Bibliographical References

[1] **AHO, ALFRED V.; SETHI, RAVI; ULLMAN, JEFFREY D.. "Compilers: Principles, techniques, and tools", Addison-wesley, 1988.**

[2] **CafeOBJ official homepage: http://www.ldl.jaist.ac.jp/cafeobj/index.html.**

[3] **CAFEZEIRO, ISABEL; HAEUSLER, EDWARD HERMANN; HAEBERER, ARMANDO M.. "From Diagram to Code via Attribute Grammar", Pontifical Catholic University of Rio de Janeiro, 1997 Rio de Janeiro.**

[4] **DA COSTA, VASTON GONÇALVES. "Compactação de Provas Lógicas", PhD Thesis, Pontifical Catholic University of Rio de Janeiro, 2007 Rio de Janeiro.**

[5] **DALEN, DIRK VAN. "Logic and Structure", Springer, 1997, 3rd ed.**

[6] **DUMMETT, MICHAEL; MINIO, ROBERTO. "Elements of Intuitionism", Claredon Press, 1977 Oxford.**

[7] **The Eclipse Project: http://www.eclipse.org.**

[8] **FELSCHER, W.. "Lectures on Mathematical Logic II: Calculi for Derivations and Deductions", Gordon and Breach Sci, 2000.**

[9] **GAMMA, ERICH; HELM, RICHARD; JOHNSON, RALPH; VLISSIDES, JOHN. "Design Patterns, Elements of Reusable Object-Oriented Software", Addison-Wesley, 1995.**

[10] **GIRARD , JEAN-YVES. "Proofs and Types", Cambridge University Press, 1989.**

[11] **GOUGUEN, JOSEPH A.; MALCOLM, GRANT. "Algebraic Semantics of Imperative Programs", MIT Press, 1996.**

[12] **GORDEEW, LEW; HAEUSLER, EDWARD HERMANN; DA COSTA, VASTON GONÇALVES. "Proof compressions with circuit-structured substitutions",** Notes of Mathematical Seminars of St.Petersburg Department of V.A.Steklov Institute of Mathematics., 2008 . Special Issue dedicated to the Festschriftt of Yuri Matiyasevich, to appear..

[13] Isabelle: http://isabelle.in.tum.de/index.html

[14] The Jape Web Site at Oxford: http://users.comlab.ox.ac.uk/bernard.sufrin/jape.html

[15] Java Programming Language: http://www.java.org/.

[16] The MaudeSystem: http://maude.cs.uiuc.edu/.

[17] MENEZES, PAULO BLAUTH; HAEUSLER, EDWARD HERMANN. "Teoria das Categorias para Ciência da Computação", Porto Alegre: Instituto de Informática da UFRGS, Editora Sagra Luzzatto, 2001.

[18] OMG UML 2.1: http://www.omg.org/.

[19] OSBORNE, Martin J.; RUBINSTEIN, Ariel. "A Course in Game Theory", MIT Press, 1994.

[20] PAGAN, FRANK G.. "Formal Scpecification of Programming Languages", Prentice-hall, 1981, New Jersey.

[21] PVS Specification and Verification System: http://pvs.csl.sri.com/

[22] SCHMIDT, DAVID A.. "Denotational Semantics: A Methodology for Language Development", WmC Brown, 1948.

[23] SCHROEDER, BRUNO. "The Aquinas Machine, Monograph of Undergraduate Computer Engineering Course Conclusion", Pontifical Catholic University of Rio de Janeiro, 2005 Rio de Janeiro.

[24] SCHROEDER, BRUNO. "Fr William of Moerbeck's Compiler, Programming Project – Final Report", Pontifical Catholic University of Rio de Janeiro, 2007 Rio de Janeiro.

[25] WINSKEL, GLYNN. "The Formal Semantics of Programming Languages an Introduction", MIT Press, 1993.

Appendix A
Algebraic Specification of the Graph Machine

The Saint Thomas Aquinas Machine specification describes the machine's assembly-like language and its semantics. A CafeOBJ [2] code follows. An OBJ program is used for defining the semantics of a program because it already has a precise mathematical meaning [11]. This mathematical meaning is a real gain while developing the specification. With it, the machine could be run while it was being specified. The specification was also a good tool for scope definition.

The specification shown below does not fully specify The Saint Thomas Aquinas Machine. The main reason for it is that CafeOBJ is list-based. The graph structure of the machine had to be shrinked to list-like structured. Secondly, more assembly commands ware added to the machine's definition and implementation without a correspondence in this Algebraic Specification.

```
mod! STORE
{
 [ Store Addr Value ] .
 op _[[_]] : Store Addr -> Value .
 op (_{_:_}) : Store Addr Value -> Store .

 vars X Y : Addr . var S : Store . var v : Value .
 eq S{X : v}[[X]] = v .
 cq S{X : v}[[Y]] = S[[Y]] if X =/= Y .
}

mod! PGM
{
 pr (STORE) .
 [ Pgm ] .
 op _;_ : Pgm Pgm -> Pgm [assoc prec 50] .
 op _;_ : Store Pgm -> Store .

 var S : Store . vars P P' : Pgm .
 eq S ; ( P ; P' ) = ( S ; P ) ; P' .
}

mod! GRAPH
{
 [ Vertex Edge ] .
 [ EdgeList < Graph ] .
 op _.s : Edge -> Vertex .
 op _.t : Edge -> Vertex .
 op (_.setS_) : Edge Vertex -> Edge .
 op (_.setT_) : Edge Vertex -> Edge .
 op _#_ : Edge Edge -> EdgeList .
```

```
  op _#_ : Edge EdgeList -> EdgeList .

  var e : Edge . var v : Vertex .
  eq (e).setS(v).s = v .
  eq (e).setT(v).s = (e).s .
  eq (e).setT(v).t = v .
  eq (e).setS(v).t = (e).t .

  --
  -- It could contain a set of vertex represented by
  -- the functions nullSet, include and contains.
  -- Then, it would support Graphs with no-referenced vertex.
  --
 {}

 mod! LANG
 {
  pr (PGM) . ex (GRAPH) . pr (STRING) . pr (BOOL) .
  [ Pgm < VVar ] .
  [ VVar EVar MVar BReg SVar < Addr ] .
  [ Vertex Edge String Bool < Value ] .
  op CR : -> BReg . op MR : -> BReg . op IP : -> Addr .
  op vcreate_ : VVar -> Pgm .
  op ecreate_ : EVar -> Pgm .
  op sto_,_ : Addr String -> Pgm .
  op link_,_,_ : VVar EVar VVar -> Pgm .
  op source_,_ : VVar EVar -> Pgm .
  op target_,_ : VVar EVar -> Pgm .
  op cmp_,_ : Addr String -> Pgm .
  op je_ : VVar -> Pgm .
  op jne_ : VVar -> Pgm .
  op screate_ : SVar -> Pgm .
  op sedges_,_ : SVar VVar -> Pgm .
  op tedges_,_ : SVar VVar -> Pgm .
  op creatematch_,_ : MVar String -> Pgm .
  op match_,_ : SVar MVar -> Pgm .
  op jm_ : VVar -> Pgm .
  op jnm_ : VVar -> Pgm .
  op push_ : Addr -> Pgm .
  op pop : -> Pgm .
  op _.t : Value -> Value .
  op _.s : Value -> Value .
  op _.setT_ : Value Value -> Value .
  op _.setS_ : Value Value -> Value .
  op _.getNext(_,_) : EVar VVar EVar -> VVar .
  op run(_,_) : Graph Addr -> Pgm .

  var S : Store . var X : Addr .
  vars V V1 V2 v vt vs : VVar . vars E e G : EVar .
  var M : MVar . var Ec : SVar .
  vars str str1 str2 : String .

  eq S{V : (str).t} = S .
  eq S{V : (str).s} = S .
  eq S ; vcreate V = S{V : ""} .
  eq S ; ecreate E = S{E : ""} .
  eq S ; sto X, str = S{X : str} .
  eq     S     ;     link     V1,     E,     V2     =     S{E     :
((S[[E]]).setS(S[[V1]])).setT(S[[V2]])} .
  eq S ; source V, E = S{V : (S[[E]]).s} .
  eq S ; target V, E = S{V : (S[[E]]).t} .
```

```
  cq S ; cmp X, str = S{CR : false} if S[[X]] =/= str .
  cq S ; cmp X, str = S{CR : true} if S[[X]] == str .
  eq S ; je V = S .
  eq S ; jne V = S .
  eq S ; screate Ec = S .
  eq S ; sedges Ec, V = S .
  eq S ; tedges Ec, V = S .
  eq S ; creatematch M, str = S{M : str} .
  eq S ; match Ec, M = S{MR : true} .
  eq S ; jm V = S .
  eq S ; jnm V = S .
  eq S ; push X = S .
  eq S ; pop = S .
  eq ((e).setS(v)).s = v .
  eq ((e).setT(v)).s = (e).s .
  eq ((e).setT(v)).t = v .
  eq ((e).setS(v)).t = (e).t .
  cq (e).getNext(vs, e) = vt if (e).s == vs and (e).t == vt .
  cq (E).getNext(vs, e) = vs if (E).s =/= vs or S[[E]] =/= S[[e]].
  cq (((G) # (E)).getNext(vs, e)) = ((G).getNext(vs, e)) if ((E).s
=/= vs) or S[[E]] =/= S[[e]] .
  eq S; run(G, X) = S{IP: S[[X]]}; run(G, G.getNext(X, "PROX") .
}

mod! RUN
{
  pr (LANG) .
  op initial : -> Store .
  ops v1 v2 v3 : -> VVar .
  ops e1 e2 e3 : -> EVar .

  eq initial[[CR]] = false .
  eq initial[[MR]] = false .
}
```

Underlying most models is an idea of state determined by what contents are in the locations [25] chapter 2. The semantics of each of the various features found in programming languages is based very squarely on the semantics of assignment. That's why the key concept in Algebraic Denotational Semantics is that of a store [11] cfr. chapters 2 and 3. Thus, the first module defined is the store. Algebraically it would be defined as the function $\sigma : Addr \to Value$ from addresses to values, because assembly languages locations are addresses referencing some computable value, instead of variables ranging in a limited set. This store, as the traditional stores, represents the machines state in a determined instant.

The CafeOBJ syntax has an algebraic notation correspondence as follows:

```
op _[[_]] : Store Addr -> Value .

eq S{A : v}[[A]] = v .
cq S{A : v}[[B]] = S[[B]] if A =/= B .
```

$$\sigma : Addr \to Value$$

$$\sigma\left[v/_{A}\right](B) = \begin{cases} v \Leftarrow B = A \\ \sigma(B) \Leftarrow B \neq A \end{cases}$$

With the definition of store, it is possible to describe the rule for program sequencing or sequential composition. There is also a correspondence, the following code:

```
op _;_ : Pgm Pgm -> Pgm [assoc prec 50] .
op _;_ : Store Pgm -> Store .
eq S ; ( P ; P' ) = ( S ; P ) ; P' .
```

Corresponds to:

$$\frac{\langle \sigma, p_0 \rangle \rightarrow \sigma'' \quad \langle \sigma'', p_0 \rangle \rightarrow \sigma'}{\langle \sigma, p_0; p_1 \rangle \rightarrow \sigma'}$$

The Saint Thomas Aquinas Machine has graph as its primitive type. All the machine does for proving the theorems is graph manipulation, using graph-oriented assembly commands. Each equation is a graph, the proof being built is a graph, and even the program being run is a graph.

Graph and Lang modules describe the machine's primitive type, and commands respectively. Most of the assembly language commands are atomic commands that only manipulate graphs using store. They change the state of the machine's memory contents that are addressed by the addresses passed to each instruction. Other commands are responsible for the control flow, they are the positive and negative jumps associated with compare and match commands. Compare checks the label associated to an address. Match searches the existence of an element with a determined label in a set. Each command has an associated flag, CR and MR respectively, to provide the conditional jump functionality. Stack commands push and pop provide context and temporary storage. Execution commands are also available: run and get next. They navigate through the instructions graph evaluating the commands.

The machine's scope is theorem proving. When a proof is being built, in a usual logic, each deduction rule applied leads the prover to new equations, more than one in most of the cases. In order to reduce these new equations, it is recommended parallel processing. Hence, it can be observed that the nature of the problem suggests a non-deterministic machine. In fact, the conclusion that the non-deterministic approach must be used.

The assembler manipulates graphs. The program being run is also a graph, commands are vertexes linked by edges labeled “PROX”. The direction of the edge indicates which command is next to which other. As vertexes in a graph support more than one leaving edge and more than one arriving edge, the commands may have more than one preceding command and more than one following command as well. It is imperative that such programs executions are made in a non-deterministic machine. And that is why The Saint Thomas Aquinas Machine is a non- deterministic machine.

Special commands concern to the control flow. In assembly languages, conditional jumps are the only responsible for control flow. Roughly speaking, conditional expressions represent forward jumps and loops represent backward jumps. In this non-deterministic machine jumps receive a list of addresses in which the execution will continue. This approach makes the machine very powerful.

The specification shown above does not describe the machine’s non-determinism. In order to specify the machine’s non-determinism the commands run, get next and the jumps (je, jne, jm, jnm) must receive a list of addresses as parameters.

The algebraic specification of The Saint Thomas Aquinas Machine was presented and the formal semantics of the machine discussed. Some key issues were made explicit: the assembly-like store; the traditional sequencing rule used; the machine’s primitive type (graph); the manipulation commands; and the non-determinism aspect of the machine, with its consequences.

Appendix B
Attribute Grammar of the Upper Level Language

The Fr Williams of Moerbecks Compiler specification formalizes the syntax of the upper level language of the graph machine. An attribute grammar [3] cfr. chapter 3 specification follows. This technique is used because it enables specification of the context-free aspects as well as its code generation [20] cfr. section 2.3. With an attribute grammar; the compiler not only is specified, but also implemented. The grammar can be easily rewritten in a framework of parsers (like JavaCC [15]) to generate the specified compiler.

The attribute grammar's main purpose is to synthesize the attribute *code*. This attribute is of the type graph, and is the program written in the assembly language, that will be input to the machine. The aim is to assign VGM-codes to each construct of the higher-language.

In order to synthesize the *code*, other attributes are used. Sets are inherited: irules, ivariables, ipointers (this last one is a set of ordered pairs); and sets are synthesized: rules, variables, pointers, parameters and errorIds. These sets hold the tables of symbols needed by the compiler in order to build the code, inserting escape code where needed and local variable references.

In vertex matching, vertices can be variables or operators. Attributes ipointers and pointers are sets of pointers created to vertices that hold operators. This is used to go back to these vertices that usually have many edges linked to it. The compiler generates code that go back to them and inspect the sub-graphs it can reach.

The attributes ivariables and variables hold ordered pairs with pairs in variables x pointers. The information in this set is used in two ways. It is used in order to know if its appearance of a variable in the code is the first. And it is used in transformation graph generated code, in order to create edges to the right vertices.

Many rules are defined in a system's definition. The attributes irules and rules guarantee that rule names are unique. Attribute parameters is synthesized

when parsing command invocations. It holds parameters passed as input to commands.

The graph machine accepts only graphs as data in its memory. This is why even the program is a graph. The attribute errorIds hold the ids of the vertices containing code that are escape points of the blocks of code. This set is synthesized and the escape code is added in batch. Non-terminal rules escape with a simple lsload assembly command. This command loads the next rule in the Local Strategy register. Terminal rules escape to a command that calls failure auxiliary function.

The attribute grammar uses inherited and synthesized sets. The sets contains the table of variables and pointers The use of these sets is absolutely necessary. The derivation trees must be visited lots of times and the variables ipointers, pointers, ivariables and variables are always consulted. Besides that, non-terminal rules have two parts and two derivation trees that have to be visited. The first derivation tree synthesizes the attributes inherited by the second. In detail, the definition is made as a graph. Each line of it is an edge, and every edge parsed is a visit to the derivation tree. The synthesized attributes of one line are inherited by the following. Circularities cannot be avoided. The presented attribute grammar makes the attribute calculi from left to right [1] cfr chapter 5.

Other unary attributes are synthesized in the attribute grammar: content, type, pointer and order. They are strings, and the last one is an integer, synthesized directly from the input upper level language program.

The attribute grammar shown bellow uses more formal devices than attribution (<-), conditionals and set theory operators (∪ and ∌). These operators cannot be found in the literature [20][1][3]. They are defined in this section, and they are very intuitive for mathematicians and computer scientists.

Under `<RuleDefinition>` specification, the occurrence of `<Commands>` is optional. The production rule appears between the standard optional operator (`()?`) from regular expressions language. The attribute `code(<Commands>)` is also between an optional device (`[]`).

Some special operators are used over the sets of ordered pairs ivariables and variables. Two projection operators ware used (π, Π). The lower case operator receives a subscript index 1, and returns the set of the first element of the ordered pairs. The upper case operator is decorated with a subscript index 2 and with a

superscript element. It returns the second element of the ordered pair whose first element is the superscript element.

A special logic "or" operator ($\{x|y\}$) was used in attributions. In this operator, one of the operands is null. The attribution is made with the non-null operand.

The graph machine accepts only graphs as data. The programs run by this machine are represented as graphs. Each vertex contains an instruction and it can have any number of leaving ordered edges, linking it to the next assembly commands. The specification creates code snippets and gathers them into the compiled code.

In order to link two instructions, an edge is created. In the specification bellow, this operation is represented by an addition operator (+). When comparison are made, two vertices are created to be linked with the comparison vertex (je and jne). This also happens with loops where sets are involved, and the jump condition is the set emptiness (jse, jsne). In these cases, a dollar sign operator (`$`) is used together with addition to show that the vertex is followed by two vertices. Large code snippets are illustrated.

The dollar sign operator is also used as a batch command under `<RuleDefinition>`. It represents the linkage of the vertices that belong to the errorIds set with the escape code. It's a terminal node linkage of the graph. Under `<Command>`, a different specification of batch usage was made. Operators "`iterate`" and "`.current`" ware used to represent the creation of one push command for every parameter passed as command input.

Vertex unique identifiers are important part of this specification. When it is the case, the id is shown before the vertex content. Id and content are separated by a number sign (`#`). Under `<RuleName>`, for example, the production rule expects a string. The string is the vertex id, whose content is the jump command.

In the specification, several parts require unique identifiers name creation. These names are used for vertex ids and for variables. These names are: v_{i+1}, v_{i+2}, e_{j+1}, S_{k+1}, $LOOP_{k+1}$, p_{p+1} and $exit_{e+1}$. The indexes are always incremented in order to show that under that production rule, a new set identifiers has to be created. The index represents an integer to show that the cardinality of the set of variables is the cardinality of the integers.

Some code synthetizations contain a figure of a visual representation of the graph structure of the code snippet. This figure is intended to help the reader's understanding, although the structure can be understood with the defined operator's.

```
<Definiton> :: = <RulesDefinitions>

      rules(<RuleDefinitions>) <- {}


<RuleDefinitions>1 :: = <RuleDefinition> |

      irules(<RuleDefinition>) <- rules(<RuleDefinitions>1)

                        <RuleDefinition> <RuleDefinitions>2

      rules(<RuleDefinitions>2) <- rules(<RuleDefinition>)
      irules(<RuleDefinition>) <- rules(<RuleDefinitions>1)

<RuleDefinition> ::=

                        <RuleName> "::="
                        "find"
                        <MatchingGraph>
                        "transform-into"
                        <TransformationGraph>
                         (<Commands>)? |

      rules(<RuleDefinition>) <- rules(<RuleName>)
      irules(<RuleName>) <- irules(<RuleDefinition>)
      ivariables(<MatchingGraph>) <- {} × {}
      ipointers(<MatchingGraph>) <- {}
      ivariables(<TransformationGraph>) <-
      variables(<MatchingGraph>)
      ipointers(<TransformationGraph>) <-
      pointers(<MatchingGraph>)
      ivariables(<Commands>) <- variables(<TransformationGraph>)
      ipointers(<Commands>) <- pointers(<TransformationGraph>)
      code(<RuleDefinition>) <- code(<RuleName>) +
      code(<MatchingGraph>) + code(<Graph>) + [code(<Commands>)] +
      int + errorIds(<MatchingGraph>) $ exit_{e+1}#lsload +
      errorIds(<TransformationGraph>) $ exit_{e+1}#lsload

                        <RuleName> ":="
                        <TerminalGraph> (<Commands>)?

      rules(<RuleDefinition>) <- rules(<RuleName>)
      irules(<RuleName>) <- irules(<RuleDefinition>)
      ivariables(<TerminalGraph>) <- {}
      ipointers(<TerminalGraph>) <- {}
      ivariables(<Commands>) <- variables(<TerminalGraph>)
      ipointers(<Commands>) <- pointers(<TerminalGraph>)
      code(<RuleDefinition>) <- code(<RuleName>) +
      code(<TerminalGraph>) + [code(<Commands>)] + call success +
      errorIds(<TerminalGraph >) $ exit_{e+1}#call failure


<RuleName> ::= string
```

```
      if irules(<RuleName>) ∄ string
            code(<RuleName>) <- string#jmp + work GP
            rules(<RuleName>) <- irules(<RuleName>) ∪ string
      else
            Redefinition Exception


<MatchingGraph>1 ::=    <MatchingCompleteEdge> |

      ivariables(<MatchingCompleteEdge>) <-
      ivariables(<MatchingGraph>)
      ipointers(<MatchingCompleteEdge>) <-
      ipointers(<MatchingGraph>)
      variables(<MatchingGraph>1) <-
      variables(<MatchingCompleteEdge>)
      pointers(<MatchingGraph>1) <-
      pointers(<MatchingCompleteEdge>)
      errorIds(<MatchingGraph>1) <-
      errorIds(<MatchingCompleteEdge>)
      code(<MatchingGraph>1) <- code(<MatchingCompleteEdge>)

                        <MatchingCompleteEdge>
                        <MatchingGraph>2

      ivariables(<MatchingCompleteEdge>) <-
      ivariables(<MatchingGraph>1)
      ipointers(<MatchingCompleteEdge>) <-
      ipointers(<MatchingGraph>1)
      variables(<MatchingGraph>2) <-
      variables(<MatchingCompleteEdge>)
      pointers(<MatchingGraph>2) <-
      pointers(<MatchingCompleteEdge>)
      variables<MatchingGraph>1) <-
      variables(<MatchingCompleteEdge>) ∪
      variables(<MatchingGraph>2)
      pointers(<MatchingGraph>1) <-
      pointers(<MatchingCompleteEdge>) ∪
      pointers(<MatchingGraph>2)
      errorIds(<MatchingGraph>1) <-
      errorIds(<MatchingCompleteEdge>) ∪
      errorIds(<MatchingGraph>2)
      code(<MatchingGraph>1) <- code(<MatchingCompleteEdge>) +
      code(<MatchingGraph>2)


<MatchingCompleteEdge> ::=
                        <MatchingVertex>1 <MatchingEdge>
                        <MatchingVertex>2 ";"

      ivariables(<MatchingVertex>1) <-
      ivariables(<MatchingCompleteEdge>)
      ipointers(<MatchingVertex>1) <-
      ipointers(<MatchingCompleteEdge>)
      ivariables(<MatchingVertex>2) <-
      variables(<MatchingVertex>1)
      ipointers(<MatchingVertex>2) <- pointers(<MatchingVertex>1)
      variables(<MatchingCompleteEdge>) <-
      variables(<MatchingVertex>1) ∪ variables(<MatchingVertex>2)
      pointers(<MatchingCompleteEdge>) <-
      pointers(<MatchingVertex>1) ∪ pointers(<MatchingVertex>2)
```

```
      errorIds(<MatchingCompleteEdge>) <-
      errorIds(<MatchingVertex>1) ∪ errorIds(<MatchingVertex>2) ∪
      errorIds(<MatchingEdge>)
      code(<MatchingGraphEdge>) <- code(<MatchingVertex>1) +
      code(<MatchingEdge>) + code(<MatchingVertex>2)


<MatchingVertex> ::=
                        <MatchingVertexId>"#"
                        <MatchingVertexContent>
      variables(<MatchingVertex>) <- ivariables(<MatchingVertex>)
      pointers(<MatchingVertex>) <- ipointers (<MatchingVertex>)
      errorIds(<MatchingVertex>) <- {}
      if type(<MatchingVertexContent>) is OPERATOR
            if ipointers(<MatchingVertex>) ∌
            CP . content(<MatchingVertexId>)
                  pointers(<MatchingVertex>) <-
                  ipointers(<MatchingVertex>) ∪
                  CP . content(<MatchingVertexId>)
                  code(<MatchingVertex>) <-
                  pcreate CP . content(<MatchingVertexId>), WV
            else
                  code(<MatchingVertex>) <-
                  work CP . content(<MatchingVertexId>)
            end if
            code(<MatchingVertex>) <- code(<MatchingVertex>) +
            cmp WV, content(<MatchingVertexContent>) $ jne + je
            errorIds(<MatchingGraph>) <-
            errorIds(<MatchingGraph>) ∪ jne
      else if π1(ivariables(MatchingVertex)) ∌ivariables(<MatchingVertex>)
content(<MatchingVertexContent>)
            code(<MatchingVertex>) <-
            pcreate content(<MatchingVertexContent>), WV
            variables(<MatchingVertex>) <-
            ivariables(<MatchingVertex>) ∪
            {(content(<MatchingVertexContent>);
            content(<MatchingVertexContent>))}
      else
            code(<MatchingVertex>) <-
            sgcmp content(<MatchingVertexContent>), WV $ jne +
            je
            errorIds(<MatchingGraph>) <-
            errorIds(<MatchingGraph>) ∪ jne

      code(<MatchingVertex>) <-
      code(<MatchingVertex>) + pcreate pp+1, WV
      variables(<MatchingVertex>) <- variables(<MatchingVertex>) ∪
{(content(<VertexId>); pp+1)}
      if type(<VertexContent>) is VARIABLE
            variables(<MatchingVertex>) <-
            variables(<MatchingVertex>) ∪
            {(content(<VertexContent>); pp+1)}



<MatchingVertexId> ::= string

      content(<MatchingVertexId>) <- string
```

```
<MatchingVertexContent> ::=   string in uppercase or lowercase |

      type(<MatchingVertexContent>) <- VARIABLE
      content(<MatchingVertexContent>) <- string

                              "\"" string "\""

      type(<MatchingVertexContent>) <- OPERATOR
      content(<MatchingVertexContent>) <- string


<MatchingEdge> ::= "$" <MatchingEdgeId> "#" <MatchingEdgeOrder>
                   ("," <MatchingEdgeContent>)? "$"

      errorIds(<MatchingeEdge>) <- {}
      if order(<MatchingEdgeOrder>) < 1 and
      content(<MatchingEdgeContent>) is null
            Parsing Error
      code(<MatchingEdge>) <- screate S_{k+1} + sedges S_{k+1}, WV +
      sresetS_{k+1} + LOOP_{k+1}#jmp $ jse S_{k+1} + jsne S_{k+1} + snext S_{k+1}, e_{j+1}
```

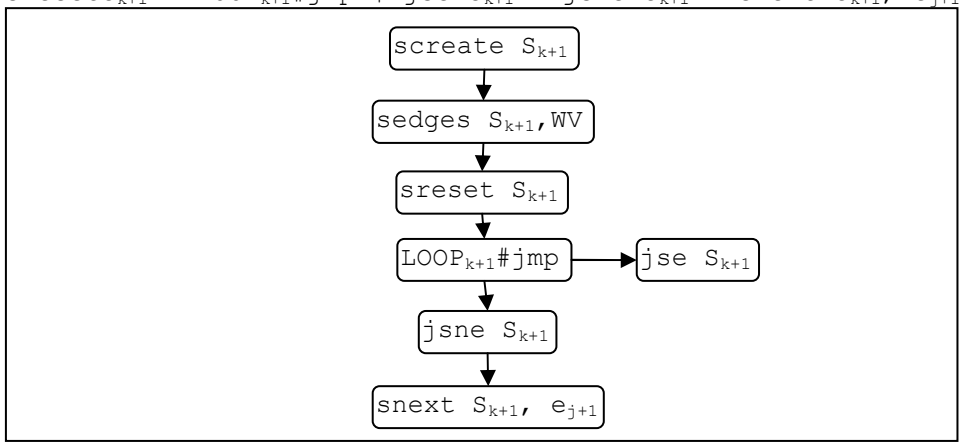


Figure 12 Matching Edge code snippet

```
      errorIds(<MatchingeEdge>) <- errorIds(<MatchingeEdge>) ∪ jse
      S_{k+1}
      if order(<MatchingEdgeOrder>) > 0 and
      content(<MatchingEdgeContent>) is not null
            code(<MatchingEdge>) <- code(<MatchingEdge>) +
            cmpeorder e_{j+1}, order(<MatchingEdgeOrder>) +
            jne $ LOOP_{k+1}#jmp +
            cmp e_{j+1}, content(<MatchingEdgeContent>)
```

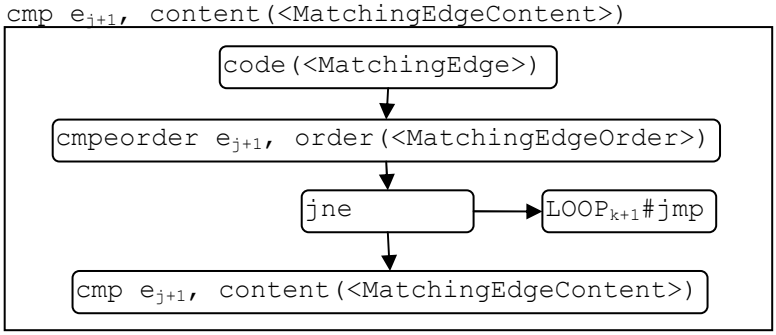


Figure 13 Matching Edge code snippet

```
      else if order(<MatchingEdgeOrder>) > 0
```

```
      code(<MatchingEdge>) <- code(<MatchingEdge>) +
      cmpeorder e_{j+1}, order(<MatchingEdgeOrder>)
else if content(<MatchingEdgeContent>) is not null
      code(<MatchingEdge>) <- code(<MatchingEdge>) +
      cmp e_{j+1}, content(<MatchingEdgeContent>)
end if
code(<MatchingEdge>) <- code(<MatchingEdge>) +
jne $ LOOP_{k+1}#jmp + je + target v_{i+2}, e_{j+1} + work v_{i+2}
```

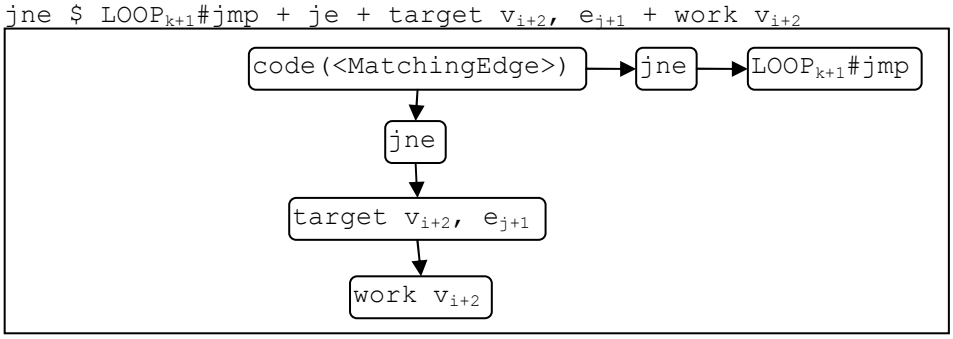


Figure 14 Matching Edge code snippet

```
<MatchingEdgeId> ::= string


<MatchingEdgeOrder> ::= int

      order(<MatchingEdgeOrder>) <- int

<MatchingEdgeContent> ::= string

      content(<MatchingEdgeContent>) <- string


<TransformationGraph>_1 ::= <CompleteEdge> |

      ivariables(<CompleteEdge>) <-
      ivariables(<TransformationGraph>)
      ipointers(<CompleteEdge>) <-
      ipointers(<TransformationGraph>)
      variables(<TransformationGraph>_1) <-
      variables(<CompleteEdge>)
      pointers(<TransformationGraph>_1) <- pointers(<CompleteEdge>)
      errorIds(<TransformationGraph>_1) <- errorIds(<CompleteEdge>)
      code(<TransformationGraph>_1) <- code(<CompleteEdge>)

                        <CompleteEdge><TransformationGraph>_2

      ivariables(<CompleteEdge>) <-
      ivariables(<TransformationGraph>)
      ipointers(<CompleteEdge>) <-
      ipointers(<TransformationGraph>)
      variables(<TransformationGraph>_2) <-
      variables(<CompleteEdge>)
      pointers(<TransformationGraph>_2) <- pointers(<CompleteEdge>)
      variables<TransformationGraph>_1) <-
      variables(<CompleteEdge>) ∪
      variables(<TransformationGraph>_2)
      pointers(<TransformationGraph>_1) <- pointers(<CompleteEdge>)
      ∪ pointers(<TransformationGraph>_2)
```

errorIds(<TransformationGraph>$_1$) <- errorIds(<CompleteEdge>) ∪ errorIds(<TransformationGraph>$_2$)
code(<TransformationGraph>$_1$) <- code(<CompleteEdge>) + code(<TransformationGraph>$_2$)

<CompleteEdge> ::= <Vertex>$_1$ <Edge> <Vertex>$_2$ ";"

ivariables(<Vertex>$_1$) <- ivariables(<CompleteEdge>)
ipointers(<Vertex>$_1$) <- ipointers(<CompleteEdge>)
ivariables(<Vertex>$_2$) <- variables(<Vertex>$_1$)
ipointers(<Vertex>$_2$) <- pointers(<Vertex>$_1$)
variables(<CompleteEdge>) <- variables(<Vertex>$_1$) ∪ variables(<Vertex>$_2$)
pointers(<CompleteEdge>) <- pointers(<Vertex>$_1$) ∪ pointers(<Vertex>$_2$)
errorIds(<CompleteEdge>) <- errorIds(<Vertex>$_1$) ∪ errorIds(<Vertex>$_2$) ∪ errorIds(<Edge>)
code(<CompleteEdge>) <- code(<Vertex>$_1$) + code(<Vertex>$_2$) + ecreate pointer(<Vertex>$_1$), pointer(<Vertex>$_2$), order(<Edge>), content(<Edge>)

<Vertex> ::= <VertexId>"#"<VertexContent> |

variables(<Vertex>) <- ivariables(<Vertex>)
pointers(<Vertex>) <- ipointers(<Vertex>)
if $\pi_1(\text{ivariables(Vertex)}) \not\ni$ content(VertexId) and $\pi_1(\text{ivariables(Vertex)}) \not\ni$ content(VertexContent)
    code(<Vertex>) <- ucreate p_{p+1}, content(<VertexContent>)
    variables(<Vertex>) <- ivariables(<Vertex>) ∪ {(content(<VertexId>); p_{p+1})}
else if $\pi_1(\text{ivariables(Vertex)}) \not\ni$ content(VertexId) and $\pi_1(\text{ivariables(Vertex)}) \ni$ content(VertexContent)
    copysq p_{p+1}, $\prod_2^{\text{content}(<VertexContent>}\text{ivariables}(\langle\text{Vertex}\rangle)$ifoundVertices
    variables(<Vertex>) <- ivariables(<Vertex>) ∪ {(content(<VertexId>); p_{p+1})}
end if
pointer(<Vertex>) <- $\left\{\prod_2^{\text{content}(<VertexId>}\text{variables}(\langle\text{Vertex}\rangle) \,\middle|\, \prod_2^{\text{content}(<VertexContent>}\text{variables}(\langle\text{Vertex}\rangle)\right\}$

<EdgeName> "(" <VertexId> ")"

code(<Vertex>) <- work $\prod_2^{\text{content}(<VertexId>}\text{variables}(\langle\text{Vertex}\rangle)$i + screate S_{k+1} + sedges S_{k+1}, WV + sreset S_{k+1} + LOOP$_{k+1}$#jmp $ jse S_{k+1} + jnse S_{k+1} + snext S_{k+1}, e_{j+1} + cmp e_{j+1}, content(<EdgeName>) + jne $ LOOP$_{k+1}$#jmp + je + target v_{i+1}, e_{j+1} + pcreate p_{p+1}, v_{i+1}

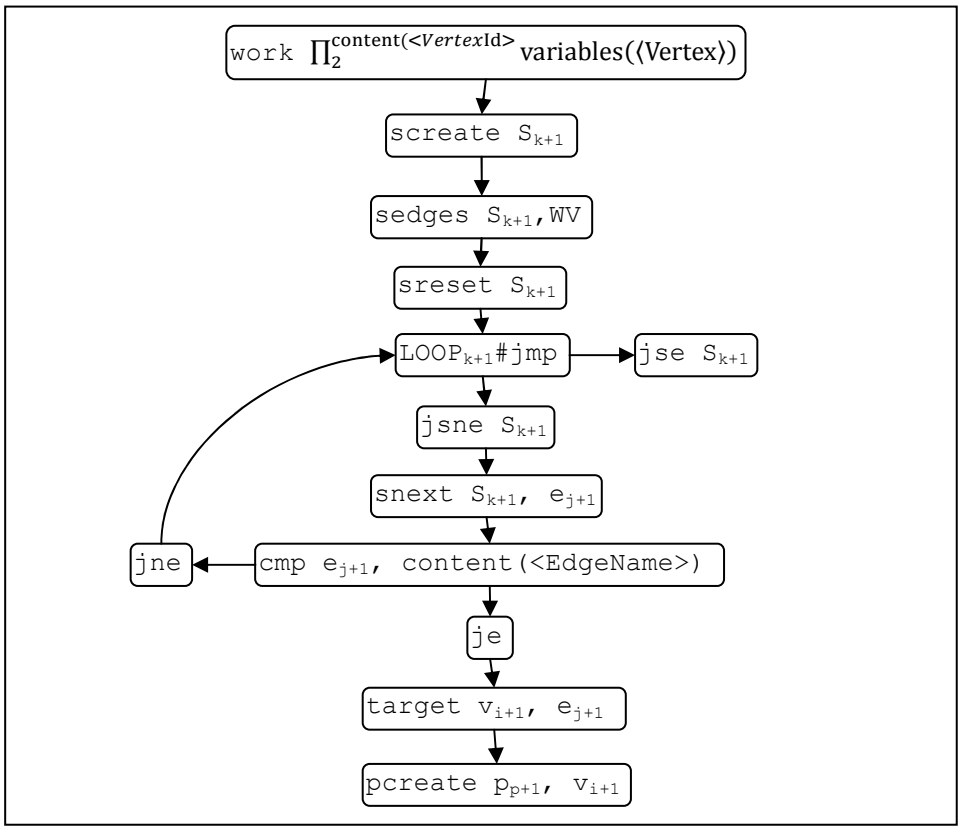


Figure 15 Vertex code snippet

```
<VertexId> ::= string

      content(<VertexId>) <- string


<VertexContent> ::= string or "\"" string "\""

      content(<VertexContent>) <- string


<EdgeName> ::= string

      content(<EdgeName>) <- string


<Edge> ::=  "$" <EdgeId> "#" <EdgeOrder> ("," <EdgeContent>)? "$"

      id(<Edge>) <- content(<EdgeId>)
      order(<Edge>) <- order(<EdgeOrder>)
      content(<Edge>) <- content(<EdgeContent>)


<EdgeId> ::= string

      content(<EdgeId>) <- string


<EdgeOrder> ::= integer
```

```
      order(<EdgeOrder>) <- integer


<EdgeContent> ::= string

      content(<EdgeContent>) <- string


<TerminalGraph>1 ::=    <TerminalGraphEdge> |

      ivariables(<TerminalCompleteEdge>) <-
      ivariables(<TerminalGraph>)
      ipointers(<TerminalCompleteEdge>) <-
      ipointers(<TerminalGraph>)
      variables(<TerminalGraph>1) <-
      variables(<TerminalCompleteEdge>)
      pointers(<TerminalGraph>1) <-
      pointers(<TerminalCompleteEdge>)
      errorIds(<TerminalGraph>1) <-
      errorIds(<TerminalCompleteEdge>)
      code(<TerminalGraph>1) <- code(<TerminalGraphEdge>)

                        <TerminalGraphEdge> <TerminalGraph>2

      ivariables(<TerminalCompleteEdge>) <-
      ivariables(<TerminalGraph>1)
      ipointers(<TerminalCompleteEdge>) <-
      ipointers(<TerminalGraph>1)
      variables(<TerminalGraph>2) <-
      variables(<TerminalCompleteEdge>)
      pointers(<TerminalGraph>2) <-
      pointers(<TerminalCompleteEdge>)
      variables<TerminalGraph>1) <-
      variables(<TerminalCompleteEdge>) ∪
      variables(<TerminalGraph>2)
      pointers(<TerminalGraph>1) <-
      pointers(<TerminalCompleteEdge>) ∪
      pointers(<TerminalGraph>2)
      errorIds(<TerminalGraph>1) <-
      errorIds(<TerminalCompleteEdge>) ∪
      errorIds(<TerminalGraph>2)
      code(<TerminalGraph>1) <- code(<TerminalGraphEdge>) +
      code(<TerminalGraph>2)


<TerminalGraphEdge> ::= <TerminalVertex>1 <TerminalEdge>
                        <TerminalVertex>2 ";"

      ivariables(<TerminalVertex>1) <-
      ivariables(<TerminalCompleteEdge>)
      ipointers(<TerminalVertex>1) <-
      ipointers(<TerminalCompleteEdge>)
      ivariables(<TerminalVertex>2) <-
      variables(<TerminalVertex>1)
      ipointers(<TerminalVertex>2) <- pointers(<TerminalVertex>1)
      variables(<TerminalCompleteEdge>) <-
      variables(<TerminalVertex>1) ∪ variables(<TerminalVertex>2)
      pointers(<TerminalCompleteEdge>) <-
      pointers(<TerminalVertex>1) ∪ pointers(<TerminalVertex>2)
```

```
      errorIds(<TerminalCompleteEdge>) <-
      errorIds(<TerminalVertex>1) ∪ errorIds(<TerminalVertex>2) ∪
      errorIds(<TerminalEdge>)
      code(<TerminalGraphEdge>) <- code(<TerminalVertex>1) +
      code(<TerminalEdge>) + code(<TerminalVertex>2)


<TerminalVertex> ::=
                  <TerminalVertexId>"#"<TerminalVertexContent>

      variables(<TerminalVertex>) <- ivariables(<TerminalVertex>)
      pointers(<TerminalVertex>) <- ipointers (<TerminalVertex>)
      errorIds(<TerminalVertex>) <- {}
      if type(<TerminalVertexContent>) is OPERATOR
            if ipointers(<TerminalVertex>) ∌
            CP . content(<TerminalVertexId>)
                  pointers(<TerminalVertex>) <-
                  ipointers(<TerminalVertex>) ∪
                  CP . content(<TerminalVertexId>)
                  code(<TerminalVertex>) <-
                  pcreate CP . content(<TerminalVertexId>), WV
            else
                  code(<TerminalVertex>) <-
                  work CP . content(<TerminalVertexId>)
            end if
            code(<TerminalVertex>) <- code(<TerminalVertex>) +
            cmp WV, content(<TerminalVertexContent>) $ jne + je
            errorIds(<TerminalGraph>) <-
            errorIds(<TerminalGraph>) ∪ jne
      else if ivariables(<TerminalVertex>) ∌
      content(<TerminalVertexContent>)
            variables(<TerminalVertex>) <-
            ivariables(<TerminalVertex>) ∪
            content(<TerminalVertexContent >)
            code(<TerminalVertex>) <-
            pcreate content(<TerminalVertexContent >), WV
      else
            code(<TerminalVertex>) <-
            sgcmp content(<TerminalVertexContent>), WV $ jne +
            je
            errorIds(<TerminalGraph>) <-
            errorIds(<TerminalGraph >) ∪ jne


<TerminalVertexId> ::= string

      content(<TerminalVertexId>) <- string


<TerminalVertexContent> ::= string in uppercase or lowercase |

      type(<TerminalVertexContent>) <- VARIABLE
      content(<TerminalVertexContent>) <- string

                              "\"" string "\""

      type(<TerminalVertexContent>) <- OPERATOR
      content(<TerminalVertexContent>) <- string


<TerminalEdge> ::=  "$" <TerminalEdgeId> "#" <TerminalEdgeOrder>
```

```
("," <TerminalEdgeContent>)? "$"

errorIds(<TerminaleEdge>) <- {}
if order(<TerminalEdgeOrder>) < 1 and
content(<TerminalEdgeContent>) is null
      Parsing Error
code(<TerminalEdge>) <- screate S_{k+1} + sedges S_{k+1}, WV +
sresetS_{k+1} + LOOP_{k+1}#jmp $ jse S_{k+1} + jsne S_{k+1} + snext S_{k+1}, e_{j+1}
```

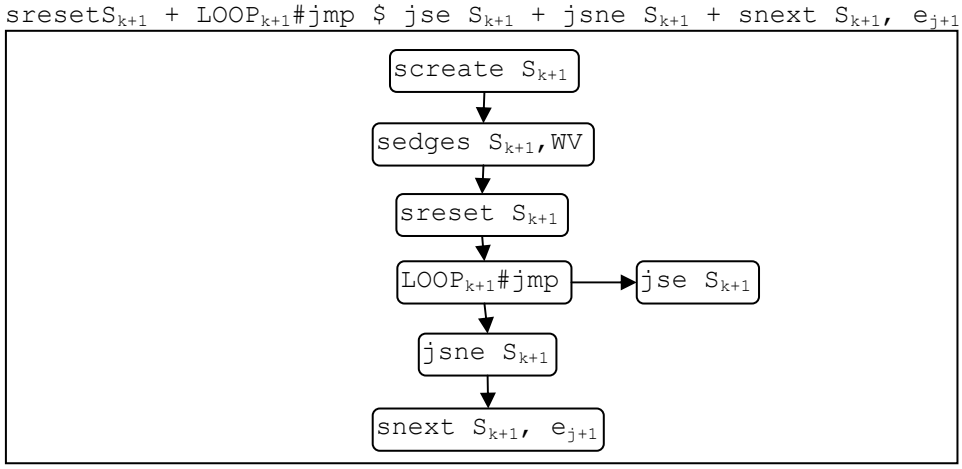


Figure 16 Terminal Edge code snippet


```
errorIds(<TerminaleEdge>) <- errorIds(<TerminaleEdge>) ∪ jse
S_{k+1}
if order(<TerminalEdgeOrder>) > 0 and
content(<TerminalEdgeContent>) is not null
      code(<TerminalEdge>) <- code(<TerminalEdge>) +
      cmpeorder e_{j+1}, order(<TerminalEdgeOrder>) +
      jne $ LOOP_{k+1}#jmp +
      cmp e_{j+1}, content(<TerminalEdgeContent>)
```

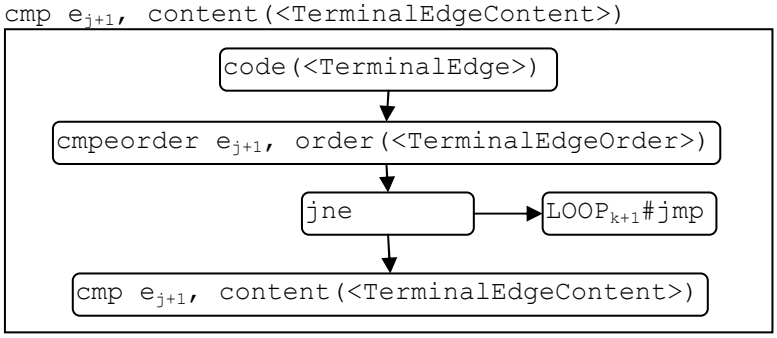


Figure 17 Terminal Edge code snippet

```
else if order(<TerminalEdgeOrder>) > 0
      code(<TerminalEdge>) <- code(<TerminalEdge>) +
      cmpeorder e_{j+1}, order(<TerminalEdgeOrder>)
else if content(<TerminalEdgeContent>) is not null
      code(<TerminalEdge>) <- code(<TerminalEdge>) +
      cmp e_{j+1}, content(<TerminalEdgeContent>)
end if
code(<TerminalEdge>) <- code(<TerminalEdge>) +
jne $ LOOP_{k+1}#jmp + je + target v_{i+2}, e_{j+1} + work v_{i+2}
```

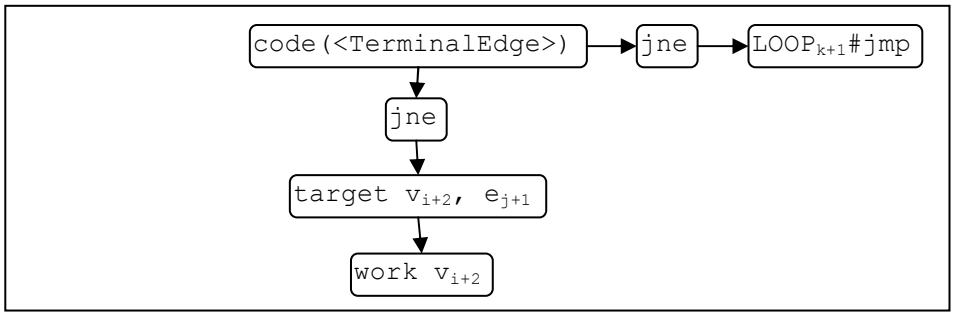


Figure 18 Terminal Edge code snippet

```
<TerminalEdgeId> ::= string

      content(<TerminalEdgeId>) <- string


<TerminalEdgeOrder> ::= integer

      order(<TerminalEdgeOrder>) <- integer


<TerminalEdgeContent> ::= string

      content(<TerminalEdgeContent>) <- string


<Commands>1 ::=    <Command> |

      code(<Commands>1) <- code(<Command>)

                  <Command> <Commands>2

      code(<Commands>1) <- code(<Command>) + code(<Commands>2)


<Command> ::= <CommandName> "[" <CommandParameters> "]"

      iterate over parameters(<CommandParameters>)
            code(<Command>) <- code(<Command>) +
            push parameters(<CommandParameters>).current
      code(<Command>) <- code(<Command>) +
      call content(<CommandName>)


<CommandName> ::= string

      content(<CommandName>) <- string


<CommandParameters>1 ::=       <CommandParameter> |

      parameters(<CommandParameters>1) <-
      content(<CommandParameter>)

                  <CommandParameter> "," <CommandParameters>2
```

```
      parameters(<CommandParameters>1) <-
      content(<CommandParameter>) ∪
      parameters(<CommandParameters>2)


<CommandParameter> ::= string

      content(<CommandParameter>) <- string
```

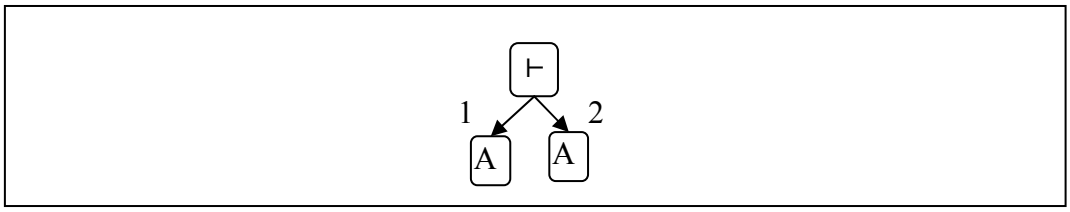
Appendix C
Logic Description Manual

This manual explains how to describe a logic deduction system using the upper level language that can be compiled by Fr William of Moerbecks Compiler. The explanation is made using examples.

Let CS be a Sequential Calculus System. In a system's specification, two elements are usually present: the WFFs identification schema and the set of inference rules. In the presented language, there is no need to specify the WFFs identification schema. The identification of WFFs comes with the rules graph matching pattern, as shown in this appendix.

The rules of the system can be divided in axioms (or terminal rules) and rules. $A \vdash A$ is the sequential calculus axiom. In order to define this axiom in the upper level language, the specifier has to represent it as a graph matching pattern. This graph matching pattern is a simple graph.

In the example the axiom will be called id, and its graph representation follows:



The representation is a graph that begins with a vertex labeled with the sequent. From this vertex, two edges leave to reach the precedent and the consequent. The specifier intention is to identify the formulas at the sides of the sequent using the order of the edges leaving it. The edge with order one, reaches the formula that is the precedent. The edge with order two, reaches the consequent.

Representation is chosen. In order to write the code, the specifier will represent each node in a line. Each vertex and edge needs a unique identifier (in the context of the rule). Vertices that are reached by more than one edge should be

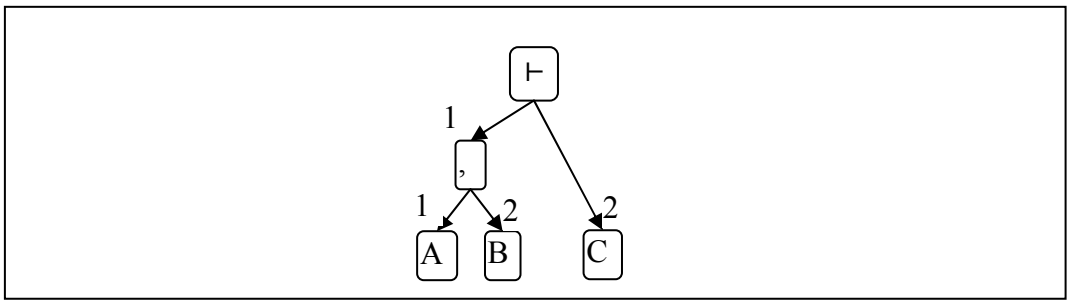
repeated (identifier and label). The code respects the syntax, and can be seen bellow.

```
id :=
v01#"⊢"$ e01#1 $v02#A ;
v01#"⊢"$ e02#2 $v03#A
```

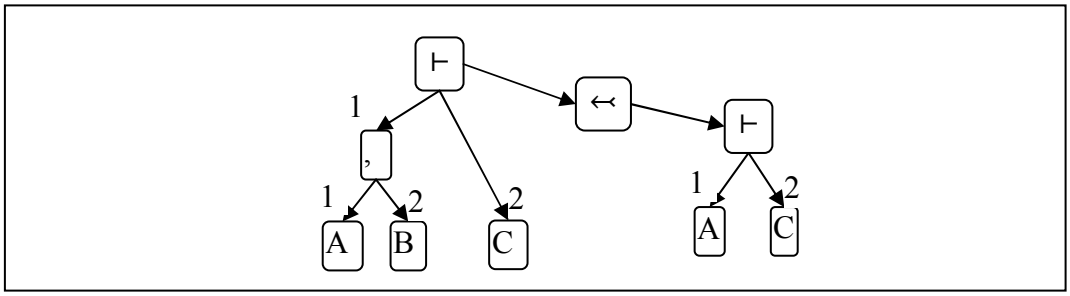
Rules are more than simple axioms. Besides matching a graph pattern; rules transform the proof, adding a inference bar and premises to the proof. The rule specification is divided in these two parts graph matching pattern and graph transformation. The graph matching pattern is defined in rules, the same way it is in axioms. A type of weakening rule will serve as example.

$$\frac{A \vdash C}{A, B \vdash C} weak_{L1}$$

One can represent the matching graph pattern in many ways. The decisions concerning representation, made in the specification phase, affect the generated theorem prover. The prover can interpret input formulae written only with the same representation of its rules. It is also important to have a set of rules with the same representation. For the example, the following pattern will be used:



Besides the graph matching pattern; the specifier defines the transformation graph. The transformation is made in the graph that matches the specified pattern. Elements are added to it. The transformation graph can be represented with the matching graph as is shown bellow.



The graphs can now be written using the language syntax. The context of the identifiers is the rule; thus vertices from the matching graph can be referenced, by its id.

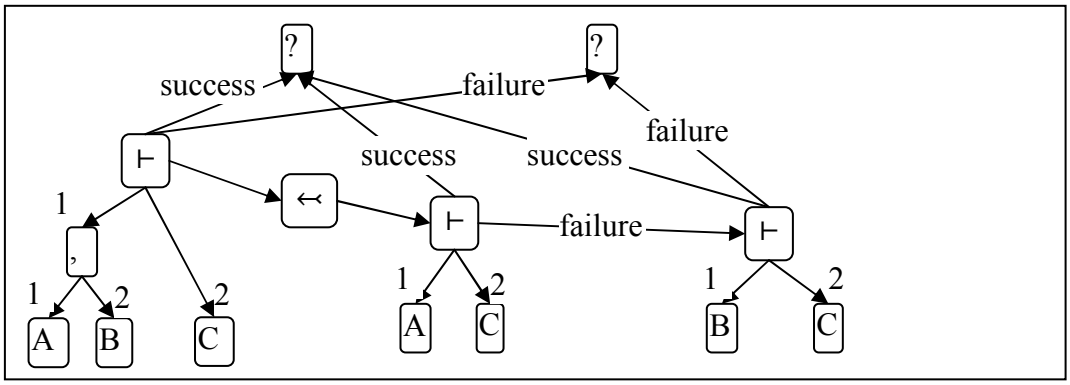
```
weak_L1 ::=
find
v01#"⊢"$ e01#1 $v02#"," ;
v01#"⊢"$ e02#2 $v03#C ;
v02#","$ e03#1 $v04#A ;
v02#","$ e04#2 $v05#B
transform-into
v01#"⊢"$ e05#-1 $v06#"↢" ;
v06#"↢"$ e06#-1 $v07#"⊢" ;
v07#"⊢"$ e07#1 $v08#A ;
v07#"⊢"$ e08#2 $v09#C
set_current_goal[v07]
```

In the code shown above; after the transformation graph, a command can be seen. This command belongs to the auxiliary library of commands. It is responsible for changing the current goal to v_{07}. Other commands can be called at this moment.

Many kinds of rules can be implemented. Another interesting kind of rule is inspired on the backtracking mechanism of [22]. With it, more than one rule can be specified as the same. For example:

$$\frac{A \vdash C}{A, B \vdash C} weak_{L1} \quad \frac{B \vdash C}{A, B \vdash C} weak_{L2}$$

The rules will be specified as one that builds L1 and L2. If L1 does not lead to a proof; L2 might do. Besides the specification of multiple rules in one, this rule will reference commands reachable by success and failure edges. These commands will be given with the formula to be proved. This way, when the proof is build a success mechanism can be started. If the proof cannot be build; a failure mechanism is started. In this approach, every rule should create references to these vertices. The desired weakening of the left side can be seen bellow:




```
weak_L ::=
find
v01#"⊢"$ e01#1 $v02#"," ;
v01#"⊢"$ e02#2 $v03#C ;
v02#","$ e03#1 $v04#A ;
v02#","$ e04#2 $v05#B
transform-into
v01#"⊢"$ e05#-1 $v06#"↢" ;
v06#"↢"$ e06#-1 $v07#"⊢" ;
v07#"⊢"$ e07#1 $v08#A ;
v07#"⊢"$ e08#2 $v09#C ;
v07#"⊢"$ e09#-4,success $success(v01) ;
v07#"⊢"$ e10#-3,failure $v10#"⊢" ;
v10#"⊢"$ e11#1 $v11#B ;
v10#"⊢"$ e12#-4,success $success(v01) ;
v10#"⊢"$ e13#-3,failure $failure(v01) ;
v10#"⊢"$ e14#2 $v12#C
set_current_goal[v07]
```

The deduction rules and axioms can be specified as shown. The compiler will generate Saint Thomas Aquinas Machine's assembly code of the theorem provers. The last step to have a good theorem prover is to invest in strategies. This is done implementing a *DecisionAgent* class to interact with the machine. More about this class is in section 4.1.