack)))),"SCName"));
%then
```

```
   funcupdate finished(topaddr(addrstack)):="True";
   extend
      extenduniverse NODE;
   withupdates
      funcupdate graph(topaddr(addrstack)):=temp(NODE,1);
      funcupdate nodetype(temp(NODE,1)):=
         nodetype(graph(pointertodef(topaddr(addrstack))));
      funcupdate nodescname(temp(NODE,1)):=
         nodescname(graph(pointertodef(topaddr(addrstack))));
   endextend
endupdates

% Sharing distinct variables.
if ( = (status, "Build-instance") &
     =/= (finished(topaddr(addrstack)),"True") &
     = (nodetype(graph(pointertodef(topaddr(addrstack)))),"LVar"));
%then
   funcupdate finished(topaddr(addrstack)):="True";
   funcupdate graph(topaddr(addrstack)):=
      graph(getsubstvalueaddr(nodevarname(graph
          (pointertodef(topaddr(addrstack))))));
endupdates

if ( = (status, "Build-instance") &
     = (finished(topaddr(addrstack)),"True") &
     (! sameaddr(topaddr(addrstack),rootofinstance)));
%then
   funcupdate addrstack:=popstack(addrstack);
endupdates

if ( = (status, "Build-instance") &
     = (finished(topaddr(addrstack)),"True") &
     sameaddr(topaddr(addrstack),rootofinstance));
%then
   funcupdate status:="Update";
endupdates

% Make a copy of the result of the reduction
if = (status, "Update");
%then
   funcupdate addrstack:=
       pushstack(popstack(popstack(addrstack)),rootofinstance);
   funcupdate status:="Unwind";
endupdates
```

A.2 The G-machine Specification

A.2.1 Specification of the Compilation

The specification to compile the supercombinator definitions into G-machine instructions is shown below:

```
% G-maskinen

% resets before loading
reset

% Signatures
signature status : STATUS
signature add: ((NUMBER x NUMBER) --> NUMBER)

signature tempstack: (TADDR *)
signature emptystack: (TADDR *)
signature initialstack: (TADDR *)
signature isemptystack: ((TADDR *) --> BOOL)
signature topaddr: ((TADDR *) --> TADDR)
signature pushstack: (((TADDR *) x TADDR) --> (TADDR *))
signature popstack: ((TADDR *) --> (TADDR *))

signature currscdefaddr: TADDR
signature valueofaddr: (TADDR --> [SCEXPR + INSTR])
signature graph: (TADDR --> NODE)

signature mainscdefname: SCNAME
signature getmainname: ((SCEXPR *) --> SCNAME)
signature allscdefs: (SCEXPR *)
signature isemptyscdefs: (SCEXPR --> BOOL)
signature getnextscdef: ((SCEXPR*) --> SCEXPR)
signature tailscdefs: ((SCEXPR*) --> (SCEXPR*))

signature getnamefromglobals: (TADDR --> SCNAME)
signature getaddrfromglobals: (SCNAME --> TADDR)

signature emptyexpr: SCEXPR
signature currscdef: SCEXPR
signature exprtype: (SCEXPR --> SCTYPE)
signature makeparams: (SCEXPR --> (SCEXPR *))
signature getscdefname: (SCEXPR --> SCNAME)
signature srcbody: (SCEXPR --> SCEXPR)
signature firstappexpr: (SCEXPR --> SCEXPR)
signature secondappexpr: (SCEXPR --> SCEXPR)
signature makenum: (SCEXPR --> NUMBER)
signature makescname: (SCEXPR --> SCNAME)
signature makevarname: (SCEXPR --> VARNAME)
```

```
signature numberofparams: ((SCEXPR *) --> NUMBER)
signature getparamlist: (TADDR --> (PARAMPOS *))
signature makeparamposlist: (SCEXPR --> (PARAMPOS *))
signature incrementposlist: ((NUMBER x (PARAMPOS *)) --> (PARAMPOS *))
signature getposition: (((PARAMPOS *) x VARNAME) --> NUMBER)

signature getoperator: (INSTR --> OPERATOR)
signature getoperand: ((NUMBER x INSTR) --> OPERAND)

signature codelist: (INSTR *)
signature instrstack: (INSTR *)
signature emptycodelist: (INSTR *)
signature makegcode: (INSTR --> (INSTR *))
signature makegcodetwo: ((INSTR x INSTR) --> (INSTR *))
signature concatcode: (((INSTR *) x (INSTR *)) --> (INSTR *))

signature instructions: (TADDR --> (INSTR *))
signature hascode: (TADDR --> BOOL)

signature nodetype: (NODE --> NTYPE)
signature defarity: (NODE --> NUMBER)
signature finishedcode: (NODE --> (INSTR *))
signature leftbranch: NUMBER
signature rightbranch: NUMBER

% To be used by the reduction part of Graph Machine
signature nodechild: ((NUMBER x NODE) --> [TADDR + {Empty}])
signature nodeparams: (NODE --> (VARNAME *))
signature nodenum: (NODE --> NUMBER)
signature nodescname: (NODE --> SCNAME)
signature nodevarname: (NODE --> VARNAME)

% Load procedure files
% Load standard user environment procedures from file!
%
loadproc "Ea-system-lib/ea-std-user-extension.scm";
loadproc "Ea-system-lib/ea-std-user-update.scm";
loadproc "Ea-system-lib/ea-std-user-lookup.scm";

% Load procedures maintaining lists
loadproc "Ea-system-lib/ea-std-user-list.scm";

% Load arithmetic procedures
loadproc "Ea-system-lib/ea-std-user-arithmetic.scm";

% Load graphical reduction procedure.
loadproc "Graf-reduksjon-lib/ea-graf-red-app-expr.scm";
```

```
loadproc "Graf-reduksjon-lib/ea-graf-red-find-type.scm";
loadproc "Graf-reduksjon-lib/ea-graf-red-number.scm";
loadproc "Graf-reduksjon-lib/ea-graf-red-scdef.scm";
loadproc "Graf-reduksjon-lib/ea-graf-red-scname.scm";
loadproc "Graf-reduksjon-lib/ea-graf-red-variable.scm";
loadproc "Graf-reduksjon-lib/ea-gcode-make-parlist.scm";

% Assignments
% assign <func-symb>, <lookup-proc>, <upd-proc>, <fmess-symb>;

% Status
assignfunc status, constant-std-lookup-data,
                    user-update-constant, std-const-dta;
% Addition
assignfunc add, add-numbers, dummy-func, upd-not-perm;
% Pointer to the graph
assignfunc graph, table-std-lookup,
                  user-update-function, std-table;
assignfunc valueofaddr, table-std-lookup,
                  user-update-function, std-table;
% The address stack
assignfunc currscdefaddr, constant-std-lookup-data,
                    user-update-constant, std-const-dta;
assignfunc tempstack, constant-std-lookup-data,
                    user-update-constant, std-const-dta;
assignfunc emptystack, constant-std-lookup-data,
                    user-update-constant, std-const-dta;
assignfunc initialstack, constant-std-lookup-data,
                    user-update-constant, std-const-dta;
assignfunc isemptystack, empty-list, dummy-func, upd-not-perm;
assignfunc topaddr, first-from-list, dummy-func, upd-not-perm;
assignfunc pushstack, add-to-list, dummy-func, upd-not-perm;
assignfunc popstack, tail-from-list, dummy-func, upd-not-perm;

% The sc definitions
assignfunc mainscdefname, constant-std-lookup-data,
                     user-update-constant, std-const-dta;
assignfunc getmainname, get-scdef-main-name,
                     dummy-func, upd-not-perm;
assignfunc allscdefs, constant-std-lookup-data,
                     user-update-constant, std-const-dta;
assignfunc isemptyscdefs, empty-list, dummy-func, upd-not-perm;
assignfunc getnextscdef, first-from-list, dummy-func, upd-not-perm;
assignfunc tailscdefs, tail-from-list, dummy-func, upd-not-perm;

% The globals
assignfunc getnamefromglobals, table-std-lookup,
             user-update-function, std-table;
```

```
assignfunc getaddrfromglobals, table-std-lookup,
            user-update-function, std-table;

% The sc expression
assignfunc emptyexpr, constant-std-lookup-data,
                    user-update-constant, std-const-dta;
assignfunc currscdef, constant-std-lookup-data,
                    user-update-constant, std-const-dta;
assignfunc getscdefname, get-scdef-name, dummy-func, upd-not-perm;
assignfunc exprtype, find-type-gen, dummy-func, upd-not-perm;
assignfunc makeparams, get-params, dummy-func, upd-not-perm;
assignfunc srcbody, get-scbody-expr, dummy-func, upd-not-perm;
assignfunc firstappexpr, get-first-app-expr,
                    dummy-func, upd-not-perm;
assignfunc secondappexpr, get-second-app-expr,
                    dummy-func, upd-not-perm;
assignfunc makenum, get-sc-number, dummy-func, upd-not-perm;
assignfunc makescname, get-sc-name, dummy-func, upd-not-perm;
assignfunc makevarname, get-sc-var, dummy-func, upd-not-perm;

% Parameter-list
assignfunc numberofparams, length-of-the-list, dummy-func, upd-not-perm;
assignfunc getparamlist, table-std-lookup,
                    user-update-function, std-table;

% * New procedures to be made.
assignfunc makeparamposlist, make-param-pos-list, dummy-func, upd-not-perm;
assignfunc incrementposlist, increm-param-pos, dummy-func, upd-not-perm;
assignfunc getposition, get-position, dummy-func, upd-not-perm;

% Contents of instructions
assignfunc getoperator, table-std-lookup,
                    user-update-function, std-table;
assignfunc getoperand, table-std-lookup,
                    user-update-function, std-table;
% The instructions
assignfunc codelist, constant-std-lookup-data,
                    user-update-constant, std-const-dta;
assignfunc instrstack, constant-std-lookup-data,
                    user-update-constant, std-const-dta;
assignfunc emptycodelist, constant-std-lookup-data,
                    user-update-constant, std-const-dta;
assignfunc makegcode, make-list, dummy-func, upd-not-perm;
assignfunc makegcodetwo, make-list, dummy-func, upd-not-perm;
assignfunc concatcode, concat-lists, dummy-func, upd-not-perm;
assignfunc instructions, table-std-lookup,
                    user-update-function, std-table;
```

```
assignfunc hascode, table-std-lookup,
                    user-update-function, std-table;

% The graph
assignfunc nodetype, table-std-lookup,
                    user-update-function, std-table;
assignfunc defarity, table-std-lookup,
                    user-update-function, std-table;
assignfunc finishedcode, table-std-lookup,
                    user-update-function, std-table;
assignfunc leftbranch, constant-std-lookup-data,
                    user-update-constant, std-const-dta;
assignfunc rightbranch, constant-std-lookup-data,
                    user-update-constant, std-const-dta;

assignfunc nodechild, table-std-lookup,
                    user-update-function, std-table;
assignfunc nodeparams, table-std-lookup,
                       user-update-function, std-table;
assignfunc nodenum, table-std-lookup,
                    user-update-function, std-table;
assignfunc nodescname, table-std-lookup,
                    user-update-function, std-table;
assignfunc nodevarname, table-std-lookup,
                      user-update-function, std-table;

% Assign to universe
% assign <universe-symbol> <universe-ext-proc> <umess-symbol>;
assignuniverse NODE, std-ext-collection, number;
assignuniverse TADDR, std-ext-collection, newsymbol;
assignuniverse INSTR, std-ext-collection, newsymbol;

% Initial values
loadalg "Graf-red-source/SC/church-partiell-two.src";
%initial allscdefs :==
%    [((= ((sc "Main") ()) (((sc "K") (num 1)) (num 2)))
%      (= ((sc "K") ((var "x") (var "y"))) ((sc "I") (var "x")))
%      (= ((sc "I") ((var "z"))) (var "z")) )]
initial emptyexpr :== [ (empty) ]

%initial status :== "Get-curr-sc-def"
initial status :== "Initial"
initial emptystack :== [()]
initial codelist :== [()]
initial emptycodelist :== [()]
initial currscdefaddr :== [start]

%initial tempstack :== [()]
```

```
% Start the compilation
if    ( = (status, "Initial") &
        (! isemptyscdefs(allscdefs)) );
%then
   funcupdate mainscdefname:=getmainname(allscdefs)
   funcupdate status:="Get-curr-sc-def"
endupdates

% Prepare for compilation of the supercombinator definition
if    ( = (status, "Get-curr-sc-def") &
        (! isemptyscdefs(allscdefs)) );
%then
   extend
      extenduniverse TADDR;
   withupdates
      funcupdate currscdefaddr:=temp(TADDR,1);
      funcupdate getnamefromglobals(temp(TADDR,1)):=
           getscdefname(getnextscdef(allscdefs));
      funcupdate getaddrfromglobals(getscdefname(getnextscdef(allscdefs))):=
           temp(TADDR,1);
      funcupdate valueofaddr(temp(TADDR,1)):=getnextscdef(allscdefs);
   endextend
   funcupdate allscdefs:=tailscdefs(allscdefs);
   funcupdate status:="Compile-sc-def";
endupdates

% Prepare for performing graph reductions
if ( = (status, "Get-curr-sc-def") &
     isemptyscdefs(allscdefs) );
%then
   extend
      extenduniverse INSTR # 2;
   withupdates
         funcupdate getoperator(temp(INSTR,1)):="Pushglobal";
         funcupdate getoperand(1,temp(INSTR,1)):=mainscdefname;
         funcupdate getoperator(temp(INSTR,2)):="Unwind";
         funcupdate instrstack:=
                 makegcodetwo(temp(INSTR,1),temp(INSTR,2));
   endextend
   funcupdate status:="Exec-code";
   funcupdate leftbranch:=1
   funcupdate rightbranch:=2
endupdates

% Supercombinator definition
if ( = (status, "Compile-sc-def") &
```

```
     = (exprtype(valueofaddr(currscdefaddr)),"SDEFexpr"));
%then
   extend
      extenduniverse TADDR # 2;
      extenduniverse NODE;
      extenduniverse INSTR # 2;
   withupdates
      funcupdate graph(currscdefaddr):=temp(NODE,1);
      funcupdate nodetype(temp(NODE,1)):="Global";
      funcupdate defarity(temp(NODE,1)):=
                  numberofparams(makeparams(valueofaddr(currscdefaddr)));
      funcupdate valueofaddr(temp(TADDR,2)):=
                srcbody(valueofaddr(currscdefaddr));
      funcupdate getparamlist(temp(TADDR,2)):=
                  makeparamposlist(valueofaddr(currscdefaddr));
      funcupdate hascode(temp(TADDR,2)):="False";
      funcupdate getoperator(temp(INSTR,1)):="Slide";
      funcupdate getoperand(1,temp(INSTR,1)):=
                add(1,numberofparams(makeparams
                           (valueofaddr(currscdefaddr))));
      funcupdate getoperator(temp(INSTR,2)):="Unwind";
      funcupdate instructions(temp(TADDR,1)):=
                makegcodetwo(temp(INSTR,1),temp(INSTR,2));
      funcupdate hascode(temp(TADDR,1)):="True";
      % Make the testpredicates below happy.
      funcupdate valueofaddr(temp(TADDR,1)):=valueofaddr(currscdefaddr);
      funcupdate tempstack:=pushstack(
                    pushstack(emptystack,temp(TADDR,1)),
                    temp(TADDR,2));
   endextend
   funcupdate status:="Compile-the-body";
endupdates

% Application
if ( = (status, "Compile-the-body") &
     (! isemptystack(tempstack)) &
     = (exprtype(valueofaddr(topaddr(tempstack))),"APexpr") &
     = (hascode(topaddr(tempstack)),"False"));
%then
   extend
      extenduniverse TADDR # 2;
      extenduniverse INSTR;
   withupdates
      % First AP expression underneath the second on the
      % instruction stack (and on top of address stack).
      funcupdate valueofaddr(temp(TADDR,1)):=
           firstappexpr(valueofaddr(topaddr(tempstack)));
      funcupdate hascode(temp(TADDR,1)):="False"
```

```
      funcupdate getparamlist(temp(TADDR,1)):=
              incrementposlist(1,getparamlist(topaddr(tempstack)));
      % Second AP expression on top of the instruction stack
      % (and second on address stack).
      funcupdate valueofaddr(temp(TADDR,2)):=
           secondappexpr(valueofaddr(topaddr(tempstack)));
      funcupdate hascode(temp(TADDR,2)):="False";
      funcupdate getparamlist(temp(TADDR,2)):=
              getparamlist(topaddr(tempstack));
      % Make the AP instruction
      funcupdate getoperator(temp(INSTR,1)):="MKap";
      funcupdate instructions(topaddr(tempstack)):=
                 makegcode(temp(INSTR,1));
      funcupdate hascode(topaddr(tempstack)):="True";
      funcupdate tempstack:=pushstack(
                    pushstack(tempstack,temp(TADDR,1)),
                    temp(TADDR,2));
   endextend
endupdates

% Number expression
if ( = (status, "Compile-the-body") &
     (! isemptystack(tempstack)) &
     = (exprtype(valueofaddr(topaddr(tempstack))),"NUMexpr") &
     = (hascode(topaddr(tempstack)),"False"));
%then
   extend
      extenduniverse INSTR;
   withupdates
      funcupdate getoperator(temp(INSTR,1)):="Pushint";
      funcupdate getoperand(1,temp(INSTR,1)):=
                 makenum(valueofaddr(topaddr(tempstack)));
      funcupdate instructions(topaddr(tempstack)):=
                 makegcode(temp(INSTR,1));
      funcupdate hascode(topaddr(tempstack)):="True";
   endextend
endupdates

% Name expression
if ( = (status, "Compile-the-body") &
     (! isemptystack(tempstack)) &
     = (exprtype(valueofaddr(topaddr(tempstack))),"SCname") &
     = (hascode(topaddr(tempstack)),"False"));
%then
   extend
      extenduniverse INSTR;
   withupdates
      funcupdate getoperator(temp(INSTR,1)):="Pushglobal";
```

```
      funcupdate getoperand(1,temp(INSTR,1)):=
                 makescname(valueofaddr(topaddr(tempstack)));
      funcupdate instructions(topaddr(tempstack)):=
                 makegcode(temp(INSTR,1));
      funcupdate hascode(topaddr(tempstack)):="True";
   endextend
endupdates

% Variable expression
if ( = (status, "Compile-the-body") &
     (! isemptystack(tempstack)) &
     = (exprtype(valueofaddr(topaddr(tempstack))),"VARname") &
     = (hascode(topaddr(tempstack)),"False"));
%then
   extend
      extenduniverse INSTR;
   withupdates
      funcupdate getoperator(temp(INSTR,1)):="Push";
      funcupdate getoperand(1,temp(INSTR,1)):=
                 getposition(getparamlist(topaddr(tempstack)),
                             makevarname(valueofaddr(topaddr(tempstack))));
      funcupdate instructions(topaddr(tempstack)):=
                 makegcode(temp(INSTR,1));
      funcupdate hascode(topaddr(tempstack)):="True";
   endextend
endupdates

% Traverse up
if ( = (status, "Compile-the-body") &
     (! isemptystack(tempstack)) &
     = (hascode(topaddr(tempstack)),"True"));
%then
   % The instruction is always appendended at the end of
   % the instruction list.
   funcupdate codelist:=concatcode(codelist,
                           instructions(topaddr(tempstack)));
   funcupdate tempstack:=popstack(tempstack);
endupdates;

% End of on supecombinator definition
if ( = (status, "Compile-the-body") &
     isemptystack(tempstack));
%then
   funcupdate status:="Get-curr-sc-def";
   funcupdate finishedcode(graph(currscdefaddr)):=codelist;
   funcupdate codelist:=emptycodelist;
endupdates;
```

```
% Loads part two
loadalg "Graf-red-source/gmaskin-red.src";
```

A.2.2 Specification of the Reduction Process

The specification to execute the G-machine instructions is shown below:

```
% To be loaded into the evolving algebra interpret
% after supercomb-comp.src

% More signatures.
%
signature topinstr: ((INSTR *) --> INSTR)
signature popinstr: ((INSTR *) --> (INSTR *))

signature addrstack: (TADDR *)
signature currglobdefaddr: TADDR
signature getnthaddr: ((NUMBER x (TADDR *)) --> TADDR)
signature popnaddrs: ((NUMBER x (TADDR *)) --> (TADDR *))
signature lengthas: ((TADDR *) --> NUMBER)

signature result: NUMBER

% More assignments
assignfunc topinstr, first-from-list, dummy-func, upd-not-perm;
assignfunc popinstr, tail-from-list, dummy-func, upd-not-perm;

assignfunc addrstack, constant-std-lookup-data,
                      user-update-constant, std-const-dta;
assignfunc currglobdefaddr, constant-std-lookup-data,
                      user-update-constant, std-const-dta;
assignfunc popnaddrs, take-nth-first-from-list, dummy-func, upd-not-perm;
assignfunc getnthaddr, pick-elem-from-list, dummy-func, upd-not-perm;
assignfunc lengthas, length-of-the-list, dummy-func, upd-not-perm;

assignfunc result, constant-std-lookup-data,
                        user-update-constant, std-const-dta;

% No assign to universe

% Initial values
initial addrstack:== [()]

if ( = (status, "Exec-code") &
     = (getoperator(topinstr(instrstack)), "Pushglobal") );
%then
   extend
      extenduniverse TADDR;
      extenduniverse NODE;
```

```
   withupdates
      funcupdate graph(temp(TADDR,1)):=temp(NODE,1);
      funcupdate nodetype(temp(NODE,1)):="SCname";
      funcupdate nodescname(temp(NODE,1)):=
                    getoperand(1,topinstr(instrstack));
      funcupdate addrstack:=pushstack(addrstack,temp(TADDR,1));
   endextend
   funcupdate instrstack:=popinstr(instrstack);
endupdates;

if ( = (status, "Exec-code") &
     = (getoperator(topinstr(instrstack)), "Pushint") );
%then
   extend
      extenduniverse TADDR;
      extenduniverse NODE;
   withupdates
      funcupdate graph(temp(TADDR,1)):=temp(NODE,1);
      funcupdate nodetype(temp(NODE,1)):="Num";
      funcupdate nodenum(temp(NODE,1)):=
                    getoperand(1,topinstr(instrstack));
      funcupdate addrstack:=pushstack(addrstack,temp(TADDR,1));
   endextend
   funcupdate instrstack:=popinstr(instrstack);
endupdates;

if ( = (status, "Exec-code") &
     = (getoperator(topinstr(instrstack)), "Push") );
%then
    funcupdate addrstack:=
         pushstack(addrstack,nodechild(rightbranch,
                   graph(getnthaddr
                    (add(getoperand(1,topinstr(instrstack)),2),
                    addrstack)) ));
   funcupdate instrstack:=popinstr(instrstack);
endupdates;

if ( = (status, "Exec-code") &
     = (getoperator(topinstr(instrstack)), "MKap") );
%then
   extend
      extenduniverse TADDR;
      extenduniverse NODE;
   withupdates
      funcupdate graph(temp(TADDR,1)):=temp(NODE,1);
      funcupdate nodetype(temp(NODE,1)):="APnode";
      funcupdate nodechild(leftbranch,temp(NODE,1)):=
          getnthaddr(1,addrstack);
```

```
      funcupdate nodechild(rightbranch,temp(NODE,1)):=
          getnthaddr(2,addrstack);
      funcupdate addrstack:=pushstack(popnaddrs(2,addrstack),temp(TADDR,1));
   endextend
   funcupdate instrstack:=popinstr(instrstack);
endupdates;

if ( = (status, "Exec-code") &
     = (getoperator(topinstr(instrstack)), "Slide") );
%then
  funcupdate addrstack:=
      pushstack(popnaddrs(add(getoperand(1,topinstr(instrstack)),1)
                           ,addrstack)
                ,topaddr(addrstack))
  funcupdate instrstack:=popinstr(instrstack);
endupdates;

% The unwind instructions.
if ( = (status, "Exec-code") &
     = (getoperator(topinstr(instrstack)), "Unwind") &
     = (nodetype(graph(topaddr(addrstack))), "APnode") );
%then
  funcupdate addrstack:=
      pushstack(addrstack,nodechild(leftbranch,graph(topaddr(addrstack))));
endupdates;

if ( = (status, "Exec-code") &
     = (getoperator(topinstr(instrstack)), "Unwind") &
     = (nodetype(graph(topaddr(addrstack))), "Num") );
%then
  funcupdate status:="Normal-form";
  funcupdate result:=nodenum(graph(topaddr(addrstack)));
  funcupdate instrstack:=popinstr(instrstack);
endupdates;

if ( = (status, "Exec-code") &
     = (getoperator(topinstr(instrstack)), "Unwind") &
     = (nodetype(graph(topaddr(addrstack))), "SCname") );
%then
  funcupdate currglobdefaddr:=
         getaddrfromglobals(nodescname(graph(topaddr(addrstack))));
  funcupdate instrstack:=popinstr(instrstack);
  funcupdate status:="Exec-sc-def";
endupdates;

if ( = (status, "Exec-sc-def") &
     = (nodetype(graph(currglobdefaddr)), "Global") &
     > (lengthas(addrstack),defarity(graph(currglobdefaddr))) );
```

```
%then
  funcupdate instrstack:=
       concatcode(finishedcode(graph(currglobdefaddr)),instrstack);
  funcupdate status:="Exec-code";
endupdates;
```

A.3 The Template Instantiation Specification extended to strict arguments and primitives

A.3.1 Specification of the Compilation

The specification to compile the supercombinator definitions extended to strict arguments and primitives into a graph representation is shown below:

```
% resets before loading
reset

% Signatures
signature status : STATUS

signature tempstack: (TADDR *)
signature emptystack: (TADDR *)
signature initialstack: (TADDR *)
signature isemptystack: ((TADDR *) --> BOOL)
signature topaddr: ((TADDR *) --> TADDR)
signature pushstack: (((TADDR *) x TADDR) --> (TADDR *))
signature popstack: ((TADDR *) --> (TADDR *))

signature currscdefaddr: TADDR
signature currprimdefaddr: TADDR
signature valueofaddr: (TADDR --> [SCEXPR + INSTR])
signature graph: (TADDR --> NODE)

signature mainscdefname: SCPRIMNAME
signature getmainname: ((SCEXPR *) --> SCPRIMNAME)
signature allscdefs: (SCEXPR *)
signature isemptyscdefs: ((SCEXPR *) --> BOOL)
signature getnextscdef: ((SCEXPR *) --> SCEXPR)
signature tailscdefs: ((SCEXPR *) --> (SCEXPR*))

signature allprimdefs: (PRIMEXPR *)
signature isemptyprimdefs: (PRIMEXPR --> BOOL)
signature getnextprimdef: ((PRIMEXPR *) --> PRIMEXPR)
signature tailprimdefs: ((PRIMEXPR *) --> (PRIMEXPR*))

signature getnamefromglobals: (TADDR --> SCPRIMNAME)
signature getaddrfromglobals: (SCPRIMNAME --> TADDR)
```

```
signature emptyexpr: SCEXPR
signature currscdef: SCEXPR
signature exprtype: (SCEXPR --> SCTYPE)
signature getscdefname: (SCEXPR --> SCPRIMNAME)
signature makeparams: (SCEXPR --> (SCEXPR *))
signature srcbody: (SCEXPR --> SCEXPR)
signature firstappexpr: (SCEXPR --> SCEXPR)
signature secondappexpr: (SCEXPR --> SCEXPR)
signature makenum: (SCEXPR --> NUMBER)
signature makebool: (SCEXPR --> BOOLVAL)
signature makedata: (SCEXPR --> DATAVAL)
signature makescname: (SCEXPR --> SCPRIMNAME)
signature makeprimname: (SCEXPR --> SCPRIMNAME)
signature makevarname: (SCEXPR --> VARNAME)

signature getprimdefname: (PRIMEXPR --> SCPRIMNAME)
signature makeprimparaminfo: (PRIMEXPR --> (SCPRIMPARAMTYPE *))
signature makeprimarity: (PRIMEXPR --> NUMBER)

signature nodechild: ((NUMBER x NODE) --> [TADDR + {Empty}])
signature nodetype: (NODE --> NTYPE)
signature nodeparams: (NODE --> (VARNAME *))
%signature nodenum: (NODE --> NUMBER)
signature nodevalue: (NODE --> [NUMBER + BOOL + DATA])
signature nodescname: (NODE --> SCPRIMNAME)
signature nodeprimname: (NODE --> SCPRIMNAME)
signature nodevarname: (NODE --> VARNAME)
signature nodearity: (NODE --> NUMBER)
signature nodeparaminfo: (NODE --> (PARAMTYPE *))

% Load procedure files
% Load standard user environment procedures from file!
%
loadproc "Ea-system-lib/ea-std-user-extension.scm";
loadproc "Ea-system-lib/ea-std-user-update.scm";
loadproc "Ea-system-lib/ea-std-user-lookup.scm";

% Load procedures maintaining lists
loadproc "Ea-system-lib/ea-std-user-list.scm";

% Load arithmetic procedures
loadproc "Ea-system-lib/ea-std-user-arithmetic.scm";

% Load graphical reduction procedure.
loadproc "Graf-reduksjon-lib/ea-graf-red-app-expr.scm";
loadproc "Graf-reduksjon-lib/ea-graf-red-find-type.scm";
loadproc "Graf-reduksjon-lib/ea-graf-red-number.scm";
loadproc "Graf-reduksjon-lib/ea-graf-red-scdef.scm";
```

```
loadproc "Graf-reduksjon-lib/ea-graf-red-scname.scm";
loadproc "Graf-reduksjon-lib/ea-graf-red-variable.scm";
loadproc "Graf-reduksjon-lib/ea-prim-def.scm";

% Assignments
% assign <func-symb>, <lookup-proc>, <upd-proc>, <fmess-symb>;

% Status
assignfunc status, constant-std-lookup-data,
                    user-update-constant, std-const-dta;
% Pointer to the graph
assignfunc graph, table-std-lookup,
                  user-update-function, std-table;
assignfunc valueofaddr, table-std-lookup,
                  user-update-function, std-table;

% The address stack
assignfunc currscdefaddr, constant-std-lookup-data,
                    user-update-constant, std-const-dta;
assignfunc currprimdefaddr, constant-std-lookup-data,
                    user-update-constant, std-const-dta;
assignfunc tempstack, constant-std-lookup-data,
                    user-update-constant, std-const-dta;
assignfunc emptystack, constant-std-lookup-data,
                    user-update-constant, std-const-dta;
assignfunc initialstack, constant-std-lookup-data,
                    user-update-constant, std-const-dta;
assignfunc isemptystack, empty-list, dummy-func, upd-not-perm;
assignfunc topaddr, first-from-list, dummy-func, upd-not-perm;
assignfunc pushstack, add-to-list, dummy-func, upd-not-perm;
assignfunc popstack, tail-from-list, dummy-func, upd-not-perm;

% The sc definitions
assignfunc mainscdefname, constant-std-lookup-data,
                     user-update-constant, std-const-dta;
assignfunc getmainname, get-scdef-main-name,
                     dummy-func, upd-not-perm;
assignfunc allscdefs, constant-std-lookup-data,
                     user-update-constant, std-const-dta;
assignfunc isemptyscdefs, empty-list, dummy-func, upd-not-perm;
assignfunc getnextscdef, first-from-list, dummy-func, upd-not-perm;
assignfunc tailscdefs, tail-from-list, dummy-func, upd-not-perm;

% The primitive definitions
assignfunc allprimdefs, constant-std-lookup-data,
                     user-update-constant, std-const-dta;
assignfunc isemptyprimdefs, empty-list, dummy-func, upd-not-perm;
assignfunc getnextprimdef, first-from-list, dummy-func, upd-not-perm;
```

```
assignfunc tailprimdefs, tail-from-list, dummy-func, upd-not-perm;

% The globals
assignfunc getnamefromglobals, table-std-lookup,
            user-update-function, std-table;

assignfunc getaddrfromglobals, table-std-lookup,
            user-update-function, std-table;

% The sc expression
assignfunc emptyexpr, constant-std-lookup-data,
                    user-update-constant, std-const-dta;
assignfunc currscdef, constant-std-lookup-data,
                    user-update-constant, std-const-dta;
assignfunc getscdefname, get-scdef-name, dummy-func, upd-not-perm;
assignfunc exprtype, find-type-gen, dummy-func, upd-not-perm;
assignfunc makeparams, get-params, dummy-func, upd-not-perm;
assignfunc srcbody, get-scbody-expr, dummy-func, upd-not-perm;
assignfunc firstappexpr, get-first-app-expr,
                    dummy-func, upd-not-perm;
assignfunc secondappexpr, get-second-app-expr,
                    dummy-func, upd-not-perm;
assignfunc makenum, get-sc-number, dummy-func, upd-not-perm;
assignfunc makebool, get-sc-boolean, dummy-func, upd-not-perm;
assignfunc makedata, get-sc-data, dummy-func, upd-not-perm;
assignfunc makescname, get-sc-name, dummy-func, upd-not-perm;
assignfunc makeprimname, get-prim-name, dummy-func, upd-not-perm;
assignfunc makevarname, get-sc-var, dummy-func, upd-not-perm;

% The primitive expression
assignfunc getprimdefname, get-primdef-name, dummy-func, upd-not-perm;
assignfunc makeprimparaminfo, get-prim-param-types, dummy-func,
           upd-not-perm;
assignfunc makeprimarity, get-prim-arity, dummy-func, upd-not-perm;

% The graph
assignfunc nodechild, table-std-lookup,
                     user-update-function, std-table;
assignfunc nodetype, table-std-lookup,
                    user-update-function, std-table;
assignfunc nodeparams, table-std-lookup,
                       user-update-function, std-table;
%assignfunc nodenum, table-std-lookup,
%                     user-update-function, std-table;
assignfunc nodevalue, table-std-lookup,
                    user-update-function, std-table;
assignfunc nodescname, table-std-lookup,
                    user-update-function, std-table;
```

```
assignfunc nodeprimname, table-std-lookup,
                    user-update-function, std-table;
assignfunc nodevarname, table-std-lookup,
                      user-update-function, std-table;
assignfunc nodearity, table-std-lookup,
                      user-update-function, std-table;
assignfunc nodeparaminfo, table-std-lookup,
                      user-update-function, std-table;

% Assign to universe
% assign <universe-symbol> <universe-ext-proc> <umess-symbol>;
assignuniverse NODE, std-ext-collection, number;
assignuniverse TADDR, std-ext-collection, newsymbol;

% Initial values
%loadalg "Graf-red-source/PRIM-SC/prim-test-dump.src";
loadalg "Graf-red-source/PRIM-SC/strict-sc-test.src";
%initial allscdefs :==
%    [((= ((sc "SEL") ()) ((((prim "if") (bool "true")) (sc "CP")) (num 4)))
%      (= ((sc "CP") ()) (((prim "plus") (num 1)) (num 2)))
%     )]

% Primitive definitions unlikely to change. More may be added.
initial allprimdefs :==
    [((primitive (prim "plus") (num 2) (number number))
      (primitive (prim "if") (num 3) (boolean nonstrict nonstrict))
       )]

% More to be added.
initial emptyexpr :== [ (empty) ]

%initial status :== "Get-curr-sc-def"
initial status :== "Initial"
initial emptystack :== [()]
initial currscdefaddr :== [start]

%initial tempstack :== [()]


% Start the compilation
if    ( = (status, "Initial") &
        (! isemptyscdefs(allscdefs)) );
%then
   funcupdate mainscdefname:=getmainname(allscdefs)
   funcupdate status:="Get-curr-sc-def"
endupdates

% Prepare for compilation of the supercombinator definition
```

```
if    ( = (status, "Get-curr-sc-def") &
        (! isemptyscdefs(allscdefs)) );
%then
   extend
      extenduniverse TADDR;
   withupdates
      funcupdate currscdefaddr:=temp(TADDR,1);
      funcupdate getnamefromglobals(temp(TADDR,1)):=
           getscdefname(getnextscdef(allscdefs));
      funcupdate getaddrfromglobals(getscdefname(getnextscdef(allscdefs))):=
           temp(TADDR,1);
      funcupdate valueofaddr(temp(TADDR,1)):=getnextscdef(allscdefs);
   endextend
   funcupdate allscdefs:=tailscdefs(allscdefs);
   funcupdate status:="Compile-sc-def";
endupdates

% Prepare for performing graph reductions
if ( isemptyscdefs(allscdefs) &
     = (status, "Get-curr-sc-def") );
%then
   funcupdate status:="Get-curr-prim-def"
endupdates

% Prepare for compilation of the primitive definition
if    ( = (status, "Get-curr-prim-def") &
        (! isemptyprimdefs(allprimdefs)) );
%then
   extend
      extenduniverse TADDR;
   withupdates
      funcupdate currprimdefaddr:=temp(TADDR,1);
      funcupdate getnamefromglobals(temp(TADDR,1)):=
           getprimdefname(getnextprimdef(allprimdefs));
      funcupdate getaddrfromglobals(getprimdefname(getnextprimdef(allprimdefs))):=
           temp(TADDR,1);
      funcupdate valueofaddr(temp(TADDR,1)):=getnextprimdef(allprimdefs);
   endextend
   funcupdate allprimdefs:=tailprimdefs(allprimdefs);
   funcupdate status:="Compile-prim-def";
endupdates

% Supercombinator definition
if = (status, "Compile-prim-def");
%then
   extend
      extenduniverse NODE;
   withupdates
```

```
      funcupdate graph(currprimdefaddr):=temp(NODE,1);
      funcupdate nodetype(temp(NODE,1)):="Primitive";
      funcupdate nodeparaminfo(temp(NODE,1)):=
                    makeprimparaminfo(valueofaddr(currprimdefaddr));
      funcupdate nodearity(temp(NODE,1)):=
                    makeprimarity(valueofaddr(currprimdefaddr));
   endextend
   funcupdate status:="Get-curr-prim-def";
endupdates

% Prepare for performing graph reductions
if ( isemptyprimdefs(allprimdefs) &
     = (status, "Get-curr-prim-def") );
%then
   funcupdate status:="Perform-graph-reds"
endupdates

% Supercombinator definition
if ( = (status, "Compile-sc-def") &
     = (exprtype(valueofaddr(currscdefaddr)),
         "SDEFexpr"));
%then
   extend
      extenduniverse TADDR;
      extenduniverse NODE;
   withupdates
      funcupdate graph(currscdefaddr):=temp(NODE,1);
      funcupdate nodetype(temp(NODE,1)):="Supercomb";
      funcupdate nodeparams(temp(NODE,1)):=
                    makeparams(valueofaddr(currscdefaddr));
      funcupdate nodechild(1,temp(NODE,1)):=temp(TADDR,1);
      funcupdate valueofaddr(temp(TADDR,1)):=
                 srcbody(valueofaddr(currscdefaddr));
      funcupdate tempstack:=pushstack(emptystack,temp(TADDR,1));
   endextend
   funcupdate status:="Compile-the-body";
endupdates

% Application
if ( = (status, "Compile-the-body") &
     (! isemptystack(tempstack)) &
     = (exprtype(valueofaddr(topaddr(tempstack))),
        "APexpr"));
%then
   extend
      extenduniverse TADDR # 2;
      extenduniverse NODE;
   withupdates
```

```
      funcupdate graph(topaddr(tempstack)):=temp(NODE,1);
      funcupdate valueofaddr(topaddr(tempstack)):=emptyexpr;
      funcupdate nodechild(1,temp(NODE,1)):=temp(TADDR,1);
      funcupdate nodechild(2,temp(NODE,1)):=temp(TADDR,2);
      funcupdate nodetype(temp(NODE,1)):="APnode";
      funcupdate valueofaddr(temp(TADDR,1)):=
         firstappexpr(valueofaddr(topaddr(tempstack)));
      funcupdate valueofaddr(temp(TADDR,2)):=
         secondappexpr(valueofaddr(topaddr(tempstack)));
      funcupdate tempstack:=pushstack(
                    pushstack(tempstack,temp(TADDR,1)),
                    temp(TADDR,2));

   endextend
endupdates

% Number expression
if ( = (status, "Compile-the-body") &
     (! isemptystack(tempstack)) &
     = (exprtype(valueofaddr(topaddr(tempstack))),
        "NUMexpr"));
%then
   extend
      extenduniverse NODE;
   withupdates
      funcupdate graph(topaddr(tempstack)):=
         temp(NODE,1);
      funcupdate valueofaddr(topaddr(tempstack)):=
         emptyexpr;
      funcupdate nodetype(temp(NODE,1)):="Num";
      funcupdate nodevalue(temp(NODE,1)):=
         makenum(valueofaddr(topaddr(tempstack)));
   endextend
endupdates

% Boolean expression
if ( = (status, "Compile-the-body") &
     (! isemptystack(tempstack)) &
     = (exprtype(valueofaddr(topaddr(tempstack))),
        "BOOLexpr"));
%then
   extend
      extenduniverse NODE;
   withupdates
      funcupdate graph(topaddr(tempstack)):=
         temp(NODE,1);
      funcupdate valueofaddr(topaddr(tempstack)):=
         emptyexpr;
```

```
      funcupdate nodetype(temp(NODE,1)):="Bool";
      funcupdate nodevalue(temp(NODE,1)):=
         makebool(valueofaddr(topaddr(tempstack)));
   endextend
endupdates

% Data expression
if ( = (status, "Compile-the-body") &
     (! isemptystack(tempstack)) &
     = (exprtype(valueofaddr(topaddr(tempstack))),
        "DATAexpr"));
%then
   extend
      extenduniverse NODE;
   withupdates
      funcupdate graph(topaddr(tempstack)):=
         temp(NODE,1);
      funcupdate valueofaddr(topaddr(tempstack)):=
         emptyexpr;
      funcupdate nodetype(temp(NODE,1)):="Data";
      funcupdate nodevalue(temp(NODE,1)):=
         makedata(valueofaddr(topaddr(tempstack)));
   endextend
endupdates

% SC Name expression
if ( = (status, "Compile-the-body") &
     (! isemptystack(tempstack)) &
     = (exprtype(valueofaddr(topaddr(tempstack))),
        "SCname"));
%then
   extend
      extenduniverse NODE;
   withupdates
      funcupdate graph(topaddr(tempstack)):=
         temp(NODE,1);
      funcupdate valueofaddr(topaddr(tempstack)):=
         emptyexpr;
      funcupdate nodetype(temp(NODE,1)):="SCName";
      funcupdate nodescname(temp(NODE,1)):=
         makescname(valueofaddr(topaddr(tempstack)));
   endextend
endupdates

% Prim Name expression
if ( = (status, "Compile-the-body") &
     (! isemptystack(tempstack)) &
     = (exprtype(valueofaddr(topaddr(tempstack))),
```

```
        "PrimName"));
%then
   extend
      extenduniverse NODE;
   withupdates
      funcupdate graph(topaddr(tempstack)):=
         temp(NODE,1);
      funcupdate valueofaddr(topaddr(tempstack)):=
         emptyexpr;
      funcupdate nodetype(temp(NODE,1)):="PRIMName";
      funcupdate nodeprimname(temp(NODE,1)):=
         makeprimname(valueofaddr(topaddr(tempstack)));
   endextend
endupdates

% Local variable name expression
if ( = (status, "Compile-the-body") &
     (! isemptystack(tempstack)) &
     = (exprtype(valueofaddr(topaddr(tempstack))),
        "VARname"));
%then
   extend
      extenduniverse NODE;
   withupdates
      funcupdate graph(topaddr(tempstack)):=
         temp(NODE,1);
      funcupdate valueofaddr(topaddr(tempstack)):=
         emptyexpr;
      funcupdate nodetype(temp(NODE,1)):="LVar";
      funcupdate nodevarname(temp(NODE,1)):=
         makevarname(valueofaddr(topaddr(tempstack)));
   endextend
endupdates

% Traverse up
if ( = (status, "Compile-the-body") &
     (! isemptystack(tempstack)) &
     = (exprtype(valueofaddr(topaddr(tempstack))),
        "EMPty"))
%then
   funcupdate tempstack:=popstack(tempstack);
endupdates;

% End of on supecombinator definition
if ( = (status, "Compile-the-body") &
     isemptystack(tempstack));
%then
   funcupdate status:="Get-curr-sc-def";
```

```
endupdates;

% Loads part two
loadalg "Graf-red-source/strict-sc-prim-red.src";
```

A.3.2 Specification of the Reduction Process

The specification to reduce the compiled graph extended with primitives is shown here:

```
% To be loaded into the evolving algebra interpret
% after supercomb-comp.src

% More signatures.
%
signature getoperator: (INSTR --> OPERATOR)
signature egraphinstr: INSTR
signature instrstack: (INSTR *)
signature makegcode: (INSTR --> (INSTR *))
signature addrstack: (TADDR *)
signature scdefaddr: TADDR
signature primdefaddr: TADDR
signature leftbranch: NUMBER
signature rightbranch: NUMBER
signature lengthas: ((TADDR *) --> NUMBER)
signature numberofparams: ((SCPARAM *) --> NUMBER)
signature rootofredex: TADDR
signature getsubstvalueaddr: (VARNAME --> TADDR)
signature getscparaminfo: ((NUMBER x (SCPARAM *)) --> SCPARAMTYPE)
signature getparamvar: ((NUMBER x (SCPARAM *)) --> VARNAME)
signature add: ((NUMBER x NUMBER) --> NUMBER)
signature subtract: ((NUMBER x NUMBER) --> NUMBER)
signature pointertodef: (TADDR --> TADDR)
signature finished:  (TADDR --> BOOL)
signature rootofinstance: TADDR
signature sameaddr: ((TADDR x TADDR) --> BOOL)
signature result: NUMBER

% The supercombinator object
signature scobj: SCOBJ
signature objscdefnode: (SCOBJ --> NODE)
signature currcounter: (SCOBJ --> NUMBER)
signature currarity: (SCOBJ --> NUMBER)
signature currparams: (SCOBJ --> (SCPARAM *))

% The primitive object
signature primobj: PRIMOBJ
signature objprimname: (PRIMOBJ --> SCPRIMNAME)
signature objprimdefnode: (PRIMOBJ --> NODE)
```

```
signature objprimcounter: (PRIMOBJ --> NUMBER)
signature objprimarglist: (PRIMOBJ --> ([NUMBER + BOOL + DATA + NODE] *))

% The arguments given to the primitive
signature emptyprimarglist: ([NUMBER + BOOL + DATA + NODE] *)
signature pusharg: ((([NUMBER + BOOL + DATA + NODE] *)
                     x [NUMBER + BOOL + DATA + NODE])
                    --> ([NUMBER + BOOL + DATA + NODE] *) )
signature matchnodeparam: ((NODETYPE x PARAMTYPE) --> BOOL)

% Compute the primitive
signature applyresultvalue: (RESULTTYPE x NODETYPE x
                        [NUMBER + BOOL + DATA + NODE])

signature applyprimitive: ((SCPRIMNAME
                x ([NUMBER + BOOL + DATA + NODE] *))
                --> (RESULTTYPE x NODETYPE x
                        [NUMBER + BOOL + DATA + NODE]))

signature resultvalueoftype: ((RESULTTYPE x NODETYPE x
                        [NUMBER + BOOL + DATA + NODE])
                        --> RESULTTYPE)

signature typeofnode: ((RESULTTYPE x NODETYPE x
                        [NUMBER + BOOL + DATA + NODE])
                        --> NODETYPE)

signature resultvalue: ((RESULTTYPE x NODETYPE x
                        [NUMBER + BOOL + DATA + NODE])
                        --> [NUMBER + BOOL + DATA + NODE])

% The parameter type (of the primitive)
signature paramtype: (NUMBER x (PARAMTYPE *)) --> PARAMTYPE

% Observing the paramenter type
signature isstrictparam: (PARAMTYPE --> BOOL)
signature isnotbasictype: (NODETYPE --> BOOL)
signature iswhnftype: (NODETYPE --> BOOL)

% The dump stack
signature dumpstack: (DUMP *)
signature emptydumpstack: ((DUMP *) --> BOOL)
signature pushdump: (((DUMP *) x DUMP) --> DUMP)
signature popdump: ((DUMP *) --> (DUMP *))
signature topdump: ((DUMP *) --> DUMP)
signature typeofdumpel: ((DUMP *) --> DUMPTYPE)

% Operations on dumpelement
```

```
signature primobjdump: (DUMP --> PRIMOBJ)
signature scobjdump: (DUMP --> SCOBJ)
signature addrstackdump: (DUMP --> (TADDR *))


% Procedures to be loaded
loadproc "Ea-system-lib/ea-std-user-misc.scm";

% More assignments
assignfunc egraphinstr, constant-std-lookup-data,
                        user-update-constant, std-const-dta;
assignfunc instrstack, constant-std-lookup-data,
                             user-update-constant, std-const-dta;
assignfunc addrstack, constant-std-lookup-data,
                       user-update-constant, std-const-dta;
assignfunc scdefaddr, constant-std-lookup-data,
                       user-update-constant, std-const-dta;
assignfunc primdefaddr, constant-std-lookup-data,
                       user-update-constant, std-const-dta;
assignfunc emptyprimarglist, constant-std-lookup-data,
                      user-update-constant, std-const-dta;
assignfunc pusharg, add-to-list, dummy-func, upd-not-perm;
assignfunc getscparaminfo, get-param-info, dummy-func, upd-not-perm;
assignfunc dumpstack, constant-std-lookup-data,
                     user-update-constant, std-const-dta;
assignfunc emptydumpstack, empty-list, dummy-func, upd-not-perm;
assignfunc pushdump, add-to-list, dummy-func, upd-not-perm;
assignfunc popdump, tail-from-list, dummy-func, upd-not-perm;
assignfunc topdump, first-from-list, dummy-func, upd-not-perm;
assignfunc primobjdump, table-std-lookup,
                        user-update-function, std-table;
assignfunc scobjdump, table-std-lookup,
                        user-update-function, std-table;
assignfunc typeofdumpel, table-std-lookup,
                        user-update-function, std-table;
assignfunc objscdefnode, table-std-lookup,
                        user-update-function, std-table;
assignfunc addrstackdump, table-std-lookup,
                        user-update-function, std-table;
assignfunc primobj, constant-std-lookup-data,
                       user-update-constant, std-const-dta;
assignfunc scobj, constant-std-lookup-data,
                      user-update-constant, std-const-dta;
assignfunc objprimname, table-std-lookup,
                        user-update-function, std-table;
assignfunc objprimdefnode, table-std-lookup,
                        user-update-function, std-table;
assignfunc objprimcounter, table-std-lookup,
```

```
                        user-update-function, std-table;
assignfunc objprimarglist, table-std-lookup,
                        user-update-function, std-table;
assignfunc paramtype, pick-elem-from-list, dummy-func, upd-not-perm;
assignfunc matchnodeparam, is-match-node-param-type, dummy-func, upd-not-perm;
assignfunc applyresultvalue, constant-std-lookup-data,
                       user-update-constant, std-const-dta;
assignfunc applyprimitive, apply-primitive, dummy-func, upd-not-perm;
assignfunc resultvalueoftype, get-result-type, dummy-func, upd-not-perm;
assignfunc typeofnode, get-type-of-node, dummy-func, upd-not-perm;
assignfunc resultvalue, get-result-value, dummy-func, upd-not-perm;
assignfunc isstrictparam, is-strict-param-type, dummy-func, upd-not-perm;
assignfunc isnotbasictype, is-not-basic-node-type, dummy-func, upd-not-perm;
assignfunc iswhnftype, is-whnf-node-type, dummy-func, upd-not-perm;
assignfunc leftbranch, constant-std-lookup-data,
                       user-update-constant, std-const-dta;
assignfunc rightbranch, constant-std-lookup-data,
                       user-update-constant, std-const-dta;
assignfunc currparams, table-std-lookup,
                       user-update-function, std-table;
assignfunc currarity,  table-std-lookup,
                        user-update-function, std-table;
assignfunc getoperator, table-std-lookup,
                        user-update-function, std-table;
assignfunc makegcode, make-list, dummy-func, upd-not-perm;
assignfunc lengthas, length-of-the-list, dummy-func, upd-not-perm;
assignfunc numberofparams, length-of-the-list, dummy-func, upd-not-perm;
assignfunc makegcode, make-list, dummy-func, upd-not-perm;
assignfunc rootofredex, constant-std-lookup-data,
                        user-update-constant, std-const-dta;
assignfunc rootofinstance, constant-std-lookup-data,
                        user-update-constant, std-const-dta;
assignfunc result, constant-std-lookup-data,
                        user-update-constant, std-const-dta;
assignfunc currcounter, table-std-lookup,
                        user-update-function, std-table;
assignfunc getsubstvalueaddr, table-std-lookup,
                        user-update-function, std-table;
assignfunc pointertodef, table-std-lookup,
                        user-update-function, std-table;
assignfunc finished, table-std-lookup,
                        user-update-function, std-table;
assignfunc getparamvar, get-param-name, dummy-func, upd-not-perm;
assignfunc add, add-numbers, dummy-func, upd-not-perm;
assignfunc subtract, subtract-numbers, dummy-func, upd-not-perm;
assignfunc sameaddr, compare-two-vars, dummy-func, upd-not-perm;
```

```
% Assign to the universe.
assignuniverse INSTR, std-ext-collection, newsymbol;
assignuniverse PRIMOBJ, std-ext-collection, newsymbol;
assignuniverse SCOBJ, std-ext-collection, newsymbol;
assignuniverse DUMP, std-ext-collection, newsymbol;

% Initial values
initial dumpstack:== [()]
initial addrstack:== [()]
initial emptyprimarglist:== [()]

if = (status, "Perform-graph-reds");
%then
   funcupdate leftbranch:=1;
   funcupdate rightbranch:=2;
   extend
      extenduniverse INSTR;
      extenduniverse TADDR;
      extenduniverse NODE;
   withupdates
      funcupdate getoperator(temp(INSTR,1)):="Egraph";
      funcupdate egraphinstr:=temp(INSTR,1);
      funcupdate instrstack:=makegcode(temp(INSTR,1));
      funcupdate graph(temp(TADDR,1)):=temp(NODE,1);
      funcupdate nodetype(temp(NODE,1)):="SCName";
      funcupdate nodescname(temp(NODE,1)):=mainscdefname;
      funcupdate addrstack:=pushstack(emptystack,temp(TADDR,1));
   endextend
   funcupdate status:="Unwind";
endupdates;

if ( = (status, "Unwind") &
     = (nodetype(graph(topaddr(addrstack))), "APnode"));
%then
   funcupdate addrstack:=pushstack(addrstack,nodechild(leftbranch,
                            graph(topaddr(addrstack))));
endupdates

% Not any more to do.
if ( = (status, "Unwind") &
     iswhnftype(nodetype(graph(topaddr(addrstack)))) &
     emptydumpstack(dumpstack) );
%then
   funcupdate result:=nodevalue(graph(topaddr(addrstack)))
   funcupdate status:="Weak-Head-Normal-form";
endupdates

% Restore from  the dump (primitive argument)
```

```
if ( = (status, "Unwind") &
     iswhnftype(nodetype(graph(topaddr(addrstack)))) &
     (! emptydumpstack(dumpstack)) &
     = (typeofdumpel(topdump(dumpstack)), "primobj"));
%then
     funcupdate addrstack:=addrstackdump(topdump(dumpstack));
     funcupdate primobj:=primobjdump(topdump(dumpstack));
     % Insert the evaluated expression instead of the graph
     funcupdate nodechild(rightbranch,graph(topaddr
                (popstack(addrstackdump(topdump(dumpstack))))):=
                topaddr(addrstack)
     funcupdate dumpstack:=popdump(dumpstack);
     funcupdate status:="Get-prim-args";
endupdates

% Restore the dump (substitution)
if ( = (status, "Unwind") &
     iswhnftype(nodetype(graph(topaddr(addrstack)))) &
     (! emptydumpstack(dumpstack)) &
     = (typeofdumpel(topdump(dumpstack)), "scobj"));
%then
     funcupdate addrstack:=addrstackdump(topdump(dumpstack));
     funcupdate scobj:=scobjdump(topdump(dumpstack));
     % Insert the evaluated expression instead of the graph
     funcupdate nodechild(rightbranch,graph(topaddr
                (popstack(addrstackdump(topdump(dumpstack))))):=
                topaddr(addrstack)
     funcupdate dumpstack:=popdump(dumpstack);
     funcupdate status:="Make-substs";
endupdates

if ( = (status, "Unwind") &
     = (nodetype(graph(topaddr(addrstack))), "PRIMName"));
%then
   funcupdate primdefaddr:=
         getaddrfromglobals(nodeprimname(graph(topaddr(addrstack))));
   funcupdate status:="Unwind-primname";
endupdates


if ( = (status, "Unwind-primname") &
     = (nodetype(graph(primdefaddr)), "Primitive") &
     > (lengthas(addrstack),nodearity(graph(primdefaddr))) )
%then
   extend
       extenduniverse PRIMOBJ;
   withupdates
       funcupdate objprimname(temp(PRIMOBJ,1)):=
```

```
                nodeprimname(graph(topaddr(addrstack)));
       funcupdate objprimdefnode(temp(PRIMOBJ,1)):=graph(primdefaddr);
       funcupdate objprimcounter(temp(PRIMOBJ,1)):=1;
       funcupdate primobj:=temp(PRIMOBJ,1);
       funcupdate objprimarglist(temp(PRIMOBJ,1)):=emptyprimarglist;
       funcupdate status:="Get-prim-args";
   endextend
endupdates

if ( = (status, "Get-prim-args") &
     <= (objprimcounter(primobj),nodearity(objprimdefnode(primobj))) &
     isstrictparam(paramtype(objprimcounter(primobj),
                  nodeparaminfo(objprimdefnode(primobj)))) &
     matchnodeparam(
         nodetype(graph(nodechild(rightbranch,
                                  graph(topaddr(popstack(addrstack)))))),
         paramtype(objprimcounter(primobj),
                 nodeparaminfo(objprimdefnode(primobj)))) );
%then
    funcupdate objprimarglist(primobj):=pusharg(objprimarglist(primobj),
               nodevalue(graph(nodechild(
                     rightbranch,graph(topaddr(popstack(addrstack)))))));
    funcupdate objprimcounter(primobj):=add(objprimcounter(primobj),1);
    funcupdate addrstack:=popstack(addrstack);
endupdates

if ( = (status, "Get-prim-args") &
     <= (objprimcounter(primobj),nodearity(objprimdefnode(primobj))) &
     (! isstrictparam(paramtype(objprimcounter(primobj),
                     nodeparaminfo(objprimdefnode(primobj))))) );
%then
    funcupdate objprimarglist(primobj):=pusharg(objprimarglist(primobj),
            graph(nodechild(rightbranch,graph(topaddr(popstack(addrstack))))));
    funcupdate objprimcounter(primobj):=
            add(objprimcounter(primobj),1);
    funcupdate addrstack:=popstack(addrstack);
endupdates


if ( = (status, "Get-prim-args") &
     <= (objprimcounter(primobj),nodearity(objprimdefnode(primobj))) &
     isstrictparam(paramtype(objprimcounter(primobj),
                    nodeparaminfo(objprimdefnode(primobj)))) &
     isnotbasictype(nodetype(graph(nodechild(rightbranch,
                       graph(topaddr(popstack(addrstack))))))) );
%then
     extend
        extenduniverse DUMP;
```

```
     withupdates
        funcupdate addrstackdump(temp(DUMP,1)):=addrstack;
        funcupdate primobjdump(temp(DUMP,1)):=primobj;
        funcupdate typeofdumpel(temp(DUMP,1)):="primobj";
        funcupdate dumpstack:=pushdump(dumpstack,temp(DUMP,1));
        funcupdate addrstack:=pushstack(emptystack,nodechild(rightbranch,
                        graph(topaddr(popstack(addrstack)))));
     endextend
     funcupdate status:="Unwind";
endupdates

if ( = (status, "Get-prim-args") &
     > (objprimcounter(primobj),nodearity(objprimdefnode(primobj))) );
%then
    funcupdate applyresultvalue:=applyprimitive(objprimname(primobj)
                    ,objprimarglist(primobj));
    funcupdate status:="Prim-result";
endupdates

if  ( = (status, "Prim-result") &
      = (resultvalueoftype(applyresultvalue), "Pass-node"));
%then
    funcupdate graph(topaddr(addrstack)):=resultvalue(applyresultvalue);
    funcupdate status:="Unwind";
endupdates

if ( = (status, "Prim-result") &
     = (resultvalueoftype(applyresultvalue), "Make-node" ));
%then
    extend
       extenduniverse NODE;
    withupdates
       funcupdate nodevalue(temp(NODE,1)):=resultvalue(applyresultvalue);
       funcupdate nodetype(temp(NODE,1)):=
                  typeofnode(applyresultvalue);
       funcupdate graph(topaddr(addrstack)):=temp(NODE,1);
    endextend
    funcupdate status:="Unwind";
endupdates

if ( = (status, "Unwind") &
     = (nodetype(graph(topaddr(addrstack))), "SCName"));
%then
   funcupdate scdefaddr:=
         getaddrfromglobals(nodescname(graph(topaddr(addrstack))));
   funcupdate status:="Unwind-scname";
endupdates
```

```
if ( = (status, "Unwind-scname") &
     = (nodetype(graph(scdefaddr)), "Supercomb") &
     > (lengthas(addrstack),numberofparams(nodeparams(graph(scdefaddr)))) )
%then
   extend
      extenduniverse SCOBJ;
   withupdates
      funcupdate objscdefnode(temp(SCOBJ,1)):=graph(scdefaddr);
      funcupdate currarity(temp(SCOBJ,1)):=
                 numberofparams(nodeparams(graph(scdefaddr)));
      funcupdate currcounter(temp(SCOBJ,1)):=0;
      funcupdate currparams(temp(SCOBJ,1)):=nodeparams(graph(scdefaddr));
      funcupdate scobj:=temp(SCOBJ,1);
   endextend
      funcupdate status:="Make-substs";
endupdates

% Extend the list of substitution pair for nonstrict parameters
if ( = (status, "Make-substs") &
     < (currcounter(scobj),currarity(scobj)) &
     = (getscparaminfo(add(currcounter(scobj),1),currparams(scobj)),
        "nonstrict") );
%then
   funcupdate getsubstvalueaddr(getparamvar
               (add(currcounter(scobj),1),currparams(scobj))):=
                    nodechild(rightbranch,graph(topaddr
                    (popstack(addrstack))));
   funcupdate currcounter(scobj):=add(currcounter(scobj),1);
   funcupdate addrstack:=popstack(addrstack);
endupdates

if ( = (status, "Make-substs") &
     < (currcounter(scobj),currarity(scobj)) &
     = (getscparaminfo(add(currcounter(scobj),1),currparams(scobj)),
        "strict") &
     iswhnftype(nodetype(graph(nodechild(rightbranch,
                graph(topaddr(popstack(addrstack)))))))  );
%then
   funcupdate getsubstvalueaddr(getparamvar
               (add(currcounter(scobj),1),currparams(scobj))):=
                    nodechild(rightbranch,graph(topaddr
                    (popstack(addrstack))));
   funcupdate currcounter(scobj):=add(currcounter(scobj),1);
   funcupdate addrstack:=popstack(addrstack);
endupdates

if ( = (status, "Make-substs") &
     < (currcounter(scobj),currarity(scobj)) &
```

```
     = (getscparaminfo(add(currcounter(scobj),1),currparams(scobj)),
        "strict") &
     isnotbasictype(nodetype(graph(nodechild(rightbranch,
                       graph(topaddr(popstack(addrstack)))))))  );
%then
     extend
        extenduniverse DUMP;
     withupdates
        funcupdate addrstackdump(temp(DUMP,1)):=addrstack;
        funcupdate scobjdump(temp(DUMP,1)):=scobj;
        funcupdate typeofdumpel(temp(DUMP,1)):="scobj";
        funcupdate dumpstack:=pushdump(dumpstack,temp(DUMP,1));
        funcupdate addrstack:=pushstack(emptystack,nodechild(rightbranch,
                       graph(topaddr(popstack(addrstack)))));
     endextend
     funcupdate status:="Unwind";
endupdates

% Finished with the substitutions
% Retain the last element of address stack until the root of redex constant
% is set.
if ( = (status, "Make-substs") &
     = (currcounter(scobj),currarity(scobj)));
%then
   funcupdate rootofredex:=topaddr(addrstack);
   funcupdate status:="Init-instance";
endupdates

if = (status, "Init-instance");
%then
   extend
     extenduniverse TADDR;
   withupdates
     funcupdate pointertodef(temp(TADDR,1)):=
                  nodechild(1,objscdefnode(scobj));
     funcupdate addrstack:=pushstack(addrstack,temp(TADDR,1));
     funcupdate rootofinstance:=temp(TADDR,1)
     funcupdate finished(temp(TADDR,1)):="False";
   endextend
   funcupdate status:="Build-instance";
endupdates

if ( = (status, "Build-instance") &
     =/= (finished(topaddr(addrstack)),"True") &
     = (nodetype(graph(pointertodef(topaddr(addrstack)))),"APnode"));
%then
   funcupdate finished(topaddr(addrstack)):="True";
   extend
```

```
      extenduniverse TADDR # 2;
      extenduniverse NODE;
   withupdates
      funcupdate graph(topaddr(addrstack)):=temp(NODE,1);
      funcupdate nodetype(temp(NODE,1)):=
         nodetype(graph(pointertodef(topaddr(addrstack))));
      funcupdate nodechild(leftbranch,temp(NODE,1)):=temp(TADDR,1);
      funcupdate nodechild(rightbranch,temp(NODE,1)):=temp(TADDR,2);
      funcupdate finished(temp(TADDR,1)):="False";
      funcupdate finished(temp(TADDR,2)):="False";
      funcupdate pointertodef(temp(TADDR,1)):=
         nodechild(leftbranch,graph(pointertodef(topaddr(addrstack))));
      funcupdate pointertodef(temp(TADDR,2)):=
         nodechild(rightbranch,graph(pointertodef(topaddr(addrstack))));
      funcupdate addrstack:=pushstack(
                     pushstack(addrstack,temp(TADDR,1)),temp(TADDR,2));
   endextend
endupdates

if ( = (status, "Build-instance") &
     =/= (finished(topaddr(addrstack)),"True") &
     = (nodetype(graph(pointertodef(topaddr(addrstack)))),"Num"));
%then
   funcupdate finished(topaddr(addrstack)):="True";
   extend
      extenduniverse NODE;
   withupdates
      funcupdate graph(topaddr(addrstack)):=temp(NODE,1);
      funcupdate nodetype(temp(NODE,1)):=
         nodetype(graph(pointertodef(topaddr(addrstack))));
      funcupdate nodevalue(temp(NODE,1)):=
         nodevalue(graph(pointertodef(topaddr(addrstack))));
   endextend
endupdates

if ( = (status, "Build-instance") &
     =/= (finished(topaddr(addrstack)),"True") &
     = (nodetype(graph(pointertodef(topaddr(addrstack)))),"Bool"));
%then
   funcupdate finished(topaddr(addrstack)):="True";
   extend
      extenduniverse NODE;
   withupdates
      funcupdate graph(topaddr(addrstack)):=temp(NODE,1);
      funcupdate nodetype(temp(NODE,1)):=
         nodetype(graph(pointertodef(topaddr(addrstack))));
      funcupdate nodevalue(temp(NODE,1)):=
         nodevalue(graph(pointertodef(topaddr(addrstack))));
```

```
   endextend
endupdates
if ( = (status, "Build-instance") &
     =/= (finished(topaddr(addrstack)),"True") &
     = (nodetype(graph(pointertodef(topaddr(addrstack)))),"Data"));
%then
   funcupdate finished(topaddr(addrstack)):="True";
   extend
      extenduniverse NODE;
   withupdates
      funcupdate graph(topaddr(addrstack)):=temp(NODE,1);
      funcupdate nodetype(temp(NODE,1)):=
         nodetype(graph(pointertodef(topaddr(addrstack))));
      funcupdate nodevalue(temp(NODE,1)):=
         nodevalue(graph(pointertodef(topaddr(addrstack))));
   endextend
endupdates


if ( = (status, "Build-instance") &
     =/= (finished(topaddr(addrstack)),"True") &
     = (nodetype(graph(pointertodef(topaddr(addrstack)))),"SCName"));
%then
   funcupdate finished(topaddr(addrstack)):="True";
   extend
      extenduniverse NODE;
   withupdates
      funcupdate graph(topaddr(addrstack)):=temp(NODE,1);
      funcupdate nodetype(temp(NODE,1)):=
         nodetype(graph(pointertodef(topaddr(addrstack))));
      funcupdate nodescname(temp(NODE,1)):=
         nodescname(graph(pointertodef(topaddr(addrstack))));
   endextend
endupdates

if ( = (status, "Build-instance") &
     =/= (finished(topaddr(addrstack)),"True") &
     = (nodetype(graph(pointertodef(topaddr(addrstack)))),"PRIMName"));
%then
   funcupdate finished(topaddr(addrstack)):="True";
   extend
      extenduniverse NODE;
   withupdates
      funcupdate graph(topaddr(addrstack)):=temp(NODE,1);
      funcupdate nodetype(temp(NODE,1)):=
         nodetype(graph(pointertodef(topaddr(addrstack))));
      funcupdate nodeprimname(temp(NODE,1)):=
         nodeprimname(graph(pointertodef(topaddr(addrstack))));
```

```
   endextend
endupdates

% Sharing distinct variables.
if ( = (status, "Build-instance") &
     =/= (finished(topaddr(addrstack)),"True") &
     = (nodetype(graph(pointertodef(topaddr(addrstack)))),"LVar"));
%then
   funcupdate finished(topaddr(addrstack)):="True";
   funcupdate graph(topaddr(addrstack)):=
      graph(getsubstvalueaddr(nodevarname(graph
          (pointertodef(topaddr(addrstack))))));
endupdates

if ( = (status, "Build-instance") &
     = (finished(topaddr(addrstack)),"True") &
     (! sameaddr(topaddr(addrstack),rootofinstance)));
%then
   funcupdate addrstack:=popstack(addrstack);
endupdates

if ( = (status, "Build-instance") &
     = (finished(topaddr(addrstack)),"True") &
     sameaddr(topaddr(addrstack),rootofinstance));
%then
   funcupdate status:="Update";
endupdates

% Make a copy of the result of the reduction
if = (status, "Update");
%then
   funcupdate addrstack:=
       pushstack(popstack(popstack(addrstack)),rootofinstance);
   funcupdate status:="Unwind";
endupdates

% Update the root of the redex
% *** Later on.
```

A.4 The G-machine Specification

A.4.1 Specification of the Compilation

The specification to compile the supercombinator definitions extended with strict arguments and primitives into G-machine instructions is shown below:

```
% resets before loading
reset
```

```
% Signatures
signature status : STATUS
signature add: ((NUMBER x NUMBER) --> NUMBER)

signature tempstack: (TADDR *)
signature emptystack: (TADDR *)
signature initialstack: (TADDR *)
signature isemptystack: ((TADDR *) --> BOOL)
signature topaddr: ((TADDR *) --> TADDR)
signature pushstack: (((TADDR *) x TADDR) --> (TADDR *))
signature popstack: ((TADDR *) --> (TADDR *))

signature currscdefaddr: TADDR
signature valueofaddr: (TADDR --> [SCEXPR + INSTR])
signature graph: (TADDR --> NODE)

signature mainscdefname: SCNAME
signature getmainname: ((SCEXPR *) --> SCNAME)
signature allscdefs: (SCEXPR *)
signature isemptyscdefs: ((SCEXPR *) --> BOOL)
signature getnextscdef: ((SCEXPR*) --> SCEXPR)
signature tailscdefs: ((SCEXPR*) --> (SCEXPR*))

signature currprimdefaddr: TADDR
signature allprimdefs: (PRIMEXPR *)
signature isemptyprimdefs: (PRIMEXPR --> BOOL)
signature getnextprimdef: ((PRIMEXPR *) --> PRIMEXPR)
signature tailprimdefs: ((PRIMEXPR *) --> (PRIMEXPR*))

% Primitive expressions
signature getprimdefname: (PRIMEXPR --> PRIMNAME)
signature makeprimparamposinfolist: (PRIMEXPR --> (PPOSINFO *))
signature makeprimarity: (PRIMEXPR --> NUMBER)

% Primitive informations
signature primexprtype: (TADDR --> PRIMEXPRTYPE)
signature nameofprimitive: (TADDR --> PRIMNAME)
signature getprimposparaminfolist: (TADDR --> (PPOSINFO *))
signature getprimposition: (((PPOSINFO *) x NUMBER) --> NUMBER)
signature primposition: (TADDR --> NUMBER)
signature getpriminfoparam: (((PPOSINFO *) x NUMBER) --> PPINFO)
signature priminfoparam: (TADDR --> PPINFO)
signature secondprimpos: (TADDR --> NUMBER)
signature thirdprimpos: (TADDR --> NUMBER)
signature defprimarity: (TADDR --> NUMBER)

signature makeprimname: (SCEXPR --> PRIMNAME)
```

```
signature getnamefromglobals: (TADDR --> SCNAME)
signature getaddrfromglobals: (SCNAME --> TADDR)

signature emptyexpr: SCEXPR
signature currscdef: SCEXPR
signature exprtype: (SCEXPR --> SCTYPE)
signature makeparams: (SCEXPR --> (SCEXPR *))
signature getscdefname: (SCEXPR --> SCNAME)
signature srcbody: (SCEXPR --> SCEXPR)
signature firstappexpr: (SCEXPR --> SCEXPR)
signature secondappexpr: (SCEXPR --> SCEXPR)
signature makenum: (SCEXPR --> NUMBER)
signature makebool: (SCEXPR --> BOOLDATA)
signature makedata: (SCEXPR --> DATA)
signature makescname: (SCEXPR --> SCNAME)
signature makevarname: (SCEXPR --> VARNAME)

signature numberofparams: ((SCEXPR *) --> NUMBER)
signature getparamlist: (TADDR --> (PARAMPOS *))
signature makeparamposlist: (SCEXPR --> (PARAMPOS *))
signature incrementposlist: ((NUMBER x (PARAMPOS *)) --> (PARAMPOS *))
signature getposition: (((PARAMPOS *) x VARNAME) --> NUMBER)

% Info about which sc parameters is strict
signature getparaminfo: (((PARAMPOS *) x VARNAME) --> SCPINFO)

signature getoperator: (INSTR --> OPERATOR)
signature getoperand: ((NUMBER x INSTR) --> OPERAND)

signature codelist: (INSTR *)
signature instrstack: (INSTR *)
signature emptycodelist: (INSTR *)
signature makegcode: (INSTR --> (INSTR *))
signature makegcodetwo: ((INSTR x INSTR) --> (INSTR *))
signature concatcode: (((INSTR *) x (INSTR *)) --> (INSTR *))

signature instructions: (TADDR --> (INSTR *))
signature hascode: (TADDR --> BOOL)

signature nodetype: (NODE --> NTYPE)
signature defarity: (NODE --> NUMBER)
signature finishedcode: (NODE --> (INSTR *))
signature leftbranch: NUMBER
signature rightbranch: NUMBER

% Load procedure files
% Load standard user environment procedures from file!
```

```
%
loadproc "Ea-system-lib/ea-std-user-extension.scm";
loadproc "Ea-system-lib/ea-std-user-update.scm";
loadproc "Ea-system-lib/ea-std-user-lookup.scm";

% Load procedures maintaining lists
loadproc "Ea-system-lib/ea-std-user-list.scm";

% Load arithmetic procedures
loadproc "Ea-system-lib/ea-std-user-arithmetic.scm";

% Load graphical reduction procedure.
loadproc "Graf-reduksjon-lib/ea-graf-red-app-expr.scm";
loadproc "Graf-reduksjon-lib/ea-graf-red-find-type.scm";
loadproc "Graf-reduksjon-lib/ea-graf-red-number.scm";
loadproc "Graf-reduksjon-lib/ea-graf-red-scdef.scm";
loadproc "Graf-reduksjon-lib/ea-graf-red-scname.scm";
loadproc "Graf-reduksjon-lib/ea-graf-red-variable.scm";
loadproc "Graf-reduksjon-lib/ea-gcode-make-parlist.scm";
loadproc "Graf-reduksjon-lib/ea-prim-def.scm";

% Assignments
% assign <func-symb>, <lookup-proc>, <upd-proc>, <fmess-symb>;

% Status
assignfunc status, constant-std-lookup-data,
                    user-update-constant, std-const-dta;
% Addition
assignfunc add, add-numbers, dummy-func, upd-not-perm;
% Pointer to the graph
assignfunc graph, table-std-lookup,
                  user-update-function, std-table;
assignfunc valueofaddr, table-std-lookup,
                  user-update-function, std-table;
% The address stack
assignfunc currscdefaddr, constant-std-lookup-data,
                    user-update-constant, std-const-dta;
assignfunc tempstack, constant-std-lookup-data,
                    user-update-constant, std-const-dta;
assignfunc emptystack, constant-std-lookup-data,
                    user-update-constant, std-const-dta;
assignfunc initialstack, constant-std-lookup-data,
                    user-update-constant, std-const-dta;
assignfunc isemptystack, empty-list, dummy-func, upd-not-perm;
assignfunc topaddr, first-from-list, dummy-func, upd-not-perm;
assignfunc pushstack, add-to-list, dummy-func, upd-not-perm;
assignfunc popstack, tail-from-list, dummy-func, upd-not-perm;
```

```
% The sc definitions
assignfunc mainscdefname, constant-std-lookup-data,
                     user-update-constant, std-const-dta;
assignfunc getmainname, get-scdef-main-name,
                     dummy-func, upd-not-perm;
assignfunc allscdefs, constant-std-lookup-data,
                     user-update-constant, std-const-dta;
assignfunc isemptyscdefs, empty-list, dummy-func, upd-not-perm;
assignfunc getnextscdef, first-from-list, dummy-func, upd-not-perm;
assignfunc tailscdefs, tail-from-list, dummy-func, upd-not-perm;

% The globals
assignfunc getnamefromglobals, table-std-lookup,
             user-update-function, std-table;

assignfunc getaddrfromglobals, table-std-lookup,
             user-update-function, std-table;

% The sc expression
assignfunc emptyexpr, constant-std-lookup-data,
                    user-update-constant, std-const-dta;
assignfunc currscdef, constant-std-lookup-data,
                    user-update-constant, std-const-dta;
assignfunc getscdefname, get-scdef-name, dummy-func, upd-not-perm;
assignfunc exprtype, find-type-gen, dummy-func, upd-not-perm;
assignfunc makeparams, get-params, dummy-func, upd-not-perm;
assignfunc srcbody, get-scbody-expr, dummy-func, upd-not-perm;
assignfunc firstappexpr, get-first-app-expr,
                    dummy-func, upd-not-perm;
assignfunc secondappexpr, get-second-app-expr,
                    dummy-func, upd-not-perm;
assignfunc makenum, get-sc-number, dummy-func, upd-not-perm;
assignfunc makebool, get-sc-boolean, dummy-func, upd-not-perm;
assignfunc makedata, get-sc-data, dummy-func, upd-not-perm;
assignfunc makescname, get-sc-name, dummy-func, upd-not-perm;
assignfunc makevarname, get-sc-var, dummy-func, upd-not-perm;

% The primitive definitions
assignfunc currprimdefaddr, constant-std-lookup-data,
                    user-update-constant, std-const-dta;
assignfunc allprimdefs, constant-std-lookup-data,
                     user-update-constant, std-const-dta;
assignfunc isemptyprimdefs, empty-list, dummy-func, upd-not-perm;
assignfunc getnextprimdef, first-from-list, dummy-func, upd-not-perm;
assignfunc tailprimdefs, tail-from-list, dummy-func, upd-not-perm;

% Operations on primitives
assignfunc primexprtype, table-std-lookup,
```

```
                    user-update-function, std-table;
assignfunc nameofprimitive, table-std-lookup,
                    user-update-function, std-table;
assignfunc getprimposparaminfolist, table-std-lookup,
                    user-update-function, std-table;
assignfunc primposition, table-std-lookup,
                    user-update-function, std-table;
assignfunc priminfoparam, table-std-lookup,
                    user-update-function, std-table;
assignfunc secondprimpos, table-std-lookup,
                    user-update-function, std-table;
assignfunc thirdprimpos, table-std-lookup,
                    user-update-function, std-table;
assignfunc defprimarity, table-std-lookup,
                    user-update-function, std-table;
assignfunc primexprtype, table-std-lookup,
                    user-update-function, std-table;

assignfunc getprimdefname, get-primdef-name,
                dummy-func, upd-not-perm;
assignfunc makeprimparamposinfolist, make-prim-param-pos-info-list,
                dummy-func, upd-not-perm;
assignfunc makeprimarity, get-prim-arity, dummy-func, upd-not-perm;
assignfunc getprimposition, get-prim-param-position,
                dummy-func, upd-not-perm;
assignfunc getpriminfoparam, get-prim-param-info,
                dummy-func, upd-not-perm;

% Primitive name
assignfunc makeprimname, get-prim-name,
                dummy-func, upd-not-perm;

% Parameter-list
assignfunc numberofparams, length-of-the-list, dummy-func, upd-not-perm;
assignfunc getparamlist, table-std-lookup,
                    user-update-function, std-table;
assignfunc getparaminfo, get-param-sc-info, dummy-func, upd-not-perm;

% * New procedures to be made.
assignfunc makeparamposlist, make-param-pos-list, dummy-func, upd-not-perm;
assignfunc incrementposlist, increm-param-pos, dummy-func, upd-not-perm;
assignfunc getposition, get-position, dummy-func, upd-not-perm;

% Contents of instructions
assignfunc getoperator, table-std-lookup,
                    user-update-function, std-table;
assignfunc getoperand, table-std-lookup,
                    user-update-function, std-table;
```

```
% The instructions
assignfunc codelist, constant-std-lookup-data,
                    user-update-constant, std-const-dta;
assignfunc instrstack, constant-std-lookup-data,
                    user-update-constant, std-const-dta;
assignfunc emptycodelist, constant-std-lookup-data,
                    user-update-constant, std-const-dta;
assignfunc makegcode, make-list, dummy-func, upd-not-perm;
assignfunc makegcodetwo, make-list, dummy-func, upd-not-perm;
assignfunc concatcode, concat-lists, dummy-func, upd-not-perm;
assignfunc instructions, table-std-lookup,
                    user-update-function, std-table;
assignfunc hascode, table-std-lookup,
                    user-update-function, std-table;

% The graph
assignfunc nodetype, table-std-lookup,
                    user-update-function, std-table;
assignfunc defarity, table-std-lookup,
                    user-update-function, std-table;
assignfunc finishedcode, table-std-lookup,
                    user-update-function, std-table;
assignfunc leftbranch, constant-std-lookup-data,
                    user-update-constant, std-const-dta;
assignfunc rightbranch, constant-std-lookup-data,
                    user-update-constant, std-const-dta;

% Assign to universe
% assign <universe-symbol> <universe-ext-proc> <umess-symbol>;
assignuniverse NODE, std-ext-collection, number;
assignuniverse TADDR, std-ext-collection, newsymbol;
assignuniverse INSTR, std-ext-collection, newsymbol;

% Initial values
loadalg "Graf-red-source/PRIM-SC/prim-test-false.src";
%initial allscdefs :==
%    [((= ((sc "SEL") ()) ((((prim "if") (bool "true")) (sc "CP")) (num 4)))
%      (= ((sc "CP") ()) (((prim "plus") (num 1)) (num 2)))
%     )]

% Primitive definitions unlikely to change. More may be added.
initial allprimdefs :==
    [((primitive (prim "plus") (num 2) (number number))
      (primitive (prim "if") (num 3) (boolean nonstrict nonstrict))
       )]

initial emptyexpr :== [ (empty) ]
```

```
%initial status :== "Get-curr-sc-def"
initial status :== "Initial"
initial emptystack :== [()]
initial codelist :== [()]
initial emptycodelist :== [()]
initial currscdefaddr :== [start]

%initial tempstack :== [()]


% Start the compilation
if    ( = (status, "Initial") &
        (! isemptyscdefs(allscdefs)) );
%then
   funcupdate mainscdefname:=getmainname(allscdefs)
   funcupdate status:="Get-curr-sc-def"
endupdates

% Prepare for compilation of the supercombinator definition
if    ( = (status, "Get-curr-sc-def") &
        (! isemptyscdefs(allscdefs)) );
%then
   extend
      extenduniverse TADDR;
   withupdates
      funcupdate currscdefaddr:=temp(TADDR,1);
      funcupdate getnamefromglobals(temp(TADDR,1)):=
           getscdefname(getnextscdef(allscdefs));
      funcupdate getaddrfromglobals(getscdefname(getnextscdef(allscdefs))):=
           temp(TADDR,1);
      funcupdate valueofaddr(temp(TADDR,1)):=getnextscdef(allscdefs);
   endextend
   funcupdate allscdefs:=tailscdefs(allscdefs);
   funcupdate status:="Compile-sc-def";
endupdates

if ( = (status, "Get-curr-sc-def") &
     isemptyscdefs(allscdefs) );
%then
   funcupdate status:="Get-curr-prim-def";
endupdates

% Prepare for compilation of the primitive definition
if    ( = (status, "Get-curr-prim-def") &
        (! isemptyprimdefs(allprimdefs)) );
%then
   extend
      extenduniverse TADDR;
```

```
   withupdates
      funcupdate currprimdefaddr:=temp(TADDR,1);
      funcupdate getnamefromglobals(temp(TADDR,1)):=
           getprimdefname(getnextprimdef(allprimdefs));
      funcupdate getaddrfromglobals(getprimdefname
          (getnextprimdef(allprimdefs))):=temp(TADDR,1);
      funcupdate valueofaddr(temp(TADDR,1)):=getnextprimdef(allprimdefs);
   endextend
   funcupdate allprimdefs:=tailprimdefs(allprimdefs);
   funcupdate status:="Compile-prim-def";
endupdates

% Primitive definition
if = (status, "Compile-prim-def");
%then
   extend
      extenduniverse TADDR # 2;
      extenduniverse NODE;
      extenduniverse INSTR # 2;
   withupdates
      funcupdate defarity(temp(NODE,1)):=
                 makeprimarity(valueofaddr(currprimdefaddr))
      funcupdate graph(currprimdefaddr):=temp(NODE,1);
      funcupdate nodetype(temp(NODE,1)):="Global";
      funcupdate primexprtype(temp(TADDR,2)):="NameParams";
      funcupdate nameofprimitive(temp(TADDR,2)):=
                 getnamefromglobals(currprimdefaddr));
      funcupdate getprimposparaminfolist(temp(TADDR,2)):=
                    makeprimparamposinfolist(valueofaddr(currprimdefaddr));
      funcupdate defprimarity(temp(TADDR,2)):=
                    makeprimarity(valueofaddr(currprimdefaddr));
      funcupdate hascode(temp(TADDR,2)):="False";
% Slide n + 1, Unwind
      funcupdate getoperator(temp(INSTR,1)):="Slide";
      funcupdate getoperand(1,temp(INSTR,1)):=
                 add(1,makeprimarity(valueofaddr(currprimdefaddr))));
      funcupdate getoperator(temp(INSTR,2)):="Unwind";
      funcupdate instructions(temp(TADDR,1)):=
                 makegcodetwo(temp(INSTR,1),temp(INSTR,2));

      funcupdate hascode(temp(TADDR,1)):="True";
      funcupdate tempstack:=pushstack(
                     pushstack(emptystack,temp(TADDR,1)),
                     temp(TADDR,2));
   endextend
   funcupdate status:="Compile-part-of-primdef";
endupdates
```

```
% Primitive definition
if ( = (status, "Compile-part-of-primdef") &
     (! isemptystack(tempstack)) &
     = (hascode(topaddr(tempstack)),"False") &
     = (primexprtype(topaddr(tempstack)),"NameParams") &
     = (defprimarity(topaddr(tempstack)),1))
%then
   extend
      extenduniverse TADDR # 2;
   withupdates
      % First parameter
      funcupdate primexprtype(temp(TADDR,2)):="Param";
      funcupdate primposition(temp(TADDR,2)):=
                 getprimposition(getprimposparaminfolist
                         (topaddr(tempstack)),1)
      funcupdate priminfoparam(temp(TADDR,2)):=
                 getpriminfoparam(getprimposparaminfolist
                         (topaddr(tempstack)),1)
      funcupdate hascode(temp(TADDR,2)):="False";
      % The build in function
      funcupdate primexprtype(temp(TADDR,1)):="Priminstr";
      funcupdate nameofprimitive(temp(TADDR,1)):=
                     nameofprimitive(topaddr(tempstack));
      funcupdate hascode(temp(TADDR,1)):="False";
      % Put adresses on the address stack
      funcupdate tempstack:=pushstack(pushstack
                 (popstack(tempstack),temp(TADDR,1)),
                                      temp(TADDR,2));
   endextend
endupdates

if ( = (status, "Compile-part-of-primdef") &
     (! isemptystack(tempstack)) &
     = (hascode(topaddr(tempstack)),"False") &
     = (primexprtype(topaddr(tempstack)),"NameParams") &
     = (defprimarity(topaddr(tempstack)),2))
%then
   extend
      extenduniverse TADDR # 3;
   withupdates
      % Second parameter
      funcupdate primexprtype(temp(TADDR,3)):="Param";
      funcupdate primposition(temp(TADDR,3)):=
                 getprimposition(getprimposparaminfolist
                         (topaddr(tempstack)),2)
      funcupdate priminfoparam(temp(TADDR,3)):=
                 getpriminfoparam(getprimposparaminfolist
                         (topaddr(tempstack)),2)
```

```
      funcupdate hascode(temp(TADDR,3)):="False";
      % First parameter
      funcupdate primexprtype(temp(TADDR,2)):="Param";
      funcupdate primposition(temp(TADDR,2)):=
                 add(getprimposition(getprimposparaminfolist
                           (topaddr(tempstack)),1),1);
      funcupdate priminfoparam(temp(TADDR,2)):=
                 getpriminfoparam(getprimposparaminfolist
                             (topaddr(tempstack)),1)
      funcupdate hascode(temp(TADDR,2)):="False";
      % The build in function
      funcupdate primexprtype(temp(TADDR,1)):="Priminstr"
      funcupdate nameofprimitive(temp(TADDR,1)):=
                     nameofprimitive(topaddr(tempstack));
      funcupdate hascode(temp(TADDR,1)):="False";
      % Put adresses on the address stack
      funcupdate tempstack:=pushstack(pushstack(
                     pushstack(popstack(tempstack),temp(TADDR,1)),
                          temp(TADDR,2)),temp(TADDR,3));
   endextend
endupdates

if ( = (status, "Compile-part-of-primdef") &
     (! isemptystack(tempstack)) &
     = (hascode(topaddr(tempstack)),"False") &
     = (primexprtype(topaddr(tempstack)),"NameParams") &
     = (defprimarity(topaddr(tempstack)),3) &
     = (nameofprimitive(topaddr(tempstack)),"if") );
%then
   extend
      extenduniverse TADDR # 2;
   withupdates
      % First parameter
      funcupdate primexprtype(temp(TADDR,2)):="Param";
      funcupdate primposition(temp(TADDR,2)):=
                 getprimposition(getprimposparaminfolist
                         (topaddr(tempstack)),1)
      funcupdate priminfoparam(temp(TADDR,2)):=
                 getpriminfoparam(getprimposparaminfolist
                         (topaddr(tempstack)),1)
      funcupdate hascode(temp(TADDR,2)):="False";
      % The build in function
      funcupdate primexprtype(temp(TADDR,1)):="Priminstr"
      funcupdate nameofprimitive(temp(TADDR,1)):=
                     nameofprimitive(topaddr(tempstack))
      funcupdate secondprimpos(temp(TADDR,1)):=
                 getprimposition(getprimposparaminfolist
                         (topaddr(tempstack)),2)
```

```
      funcupdate thirdprimpos(temp(TADDR,1)):=
                 getprimposition(getprimposparaminfolist
                         (topaddr(tempstack)),3)
      funcupdate hascode(temp(TADDR,1)):="False";
      % Put adresses on the address stack
      funcupdate tempstack:=pushstack(pushstack
                    (popstack(tempstack),temp(TADDR,1)),
                                        temp(TADDR,2));
   endextend
endupdates

if ( = (status, "Compile-part-of-primdef") &
     (! isemptystack(tempstack)) &
     = (hascode(topaddr(tempstack)),"False") &
     = (primexprtype(topaddr(tempstack)),"Param") &
     = (priminfoparam(topaddr(tempstack)),"strict"));
%then
   extend
      extenduniverse INSTR # 2;
   withupdates
      funcupdate getoperator(temp(INSTR,1)):="Push";
      funcupdate getoperand(1,temp(INSTR,1)):=
                     primposition(topaddr(tempstack));
      funcupdate getoperator(temp(INSTR,2)):="Eval";
      funcupdate instructions(topaddr(tempstack)):=
                 makegcodetwo(temp(INSTR,1),temp(INSTR,2));
      funcupdate hascode(topaddr(tempstack)):="True";
   endextend
endupdates

if ( = (status, "Compile-part-of-primdef") &
     (! isemptystack(tempstack)) &
     = (hascode(topaddr(tempstack)),"False") &
     = (primexprtype(topaddr(tempstack)),"Param") &
     = (priminfoparam(topaddr(tempstack)),"nonstrict"));
%then
   extend
      extenduniverse INSTR;
   withupdates
      funcupdate getoperator(temp(INSTR,1)):="Push";
      funcupdate getoperand(1,temp(INSTR,1)):=
                     primposition(topaddr(tempstack));
      funcupdate instructions(topaddr(tempstack)):=
                 makegcode(temp(INSTR,1));
      funcupdate hascode(topaddr(tempstack)):="True";
   endextend
endupdates
```

```
% Addition
if ( = (status, "Compile-part-of-primdef") &
     (! isemptystack(tempstack)) &
     = (hascode(topaddr(tempstack)),"False") &
     = (primexprtype(topaddr(tempstack)),"Priminstr") &
     = (nameofprimitive(topaddr(tempstack)),"plus"));
%then
   extend
      extenduniverse INSTR;
   withupdates
      funcupdate getoperator(temp(INSTR,1)):="Add";
      funcupdate instructions(topaddr(tempstack)):=
                 makegcode(temp(INSTR,1));
      funcupdate hascode(topaddr(tempstack)):="True";
   endextend
endupdates

% Negation
if ( = (status, "Compile-part-of-primdef") &
     (! isemptystack(tempstack)) &
     = (hascode(topaddr(tempstack)),"False") &
     = (primexprtype(topaddr(tempstack)),"Priminstr") &
     = (nameofprimitive(topaddr(tempstack)),"negate"));
%then
   extend
      extenduniverse INSTR;
   withupdates
      funcupdate getoperator(temp(INSTR,1)):="Neg";
      funcupdate instructions(topaddr(tempstack)):=
                 makegcode(temp(INSTR,1));
      funcupdate hascode(topaddr(tempstack)):="True";
   endextend
endupdates

% Cond primitive
if ( = (status, "Compile-part-of-primdef") &
     (! isemptystack(tempstack)) &
     = (hascode(topaddr(tempstack)),"False") &
     = (primexprtype(topaddr(tempstack)),"Priminstr") &
     = (nameofprimitive(topaddr(tempstack)),"if"));
%then
   extend
      extenduniverse INSTR # 3;
   withupdates
      funcupdate getoperator(temp(INSTR,1)):="Push";
      funcupdate getoperand(1,temp(INSTR,1)):=
                   secondprimpos(topaddr(tempstack));
      funcupdate getoperator(temp(INSTR,2)):="Push";
```

```
      funcupdate getoperand(1,temp(INSTR,2)):=
                   thirdprimpos(topaddr(tempstack));
      funcupdate getoperator(temp(INSTR,3)):="Cond";
      funcupdate getoperand(1,temp(INSTR,3)):=
                      makegcode(temp(INSTR,1));
      funcupdate getoperand(2,temp(INSTR,3)):=
                      makegcode(temp(INSTR,2));
      funcupdate instructions(topaddr(tempstack)):=
                 makegcode(temp(INSTR,3));
      funcupdate hascode(topaddr(tempstack)):="True";
   endextend
endupdates

% Traverse up
if ( = (status, "Compile-part-of-primdef") &
     (! isemptystack(tempstack)) &
     = (hascode(topaddr(tempstack)),"True"));
%then
   % The instruction is always appendended at the end of
   % the instruction list.
   funcupdate codelist:=concatcode(codelist,
                             instructions(topaddr(tempstack)));
   funcupdate tempstack:=popstack(tempstack);
endupdates;

% End of on supecombinator definition
if ( = (status, "Compile-part-of-primdef") &
     isemptystack(tempstack));
%then
   funcupdate status:="Get-curr-prim-def";
   funcupdate finishedcode(graph(currprimdefaddr)):=codelist;
   funcupdate codelist:=emptycodelist;
endupdates;

% Prepare for performing graph reductions
if ( = (status, "Get-curr-prim-def") &
     isemptyprimdefs(allprimdefs) );
%then
   extend
      extenduniverse INSTR # 2;
   withupdates
         funcupdate getoperator(temp(INSTR,1)):="Pushglobal";
         funcupdate getoperand(1,temp(INSTR,1)):=mainscdefname;
         funcupdate getoperator(temp(INSTR,2)):="Unwind";
         funcupdate instrstack:=
                 makegcodetwo(temp(INSTR,1),temp(INSTR,2));
   endextend
   funcupdate status:="Exec-code";
```

```
   funcupdate leftbranch:=1
   funcupdate rightbranch:=2
endupdates

% Supercombinator definition
if ( = (status, "Compile-sc-def") &
     = (exprtype(valueofaddr(currscdefaddr)),"SDEFexpr"));
%then
   extend
      extenduniverse TADDR # 2;
      extenduniverse NODE;
      extenduniverse INSTR # 2;
   withupdates
      funcupdate graph(currscdefaddr):=temp(NODE,1);
      funcupdate nodetype(temp(NODE,1)):="Global";
      funcupdate defarity(temp(NODE,1)):=
                    numberofparams(makeparams(valueofaddr(currscdefaddr)));
      funcupdate valueofaddr(temp(TADDR,2)):=
                 srcbody(valueofaddr(currscdefaddr));
      funcupdate getparamlist(temp(TADDR,2)):=
                    makeparamposlist(valueofaddr(currscdefaddr));
      funcupdate hascode(temp(TADDR,2)):="False";
      funcupdate getoperator(temp(INSTR,1)):="Slide";
      funcupdate getoperand(1,temp(INSTR,1)):=
                 add(1,numberofparams(makeparams
                             (valueofaddr(currscdefaddr))));
      funcupdate getoperator(temp(INSTR,2)):="Unwind";
      funcupdate instructions(temp(TADDR,1)):=
                 makegcodetwo(temp(INSTR,1),temp(INSTR,2));

      funcupdate hascode(temp(TADDR,1)):="True";
      % Make the testpredicates below happy.
      funcupdate valueofaddr(temp(TADDR,1)):=valueofaddr(currscdefaddr);
      funcupdate tempstack:=pushstack(
                     pushstack(emptystack,temp(TADDR,1)),
                     temp(TADDR,2));
   endextend
   funcupdate status:="Compile-the-body";
endupdates

% Application
if ( = (status, "Compile-the-body") &
     (! isemptystack(tempstack)) &
     = (exprtype(valueofaddr(topaddr(tempstack))),"APexpr") &
     = (hascode(topaddr(tempstack)),"False"));
%then
   extend
      extenduniverse TADDR # 2;
```

```
      extenduniverse INSTR;
   withupdates
      % First AP expression underneath the second on the
      % instruction stack (and on top of address stack).
      funcupdate valueofaddr(temp(TADDR,1)):=
           firstappexpr(valueofaddr(topaddr(tempstack)));
      funcupdate hascode(temp(TADDR,1)):="False"
      funcupdate getparamlist(temp(TADDR,1)):=
              incrementposlist(1,getparamlist(topaddr(tempstack)));
      % Second AP expression on top of the instruction stack
      % (and second on address stack).
      funcupdate valueofaddr(temp(TADDR,2)):=
           secondappexpr(valueofaddr(topaddr(tempstack)));
      funcupdate hascode(temp(TADDR,2)):="False";
      funcupdate getparamlist(temp(TADDR,2)):=
              getparamlist(topaddr(tempstack));
      % Make the AP instruction
      funcupdate getoperator(temp(INSTR,1)):="MKap";
      funcupdate instructions(topaddr(tempstack)):=
                 makegcode(temp(INSTR,1));
      funcupdate hascode(topaddr(tempstack)):="True";
      funcupdate tempstack:=pushstack(
                    pushstack(tempstack,temp(TADDR,1)),
                    temp(TADDR,2));
   endextend
endupdates

% Number expression
if ( = (status, "Compile-the-body") &
     (! isemptystack(tempstack)) &
     = (exprtype(valueofaddr(topaddr(tempstack))),"NUMexpr") &
     = (hascode(topaddr(tempstack)),"False"));
%then
   extend
      extenduniverse INSTR;
   withupdates
      funcupdate getoperator(temp(INSTR,1)):="Pushint";
      funcupdate getoperand(1,temp(INSTR,1)):=
                 makenum(valueofaddr(topaddr(tempstack)));
      funcupdate instructions(topaddr(tempstack)):=
                 makegcode(temp(INSTR,1));
      funcupdate hascode(topaddr(tempstack)):="True";
   endextend
endupdates

% Boolean expression
if ( = (status, "Compile-the-body") &
     (! isemptystack(tempstack)) &
```

```
     = (exprtype(valueofaddr(topaddr(tempstack))),"BOOLexpr") &
     = (hascode(topaddr(tempstack)),"False"));
%then
   extend
      extenduniverse INSTR;
   withupdates
      funcupdate getoperator(temp(INSTR,1)):="Pushbool";
      funcupdate getoperand(1,temp(INSTR,1)):=
                 makebool(valueofaddr(topaddr(tempstack)));
      funcupdate instructions(topaddr(tempstack)):=
                 makegcode(temp(INSTR,1));
      funcupdate hascode(topaddr(tempstack)):="True";
   endextend
endupdates

% Number expression
if ( = (status, "Compile-the-body") &
     (! isemptystack(tempstack)) &
     = (exprtype(valueofaddr(topaddr(tempstack))),"DATAexpr") &
     = (hascode(topaddr(tempstack)),"False"));
%then
   extend
      extenduniverse INSTR;
   withupdates
      funcupdate getoperator(temp(INSTR,1)):="Pushdata";
      funcupdate getoperand(1,temp(INSTR,1)):=
                 makedata(valueofaddr(topaddr(tempstack)));
      funcupdate instructions(topaddr(tempstack)):=
                 makegcode(temp(INSTR,1));
      funcupdate hascode(topaddr(tempstack)):="True";
   endextend
endupdates

% Name expression
if ( = (status, "Compile-the-body") &
     (! isemptystack(tempstack)) &
     = (exprtype(valueofaddr(topaddr(tempstack))),"SCname") &
     = (hascode(topaddr(tempstack)),"False"));
%then
   extend
      extenduniverse INSTR;
   withupdates
      funcupdate getoperator(temp(INSTR,1)):="Pushglobal";
      funcupdate getoperand(1,temp(INSTR,1)):=
                 makescname(valueofaddr(topaddr(tempstack)));
      funcupdate instructions(topaddr(tempstack)):=
                 makegcode(temp(INSTR,1));
      funcupdate hascode(topaddr(tempstack)):="True";
```

```
   endextend
endupdates;

% Primitive name expression
if ( = (status, "Compile-the-body") &
     (! isemptystack(tempstack)) &
     = (exprtype(valueofaddr(topaddr(tempstack))),"PrimName") &
     = (hascode(topaddr(tempstack)),"False"));
%then
   extend
      extenduniverse INSTR;
   withupdates
      funcupdate getoperator(temp(INSTR,1)):="Pushprimglobal";
      funcupdate getoperand(1,temp(INSTR,1)):=
                 makeprimname(valueofaddr(topaddr(tempstack)));
      funcupdate instructions(topaddr(tempstack)):=
                 makegcode(temp(INSTR,1));
      funcupdate hascode(topaddr(tempstack)):="True";
   endextend
endupdates;

% Strict variable expression
if ( = (status, "Compile-the-body") &
     (! isemptystack(tempstack)) &
     = (exprtype(valueofaddr(topaddr(tempstack))),"VARname") &
     = (getparaminfo(getparamlist(topaddr(tempstack)),
           makevarname(valueofaddr(topaddr(tempstack)))),"strict") &
     = (hascode(topaddr(tempstack)),"False"));
%then
   extend
      extenduniverse INSTR # 2;
   withupdates
      funcupdate getoperator(temp(INSTR,1)):="Push";
      funcupdate getoperand(1,temp(INSTR,1)):=
                 getposition(getparamlist(topaddr(tempstack)),
                             makevarname(valueofaddr(topaddr(tempstack))));
      funcupdate getoperator(temp(INSTR,2)):="Eval";
      funcupdate instructions(topaddr(tempstack)):=
                 makegcodetwo(temp(INSTR,1),temp(INSTR,2));
      funcupdate hascode(topaddr(tempstack)):="True";
   endextend
endupdates;

% Variable expression
if ( = (status, "Compile-the-body") &
     (! isemptystack(tempstack)) &
     = (exprtype(valueofaddr(topaddr(tempstack))),"VARname") &
     = (getparaminfo(getparamlist(topaddr(tempstack)),
```

```
                  makevarname(valueofaddr(topaddr(tempstack)))),
        "nonstrict") &
     = (hascode(topaddr(tempstack)),"False") );
%then
   extend
      extenduniverse INSTR;
   withupdates
      funcupdate getoperator(temp(INSTR,1)):="Push";
      funcupdate getoperand(1,temp(INSTR,1)):=
                 getposition(getparamlist(topaddr(tempstack)),
                             makevarname(valueofaddr(topaddr(tempstack))));
      funcupdate instructions(topaddr(tempstack)):=
                 makegcode(temp(INSTR,1));
      funcupdate hascode(topaddr(tempstack)):="True";
   endextend
endupdates;

% Traverse up
if ( = (status, "Compile-the-body") &
     (! isemptystack(tempstack)) &
     = (hascode(topaddr(tempstack)),"True") );
%then
   % The instruction is always appendended at the end of
   % the instruction list.
   funcupdate codelist:=concatcode(codelist,
                            instructions(topaddr(tempstack)));
   funcupdate tempstack:=popstack(tempstack);
endupdates;

% End of on supecombinator definition
if ( = (status, "Compile-the-body") &
     isemptystack(tempstack));
%then
   funcupdate status:="Get-curr-sc-def";
   funcupdate finishedcode(graph(currscdefaddr)):=codelist;
   funcupdate codelist:=emptycodelist;
endupdates;

% Loads part two
loadalg "Graf-red-source/gmaskin-prim-red.src";
```

A.4.2 Specification of the Reduction Process

The specification to execute the G-machine instructions is shown below:

```
% To be loaded into the evolving algebra interpret
% after supercomb-comp.src

% More signatures.
```

```
%
signature topinstr: ((INSTR *) --> INSTR)
signature popinstr: ((INSTR *) --> (INSTR *))

signature addrstack: (TADDR *)
signature currglobdefaddr: TADDR
signature getnthaddr: ((NUMBER x (TADDR *)) --> TADDR)
signature popnaddrs: ((NUMBER x (TADDR *)) --> (TADDR *))
signature lengthas: ((TADDR *) --> NUMBER)

% To be used by the reduction part of Graph Machine
signature nodechild: ((NUMBER x NODE) --> [TADDR + {Empty}])
signature nodeparams: (NODE --> (VARNAME *))
signature nodenum: (NODE --> NUMBER)
signature nodescname: (NODE --> SCNAME)
signature nodeprimname: (NODE --> PRIMNAME)
signature nodevarname: (NODE --> VARNAME)
signature nodevalue: (NODE --> [NUMBER + BOOLDATA + DATA])

signature iswhnftype: (NTYPE --> BOOL)
signature result: NUMBER

% Primitives
signature applyprimitive: ((PRIMNAME x PRIMARGS) --> PRIMRESULT)
signature getresultvalue: (PRIMRESULT --> [NUMBER + BOOLDATA + DATA + INSTRS])
signature resultvalue: [NUMBER + BOOLDATA + DATA + INSTRS]
signature makenumarglist: ((NUMBER x NUMBER) --> PRIMARGS)
signature makecondarglist: ((BOOLDATA x (INSTR *) x (INSTR *))
                            --> PRIMARGS)
signature concatcondcode: ((INSTRS x (INSTR *)) --> (INSTR *))

% The dump stack
signature dumpstack: (DUMP *)
signature emptydumpstack: ((DUMP *) --> BOOL)
signature pushdump: (((DUMP *) x DUMP) --> DUMP)
signature popdump: ((DUMP *) --> (DUMP *))
signature topdump: ((DUMP *) --> DUMP)

% Operation on the dump stack
signature addrstackdump: (DUMP --> (TADDR *))
signature instrstackdump: (DUMP --> (INSTR *))

% More assignments
assignfunc topinstr, first-from-list, dummy-func, upd-not-perm;
assignfunc popinstr, tail-from-list, dummy-func, upd-not-perm;

assignfunc addrstack, constant-std-lookup-data,
                      user-update-constant, std-const-dta;
```

```
assignfunc currglobdefaddr, constant-std-lookup-data,
                     user-update-constant, std-const-dta;
assignfunc popnaddrs, take-nth-first-from-list, dummy-func, upd-not-perm;
assignfunc getnthaddr, pick-elem-from-list, dummy-func, upd-not-perm;
assignfunc lengthas, length-of-the-list, dummy-func, upd-not-perm;


assignfunc nodechild, table-std-lookup,
                    user-update-function, std-table;
assignfunc nodeparams, table-std-lookup,
                       user-update-function, std-table;
assignfunc nodenum, table-std-lookup,
                    user-update-function, std-table;
assignfunc nodescname, table-std-lookup,
                    user-update-function, std-table;
assignfunc nodeprimname, table-std-lookup,
                    user-update-function, std-table;
assignfunc nodevarname, table-std-lookup,
                      user-update-function, std-table;
assignfunc nodevalue, table-std-lookup,
                    user-update-function, std-table;

assignfunc applyprimitive, apply-primitive, dummy-func, upd-not-perm;
assignfunc getresultvalue, get-result-value, dummy-func, upd-not-perm;
assignfunc makenumarglist, make-list, dummy-func, upd-not-perm;
assignfunc makecondarglist, make-list, dummy-func, upd-not-perm;
assignfunc concatcondcode, concat-lists, dummy-func, upd-not-perm;
assignfunc resultvalue, constant-std-lookup-data,
                       user-update-constant, std-const-dta;

assignfunc result, constant-std-lookup-data,
                       user-update-constant, std-const-dta;

assignfunc dumpstack, constant-std-lookup-data,
                    user-update-constant, std-const-dta;
assignfunc emptydumpstack, empty-list, dummy-func, upd-not-perm;
assignfunc pushdump, add-to-list, dummy-func, upd-not-perm;
assignfunc popdump, tail-from-list, dummy-func, upd-not-perm;
assignfunc topdump, first-from-list, dummy-func, upd-not-perm;
assignfunc addrstackdump, table-std-lookup,
                       user-update-function, std-table;
assignfunc instrstackdump, table-std-lookup,
                       user-update-function, std-table;

assignfunc iswhnftype, is-whnf-node-type, dummy-func, upd-not-perm;

% Assign to universe
assignuniverse DUMP, std-ext-collection, newsymbol;
```

```
% Initial values
initial addrstack:== [()]
initial dumpstack:== [()]

if ( = (status, "Exec-code") &
     = (getoperator(topinstr(instrstack)), "Pushglobal") );
%then
   extend
      extenduniverse TADDR;
      extenduniverse NODE;
   withupdates
      funcupdate graph(temp(TADDR,1)):=temp(NODE,1);
      funcupdate nodetype(temp(NODE,1)):="SCname";
      funcupdate nodescname(temp(NODE,1)):=
                  getoperand(1,topinstr(instrstack));
      funcupdate addrstack:=pushstack(addrstack,temp(TADDR,1));
   endextend
   funcupdate instrstack:=popinstr(instrstack);
endupdates;

if ( = (status, "Exec-code") &
     = (getoperator(topinstr(instrstack)), "Pushprimglobal") );
%then
   extend
      extenduniverse TADDR;
      extenduniverse NODE;
   withupdates
      funcupdate graph(temp(TADDR,1)):=temp(NODE,1);
      funcupdate nodetype(temp(NODE,1)):="PRIMName";
      funcupdate nodeprimname(temp(NODE,1)):=
                  getoperand(1,topinstr(instrstack));
      funcupdate addrstack:=pushstack(addrstack,temp(TADDR,1));
   endextend
   funcupdate instrstack:=popinstr(instrstack);
endupdates;

if ( = (status, "Exec-code") &
     = (getoperator(topinstr(instrstack)), "Pushint") );
%then
   extend
      extenduniverse TADDR;
      extenduniverse NODE;
   withupdates
      funcupdate graph(temp(TADDR,1)):=temp(NODE,1);
      funcupdate nodetype(temp(NODE,1)):="Num";
      funcupdate nodevalue(temp(NODE,1)):=
                  getoperand(1,topinstr(instrstack));
```

```
      funcupdate addrstack:=pushstack(addrstack,temp(TADDR,1));
   endextend
   funcupdate instrstack:=popinstr(instrstack);
endupdates;

if ( = (status, "Exec-code") &
     = (getoperator(topinstr(instrstack)), "Pushbool") );
%then
   extend
      extenduniverse TADDR;
      extenduniverse NODE;
   withupdates
      funcupdate graph(temp(TADDR,1)):=temp(NODE,1);
      funcupdate nodetype(temp(NODE,1)):="Bool";
      funcupdate nodevalue(temp(NODE,1)):=
                    getoperand(1,topinstr(instrstack));
      funcupdate addrstack:=pushstack(addrstack,temp(TADDR,1));
   endextend
   funcupdate instrstack:=popinstr(instrstack);
endupdates;

if ( = (status, "Exec-code") &
     = (getoperator(topinstr(instrstack)), "Pushdata") );
%then
   extend
      extenduniverse TADDR;
      extenduniverse NODE;
   withupdates
      funcupdate graph(temp(TADDR,1)):=temp(NODE,1);
      funcupdate nodetype(temp(NODE,1)):="Data";
      funcupdate nodevalue(temp(NODE,1)):=
                    getoperand(1,topinstr(instrstack));
      funcupdate addrstack:=pushstack(addrstack,temp(TADDR,1));
   endextend
   funcupdate instrstack:=popinstr(instrstack);
endupdates;

if ( = (status, "Exec-code") &
     = (getoperator(topinstr(instrstack)), "Push") );
%then
    funcupdate addrstack:=
         pushstack(addrstack,nodechild(rightbranch,
                   graph(getnthaddr
                    (add(getoperand(1,topinstr(instrstack)),2),
                    addrstack)) ));
   funcupdate instrstack:=popinstr(instrstack);
endupdates;
```

```
% Tha Add primitive
if ( = (status, "Exec-code") &
     = (getoperator(topinstr(instrstack)), "Add") );
%then
   funcupdate resultvalue:=getresultvalue(applyprimitive("plus",
                  makenumarglist(
                       nodevalue(graph(topaddr(popstack(addrstack)))),
                       nodevalue(graph(topaddr(addrstack))) )));
   funcupdate status:="Make-num-node";

endupdates;

% Processing numeric result
if   = (status, "Make-num-node");
%then
   extend
      extenduniverse NODE;
      extenduniverse TADDR;
   withupdates
      funcupdate graph(temp(TADDR,1)):=temp(NODE,1);
      funcupdate nodetype(temp(NODE,1)):="Num";
      funcupdate nodevalue(temp(NODE,1)):=resultvalue;
      funcupdate addrstack:=pushstack(popstack(popstack(addrstack)),
                               temp(TADDR,1));
   endextend;
   funcupdate instrstack:=popinstr(instrstack);
   funcupdate status:="Exec-code";
endupdates;

% The Cond Primitive
if ( = (status, "Exec-code") &
     = (getoperator(topinstr(instrstack)), "Cond") );
%then
   funcupdate instrstack:=concatcondcode(
                            getresultvalue(applyprimitive("if",
                  makecondarglist(getoperand(2,topinstr(instrstack)),
                                  getoperand(1,topinstr(instrstack)),
                                  nodevalue(graph(topaddr(addrstack))) )),
                               popinstr(instrstack));
    funcupdate addrstack:=popstack(addrstack);
endupdates;

if ( = (status, "Exec-code") &
     = (getoperator(topinstr(instrstack)), "Eval") &
     (! iswhnftype(nodetype(graph(topaddr(addrstack))))) );
%then
    extend
      extenduniverse DUMP;
```

```
      extenduniverse INSTR;
    withupdates
      funcupdate addrstackdump(temp(DUMP,1)):=addrstack;
      funcupdate instrstackdump(temp(DUMP,1)):=popinstr(instrstack);
      funcupdate dumpstack:=pushdump(dumpstack,temp(DUMP,1));
      funcupdate addrstack:=pushstack(emptystack,topaddr(addrstack));
      funcupdate getoperator(temp(INSTR,1)):="Unwind";
      funcupdate instrstack:=makegcode(temp(INSTR,1));
    endextend;
endupdates;

% The node is already in whnf form, so we do not make any dump.
if ( = (status, "Exec-code") &
     = (getoperator(topinstr(instrstack)), "Eval") &
     iswhnftype(nodetype(graph(topaddr(addrstack)))) );
%then
     funcupdate instrstack:=popinstr(instrstack);
endupdates

if ( = (status, "Exec-code") &
     = (getoperator(topinstr(instrstack)), "MKap") );
%then
   extend
      extenduniverse TADDR;
      extenduniverse NODE;
   withupdates
      funcupdate graph(temp(TADDR,1)):=temp(NODE,1);
      funcupdate nodetype(temp(NODE,1)):="APnode";
      funcupdate nodechild(leftbranch,temp(NODE,1)):=
          getnthaddr(1,addrstack);
      funcupdate nodechild(rightbranch,temp(NODE,1)):=
          getnthaddr(2,addrstack);
      funcupdate addrstack:=pushstack(popnaddrs(2,addrstack),temp(TADDR,1));
   endextend
   funcupdate instrstack:=popinstr(instrstack);
endupdates;

if ( = (status, "Exec-code") &
     = (getoperator(topinstr(instrstack)), "Slide") );
%then
  funcupdate addrstack:=
      pushstack(popnaddrs(add(getoperand(1,topinstr(instrstack)),1)
                          ,addrstack)
               ,topaddr(addrstack))
  funcupdate instrstack:=popinstr(instrstack);
endupdates;

% The unwind instructions.
```

```
if ( = (status, "Exec-code") &
     = (getoperator(topinstr(instrstack)), "Unwind") &
     = (nodetype(graph(topaddr(addrstack))), "APnode") );
%then
  funcupdate addrstack:=
      pushstack(addrstack,nodechild(leftbranch,graph(topaddr(addrstack))));
endupdates;

if ( = (status, "Exec-code") &
     = (getoperator(topinstr(instrstack)), "Unwind") &
     iswhnftype(nodetype(graph(topaddr(addrstack)))) &
     emptydumpstack(dumpstack) );
%then
  funcupdate status:="Weak-Head-Normal-form";
  funcupdate result:=nodevalue(graph(topaddr(addrstack)));
  funcupdate instrstack:=popinstr(instrstack);
endupdates;

if ( = (status, "Exec-code") &
     = (getoperator(topinstr(instrstack)), "Unwind") &
     iswhnftype(nodetype(graph(topaddr(addrstack)))) &
     (! emptydumpstack(dumpstack)) );
%then
     % Set the address to the evaluated node on the top of the addrstack
     funcupdate addrstack:=pushstack(popstack(addrstackdump
                              (topdump(dumpstack))),topaddr(addrstack));
     funcupdate instrstack:=instrstackdump(topdump(dumpstack));
     funcupdate dumpstack:=popdump(dumpstack);
endupdates

if ( = (status, "Exec-code") &
     = (getoperator(topinstr(instrstack)), "Unwind") &
     = (nodetype(graph(topaddr(addrstack))), "SCname") );
%then
  funcupdate currglobdefaddr:=
         getaddrfromglobals(nodescname(graph(topaddr(addrstack))));
  funcupdate instrstack:=popinstr(instrstack);
  funcupdate status:="Exec-sc-def";
endupdates;

if ( = (status, "Exec-code") &
     = (getoperator(topinstr(instrstack)), "Unwind") &
     = (nodetype(graph(topaddr(addrstack))), "PRIMName") );
%then
  funcupdate currglobdefaddr:=
         getaddrfromglobals(nodeprimname(graph(topaddr(addrstack))));
  funcupdate instrstack:=popinstr(instrstack);
  funcupdate status:="Exec-sc-def";
```

```
endupdates;

if ( = (status, "Exec-sc-def") &
     = (nodetype(graph(currglobdefaddr)), "Global") &
     > (lengthas(addrstack),defarity(graph(currglobdefaddr)))) );
%then
  funcupdate instrstack:=
       concatcode(finishedcode(graph(currglobdefaddr)),instrstack);
  funcupdate status:="Exec-code";
endupdates;
```

Appendix B

The Evolving Algebra Interpret

The system description of the Evolving Algebra interpret is enclosed here as a separate report.