THE SEMANTICS AND TRANSFORMATION OF IMPERATIVE PROGRAMS USING HORN CLAUSES

By

BRIAN JAMES ROSS

B.C.Sc., University of Manitoba, 1984

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES
(DEPARTMENT OF COMPUTER SCIENCE)

We accept this thesis as conforming
to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

July 1988

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of computer science

The University of British Columbia
1956 Main Mall
Vancouver, Canada
V6T 1Y3

Date Aug. 15, 1988

Abstract

The feasibility of using Horn clauses as a means of describing and transforming imperative programs is explored. A logical semantics is derived for a typical imperative language. The style of this semantics permits direct translations between the source language and semantical representations. Given the use of Horn clause logic in logic programming, the semantics is particularly useful since models of computation associated with logic programs can be applied to it. Treating the semantics as a logic program means that, in a program transformation context, the semantic representation is particularly well suited to partial evaluation. To support these ideas, an automatic translation system has been implemented which permits translation between programs written in an imperative language and their logical equivalents. In addition, a partial evaluator has been written which meta-interprets the logical representation of an imperative program to produce a partially evaluated result. Using predicative programming, this result can be translated back into the original source code. Thus the system as a whole performs source-to-source transformations of imperative programs.

Contents

List of Figures

Acknowledgement

I owe special thanks to my supervisor, Harvey Abramson. His advice and guidance were directly responsible for the success of this thesis. Thanks are also due to Wolfgang Bibel for his helpful suggestions, and for reading the thesis. I also owe thanks to Paul Voda for directing me to Hehner's work; to Doug Westcott for technical assistance and the use of his lexical analysis code; to Matthew Crocker for helpful discussions and moral support; and to the Department of Computer Science for its financial assistance. Finally, my gratitude to my parents, Jack and Marlene Ross, for their unwavering love and support throughout the years.

Chapter 1

Introduction

An imperative programming language is one whose design is motivated by the architecture of a von Neumann machine [Backus 78]. Such languages typically involve the update of a "state", which is usually taken to be the current state of the machine's memory. The vast majority of computer applications still make predominant use of imperative programming. Engineering and scientific applications rely on the use of FORTRAN. The "C" language has found a niche as a systems programming language, to a large extent replacing assembler language programming. The Pascal language is currently in vogue as a general purpose programming language. Given the widespread existence of imperative programs, there is still a need to explore new techniques of analysis for this class of programs.

Predicate logic continues to enjoy extensive use in the verification and transformation of imperative–style programs [Loeckx *et al.* 84] [Hoare 85]. With the advent of logic programming and other computational models for predicate logic there are increased opportunities to apply the logical analysis of programs in semi–automated environments. This thesis explores the feasibility of using Horn clause logic to describe and transform imperative programs. Although declarative logic programming and imperative programming represent opposite stylistic philosophies, we will demonstrate that logic programming offers interesting possibilities as a tool for analyzing its "imperative ancestor".

The motivating principle behind this thesis is that *valid logical manipulations of a*

program's logical semantics reflect correctness–preserving transformations of the program's source. In particular, we will show that imperative programs can be economically described in Horn clause logic, and that there exist logically valid transformations of these Horn clause axioms which represent useful program transformations of the source. First, the thesis introduces a new logical semantics for imperative–style programs. Given a correct and executable program written in an imperative language, we define for it a logical semantics through the inspection of its syntax. The style of this semantics is most related to ones given by Hehner [Hehner 84a] [Hehner *et al.* 86], but is somewhat simplified, given its intended use for program transformation. Using the predicative programming paradigm, the logical semantics and source language are intertranslatable. Thus, changes to the logical representation reflect corresponding changes to the source program. Interpreting the semantics as a logic program means that one specific type of program transformation – partial evaluation – is particularly suitable, since the semantics can be executed using SLDNF resolution.

To support the above argument, we implement an imperative program transformation system. We first define a Horn clause semantics for a generic imperative language. Given an operational semantics for the language defined in terms of an abstract machine, we define logical axioms for basic language constructs which describe their operational behavior. The semantics for a program takes the form of a logical theory which describes the complete computational behavior of the program. We prove the feasibility of this semantics by implementing an automatic translation system which transforms imperative programs into their logical equivalents, and vice versa. We then implement a partial evaluator which, given a logic program representation of a program, meta–interprets it to produce a partially evaluated result. Using predicative programming, this partially evaluated result is translatable into its imperative equivalent. This implementation as a whole is thus a source–to–source transformation system for imperative programs.

Chapter 2 reviews the areas of formal program semantics, program transformation, and partial evaluation – topics all relevant to this thesis. Chapter 3 defines a generic

imperative language called Bruce. Design motivations for the language are discussed, and the language syntax and operational semantics are presented. Chapter 4 presents a new Horn clause semantics for Bruce. Motivations behind the design of the semantics are discussed. Then the semantics themselves are presented, along with some example translations. An implementation of a Bruce–logic translation system is outlined in chapter 5. A technique for optimizing Bruce programs is presented in chapter 6. An evaluation of the work concludes the thesis in chapter 7.

Chapter 2

Review

Brief reviews of some areas of computer science relevant to this thesis are now presented. Section 2.1 surveys the field of formal program semantics. Program transformation is discussed in section 2.2. Section 2.3 introduces the concept of partial evaluation.

2.1 Formal Program Semantics

2.1.1 Goals of Program Semantics

Programming languages are complex systems. Since conventional programming languages are tailored around the architecture of von Neumann machines, operations in these languages are primarily motivated by hardware considerations. The functionality of program segments as they relate to the original problem specification is often not obvious to the programmer. This occurs because people do not normally define problems and solutions in terms of the hardware of a machine, but instead use natural language, mathematics, or some other human–oriented language. The task of computer programming is to convert a human–oriented specification into a machine–oriented programming language.

The difference of expressiveness between programs and problem specifications becomes especially relevant when one wants to formally verify the correctness of a program. Formally proving facts about programs is difficult, since there are two levels of formal abstraction being addressed: (1) the program specification language, and (2) the programming lan-

guage itself, which is a formal language. Relating these two formal systems together is the task of formal program semantics, which have been established to allow people to more easily reason about programs and programming languages. Program semantics provide a means of associating programs and programming languages with objects representing their meaning. For example, these objects might be integers representing the input and output of a program module, and the means of deriving facts about the integers might take the form of functions, logical predicates, or algebraic equations. The main point is that any semantic system adopted must allow correct conclusions to be reached about the program being analyzed, and should ideally provide useful calculi with which these conclusions can be derived with the least amount of effort.

A number of methodologies exist which differ in both their models of implementation and their intended purposes as tools for program and language analysis. The main semantic models are: (1) logical semantics, (2) denotational semantics, (3) operational semantics, and (4) algebraic semantics. These systems have found uses in the areas of language description and design, and program verification, synthesis, and transformation. Since some semantic methodologies are specialized for particular analytical purposes, not all semantic paradigms are used for the entire aforementioned list of applications. On the other hand, systems such as the denotational semantics have proved to be quite general in their applicability.

We now briefly survey the four major semantic methodologies. Since this thesis focuses on the use of logical semantics, the section on this formalism will be emphasized accordingly. For a good comparative discussion of semantics, see [Moss 81]. [Loeckx *et al.* 84] presents a comprehensive overview, as well as many example applications.

2.1.2 Logical Semantics

The term "logical semantics" can easily lead to confusion, as predicate logic has been used in analyzing programs in a variety of different ways. The term has historically come to represent the use of a specialized form of logic known as *Hoare logic.* However, the

last decade has seen other applications of predicate logic as a semantic formalism for programming languages and computation. We distinguish three distinct uses of logic in the field of program semantics: (i) Hoare logics and program verification, (ii) the logical description of computation, and (iii) logic programming.

Seminal papers by [Floyd 67] [Hoare 69] [Burstall 69] established the utility of first–order predicate logic as a tool for analyzing the behavior of imperative programs. Out of this work, Hoare logics became established as a standard methodology for program verification. [Apt 81] has a comprehensive review of the field, and [Loeckx *et al.* 84] has example applications. Hoare logic has defined for it a standard set of axioms and inference rules which are common to most imperative languages; new languages must have new axioms and calculi defined for them (for example, [Hoare *et al.* 73] has a treatment of Pascal). The axioms specify the computational behavior of program constructs under particular conditions, outlining any logical inferences derivable when these conditions are satisfied. Once an axiom set is established for a language, it is applied to a program to formally prove some property of the code.

Hoare logic is a major program verification tool. Its predicate logic basis gives it intuitive appeal. It uses a "flatter" domain space than denotational semantics (see below), the domain typically being variable values, which lends considerable simplicity to users interested in program verification. In addition, work in theorem proving has shown the applicability of Hoare–style logic in semi–automated environments [Manna *et al.* 77]. However, Hoare logic has been criticized for its often unsound and *ad hoc* axioms, and dissatisfaction has arisen to the ways in which different program constructs have been handled [O'Donnell 82]. Hoare logic is also unsuitable in applications requiring more abstract representations of programming languages, and does not address language implementation at all.

Whereas Hoare logic is a specialized logic useful in verification, unconstrained first–order predicate logic has found a niche as a general purpose description language for computation and the programming process. One particular paradigm founded in logic

is *predicative programming.* Generally speaking, predicative programming is a programming discipline in which programs are constructed from logical specifications using first–order logic. Bibel established the term in [Bibel 75], and further illustrates the formal synthesis of programs from logical specifications in [Bibel 85]. Hehner uses the term in a similar sense, with the additional perspective that it is possible to freely intersperse conventional imperative–style program code with logic during program analysis and synthesis if one treats syntactic constructs of a programming language as predicates [Hehner 84a] [Hehner *et al.* 86]. In his system, each construct in a language has a semantic defined for it which can be (i) composed in first–order predicate calculus, (ii) defined in terms of subsidiary language constructs, which are likewise logically defined, or (iii) a combination of language constructs and logic. In a mixture of program code and logic, the code portions represent *implementable* parts of the computation, while the logic represents *semantic descriptions.* Using predicative programming, the transformation and analysis of code is assured of being logically sound. In addition, predicative programming provides a degree of elegance in merging these two different formal languages into one analytical framework.

Hehner defines logical semantics for a typical imperative language in [Hehner 84a] and [Hehner *et al.* 86]. Examples are given using these semantics in verification and synthesis, as well as illustrating sophisticated aspects of programming. This style of semantics is extended in [Hehner 84b] and [Hoare 85] to describe computations which utilize interprocess communication.

A key advantage of logic worth mentioning is that it is easily automated. Work in theorem proving has enabled the implementation of program transformation and verification systems. Systems such as those described in [Boyer *et al.* 79] and [Manna *et al.* 77] utilize logical semantics as an underlying formalism.

Logic programming is an active field of computer science. It is a paradigm which uses Robinson's resolution principle [Robinson 65] to solve programs written as Horn clauses, a subset of first–order predicate logic. A logic program can have many semantic interpretations applied to it, namely, a declarative or model theoretic semantics, and a procedural

or proof theoretic semantics [Clark *et al.* 81]. The programming language Prolog is the current realization of the logic programming ideal [Clocksin *et al.* 81]. The relevance logic programming has to program semantics is that the statements of a declarative Prolog program can be directly interpreted as statements of logic. This means that there is no need for Hoare-style logics to be applied to the program when analyzing it, since the program's semantics are explicit in the program text.

2.1.3 Denotational Semantics

Denotational semantics is a formalism invented by Scott and Strachey (see [Stoy 77] for an introduction). In general terms, denotational semantics refers to the interpretation of the meaning of a program strictly in terms of the meaning of its constituent parts. It is a mathematically founded model of program semantics concerned with the theory of domains, and whose basis language is the lambda calculus. In denotational semantics, the meaning of a program is described in terms of a function from some domain to another. The definition of this function may be in terms of other functions, many of which might have complex domains. The strength of denotational semantics lies in its generality. It has been used to describe many different programming languages, such as Algol–like languages [Gordon 79] and Prolog [Allison 86]. Denotational semantics is arguably more descriptively powerful than semantics based in logic due to the high level of abstraction available. Consequently, it has been used in program verification and transformation [Stoy 77], as well as language description and design [Tennent 77] However, its simplicity has suffered at the sake of generality. Denotational descriptions of languages are often very complicated, suggesting that language description is not necessarily a natural application [Ashcroft *et al.* 82]. The domains which might be required by a particular language are often too abstract for the programmer trying to prove a simple fact about a program segment. In addition, implementability issues are not dealt with.

2.1.4 Operational Semantics

Operational semantics characterize languages in terms of the machine's behavior when processing them. An operational semantics requires the definition of an abstract machine or interpreter for the language in question. This abstract machine must have incorporated within it precise definitions of behavior for the language constructs. Operational semantics are important in describing programming languages from a design and implementation standpoint, but are typically of less use in program verification and transformation. See [Cook 78], [de Bruin 81], and [Moss 81] for examples of operational definitions of languages.

2.1.5 Algebraic Semantics

The latest semantic formalism is algebraic semantics [Guessarian 81]. Algebraic semantics attempts to simplify the semantic notation and analytical calculi of program analysis to its most basic elements. In this scheme, programs are recast into algebraic expressions. Proving program properties is then transformed into proving algebraic problems. For example, to prove the equivalence of two programs, one would define algebraic schemas for the programs, and then through the use of analytical techniques such as induction and term rewriting prove that the program schema are reducible to a common algebraic representation. Algebraic semantics are used in program verification and transformation [Guessarian 81], and synthesis [Broy 84].

2.2 Program Transformation

A comprehensive review of program transformation systems up to 1983 is in [Partsch *et al.* 83], upon which most of the following discussion is based. We forego mentioning many specific transformation systems in this section, and instead survey the systems most relevant to this thesis in section 2.3. [Pepper 84] also contains a detailed treatment of the field.

Software engineering up to now has relied on the programmer's expertise in encoding and debugging. This is a process which has proved error prone, and as a consequence,

expensive. As a result, much research effort is currently being directed into developing more automated programming environments, and is inspiring coordinated developments in software engineering, artificial intelligence, and programming languages. These automatic programming systems, although still at the experimental stage, will eventually allow reliable programs to be produced with the least amount of technical intervention on the part of the programmer.

One key area of automatic programming is program transformation. The term "program transformation" is a general one; it has been used in reference to program transformation proper, as well as program synthesis and verification. We use the term to mean the conversion of a program into one which is more efficient or contains some desired quality not found in the original program. Program synthesis, on the other hand, is the creation of a program from a formal specification. Both these applications share much basic methodology, as one can consider the original program given to optimizing transformation systems as the "specification" used by synthesis systems. Program transformation differs from program compilation in that transformations are typically applied at a higher level than the translations done by compilers. The result of a program transformation is usually a program in the same source language exhibiting profound improvements, whereas compilers produce code which is more elementary than the source, and is "peep hole" optimized at best.

Two basic approaches to program manipulation are found in transformation systems. The *generative* approach takes a small kernel of basic program transformation schemas and generates more complex ones from them. These schemas might be implemented as general principles of transformation, as is done in section 2.3, rather than through the use of rules *per sae*. The *catalog* approach assumes the existence of a knowledge base of program transformations. This collection of transformations might contain programming and data domain knowledge, language specific features, and efficiency information. An advantage of knowledge–based systems is that they can be upgraded with new rules when necessary.

An important factor in any transformation system is the type of rule set used, as the types and scope of the transformation rules ultimately determine the power of the system as a whole. The most elementary class of rules are the schematic or pattern replacement ones. They specify local optimizations of a program, such as boolean and algebraic simplifications. A more powerful class are those which are global. These rules have repercussions on an algorithmic scale, and may specify information dealing with flow analysis and general programming principles. Lastly, there exist rules which are both local and global in applicability, such as *fold* and *unfold* transformations [Darlington *et al.* 73].

The internal representation of programs and rules is also of importance. Some systems rely on formal program semantics as an underlying representation for programs and transformation rules [Manna *et al.* 77]. Other systems simply use syntactic representations for programs on which textual "source–to–source" transformations are applied [Loveman 77]. There are philosophical differences between these approaches. Formal approaches are motivated by correctness considerations; transformations applied to programs are always assured of being sound, sometimes at the expense of transformational efficiency. Syntactic approaches, on the other hand, are more intuitionistic and efficient, but are prone to erroneous modifications should a transformation rule be faulty. A reconciliation is in approaches founded in algebraic semantics [Broy 84], which simplify the representation of programs and rule sets while maintaining correctness considerations.

Different degrees of automation exist in transformation systems. User controlled systems give the user the responsibility of applying transformations to a program. The system acts as a repository of powerful editing commands, and does assorted bookkeeping tasks in order to simplify programming. At the other extreme are the fully automated systems. They require the use of heuristics to control the transformation process in order to be practical. Semi–automated systems share features of both these approaches, acting as intelligent editors for the programmer.

2.3 Partial Evaluation

Partial evaluation, or mixed computation, is a computational paradigm which is currently gaining more attention in research in logic and functional programming, as well as conventional program translation. Partial evaluation can be applied in areas such as program execution, program optimization and transformation, compiler generation, compiler–compiler generation, and meta–programming.

[Ershov 82] gives an introduction to the concept, and outlines how partial evaluation can be characterized in either a functional or operational fashion. (1) Functionally: given a program P which computes some function $f(X,Y)$, partial evaluation is the process of obtaining a *residual* program P_x which computes the function $f(x,Y) = g(Y)$, where the data value of X is x. P_x is also known as the *projection* of P onto x. (2) Operationally: a program P defines a set C of elementary computations. C itself is partitionable into subsets $C' \cup C''$, where C' are called *permissible* computations, and C'' are *suspendable* computations. Partial evaluation is the process which executes C' and forms a residual program P_μ which defines the computation C''.

These two characterizations are generalized in [Ershov 82] [Ershov 85], in which partial evaluation is defined by a mapping:

$$M : P \times D \rightarrow P \times D$$

where P are programs and D are data domains. This mapping can be parameterized by an ordered set $\{\mu\}$ in which μ_0 and μ_1 are minimal and maximal values in the set respectively. μ_0 represents the case when no data is available to the partial evaluator; μ_1 is the case when all the data is available. This metric – the availability of data – effectively describes the nature of partial evaluation: depending on the availability of data, some parts of a program can be executed, while other portions which need undefined data values are left as residual results.

The most difficult problem faced when partially evaluating a program is controlling the evaluation process. Partial evaluation first requires the fixing of data values , usually

meaning that program parameters are set. Processing commences by executing all the code in which the necessary data is defined, while retaining the code which uses unknown data values. However, undefined values in a computation escalate during processing according to the data flow in the program: any value which uses an undefined value subsequently becomes undefined, and computations using it become suspendable. Program constructs such as loops thus present the possibility of the infinite generation of residual code. The user is given the responsibility of indicating the halting conditions for a particular application.

Most implementations of partial evaluators have been done in either Lisp or Prolog. This is because programs can be treated as data quite naturally within these languages, which is an advantage when implementing meta–programming environments. An early Lisp–based implementation of a partial evaluator is in [Beckman *et al.* 76], which is used for program optimization. [Futamura 71] first suggested the use of partial evaluation in the creation of compiler–compilers, which are programs that transform interpreters into compilers. [Jones *et al.* 85] implemented a partial evaluator in Lisp which was used to generate compilers and a compiler–compiler. Two partial evaluators based in Prolog are given in [Venken 85] and [Takeuchi *et al.* 85]. The Venken system can process full Prolog programs; cuts and system predicates with side–effects are handled. However, control facilities are limited to simply recursion detection and predicate tagging. The Takeuchi system does not handle meta–logical operators such as cut, but has extensive facilities with which the user can control the partial evaluation.

Chapter 3

An Imperative Language Called Bruce

3.1 Design Motivations

This thesis concentrates on the semantics and transformation of imperative programs, which are those written in languages tailored around the architecture of von Neumann machines [Backus 78]. A von Neumann machine is an abstract simplification of the conventional computer, having a central processing unit (CPU), a memory or store, and a two–way data bus between them. Imperative languages consequently make predominant use of: (a) variables, which represent values in the store (b) control statements, which are extensions of the CPU's jump and test abilities, and (c) assignment statements, which imitate data passing between the CPU and store. All programs written in imperative languages share a fundamental computational methodology based on the above features, and the mappings between different imperative languages are usually trivial. A very general view of a computation on a von Neumann machine is:

1. the computer is initialized with variable values
2. as execution proceeds, the variable values are processed
3. execution eventually terminates with result values in particular variables.

BoolExpr:

b ::= $true \mid false \mid r(e_1, ..., e_k) \mid (b_1 \vee b_2) \mid (b_1 \wedge b_2) \mid (\neg b)$

where r is a relational operator (eg. <, >, <=, etc.),

Expr:

e ::= $x \mid c \mid f(e_1, ..., e_k)$

where x is a variable, c is a constant, f is a function (eg. +, *, /, etc.),

Figure 3.1: Syntax of expressions

Other side effects usually result during a computation, such as output to a printer or screen, or movement of a robot arm or disk head. However, our view of computation will be restricted solely to the observable effects to the program variable values kept in the store.

The rest of this chapter introduces an imperative language called *Bruce*[1]. Bruce is intended to be a generic language incorporating many traditional imperative features, such as destructive variable assignments, structured control mechanisms, and subprogram modules. For expository reasons, Bruce also includes branches or *goto* statements as a more primitive control construct. Although Bruce is by no means as robust as languages such as Pascal or "C", it is sophisticated enough to permit interesting and nontrivial programs to be written in it. In addition, the semantics and transformations we apply to Bruce programs later in this thesis should be easily extended to real programming languages used in production environments.

The definition of Bruce will be in two parts. Section 3.2 outlines the Bruce syntax. This is followed by a presentation of Bruce's operational semantics in section 3.3. This operational semantics will be the basis for the logical semantics introduced in chapter 4.

3.2 Syntax

[1] Brian Ross's Unfriendly Computational Environment

The syntax of Bruce is similar to that of Pascal [Wirth *et al.* 78] and "C"[Kernighan *et al.* 78]. For the sake of brevity, we forego minute syntactic details in the following definitions with no loss of generality. As a result, labels and constants are assumed to have conventional denotations.

Figure 3.1 has a BNF–style grammar indicating the construction of boolean and mathematical expressions. Most common relational and arithmetic operators may be used. We assume that conventional operator bindings are adopted, and that parentheses can be dropped without resulting in interpretive ambiguities.

Figure 3.2 has a BNF–style syntax of Bruce programs. Items enclosed in brackets, such as statement labels, are optional. Items with an overbar (as in $\bar{a}$) represent lists of items. An environment E contains all the programs to be executed in a particular application, and represents the entire computational environment. A program P is a conventional program or subprogram. A program is defined by a unique program name $Progname_i$, parameter and local variable definitions, and a program body. Parameters may be *variable* or *call by address* parameters (the vector $\bar{a}$), or *call by value* parameters (the vector $\bar{e}$). Note that there are no data types in Bruce. A chain Q (also referred to as a *train*) is a sequence of one or more statements separated by semi–colons. Each statement can be optionally labelled. Finally, a statement S represents a basic Bruce construct. Standard assignments and tests are allowed. *Skip* commands are null instructions which have no effect whatsoever. Control constructs such as *while* and *repeat* loops may be used, as well as more primitive constructs such as branches. Lastly, subprograms can be invoked with *call* statements, where the first list of parameters are variables to be passed by address, and the second list are expressions to be passed by value.

3.3 Operational Semantics

Establishing an operational semantics involves the definition of an abstract machine or interpreter. The style of this operational semantics is based on those in [de Bruin 81]

Environment:

$E ::= \cup P$

Programs:

$P ::= Progname_i(\bar{a}, \bar{e}) : [local\ \bar{x}]\ \{\ Q\ \}$

where $Progname_i$ and each variable name $v \in \{\bar{a}, \bar{e}, \bar{x}\}$ is unique in E

Chains:

$Q ::= [label_i :]\ S \mid [label_i :]\ S; Q$

where each label $label_i$ is unique in E

Statements:

$S ::= skip \mid v := e \mid if\ (b)\ then\ \{\ Q\ \}\ else\ \{\ Q'\ \} \mid if\ (b)\ then\ \{\ Q\ \} \mid$
$call\ p_i(\bar{v}, \bar{e}) \mid while\ (b)\ \{\ Q\ \} \mid repeat\ \{\ Q\ \}\ until\ (b) \mid$
$goto\ label_i$

where $b \in BoolExpr, e \in Expr, v \in$ *variable names*,
$p_i \in Programs$

Figure 3.2: Syntax of higher–level constructs

and [Cook 78]. This semantics is semi–formal in the sense that mathematical notation is descriptively useful in the operational definitions. However, there is no attempt in utilizing unnecessary rigor in this semantics, as the intent is to illustrate in elementary terms the effects of program statements on the store.

We will characterize a computation by its effect on the store, and in particular, the effect on the values of the program variables. A state,σ, is a static representation of the computer store at an instant in time of the computation, and is our basic semantic unit. σ is defined as a function which maps program variables to their current values in the domain space *Domain*:

$$\sigma : Var \rightarrow Domain$$

A variant $\sigma[e/v]$ of a state σ is a state σ' in which σ' differs from σ only in the mapping of variable v, which is mapped to e. $\sigma[\bar{e}/\bar{v}]$ represents the multiple application of a variant to a vector of elements. $\sigma[\mathcal{V}(\bar{e})/\bar{v}]$ represents the multiple application of a variant in which the valuations of $\bar{e}$ are substituted. A computation sequence, denoted Σ, is an ordered list of successive states produced during the execution of a program:

$$\Sigma =< \sigma_1, \sigma_2, ..., \sigma_k, ... >$$

A function $\mathcal{K}$ is defined as:

$$\mathcal{K}(\Sigma) = \begin{cases} \perp & if\ \Sigma\ does\ not\ terminate \\ \sigma_k & if\ \Sigma^i =< \sigma_0, ..., \sigma_k > \end{cases}$$

$\mathcal{K}$ simply returns the final result state of terminating computation sequences. A concatenation operator $\star$ will be used to construct computation sequences from other sequences. If

$$\Sigma^1 =< \sigma_a, ..., \sigma_b >$$

and

$$\Sigma^2 =< \sigma_y, ..., \sigma_z >$$

then

$$\Sigma^1 \star \Sigma^2 =< \sigma_a, ..., \sigma_b, \sigma_y, ..., \sigma_z >$$

1. $Comp \ll E, P,\ S \gg \sigma = Comp \ll E, P,\ S; skip \gg \sigma$

2. $Comp \ll E, P,\ skip \gg \sigma = < \sigma >$

3. $Comp \ll E, P,\ skip; Q \gg \sigma = < \sigma > \star Comp \ll E, P,\ Q \gg \sigma$

4. $Comp \ll E, P,\ x := e; Q \gg \sigma = < \sigma[\mathcal{V} \ll e \gg \sigma / x] \equiv \sigma' > \star Comp \ll E, P,\ Q \gg \sigma'$

5. $Comp \ll E, P, call\ P_i(\bar{v}, \bar{e}); Q \gg \sigma = < \sigma > \star (Comp \ll E, P_i, Q^i \gg \sigma' \equiv \Sigma^i) \star Comp \ll E, P, Q \gg \sigma''$
   where $P_i(\bar{x}, \bar{y}) : \{Q^i\} \in E$,
   σ_i is a fresh state for P_i,
   $\sigma' = (\sigma_i[\bar{v}/\bar{x}])[\mathcal{V}(\bar{e})/\bar{y}]$
   $\sigma'' = \alpha(\sigma, \mathcal{K}(\Sigma^i), \bar{v})$

6. $Comp \ll E, P,\ if(b) then\{Q_1\} else\{Q_2\}; Q_3 \gg \sigma = Comp \ll E, P,\ Q_1; Q_3 \gg \sigma$ (if $\mathcal{W} \ll b \gg = true$)
   $= Comp \ll E, P,\ Q_2; Q_3 \gg \sigma$ (if $\mathcal{W} \ll b \gg = false$)

7. $Comp \ll E, P,\ if(b) then\{Q_1\}; Q_2 \gg \sigma = Comp \ll E, P,\ if(b) then\{Q_1\} else\{skip\}; Q_2 \gg \sigma$

8. $Comp \ll E, P,\ while(b)\{Q_1\}; Q_2 \gg \sigma = Comp \ll E, P, if(b)\{Q_1; while(b)\{Q_1\}\}; Q_2 \gg \sigma$

9. $Comp \ll E, P, repeat\{Q_1\} until(b); Q_2 \gg \sigma = Comp \ll E, P, Q_1; if(\neg b)\{repeat\{Q_1\} until(b)\}; Q_2 \gg$

10. $Comp \ll E, P, goto\ l_i; Q \gg \sigma = < \sigma > \star Comp \ll E, P, Q' \gg \sigma$
    where $Q' = Buildenv(P, Q_i), l_i : Q_i \in P$

Figure 3.3: Definition of *Comp*

Now that the notational tools are defined, the actual operation semantics for Bruce can be established. The first step is to provide a means of interpreting expressions. Two functions, $\mathcal{V}$ and $\mathcal{W}$, evaluate arithmetic and boolean expressions respectively "in the usual way":

$$\mathcal{V} : Exp \rightarrow \sigma \rightarrow Domain$$
$$\mathcal{W} : BoolExp \rightarrow \sigma \rightarrow \{true, false\}$$

Here, *Domain* might refer to the natural numbers.

A function *Comp* is used to build Σ (figure 3.3). The state sequence Σ for a terminating computation starts at an initial state σ_0, and ends at a state σ_n. *Comp* takes as its arguments the environment E, a program P, a chain Q, and a state σ. E is a source where all available program definitions may be found – a necessity when processing subprogram

$$Buildenv(P,Q) = \begin{cases} \ll Q \gg & \text{if } P : \{Q';Q\} \\ \ll Q; while(b)\{Q';Q\}; Buildenv(P,Q^*) \gg & \text{if } Q \text{ belongs to a } while \\ \ll Q; if(\neg b)\{repeat\{Q';Q\}until(b)\}; Buildenv(P,Q^*) \gg & \text{if } Q \text{ belongs to a } repeat \\ \ll Q; Buildenv(P,Q^*) \gg & \text{if } Q \text{ belongs to an } if \end{cases}$$

where $S\{Q';Q\};Q^*$,
ie. S is the parent statement containing subchain Q, and Q^* is the chain following S

Figure 3.4: Definition of *Buildenv*

calls. P defines which labels may be used as branch destinations. The chain Q is the program code currently being processed. σ is the current state of the store.

Some features of *Comp* are worth explanation. Line 1 simplifies *Comp* so that operational definitions are not duplicated for single– and multiple–statement chains. Lines 2 and 3 define the *skip* instruction. In line 4, an assignment results in a state variant's creation of a new state σ', which is used by the following chain. Tests (lines 6 and 7) are handled by the execution of the appropriate chain of code according to the value of the test expression. While loops (line 8) are processed by reducing the construct into a test. The body of the test is composed to simulate the iterative nature of the loop should the loop test be positive. Repeat loops (line 9) are handled similarly.

Subprogram invocation (line 5) is more complex. A stack mechanism is assumed to exist so that nested subprogram calling, including recursion, can occur. When calling a new program, the pre–existing arguments to *Comp* are implicitly saved on the stack, and a "fresh" state σ_i is created for the new subprogram to be executed. Before executing the subprogram, multiple variants are performed on σ_i in order to initialize the store with the values of the variable and value parameters being passed. The subprogram is then executed, eventually producing a state sequence Σ^i. Finally, changes in the variable parameters which may have occurred during the subprogram call must be accounted for. This is a mechanical process which we give to a function α. α takes as arguments the old state σ, the final state of Σ^i, and the variable parameter vector $\bar{v}$, and returns a new state with σ set appropriately. State variants would be used in the creation of this new state.

```
p([a, b], [ ]) {
  a := 1;
  goto g1;
  while (a < b) {
    repeat {
      g1 : a := a + 2
    } until (a == b);
    a := a + 3;
  };
  a := a + 4
}
```

Figure 3.5: Example Program

```
a := a + 2;
if (¬(a == b)) {
  repeat {
    a := a + 2
  } until (a == b);
};
a := a + 3;
while (a < b) {
  repeat {
    a := a + 2
  } until (a == b);
  a := a + 3;
};
a := a + 4
```

Figure 3.6: Buildenv Output

When processing branches (line 10), *Comp* makes use of an auxiliary function *Buildenv* (figure 3.4). The branch itself results in no change to the state, and the chain following the branch is ignored, which reflects the destructive nature branches have on command sequencing. Following the branch is the computation of the chain of code at $label_i$. Using *Buildenv*, the control context normally found at the branch destination is recreated. *Buildenv* creates a chain of program code by taking the chain at the destination label and appending appropriate control code for successive parent constructs, resulting in a chain of code which has an appropriate control context. For example, figure 3.5 is a Bruce program containing a branch into two nested loops. Figure 3.6 contains the code generated by *Buildenv* for the label $g1$.

Once *Comp* is defined, we complete the operational semantics of a program using a function $\mathcal{M}$, which is the *meaning* function of a program:

$$\mathcal{M} \ll P \gg \sigma_0 = \mathcal{K}(Comp \ll E, Q, Q \gg \sigma_0)$$

where $P : \{Q\} \in E$. This can be thought of as the initial module executed in order to start *Comp*'s evaluation of the program.

In summary, the Bruce abstract machine (BAM) executes by code manipulation ($Comp$), expression evaluation ($\mathcal{V}$ and $\mathcal{W}$), and writing values to the store (state variants). The feature of this operational semantics worth special attention is the way control is processed by *Comp*. If one were to observe the contents of *Comp*'s chain argument, one would see an ever–changing chain of program instructions. This chain reflects the control context of the program at any particular moment in the computation. This somewhat parallels the way people "hand compute" programs, in that tracing a program's flow of control involves the implicit textual context of statements, in particular, the constructs in which they are nested and the code textually following them. The Bruce machine does this by explicitly constructing this context.

Chapter 4

A Horn Clause Semantics

This chapter introduces a new semantic methodology for imperative programs. In particular, a Horn clause semantics is presented for the Bruce language of chapter 3. Section 4.1 discusses some of the philosophical motivations behind the design of the semantics. The logical semantics for basic Bruce programs, that is, ones in which branches are not used, is given in section 4.2. The semantics of branches is then detailed in section 4.3. Section 4.4 discusses the interpretation of the semantics as logic programs, and addresses some completeness issues. Some examples of actual translations of Bruce programs conclude the chapter in section 4.6.

4.1 Design Goals

There are a number of design motivations for the logical semantics being introduced in this chapter. Since computation is to be defined solely by the state of the store, the minimal requirement for the semantics is that it has enough expressive capability with which to describe this characterization of computation. This requirement should not be an obstacle, as predicate logic has proved to be a powerful tool in the semantics of computation and programming languages (see section 2.1.2).

Our main motivation, however, is to define a semantic system which allows efficient and powerful transformations of imperative–style programs, namely, those written in Bruce.

The fact that the semantics is to be used in a program transformation environment means that its semantic scope can be constrained somewhat, especially when compared to the scope of semantic systems used in program verification environments. The program transformation system to be used in this thesis will take as input an imperative program, and will produce a transformed, optimized program as output. A key assumption is that the source program being input is to be considered complete and correct, both syntactically and semantically. This assumption greatly simplifies the scope of the semantics, as we do not have to deal with such issues as the well–formedness of the source code, the termination of mechanisms, the evaluability of expressions, and the matching of data types. More general semantic systems, such as that presented in [Hehner 84a], address such issues.

A major goal of the semantics is that there should exist a strong and natural relationship between the source language and its corresponding semantic representation. The predicative programming paradigm offers a solution in this regard. The scheme is to define for every construct of the source language a predicative meaning. This proves to be advantageous when converting between the source and logic, as program constructs can be considered to be predicate labels for their corresponding semantic definitions. Such a mapping between the source language and logic can be utilized in both directions, which is an ideal situation since such conversions will be required by the program transformation process.

Finally, it is prudent to consider the recent accomplishments in defining models of computation for first–order predicate logic. In particular, work in logic programming has shown the suitability and practicality of Prolog in various theoretical and practical applications. What is especially worth noting is that Prolog is well suited to many types of program transformations. A program written in pure Prolog (one without meta–logical operators) can have many types of transformations applied to it, in fact, any transformation which can be proved to be logically valid. Ascribing a logic programming model to our semantics means that a richer, sounder, and more automatable transformation system might be possible.

4.2 Basic Bruce Semantics

We now present the semantics of Bruce programs having no branches[1]. In section 3.3 a computational environment E is defined as a particular set of programs. In logic, this environment defines a theory

$$Th(E) = (B, A)$$

where $B = (F, P)$ is a predicative basis, and A is a set of axioms. The basis is composed of a set of function symbols F and a set of predicate symbols P. F contains all the arithmetic, relational, and boolean functors, as well as the domain dependent constants, used in Bruce. P contains the atoms *asgn* and *callasgn*, the propositional constants *true* and $false$, plus additional predicates dependent upon the structure of the programs in E. The axioms A, collectively known as the *program predicate*, are created through the principles of predicative programming. The composition of these axioms reflect the syntactic structure of the program, and will be detailed shortly.

A *mechanism* is a basic module of computation which has observable, well defined behavior. Mechanisms are defined by particular syntactic structures of the programming language whose behavior have been predefined in logic. Bruce constructs which exhibit well defined behavior, such as programs, assignments, if tests, loops, and branches, are treated as mechanisms. Mechanisms are also defined by computational events which occur during the execution of a program, such as the evaluation of value parameters during subprogram invocation. Mechanisms fall into two categories: (1) elementary mechanisms, such as assignments, which are described in logic by atoms, and (2) complex mechanisms, which is any construct which requires a predicative definition.

Satisfiability in this logical semantics denotes that a mechanism is *implementable*. An unsatisfiable program predicate, one which evaluates to logical $false$, is *unimplementable* or *underspecified*. A consequence of this convention is that the logical values *true* and $false$ are themselves program specifications, being the universally satisfiable and unsatis-

[1]This semantics is used in the initial stages of defining the semantics of program branches in section 4.3.

fiable program specifications respectively.

We restrict our logical semantics so that all axioms for complex mechanisms are written as *Horn clauses.* A Horn clause is a formula of predicate logic having the form

$$\forall(H \Leftarrow C_1 \wedge C_2 \wedge ... \wedge C_k)$$

where H is termed the *head*, and the conjunction of C_i's is called the *body.* The C_i's are typically input–output relations in most axioms. The informal semantics of this formula is: if the relations C_i all hold for some particular instances of memory states, then H holds. The C_i relations which represent complex mechanisms require their own subsequent Horn clause definitions, and so on. A program predicate thus has the form:

$$\models \forall((P \Leftarrow S_1 \wedge ... \wedge S_k) \wedge (S_1 \Leftarrow T_1 \wedge ... \wedge T_m)... \wedge (T_j \Leftarrow W_1 \wedge ...) \wedge ...)$$

where the S_i's, T_i's, and W_i's are input–output relations.

Input–output relations have included as arguments two state vectors, one for the input state of the computation, and the other for the output state. For example, in the relation $if2(\bar{v}_i, \bar{v}_f)$ the vectors $\bar{v}_i$ and $\bar{v}_f$ represent the initial and final store states during the execution of the mechanism $if2$. Each input–output vector represents the state σ of the computer's store at a particular instant in time. The elements of input–output vectors are logical variables whose scope is the clause within which they are used, and whose domains are the value domains of the variables used in Bruce. Each element represents the value of a particular variable parameter, value parameter, or local variable during an instant of the computation. Under some circumstances vector elements might be set to domain values[2]. Such values represent implicit assignments to the Bruce objects that the vector elements represent.

Figure 4.1 presents the axiomatic semantics of Bruce constructs without branches. The semantics are logical descriptions of the behavior of *Comp* in Bruce's operational semantics (figure 3.3). Generally speaking, each complex mechanism in a program is represented by

[2]We will see such a phenomenon occur in chapter 6.

1. Programs:

$$Progname(\bar{a},\bar{e}) : local\ \bar{x}\ \{\ Q\ \} \stackrel{\text{def}}{=} \models Progname(<\bar{a}_i,\bar{e}_i>,<\bar{a}_f,\bar{e}_i>) \Leftarrow Q(<\bar{a}_i,\bar{e}_i,\bar{x}_i>,<\bar{a}_f,\bar{e}_f,\bar{x}_f>)$$

2. Chains:

$$Q \equiv S_1; S_2; ...; S_k \stackrel{\text{def}}{=} \models Q(\bar{v}_i,\bar{v}_f) \equiv S_1(\bar{v}_i,\bar{v}_1) \wedge S_2(\bar{v}_1,\bar{v}_2) \wedge ... \wedge S_k(\bar{v}_{k-1},\bar{v}_k)$$

3.Null operation:

$$skip \stackrel{\text{def}}{=} true$$

4. Assignment:

$$x := e \stackrel{\text{def}}{=} \models asgn(\bar{v}_i, v_x, e, \bar{v}_f)$$

5.Program invocation:

$$call\ P(<v_1,...,v_j>,<e_1,...,e_k>) \stackrel{\text{def}}{=} \models callasgn(\bar{v},x_1,e_1) \wedge ... \wedge callasgn(\bar{v},x_k,e_k) \wedge P(<v_1,...,v_j,x_1,...,x_k>,\bar{v}_f)$$

6.(a) Test:

$$if\ (b)\ then\ \{\ Q_1\ \}\ else\ \{\ Q_2\ \} \stackrel{\text{def}}{=} \models (if_i(\bar{v}_i,\bar{v}_f) \Leftarrow b \wedge Q_1(\bar{v}_i,\bar{v}_f)) \wedge (if_i(\bar{v}_i,\bar{v}_f) \Leftarrow \neg b \wedge Q_2(\bar{v}_i,\bar{v}_f))$$

6.(b) Test:

$$if\ (b)\ then\ \{\ Q_1\ \} \stackrel{\text{def}}{=} \models (if_i(\bar{v}_i,\bar{v}_f) \Leftarrow b \wedge Q_1(\bar{v}_i,\bar{v}_f)) \wedge (if_i(\bar{v}_i,\bar{v}_i) \Leftarrow \neg b)$$

7. While loops:

$$while\ (b)\ \{\ Q\ \} \stackrel{\text{def}}{=} \models (while_i(\bar{v}_i,\bar{v}_f) \Leftarrow b \wedge Q(\bar{v}_i,\bar{v}_2) \wedge while_i(\bar{v}_2,\bar{v}_f)) \wedge (while_i(\bar{v}_i,\bar{v}_i) \Leftarrow \neg b)$$

8. Repeat loops:

$$repeat\ \{\ Q\ \}\ until\ (b) \stackrel{\text{def}}{=} \models (repeat_i(\bar{v}_i,\bar{v}_f) \Leftarrow Q(\bar{v}_i,\bar{v}_2) \wedge \neg b \wedge repeat_i(\bar{v}_2,\bar{v}_f)) \wedge (repeat_i(\bar{v}_i,\bar{v}_f) \Leftarrow Q(\bar{v}_i,\bar{v}_f) \wedge b)$$

Figure 4.1: Logical semantics of Bruce constructs (no branching)

a uniquely named predicate. Our scheme is to generate unique predicate names through numbering. For example, a program with three if tests has generated for it three predicates named $if1$, $if2$, and $if3$. The only relevance that a predicate name has is that it is unique; the label name itself is only for notational convenience. When a predicate is being converted into Bruce code, the fact that the predicate might be called $if2$ does not necessarily imply that it represents an if test, since only the logical form of the predicate is used for the translation.

Line 1 specifies the high level semantics of programs, as well as the composition of input–output vectors. A program *Progname* is defined by the predicative meaning of its main program chain Q. The change to the store when executing *Progname* is indicated by the changes to the input–output vectors used within the relations for *Progname* and Q. Within all predicates, input–output vector elements are mapped to their Bruce objects by their positions in the vector. In particular, a program *Progname* having variable parameters $\bar{a}$, value parameters $\bar{e}$, and local variables $\bar{x}$ uses the vector format $< \bar{a}, \bar{e}, \bar{x} >$ throughout the rest of the program predicate. The manner in which vector values are shared between *Progname* and Q depends upon the nature of the store objects in question. Since local variables ($\bar{x}$) are only active during the execution of the program body, they are not used in the predicate header. In addition, value parameters ($\bar{e}$) are not changed by the program's execution; thus their final values in the program header are the same as their initial values.

The composition of statements is represented in Bruce by chains, which are statements separated by the sequencing operator ";". In logic, this is represented by conjuncting the relations for each statement in the chain (line 2). Intermediate input–output vectors are used to "pass" the value of the store among the relations. As was mentioned above, the relations themselves often have auxiliary definitions.

Elementary relations, represented by atoms, define the behavior of elementary mechanisms. There are four elementary relations representing assignments (line 4), boolean tests, *skip* statements (line 3), and value parameter evaluation when invoking subprograms. $asgn(\bar{v}_i, v_x, e, \bar{v}_f)$ is an elementary relation which represents a state variant. v_x

is a logical variable which represents the element in the output vector $\bar{v}_f$ corresponding to the store location for variable x. $\bar{v}_f$ is the state which results after substituting the value of the logical variable v_x in $\bar{v}_f$ with the value of expression e in the context of input state $\bar{v}_i$. Boolean tests, which are represented by the boolean expression b throughout the definitions, are also elementary relations. They implicitly use the input vector appropriate to the context of the test within each definition. *skip* instructions are simply defined by logical *true*.

Subprogram invocation (line 5) is represented by an input–output relation for the subprogram in question. The input and output vectors within this relation are constructed to match the particular parameter conventions of the program being invoked. An input–output vector with the variable parameters followed by the value parameters is constructed, as is shown in the definition. However, the value parameter elements first require that each expression being passed is evaluated with respect to the current state of the store ($\bar{v}$). $callasgn(\bar{v}, x, e)$ is an elementary relation which equates the value of expression e in the context of the next state $\bar{v}$ with a logical variable x. These expression values are then placed into the subprogram relation's input vector. The output vector of the subprogram relation represents the final state of the parameters after execution has completed. Since only the variable parameters could have possibly changed, the final values of these elements are appropriately distributed to the current program state. This distribution of parameter values means that any relation following the subprogram call has the final parameter values placed within its input vector. Should the call be the last statement of a chain, then the values may affect the output vector of the predicative header to which the chain belongs.

Test constructs have straight–forward definitions. A Bruce *if* construct with *then* and *else* alternatives (line 6(a)) always has one of the two test bodies executed according to the outcome of the boolean test. The corresponding logical semantic mirrors this fact: if the boolean test is *true*, then chain Q_1 is executed, which is abbreviated in the definition by an input–output relation for the chain. Otherwise chain Q_2 is executed. In actuality, these chains are represented by conjunctions of relations (as in line (2)). An *if* construct

with no *else* alternative is represented by a clause with a null body (line 6(b)). This is a simplified representation for a chain containing the *skip* statement used by *Comp*, which in our semantics is defined as *true* (line 3).

While loops (line 7), are defined by two clauses, one defining the iterative case, and the other the exit condition. The iterative clause specifies a relation which holds if the *while* test is *true*, in which case the loop body is executed, and the loop itself iterates. Iteration is defined by a recursive restatement of the relation for the *while* construct. The treatment of *repeat* loops (line 8) is similar to that of *while* loops.

4.3 Program Branches

Branches add a dimension of complexity to the semantics of Bruce. A program branch has consequences on the composition of commands, as it destroys statement sequencing. In addition, branching alters the logical semantics of constructs. For example, a branch can be used to jump out of one or more nested loops – an action which dramatically corrupts their normal semantics. The semantics of such a loop is now dependent on the contents of that loop's body. The existence of branches therefore means that the syntactic "surface structure" of a statement is not sufficient for derivation of the semantics. Instead, the semantics are now dependent on code within the statement.

Two distinct phenomena occur during the execution of a branch. First, when a branch is executed, every mechanism which is active is in effect halted. In other words, the current control context of the program is terminated. Second, a new control context found at the branch destination is then used to commence subsequent execution.

Defining the logical semantics of branches is likewise done in two steps: defining new control contexts, and terminating old control contexts. First, the activation of new control contexts is defined. Our treatment of branches makes use of the concept of *continuations* [Clint *et al.* 72] [Stoy 77] [Tennent 81]. In a language with branching, a continuation for a statement is the expected results of all the program computation which will temporally

follow the statement's execution. The meaning of a branch is therefore the direct effect on the store, which is none, plus the effect of the continuation which, in the case of branches, is the effect of executing the code at the statement label. We treat a branch as a call to the *label program* identified by the branch destination label:

$$goto\ label_i \stackrel{\text{def}}{=} label_i(\bar{v}_i, \bar{v}_f)$$

Here, $\bar{v}_f$ represents the final state of the store for the rest of the mechanism's execution, in effect, the continuation for the program. The label program itself is defined in Bruce's operational semantics by the output of the *Buildenv* function, which recreates the control context of a labelled statement. Using predicative programming, we define the logical meaning of a branch destination by simply generating a predicate for the code created by *Buildenv*. Hence, we define for each labelled statement in a program a label program predicate:

$$label_i(\bar{v}_i, \bar{v}_f) \Leftarrow Q(\bar{v}_i, \bar{v}_f)$$

in which $Q(\bar{v}_i, \bar{v}_f)$ is the predicative meaning of the code generated by *Buildenv* for that label.

The next step is to account for the termination of the current control context caused by a branch. The control context of any statement of a program can be determined as follows. Consider a statement S whose control context is to be found. We expand the program predicate, starting from the main chain, by replacing each parent construct of S by its logical definition. This continues until we reach S. Next, distributivity about logical *or*($\vee$) is used to create a disjunction of terms. This distribution about "$\vee$" should be done so that the sequencing of terms is not altered, as this sequence reflects the relative order in which tests and statements are executed during computation. Each disjunct is composed of a mixture of chains of code conjuncted with boolean tests, which is a situation acceptable in predicative programming given the interchangeability of code and logic. Semantically, each disjunct describes a trace of the program control that can occur during execution. The control context of the statement S is represented by the conjunction of relations following each instance of S in the predicate.

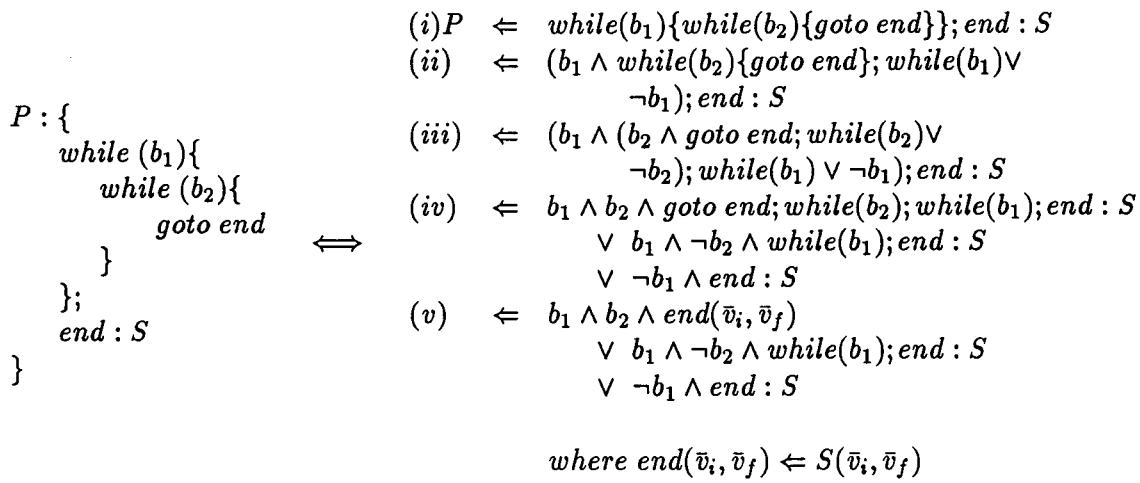

Figure 4.2: Branching out of two loops

To determine the effect that a branch has on the control context, we must first expand the program predicate as previously described. This expansion produces the semantic description of the control context following the branch. Given the appearance of a branch in a disjunct of the predicate, the termination of the control context at that branch is defined as

$$R_i \wedge ... \wedge R_{j-1} \wedge goto\ label_i \wedge R_j \wedge ... \wedge R_k \stackrel{\text{def}}{=} R_i \wedge ... \wedge R_{j-1} \wedge label_i(\bar{v}_i, \bar{v}_f)$$

where $R_j \wedge ... \wedge R_k$ is the control context for the branch. The disappearance of $R_j \wedge ... \wedge R_k$ from the predicate is equivalent to removing its effect on control. This restructuring is then applied to each disjunct containing the branch. The net effect of the restructuring is to "pre–compile" the effects of the branch on sequencing into the predicate.

In summary, the predicative restructuring procedure is as follows:

1. Expand statements by their semantic definitions until a branch is revealed.

2. Distribute the chains of code over the logical *or* ($\vee$) connective until we get a disjunction of terms.

3. Remove terms and chains of code which sequentially follow the branch.

4. Replace the branch by a call to its label program.

5. (Optional) Simplify the disjunction by factoring common terms using distributivity.

Of course, logical manipulations can be economized and predicative equivalences can be preserved when expanding and simplifying the program predicate.

Figure 4.2 shows a program where a branch removes control out of two nested loops (relations are abbreviated). One expansion of each loop (step (iii)) is required in order to produce the chain containing the branch. In step (iv), the chains are distributed through the internal disjunctions, creating a single disjunction of chains. Finally, step (v) shows the removal of the chain following the branch, and the replacement of the branch with a call to the label program (see first disjunct).

With respect to the theory $Th(E)$ outlined in section 4.2, the incorporation of branches in the semantics means that the predicative basis B is supplemented with relations for the label programs, and that the axioms A are supplemented with label program predicates. However, an unfortunate aspect of branches is that we can no longer rely on a straightforward inspection of the program syntax in order to derive program axioms, as is used by the basic semantics, but must instead use the restructuring procedure outlined above. Even though the methodology of deriving axioms has become much less elegant, we still end up with a semantic description for the program which is as logically sound as for programs without branches. Only the formula for axiom derivation has suffered in terms of complexity.

4.4 Logic Program Interpretation and Completeness

Given a program P, we define for it a set of Horn clauses which define the input–output behavior of P and its constituent parts. One last clause to be added to the semantics is a

goal clause, which represents an invocation of the program:

$$\models \neg P(\bar{v}_i, \bar{v}_f)$$

The input parameters to P would be appropriately set in the $\bar{v}_i$ input vector. The axioms and goal clause for P has a logic program interpretation [Clocksin *et al.* 81], and has both a procedural and declarative semantics [van Emden *et al.* 76]. Thus, through the logic programming paradigm, the complete logical semantics of a program can be treated as a logic program, and can be executed using SLDNF resolution [Lloyd 84].

In order to more closely ally our semantics with the logic programming concept, we henceforth use the Prolog notation from [Clocksin *et al.* 81] with which to write the semantics. The typical form of a Horn clause is now:

$$P \Leftarrow C_1 \wedge C_2 \wedge ... \wedge C_k \quad \stackrel{\text{def}}{=} \quad P :- C_1, C_2, ..., C_k.$$

Also, input-output vectors will now be written in terms of Prolog lists:

$$\bar{v} \equiv < v_1, ..., v_k > \quad \stackrel{\text{def}}{=} \quad [v_1, v_2, ..., v_k]$$

We assume that any Prolog interpreter used to execute the semantics has available to it the arithmetic and boolean expression evaluators $\mathcal{V}$ and $\mathcal{W}$ from the operational semantics.

The logic program interpretation of the semantics also makes the axioms logically stronger. The problem with the "bare" logical interpretation of the axioms is that the logical implications used in the formulas are only partial specifications of the behavior of mechanisms. For example, a mechanism M described by two clauses is logically equivalent to the following:

$$\models (M \Leftarrow C_1) \wedge (M \Leftarrow C_2) \quad \equiv \quad \models M \Leftarrow (C_1 \vee C_2)$$

This states that M is satisfied if C_1 or C_2 are satisfied, or both. However, this is only a partial specification of the behavior of M, as when C_1 and C_2 are both false, M is still satisfied. What we need is for the implication in each predicate to be read as a logical equivalence, or *iff*.

Fortunately, when the semantics are interpreted as a logic program, the predicates become logically complete specifications. The *completion* of a logic program is a logical characterization given the effects of SLDNF resolution on program interpretation [Lloyd 84]. If

$$p(t_1, ..., t_n) \Leftarrow L_1, ..., L_m$$

is one of k clauses defining p, we first transform each clause into

$$p(x_1, ..., x_n) \Leftarrow \exists y_1 ... \exists y_d((x_1 = t_1) \wedge ... \wedge (x_n = t_n) \wedge L_1 \wedge .. \wedge L_m)$$

where $x_1, ..., x_n$ are new variables, and $y_1, ..., y_d$ are the variables of the original clause. The predicate for p is thus

$$p(x_1, ..., x_n) \Leftarrow E_1 \vee ... \vee E_k$$

where each E_i has the form

$$\exists y_1 ... \exists y_d((x_1 = t_1) \wedge ... \wedge (x_n = t_n) \wedge L_1 \wedge .. \wedge L_m)$$

The completed definition of p is then

$$\forall x_1 ... \forall x_n(p(x_1, ..., x_n) \Leftrightarrow E_1 \vee ... \vee E_k)$$

This is the logical meaning of a predicate given the effects of SLDNF resolution, and is assumed implicit in all pure Prolog programs. This means that our logical axioms are complete specifications.

4.5 Translating Logic to Bruce

Once a program is defined in terms of the preceding semantics, its semantic representation can be transformed and analyzed using the laws of first–order predicate logic. In a program transformation environment, the intent is for the semantics to be manipulated to produce a result which yields greater efficiency or some other desired quality not found in the original source program. This optimized representation should then be transformed back into the source language using predicative programming.

Compiling a logical representation back into Bruce is the inverse of generating the logic from Bruce. Given a set of axioms, they are matched "backwards" to the Bruce constructs using the axiom schema in figure 4.1 of section 4.2. However, it is possible that a direct match is not possible between the axioms being matched and the definitions for Bruce constructs. As a consequence, logical manipulation of the axioms might be required until a match can be obtained.

The conversion of basic Bruce constructs is straight–forward. However, the conversion of branches and branch destinations needs clarification. Branches are treated as label program relations in our semantics. These label programs have embedded in them the continuation of the computation, or in other words, the result of the rest of the computation. As a result, there should never be instances of other goals making use of the output vector of a branch relation. Instead, the output of a label relation is given to the output of the clause within which it resides:

$$P(\bar{v}_i, \bar{v}_f) \Leftarrow C_1(\bar{v}_i, \bar{v}_2) \wedge ... \wedge Branch(\bar{v}_k, \bar{v}_f)$$

Therefore, any complex relation (that is, one with a predicative definition) which is the final goal of a clause can be considered as a branch. The branch destination is simply the Bruce code derived for the predicate describing the branch relation, prepended with a unique destination label. The placement of the destination code within the Bruce program can become somewhat involved, as the code should not interfere with the control of other program constructs. A solution to this is to unfold the first instance of a branch. When a branch relation is first discovered, instead of creating a branch statement to it, the relation is replaced by the code defining the destination, which is labelled with a destination label. Subsequent branch relations referring to this destination will simply refer to this unfolded code, and can be replaced with branches to it.

An example of branch translation is in figure 4.3. Predicate A contains the logical semantics to be converted. Predicate B has the predicate for c_1 converted to its Bruce realization. This assignment is unfolded into p in predicate C, and is given the label c_1. D has the reference to c_2 replaced by its equivalent definition. Finally, this reference to c_1 is

$$
\begin{array}{rl}
A: \models (p(\bar{v}_i, \bar{v}_f) & \Leftarrow c_1(\bar{v}_i, \bar{v}_2) \wedge c_2(\bar{v}_2, \bar{v}_f)) \\
\wedge (c_1(\bar{v}_i, \bar{v}_f) & \Leftarrow asgn(\bar{v}_i, X, Y+1, \bar{v}_f)) \\
\wedge (c_2(\bar{v}_i, \bar{v}_f) & \Leftarrow c_1(\bar{v}_i, \bar{v}_f))
\end{array}
$$

$$
\begin{array}{rl}
B: \models (p(\bar{v}_i, \bar{v}_f) & \Leftarrow c_1(\bar{v}_i, \bar{v}_2) \wedge c_2(\bar{v}_2, \bar{v}_f)) \\
\wedge (c_1(\bar{v}_i, \bar{v}_f) & \Leftarrow x := y+1) \\
\wedge (c_2(\bar{v}_i, \bar{v}_f) & \Leftarrow c_1(\bar{v}_i, \bar{v}_f))
\end{array}
$$

$$
\begin{array}{rl}
C: \models (p(\bar{v}_i, \bar{v}_f) & \Leftarrow c1 : x := y+1 \wedge c_2(\bar{v}_2, \bar{v}_f)) \\
\wedge (c_2(\bar{v}_i, \bar{v}_f) & \Leftarrow c_1(\bar{v}_i, \bar{v}_f))
\end{array}
$$

$$
D: \models (p(\bar{v}_i, \bar{v}_f) \Leftarrow c1 : x := y+1 \wedge c_1(\bar{v}_2, \bar{v}_f))
$$

$$
E: \models (p(\bar{v}_i, \bar{v}_f) \Leftarrow (c1 : x := y+1; goto\ c1)
$$

Figure 4.3: Converting Branches

$$
\begin{array}{l}
prog([x],[\,]) : \{ \\
\quad x := x+1; \\
\quad x := x+2; \\
\quad x := x+3 \\
\}
\end{array}
\quad \Longleftrightarrow \quad
\begin{array}{l}
prog([X1],[X2]) :- \\
\quad asgn([X1],X3,X1+1,[X3]), \\
\quad asgn([X3],X4,X3+2,[X4]), \\
\quad asgn([X4],X2,X4+3,[X2]).
\end{array}
$$

Figure 4.4: Consecutive assignments

replaced with a branch, since the code for c_1 has already been converted.

4.6 Example Translations

This section presents some example translations of Bruce programs into logic, and vice versa. All the examples are generated by an automatic translation system, particulars of which are discussed in chapter 5.

Figure 4.4 contains a simple Bruce program with three consecutive assignments to a variable parameter x, along with its semantics. The sequential nature of the Bruce program is captured in the semantics by the naming of the logical variables in the input–output vectors. The result of the final assignment is given to the header for *prog*. The converted result for the semantics is identical to the original program.

```
p([y],[x,n]) : {
   y := 1;
   while (n > 0) {
      n := n − 1;
      y := y * x
   }
}
```

$\Longleftrightarrow$

$p([Y1,X1,N1],[Y2,X1,N1])\ :-$
$\quad asgn([Y1,X1,N1],Y3,1,[Y3,X1,N1]),$
$\quad while1([Y3,X1,N1],[Y2,X2,N2]).$

$while1([Y1,X1,N1],[Y2,X2,N2])\ :-$
$\quad N1 > 0,$
$\quad asgn([Y1,X1,N1],N3,N1-1,[Y1,X1,N3]),$
$\quad asgn([Y1,X1,N3],Y3,Y1*X1,[Y3,X1,N3]),$
$\quad while1([Y3,X1,N3],[Y2,X2,N2]).$

$while1([Y1,X1,N1],[Y1,X1,N1])\ :-$
$\quad \neg N1 > 0.$

Figure 4.5: While loop

```
p([y],[x,n]) : {
   y := 1;
   while (n > 0) {
      n := n − 1;
      y := y * x
   }
}
```

– OR –

```
p([y],[x,n]) : {
   y := 1;
   while1 : if (n > 0) {
      n := n − 1;
      y := y * x;
      goto while1
   }
}
```

Figure 4.6: Two alternate Bruce realizations

A more complex Bruce program is in figure 4.5. Two clauses, collectively defining the predicate *while*1, describe the *while* loop. Because changes to the value parameters x and n are not available to any calling mechanisms, the two input–output vector elements corresponding to these objects are not altered in the header for p. Two possible Bruce realizations for the semantics are in figure 4.6. The first program is the same as the original source. The second one, however, replaces the *while* loop by a branch and test mechanism.

Figure 4.7 contains the treatment of subprogram invocations. The four expressions making up the call–by–value argument list are each given *callasgn* relations, and the logical variables representing the valuation of the expressions are passed to the subprogram. Note the distribution of the new variable argument values into the subsequent assignments occurring after the subprogram call.

```
progA([a,b,c],[x,y,z]) :
local[games,work] {
   x := 3;
   work := 6;
   call proc([a,b,work],[5,x,y,(3 + 4)]);
   c := 8;
   z := 10
}

proc([q,w,e],[rr,r,t,y]) : {
   q := w
}
```

```
progA([A1,B1,C1,X1,Y1,Z1],[A2,B2,C2,X1,Y1,Z1]) :-
   asgn([A1,B1,C1,X1,Y1,Z1,Ga1,Wo1],X2,3,[A1,B1,C1,X2,Y1,Z1,Ga1,Wo1]),
   asgn([A1,B1,C1,X2,Y1,Z1,Ga1,Wo1],Wo2,6,[A1,B1,C1,X2,Y1,Z1,Ga1,Wo2]),
   callasgn([A1,B1,C1,X2,Y1,Z1,Ga1,Wo2],T1,5),
   callasgn([A1,B1,C1,X2,Y1,Z1,Ga1,Wo2],T2,X2),
   callasgn([A1,B1,C1,X2,Y1,Z1,Ga1,Wo2],T3,Y1),
   callasgn([A1,B1,C1,X2,Y1,Z1,Ga1,Wo2],T4,3 + 4),
   proc([A1,B1,Wo2,T1,T2,T3,T4],[A2,B2,Wo3,T5,T6,T7,T8]),
   asgn([A2,B2,C1,X2,Y1,Z1,Ga1,Wo3],C2,8,[A2,B2,C2,X2,Y1,Z1,Ga1,Wo3]),
   asgn([A2,B2,C2,X2,Y1,Z1,Ga1,Wo3],Z2,10,[A2,B2,C2,X2,Y1,Z2,Ga1,Wo3]).

proc([Q1,W1,E1,Rr1,R1,T1,Y1],[Q2,W1,E1,Rr1,R1,T1,Y1]) :-
   asgn([Q1,W1,E1,Rr1,R1,T1,Y1],Q2,W1,[Q2,W1,E1,Rr1,R1,T1,Y1]).
```

Figure 4.7: Subprogram invocation

```
prog([x,y],[]) : {
   goto l1;
   l0 : x := y + 1;
   l2 : y := x + 2;
   goto l3;
   l4 : x := y + 4;
   goto end;
   l3 : y := x + 3;
   goto l4;
   l1 : x := y + 1;
   goto l2;
   end : y := x
}
```

$\Longleftrightarrow$

```
prog([X1,Y1],[X2,Y2]) :-
   l1([X1,Y1],[X2,Y2]).

l0([X1,Y1],[X2,Y2]) :-
   asgn([X1,Y1],X3,Y1 + 1,[X3,Y1]),
   asgn([X3,Y1],Y3,X3 + 2,[X3,Y3]),
   l3([X3,Y3],[X2,Y2]).

l1([X1,Y1],[X2,Y2]) :-
   asgn([X1,Y1],X3,Y1 + 1,[X3,Y1]),
   l2([X3,Y1],[X2,Y2]).

l2([X1,Y1],[X2,Y2]) :-
   asgn([X1,Y1],Y3,X1 + 2,[X1,Y3]),
   l3([X1,Y3],[X2,Y2]).

l3([X1,Y1],[X2,Y2]) :-
   asgn([X1,Y1],Y3,X1 + 3,[X1,Y3]),
   l4([X1,Y3],[X2,Y2]).

l4([X1,Y1],[X2,Y2]) :-
   asgn([X1,Y1],X3,Y1 + 4,[X3,Y1]),
   end([X3,Y1],[X2,Y2]).

end([X1,Y1],[X1,Y2]) :-
   asgn([X1,Y1],Y2,X1,[X1,Y2]).
```

Figure 4.8: Branching within a chain

```
prog([x,y],[]) : {
   x := y + 1;
   y := x + 2;
   y := x + 3;
   x := y + 4;
   y := x
}
```

Figure 4.9: Bruce equivalent

```
prog([y, n], [ ]) : local [x, eof] {
  n := 0;
  while (true) {
    call read([x], [ ]);
    n := n + 1;
    if (eof =:= true) {
      goto end
    };
    y := y + x
  };
  end : n := n − 1
}
```

Figure 4.10: End of file control scheme (initial)

A treatment of program branches is illustrated in figure 4.8. A label predicate is created for each labelled statement in the program. The converted Bruce program in figure 4.9 illustrates an interesting side–effect of the conversion algorithm: since the first instance of any branch is unfolded, the branches in the original program are lost.

A more complex branching scheme is illustrated in figure 4.10. This program uses a common control strategy for exiting loops based on an end–of–file condition. The reference to *eof* represents a system call which signals the end of the input stream. Figure 4.11 has the logical equivalent of the source, and the Bruce program derived from it is in figure 4.12. The old loop structure is lost, as it never represented a true while loop anyway. This new program outlines the control which was implicit in the original source. Note that it would have been possible to simplify the program predicate (for example, the tests *true* and $\neg true$) before generating the final program.

One final example is the program in figure 4.13, which contains a jump into the middle of a *repeat* loop. The program derived after translation into logic (figure 4.14) illustrates the complexity in control which may arise when using jumps.

$$
\begin{aligned}
&prog([Y1, N1], [Y2, N2]) \ :- \\
&\quad asgn([Y1, N1, X3, Eof3], N4, 0, [Y1, N4, X3, Eof3]), \\
&\quad while1([Y1, N4, X3, Eof3], [Y2, N2, X7, Eof7]).
\end{aligned}
$$

$$
\begin{aligned}
&while1([Y1, N1, X1, Eof1], [Y2, N2, X2, Eof2]) \ :- \\
&\quad true, \\
&\quad read([X1], [X4]), \\
&\quad asgn([Y1, N1, X4, Eof1], N6, N1 + 1, [Y1, N6, X4, Eof1]), \\
&\quad test1([Y1, N6, X4, Eof1], [Y2, N2, X2, Eof2]).
\end{aligned}
$$

$$
\begin{aligned}
&while1([Y1, N1, X1, Eof1], [Y1, N2, X1, Eof1]) \ :- \\
&\quad \neg true, \\
&\quad asgn([Y1, N1, X1, Eof1], N2, N1 - 1, [Y1, N2, X1, Eof1]).
\end{aligned}
$$

$$
\begin{aligned}
&test1([Y1, N1, X1, Eof1], [Y2, N2, X2, Eof2]) \ :- \\
&\quad Eof1 =:= true, \\
&\quad end([Y1, N1, X1, Eof1], [Y2, N2, X2, Eof2]).
\end{aligned}
$$

$$
\begin{aligned}
&test1([Y1, N1, X1, Eof1], [Y2, N2, X2, Eof2]) \ :- \\
&\quad \neg Eof1 =:= true, \\
&\quad asgn([Y1, N1, X1, Eof1], Y4, Y1 + X1, [Y4, N1, X1, Eof1]), \\
&\quad while1([Y4, N1, X1, Eof1], [Y2, N2, X2, Eof2]).
\end{aligned}
$$

$$
\begin{aligned}
&end([Y1, N1, X1, Eof1], [Y1, N2, X1, Eof1]) \ :- \\
&\quad asgn([Y1, N1, X1, Eof1], N2, N1 - 1, [Y1, N2, X1, Eof1]).
\end{aligned}
$$

Figure 4.11: Logical Form

```
prog([y,n],[ ]) : local [x,eof]{
  n := 0;
  while1 : if (true) {
    call read([x],[ ]);
    n := n + 1;
    if (eof =:= true) {
      n := n - 1
    }
    else {
      y := y + x;
      goto while1
    }
  }
  else {
    n := n - 1
  }
}
```

Figure 4.12: Derived result

```
prog1([x, y], [ ]) : {
   x := x + 1;
   y := y + 2;
   repeat {
      x := x + 3;
      l1 : y := x + 4
   } until (x =:= y);
   goto l1
}
```

$\Longleftrightarrow$

$$
\begin{array}{l}
prog1([X1,Y1],[X2,Y2]) \ :- \\
\quad asgn([X1,Y1],X3,X1+1,[X3,Y1]), \\
\quad asgn([X3,Y1],Y3,Y1+2,[X3,Y3]), \\
\quad repeat1([X3,Y3],[X4,Y4]), \\
\quad l1([X4,Y4],[X2,Y2]).
\end{array}
$$

$$
\begin{array}{l}
l1([X1,Y1],[X2,Y2]) \ :- \\
\quad asgn([X1,Y1],Y3,X1+4,[X1,Y3]), \\
\quad test1([X1,Y3],[X3,Y4]), \\
\quad l1([X3,Y4],[X2,Y2]).
\end{array}
$$

$$
\begin{array}{l}
test1([X1,Y1],[X2,Y2]) \ :- \\
\quad \neg X1 =:= Y1, \\
\quad repeat1([X1,Y1],[X2,Y2]).
\end{array}
$$

$$
\begin{array}{l}
test1([X1,Y1],[X1,Y1]) \ :- \\
\quad X1 =:= Y1.
\end{array}
$$

$$
\begin{array}{l}
repeat1([X1,Y1],[X2,Y2]) \ :- \\
\quad asgn([X1,Y1],X3,X1+3,[X3,Y1]), \\
\quad asgn([X3,Y1],Y3,X3+4,[X3,Y3]), \\
\quad \neg X3 =:= Y3, \\
\quad repeat1([X3,Y3],[X2,Y2]).
\end{array}
$$

$$
\begin{array}{l}
repeat1([X1,Y1],[X3,Y7]) \ :- \\
\quad asgn([X1,Y1],X3,X1+3,[X3,Y1]), \\
\quad asgn([X3,Y1],Y3,X3+4,[X3,Y3]), \\
\quad X3 =:= Y3.
\end{array}
$$

Figure 4.13: Jumping into a repeat loop

```
prog1([x, y], [ ]) : {
    x := x + 1;
    y := y + 2;
    repeat1 : repeat {
        x := x + 3;
        y := x + 4
    } until (x =:= y);
    l1 : y := x + 4;
    if (¬x =:= y) {
        goto repeat1
    };
    goto l1
}
```

Figure 4.14: Bruce equivalent

Chapter 5

A Bruce–Logic Translation System

A Bruce–Logic translation system has been implemented in CProlog [Pereira *et al.* 84]. This system comprises the front–end (logic generator) and back–end (Bruce generator or *Brucifier*) of a Bruce transformation system discussed in chapter 6. The complete source listing of the translation system can be found in the appendices.

5.1 The logic generator

The logic generator reads a Bruce program and generates its logical equivalent. It operates in three stages:

1. A Bruce program is read and parsed using a definite clause translation grammar (DCTG) [Abramson 84]. Then, an initial semantics is created directly from the parse tree of the DCTG. This semantics is complete and adequate in the case when no branching exists in the program, pointedly illustrating the semantic clarity of structured, branchless programming.

2. Should branching be found in the program, label program definitions are created for all the labelled code in the program.

```
chain ::= stat^^S
<:>
code(Code,In,Out,Defns) :: - S^^code(Code,In,Out,Defns).

chain ::= stat^^S, [';'], chain^^Q
<:>
code(Code,In,Out,Defns) :: -
    gen_chain(Code,In,Out,Defns,S,Q).

gen_chain(Code,In,Out,Defns,S,Q) : -
    S^^code(S1,In,X,D1),
    Q^^code(S2,X,Out,D2),
    fuse((S1,S2),Code),
    append(D1,D2,Defns).
```

Figure 5.1: Chain rules

3. Again, in the case of branches, the semantics are restructured to reflect the effect of branches on program control.

The rest of this section discusses these steps in more detail.

The creation of semantics for basic Bruce programs (section 4.2) is efficiently done through the use of a DCTG. The DCTG specifies how Bruce programs are to be parsed, and once parsed, how semantics are to be generated for the Bruce constructs. Each DCTG rule has associated with it a *syntactic* component and a *semantic* component. The syntactic component contains a BNF–style grammar rule for a portion of the Bruce syntax. The semantic component specifies actions which are to take place at the node of the parse tree associated with that grammar rule. In our implementation, a semantic rule builds up a portion of the logical axiom associated with the context of the grammar rule in question.

Two rules describing the composition of chains are in figure 5.1. Items within braces are matched with strings in the input, while *stat* and *chain* are references to grammar rules. The first rule parses single statement chains, and the second parses multiple statement ones. In the case of single statement chains, the code is directly retrieved at the grammar

```
while ::= [while,'('], bool_expr^^B, [')','{'], chain^^Q, ['}']
<:>
code(Code, In, Out, Defns) :: -
    gen_whilecode(Code, In, Out, Defns, B, Q, Loop_label),
    add_curr_labellist(while, Loop_label).
```

Figure 5.2: While-loop rule

rule for that statement. In the multiple–statement case, the semantic rule calls a Prolog predicate *gen_chain* which obtains the code for the statement at node S in the parse tree and adds it to the code for the rest of the chain at node Q. Obtaining the code for chain Q results in a recursive invocation of the same grammar rule *chain*.

A typical DCTG rule which parses a statement is the one for *while* loops in figure 5.2. The syntactic component uses the grammar rules *bool_expr* and *chain*. The argument *Code* for the semantic rule *code* returns the axioms created for the *while* loop, and the argument *Defns* returns axioms generated when parsing the loop body. The semantic rule uses a predicate *gen_whilecode*, which generates the predicative text of the axiom for the loop. Other Bruce constructs are handled similarly.

Some bookkeeping is done to ease recompilation of the semantics into Bruce later. Program parameters and variables are mapped to input–output vectors by their relative positions within the vectors. Consequently, the parameter and variable names, as well as their relative ordering in the program, are retained. In addition, some features of the CProlog interpreter itself are exploited. It is convenient for Bruce to share the builtin arithmetic and relational operators used by CProlog, as this eases the evaluation of expressions later.

The existence of a branch in a Bruce program requires the generation of label predicates and the restructuring of the axioms. Both these operations use the existing axiom set during their processing, rather than operate on the original Bruce source program. Label predicate generation involves creating a predicate for each labelled statement in the program. The definition of a label predicate is the predicative meaning of the code generated by the

```
prog([x, y], [ ]) : {
  repeat {
    while (x < 0) {
      x := x + 1;
      l1 : y := x + 2
    };
    x := y + 3
  } until (x =:= y);
  y := y + 4;
  goto l1;
  y := y + 5
}
```

Figure 5.3: Branching into two loops

Buildenv function from Bruce's operational semantics. To derive these predicates from the existing axioms, conjunctions of goals which compose the control context of the labelled statement are collected. Collecting these goals requires traversing the axioms from the statement up through its parent mechanisms until the top program axiom is reached. The predicate generated is refined somewhat by using the fact that branches are semantically represented by label program calls, which can be treated as program continuations. The predicate can thus be terminated as soon as a branch is reached, since the continuation for the branch accounts for the rest of the computation.

Figure 5.3 has a program which contains a branch into the middle of two nested loops. The semantics initially generated are in figure 5.4. The program branch is represented by a goal $goto(\bar{v}_i, l1, \bar{v}_f)$, which is an "unprocessed" branch. The label $l1$ is tagged by a goal $label(l1)$ in the first clause of $while1$, which is removed after a label program predicate has been created for it. The label program as defined by *Buildenv* for $l1$ is in figure 5.5, and the corresponding label program predicate is in figure 5.6. The predicate is much more terse than the label program, since code in the label program is logically described by existing predicates from the basic semantics. Note the addition of a predicate $test1$, which is used for describing the control which occurs when branching into the *repeat* loop.

$$
\begin{array}{l}
prog([X1,Y1],[X2,Y2]) \ :- \\
\quad repeat1([X1,Y1],[X3,Y3]), \\
\quad asgn([X3,Y3],Y4,Y3+4,[X3,Y4]), \\
\quad goto([X3,Y4],l1,[X2,Y5]), \\
\quad asgn([X2,Y5],Y2,Y5+5,[X2,Y2]).
\end{array}
$$

$$
\begin{array}{l}
repeat1([X1,Y1],[X2,Y2]) \ :- \\
\quad while1([X1,Y1],[X3,Y3]), \\
\quad asgn([X3,Y3],X4,Y3+3,[X4,Y3]), \\
\quad \neg X4 =:= Y3, \\
\quad repeat1([X4,Y3],[X2,Y2]).
\end{array}
$$

$$
\begin{array}{l}
repeat1([X1,Y1],[X2,Y2]) \ :- \\
\quad while1([X1,Y1],[X3,Y2]), \\
\quad asgn([X3,Y2],X4,Y2+3,[X2,Y2]), \\
\quad X2 =:= Y2.
\end{array}
$$

$$
\begin{array}{l}
while1([X1,Y1],[X2,Y2]) \ :- \\
\quad X1 < 0, \\
\quad asgn([X1,Y1],X3,X1+1,[X3,Y1]), \\
\quad label(l1), \\
\quad asgn([X3,Y1],Y3,X4+2,[X3,Y3]), \\
\quad while1([X3,Y3],[X2,Y2]).
\end{array}
$$

$$
\begin{array}{l}
while1([X1,Y1],[X1,Y1]) :- \\
\quad \neg X1 < 0.
\end{array}
$$

Figure 5.4: Basic semantics

$$Buildenv(prog\{...\}, l1 : y := x + 2) \iff$$

```
y := x + 2;
while(x < 0) {
   x := x + 1;
   y := x + 2
};
x := y + 3;
if (¬(x =:= y)) {
   repeat {
      while (x < 0) {
         x := x + 1;
         y := x + 2
      };
      x := y + 3
   } until (x =:= y)
}
y := y + 4;
goto l1;
y := y + 5
```

Figure 5.5: Label program

```
l1([X1,Y1],[X2,Y2]) :-
   asgn([X1,Y1],Y3,X1 + 2,[X1,Y3]),
   while1([X1,Y3],[X3,Y4]),
   asgn([X3,Y4],X4,Y4 + 3,[X4,Y4]),
   test1([X4,Y4],[X5,Y5]),
   asgn([X5,Y5],Y6,Y5 + 4,[X5,Y6]),
   goto([X5,Y6],l1,[X2,Y2]).

test1([X1,Y1],[X2,Y2]) :-
   ¬X1 =:= Y1,
   repeat1([X1,Y1],[X2,Y2]).

test1([X1,Y1],[X1,Y1]) :-
   X1 =:= Y1.
```

Figure 5.6: Label program predicate

The branch restructuring algorithm is a refined version of the distribution procedure outlined in section 4.3. The process begins by searching for an instance of an unprocessed branch goal. When such a goal is discovered, the goals following that goal are thrown away, and the goal is replaced by a label program call. Then the ancestor clauses of the predicate containing the branch are restructured. However, instead of distributing terms of the whole program predicate, only the clauses whose control is affected by a branch are inspected. In addition, predicative formulas are retained where possible, in order to minimize textual changes to the semantics. The net effect of the restructuring is that every axiom whose control is affected by the branch is altered as specified in section 4.3.

In terms of the program in figure 5.3, the resulting branch restructuring in figure 5.7 is not very interesting. The main program predicate has its branch goal replaced by a label program relation for $l1$, and the goal following it has been lost. The branch in the label program predicate has also been replaced.

What is of more interest is the program branch in figure 5.8, whose basic semantics are in figure 5.9. The restructured semantics in figure 5.10 illustrates the effect of the single branch on the control of the two loops. These predicates are a derivation of the distributed program predicate in figure 5.11, in which common terms have been factored out and assigned to existing predicate names.

5.2 The Brucifier

Compiling a logical representation back into Bruce is the inverse of generating the logic from Bruce. Pattern matching is performed on the axioms of a semantic representation to search for matches with logical definitions corresponding to Bruce constructs. The schema matching used when converting predicates into Bruce is basically a backwards interpretation of the basic semantics of figure 4.1 in chapter 4. A measure of nondeterminism is encoded into the Brucifier in order to account for the fact that there is often more than one Bruce implementation derivable for a given set of axioms. This nondeterminism means

$$
\begin{array}{l}
prog([X1,Y1],[X2,Y2]) \ :- \\
\quad repeat1([X1,Y1],[X3,Y3]), \\
\quad asgn([X3,Y3],Y4,Y3+4,[X3,Y4]), \\
\quad l1([X3,Y4],[X2,Y2]).
\end{array}
$$

$$
\begin{array}{l}
repeat1([X1,Y1],[X2,Y2]) \ :- \\
\quad while1([X1,Y1],[X3,Y3]), \\
\quad asgn([X3,Y3],X4,Y3+3,[X4,Y3]), \\
\quad \neg X4 =:= Y3, \\
\quad repeat1([X4,Y3],[X2,Y2]).
\end{array}
$$

$$
\begin{array}{l}
repeat1([X1,Y1],[X2,Y2]) \ :- \\
\quad while1([X1,Y1],[X3,Y2]), \\
\quad asgn([X3,Y4],X2,Y2+3,[X2,Y2]), \\
\quad X6 =:= Y2.
\end{array}
$$

$$
\begin{array}{l}
while1([X1,Y1],[X2,Y2]) \ :- \\
\quad X1 < 0, \\
\quad asgn([X1,Y1],X3,X1+1,[X3,Y1]), \\
\quad asgn([X3,Y1],Y3,X4+2,[X3,Y3]), \\
\quad while1([X3,Y3],[X2,Y2]).
\end{array}
$$

$$
\begin{array}{l}
while1([X1,Y1],[X1,Y1]) \ :- \\
\quad \neg X1 < 0.
\end{array}
$$

$$
\begin{array}{l}
l1([X1,Y1],[X2,Y2]) \ :- \\
\quad asgn([X1,Y1],Y3,X1+2,[X1,Y3]), \\
\quad while1([X1,Y3],[X3,Y4]), \\
\quad asgn([X3,Y4],X4,Y4+3,[X4,Y4]), \\
\quad test1([X4,Y4],[X5,Y5]), \\
\quad asgn([X5,Y5],Y6,Y5+4,[X5,Y6]), \\
\quad l1([X5,Y6],[X2,Y2]).
\end{array}
$$

$$
\begin{array}{l}
test1([X1,Y1],[X2,Y2]) \ :- \\
\quad \neg X1 =:= Y1, \\
\quad repeat1([X1,Y1],[X2,Y2]).
\end{array}
$$

$$
\begin{array}{l}
test1([X1,Y1],[X1,Y1]) \ :- \\
\quad X1 =:= Y1.
\end{array}
$$

Figure 5.7: Final semantics

```
prog2([x, y], [ ]) : {
  while (x > 0) {
    while (x < 0) {
      x := x + 1;
      goto l2
    };
    x := y + 2
  };
  l2 : y := y + 3
}
```

Figure 5.8: Branch out of two loops

$$
\begin{array}{l}
prog2([X1, Y1], [X2, Y2]) \ :- \\
\quad while2([X1, Y1], [X2, Y3]), \\
\quad label(l2), \\
\quad asgn([X2, Y3], Y2, Y3 + 5, [X2, Y2]).
\end{array}
$$

$$
\begin{array}{l}
while2([X1, Y1], [X2, Y2]) \ :- \\
\quad X1 > 0, \\
\quad while1([X1, Y1], [X3, Y3]), \\
\quad asgn([X3, Y3], X4, Y3 + 3, [X4, Y3]), \\
\quad while2([X4, Y3], [X2, Y2]).
\end{array}
$$

$$
\begin{array}{l}
while2([X1, Y1], [X1, Y1]) \ :- \\
\quad \neg X1 > 0.
\end{array}
$$

$$
\begin{array}{l}
while1([X1, Y1], [X2, Y2]) \ :- \\
\quad X1 < 0, \\
\quad asgn([X1, Y1], X3, X1 + 1, [X3, Y1]), \\
\quad goto([X3, Y1], l2, [X4, Y3]), \\
\quad while1([X4, Y3], [X2, Y2]).
\end{array}
$$

$$
\begin{array}{l}
while1([X1, Y1], [X1, Y1]) \ :- \\
\quad \neg X1 < 0.
\end{array}
$$

Figure 5.9: Basic semantics

$$
\begin{array}{l}
prog2([X1,Y1],[X2,Y2]) \ :- \\
\quad while2([X1,Y1],[X2,Y2]).
\end{array}
$$

$$
\begin{array}{l}
while2([X1,Y1],[X2,Y2]) \ :- \\
\quad X1 > 0, \\
\quad while1([X1,Y1],[X2,Y2]).
\end{array}
$$

$$
\begin{array}{l}
while2([X1,Y1],[X1,Y2]) \ :- \\
\quad \neg X1 > 0, \\
\quad asgn([X1,Y1],Y2,Y1+5,[X1,Y2]).
\end{array}
$$

$$
\begin{array}{l}
while1([X1,Y1],[X2,Y2]) \ :- \\
\quad X1 < 0, \\
\quad asgn([X1,Y1],X3,X1+1,[X3,Y1]), \\
\quad l2([X3,Y1],[X2,Y2]).
\end{array}
$$

$$
\begin{array}{l}
while1([X1,Y1],[X2,Y2]) \ :- \\
\quad \neg X1 < 0, \\
\quad asgn([X1,Y1],X3,Y1+3,[X3,Y1]), \\
\quad while2([X3,Y1],[X2,Y2]).
\end{array}
$$

$$
\begin{array}{l}
l2([X1,Y1],[X1,Y2]) : - \\
\quad asgn([X1,Y1],Y2,Y1+5,[X1,Y2]).
\end{array}
$$

Figure 5.10: Final semantics

$$
\begin{array}{rl}
prog2 \Leftarrow & ((x > 0) \wedge (x < 0); x := x + 1 \wedge l2(\bar{v}_i, \bar{v}_f)) \\
& \vee((x > 0) \wedge \neg(x < 0); x := y + 2; while(x > 0); l2 : y := y + 3) \\
& \vee(\neg(x > 0) \wedge l2 : y := y + 3)
\end{array}
$$

Figure 5.11: Distributed program predicate

```
template_match(Progname, Vec, Prolog, [A, B], Bruce, Labels, Newlabels) :-
    is_a_while(Progname, A, B, Test, Body),
    get_IO(A, _, Out),
    template_match_goals(Progname, Out, Prolog, Body, Bruce_body, Labels, Newlabels),
    append_list([['while(', Test, ')', ' {'], Bruce_body, ['}']], Bruce).
```

Figure 5.12: While loop matching

that repeated calls to the Brucifier for a given set of axioms may result in different Bruce programs.

There are two main matching modules used by the Brucifier. A module called *template_match* takes the axioms for a goal and matches them with the logical definitions of Bruce constructs. Such matching often requires the logical manipulation of predicates, for example, factoring out common terms and unfolding predicate definitions. This module also assures that the Bruce statements generated for a list of goals are sequenced properly. Depending on the nature of the transformations performed on the semantics, goals may have to be ordered with respect to the data flow of their input–output vectors. A typical matching rule is in figure 5.12. The predicate *is_a_while* succeeds if clauses A and B represent an instance of a *while* loop. *Body* are the goals in the loop's body, and are matched through use of the other matching module (called *template_match_goals* in our implementation). When this succeeds, the Bruce code defining this loop is constructed.

The other module, *template_match_goals*, is used for matching goals in chains. After the goals are ordered, a Bruce construct is derived for each goal. Elementary relations such as assignments are converted directly into Bruce at this stage. More complex mechanisms (those with external definitions) are converted to Bruce by invoking *template_match* on the axioms defining the predicate to which the goal is referring.

One peculiarity of our matching algorithm is that branches always refer to code which has been previously converted. In other words, the first branch in a program will be unfolded, and subsequent references to it will refer to this unfolded portion of code. A

```
prog([x, y], [ ]) : {
  repeat {
    while1 : while (x < 0) {
      x := x + 1;
      y := x + 2
    };
    x := y + 3
  } until (x =:= y);
  y := y + 4;
  y := x + 2;
  goto while1
}
```

Figure 5.13: Derived Bruce program

consequence of this is that branches among statements in a single chain tend to be unfolded, and might just disappear as a result. For an example of this phenomena, see the program in figure 4.9 of section 4.6

Lastly, the conversion of Bruce expressions, such as those in assignments and boolean tests, requires the use of information about the names of variables and parameters and their relative ordering in the input–output vectors. This information was saved during the parsing of the program.

The Brucified results for the logic programs in figures 5.7 and 5.10 are in figures 5.13 and 5.14 respectively.

```
prog2([x, y], [ ]) : {
  while2 : if (x > 0) {
    if (x < 0) {
      x := x + 1;
      y := y + 5
    }
    else {
      x := y + 3;
      goto while2
    }
  }
  else {
    y := y + 5
  }
}
```

Figure 5.14: Derived Bruce program

Chapter 6

The Partial Evaluation of Bruce Programs

6.1 Partial evaluation and program transformation

Once a program is translated into its logical representation, many types of transformations are possible. Deductive transformations and term rewriting are two possibilities which would likely incorporate a mixture of heuristics and user control in order to be practical. We chose partial evaluation as our program transformation scheme, as it is easily automated. The partial evaluation of logic programs is also a logically sound transformation formalism, since the meta–interpretation of logic programs in a partial evaluation context is based on SLDNF resolution, which is sound. This means that resulting transformed Bruce programs are correct. Most importantly, partial evaluation most clearly illustrates the utility of our Horn clause semantics and its underlying execution model.

Partial evaluation is a computational model which distinguishes executable, *permissible* code from non–executable, *suspendable* code [Ershov 82] [Ershov 85]. In imperative programs, permissible code is that which has all the necessary store values defined in order to, for example, evaluate an expression or test. Suspendable code is that in which a needed value is undefined. In a program transformation setting, input parameters to the program are usually fixed so that the partial evaluator specializes the program with respect to that

set of input. Partial evaluation then entails executing all permissible code, during which all suspendable code is retained as a partially evaluated program. This *residual* program is an optimized, transformed version of the original program.

The following terminology and results are from [Lloyd *et al.* 87]. The partial evaluation of logic programs has been formally shown to be correct and complete, albeit under particular conditions. A semantic theory composed in our Horn clause semantics can be considered to be a *definite* logic program, since the negation operator used in negated goals only operates on closed boolean expressions in which resolution is never used; we could have used a weaker operator, but chose not to for convenience sake. Let P' be a partially evaluated program wrt a source program P and goal G. P' is *sound* if correct answers for P' and G are also correct for P and G. The partial evaluation of our semantics is therefore sound, since the partial evaluation of all definite logic programs is sound.

The partial evaluation of definite programs is not necessarily complete. P' is *complete* if correct answers for P and G are correct for P' and G. However, P' may be specialized wrt goal G, which means that some reference in P' to a specialized clause of P may no longer be satisfiable. Lloyd's method of ensuring completeness is to enforce a "closedness" condition on P'. This condition states that any predicate in P' must remain general enough to satisfy all goals which refer to it, which essentially inhibits the partial evaluation from becoming too specialized. We maintain completeness in a different manner. Given a program P, a general predicate h of P, and a goal G, we partially evaluate P wrt G to obtain a program P' containing a specialized predicate h'. Should references to h exist in P' which are too general for the specialized predicate h', we supplement P' with h:

$$P'' = P' \cup \{h\}$$

With respect to the theory $Th(P)$ defined by the axioms for P, P' being sound means that

$$\models (Th(P') \models h \;\Rightarrow\; Th(P) \models h)$$

However, P''s incompleteness implies

$$\models \neg(Th(P) \models h \;\Rightarrow\; Th(P') \models h)$$

The supplemented theory $Th(P'')$ is totally correct:

$$\models (Th(P) \models h \;\Leftrightarrow\; Th(P'') \models h)$$

6.2 Implementation

We transform Bruce programs by partially evaluating the Horn clause semantics for them. Once a Bruce program is translated into logic, its representation is asserted into the Prolog environment, and the partial evaluator is applied to it with respect to given parameter settings. A residual program is eventually created. Should this program be incomplete, it is made complete by adding to it required clauses from the original general program. Then, using predicative programming, the derived predicate is compiled back into Bruce, resulting in a transformed, optimized Bruce program.

Our partial evaluator takes the form of a meta–interpreter written in CProlog, and is based on implementations in [Venken 85], [Takeuchi *et al.* 85], and [Pereira *et al.* 87]. The core of the evaluator is in figure 6.1. The first clause of *mix* partially evaluates conjunctions of goals, the results of which are combined together. Clause (2) executes goals which have been determined to be builtin by the *builtin_executable* utility. Examples of such goals are the boolean and relational operators used in Bruce tests. Clause (4) solves goals by unifying a goal with a clause, and partially evaluating the resulting body. CProlog's *clause* utility automatically unifies the goal with a clause each time *clause* succeeds.

The most difficult problem faced when partially evaluating a program is determining when to inhibit the partial evaluation process. Goal inhibition is done in clause (3) of *mix* by the *inhibited* utility (see figure 6.2). Our system provides both automatic and user–controlled goal inhibition. Builtin goals and subprogram calls are inhibited by the first two clauses of *inhibited*. Another automatic feature checks that the variables used in

```
%(1) split multiple goals
mix((Goal, Rest), Result, Recur) :-
    !,
    mix(Goal, GResult, Recur),
    mix(Rest, RResult, Recur),
    fuse((GResult, RResult), Result).

%(2) execute builtin's
mix(Goal, true, _) :-
    builtin_executable(Goal),
    !,
    Goal.

%(3) halt interpretation
mix(Goal, Goal, Recur) :-
    inhibited(Goal, Recur),
    !.

%(4) interpret goal
mix(Goal, Result, Recur) :-
    clause(Goal, Body),
    mix(Body, Result2, [Goal|Recur]),
    set_mix_result(Result2, Result).
```

Figure 6.1: Partial evaluator

```
%(1) builtin goals
inhibited(Goal,_) :-
    builtin(Goal),
    !.

%(2) subprogram calls
inhibited(Goal,_) :-
    functor(Goal,Name,_),
    storematch(Name,_,_,_,_),
    !.

%(3) key variables are uninstantiated
inhibited(Goal,_) :-
    not var_inst(Goal),
    !.

%(4) loop detected
inhibited(Goal,Recur) :-
    member3(Goal,Recur),
    !.

%(5) no clause for goal
inhibited(Goal,_) :-
    not clause(Goal,_),
    !.
```

Figure 6.2: Goal inhibition

a loop test are ground before that loop is unfolded (clause (3) of *inhibited*). When a Bruce program is initially parsed, a record is kept of the variables used by the test expression of each *if* statement and loop. The partial evaluator uses this information to inhibit goals corresponding to these constructs should their test variables be uninstantiated. The fourth clause of *inhibited* checks for loops. Finally, any goal which cannot be unified with a clause is inhibited. In addition to these five strategies, the user is allowed to enforce his or her own inhibition schemes simply by asserting into Prolog their own *inhibited* clause. This is vital when the program contains complex looping mechanisms caused by the use of branches. It is also possible for the user to override any automatic inhibition strategy.

6.3 Example transformations

Figure 6.3 has a Bruce program *power* which computes integral powers using a binary power algorithm (from [Ershov 82]). The two *varinst* clauses, generated during parsing, are used to indicate to the partial evaluator when to inhibit the clauses corresponding to the two loops in the program. Both clauses specify that the third data object (the value parameter n) must be instantiated before partially evaluating the loops. When *power* is partially evaluated with respect to $n = 5$, partial execution yields the residual logic program and Bruce program in figure 6.4. Note that a side effect of partial evaluation is that some vector values become ground. When converting such a relation back into Bruce, the ground terms implicitly represent assignments of constants to variables. Partially evaluating *power* with both $x = 2$ and $n = 5$ produces a program which simply sets the output parameter to the result (figure 6.5).

An example of code simplification is the binary gcd program of figure 6.6 (from [Knuth 81]). We assert goal inhibition clauses specifying that u must be ground before the label programs $b1$ and $b2$ are unfolded, and t must be ground before $b4$ is unfolded. Evaluating the program with $u = 7$ results in a residual program which is somewhat simpler than the original, though not a great deal more efficient. The logical form of *bingcd* and its transformed equivalent can be found in appendices C and D respectively.

```
power([y],[x,n]) : {
  y := 1;
  while (n > 0) {
    while (((n//2) * 2) =:= n) {
      n := n//2;
      x := x * x
    };
    n := n - 1;
    y := y * x
  }
}
```

$\Longleftrightarrow$

```
power([Y1,X1,N1],[Y2,X1,N1]) :-
  asgn([Y1,X1,N1],Y3,1,[Y3,X1,N1]),
  while2([Y3,X1,N1],[Y2,X2,N2]).

while1([Y1,X1,N1],[Y2,X2,N2]) :-
  N1//2 * 2 =:= N1,
  asgn([Y1,X1,N1],N3,N1//2,[Y1,X1,N3]),
  asgn([Y1,X1,N4],X3,X1 * X1,[Y1,X3,N3]),
  while1([Y1,X3,N3],[Y2,X2,N2]).

while1([Y1,X1,N1],[Y1,X1,N1]) :-
  ¬N1//2 * 2 =:= N1.

while2([Y1,X1,N1],[Y2,X2,N2]) :-
  N1 > 0,
  while1([Y1,X1,N1],[Y3,X3,N3]),
  asgn([Y3,X3,N3],N4,N3 - 1,[Y3,X3,N4]),
  asgn([Y3,X3,N4],Y4,Y3 * X3,[Y4,X3,N4]),
  while2([Y4,X3,N4],[Y2,X2,N2]).

while2([Y1,X1,N1],[Y1,X1,N1]) :-
  ¬N1 > 0.

varinst(while1,[f,f,i]).
varinst(while2,[f,f,i]).
```

Figure 6.3: Binary powers

```
power([Y1,X1,5],[Y2,X1,5]) :-
  asgn([1,X1,4],Y3,1 * X1,[Y3,X1,4]),
  asgn([Y3,X1,2],X2,X1 * X1,[Y3,X2,2]),
  asgn([Y3,X2,1],X3,X2 * X2,[Y3,X3,1]),
  asgn([Y3,X3,0],Y2,Y4 * X3,[Y2,X3,0]).
```

$\Longleftrightarrow$

```
power2([y],[x,n]) : {
  y := 1 * x;
  x := x * x;
  x := x * x;
  y := y * x
}
```

Figure 6.4: Residual programs ($n = 5$)

```
power([Y1,2,5],[32,2,5]).
```

$\Longleftrightarrow$

```
power3([y],[x,n]) : {
  y := 32
}
```

Figure 6.5: Residual programs ($x = 2, n = 5$)

```
bingcd([gcd],[u,v]) : local [k,p,t]{
    k := 0;
b1 : if (((u//2) * 2) = \ = u) {
       goto b2
    };
    if (((v//2) * 2) = \ = v) {
       goto b2
    };
    k := k + 1;
    u := u//2;
    v := v//2;
    goto b1;
b2 : if (((u//2) * 2) = \ = u) {
       t := 0 - v;
       goto b4
    };
    t := u;
b3 : t := t//2;
b4 : if (((t//2) * 2) =:= t) {
       goto b3
    };
    if (t > 0) {
       u := t
    }
    else {
       v := 0 - t
    };
    t := u - v;
    if (t = \ = 0) {
       goto b3
    };
    call power([p],[2,k]);
    gcd := u * p
}
```

$\Longrightarrow$

```
bingcd([gcd],[u,v]) :
local [k,p,t] {
        t := 0 - v;
        u := 7;
        k := 0;
test4 : if (((t//2) * 2) =:= t) {
b3 :        t := t//2;
              goto test4
        }
        else {
          if (t > 0) {
            u := t
          }
          else {
            v := 0 - t
          };
          t := u - v;
          if (t = \ = 0) {
            goto b3
          }
          else {
            call power([p],[2,k]);
            gcd := u * p
          }
        }
}
```

Figure 6.6: Binary gcd ($u = 7$)

```
p([y, z], [x, c1, c2]) :
local [r1, r2]{
   if (x > z) {
      r1 := c1 + 1;
      r2 := c1 − 1
   }
   else {
      r1 := c2 − 1;
      r2 := c2 + 1
   };
   y := y + r1;
   z := z + r2
}
```

Figure 6.7: A suspended test

```
p([Y1, Z1, X1, 3, 5], [Y2, Z2, X1, 3, 5]) : −
   ¬X1 > Z1,
   asgn([Y1, Z1, X1, 3, 5, 4, 6], Y2, Y1 + 4, [Y2, Z1, X1, 3, 5, 4, 6]),
   asgn([Y2, Z1, X1, 3, 5, 4, 6], Z2, Z1 + 6, [Y2, Z2, X1, 3, 5, 4, 6]).

p([Y1, Z1, X1, 3, 5], [Y2, Z2, X1, 3, 5]) : −
   X1 > Z1,
   asgn([Y1, Z1, X1, 3, 5, 4, 2], Y2, Y1 + 4, [Y2, Z1, X1, 3, 5, 4, 2]),
   asgn([Y2, Z1, X1, 3, 5, 4, 2], Z2, Z1 + 2, [Y2, Z2, X1, 3, 5, 4, 2]).
```

Figure 6.8: Polyvariant logical result

```
p2([y, z], [x, c1, c2]) :
local [r1, r2] {
  if (x > z) {
    y := y + 4;
    z := z + 2
  }
  else {
    y := y + 4;
    z := z + 6
  }
}
```

Figure 6.9: Polyvariant Bruce result

Logic elegantly handles *polyvariant computational schemes*, which are simply those which entail partially evaluating the bodies of tests and loops whose tests have been suspended. In logic, this is naturally handled by unfolding the suspended test's defining clauses, and retaining the suspended test expression as a goal. Figure 6.7 shows a Bruce program with a suspended test (assume x is undefined). Figure 6.9 shows the partially evaluated logical result for $c1 = 3$ and $c2 = 5$, and its Bruce realization is in figure 6.3.

Chapter 7

Discussion

Comparisons are given between the work in this thesis and related work. This is followed by a general discussion of the thesis, along with future research possibilities.

7.1 Related work

The Horn clause semantics we use is semantically founded on those given in [Hehner 84a] and [Hehner *et al.* 86], which in turn is similar in spirit with logics used in program verification [Burstall 69] [Floyd 67] [Hoare 69]. Hehner's semantics describe the behavior of most of Bruce – descriptions which we borrow, albeit in a simplified and stylistically altered form. We also extend the imperative constructs handled by including program branches and call–by–value subprogram arguments. Since Hehner uses his logical semantics to study general characteristics of programs and computation, his semantics is much more descriptive than ours. His system has provisions for describing the evaluability of expressions, the termination of mechanisms, and the type checking of data. Our semantics ignores such features, as we are concerned with the representation of correct and terminating programs which are to be transformed. He also employs full first–order predicate calculus in his semantics, whereas we use a subset of predicate calculus – Horn clauses. A stylistic difference between his system and ours is that he uses fairly informal naming schemes when describing syntactic entities of a program, where we explicitly represent syntactic constructs with

uniquely named predicates.

Moss also uses Horn clause logic to semantically describe imperative programs [Moss 81] [Moss 82]. The description of a language in his system takes the form of a sophisticated interpreter which fully describes the syntax and operational semantics of the language. The syntax is written using a metamorphosis grammar, which is closely akin to the DCTG we use. His semantic rules, like ours, take the form of pure Prolog predicates. However, the nature of his semantics differs substantially from ours. Moss's semantic methodology is very robust, as his prime motivation is to describe the complete syntax and operational semantics of different programming languages using pure Horn clause logic, thereby proving the utility and generality of Prolog's descriptive capabilities. A theory in his system is a set of axioms which fully describe the operational semantics of a language, and the domain of discourse of the theory is the universe of possible programs written in the language. Programs are translated into various first–order terms which take the form of convenient data structures. These terms are in turn manipulated by predicates describing various functions of the operational semantics for the language. The domain of discourse of his logical axioms is thus programs and their abstract operational components. He essentially creates pure Prolog interpreters for languages.

Our semantics uses a lower level of abstraction than that of Moss, since a theory in our system describes the computational behavior of a single imperative program. We use logical axioms to represent the functionality of the program's components, and reserve as our domain of discourse the domains of the store and expressions. The immediate advantage of such a low level description of computation is the ability of mapping, in both directions, between the source language and logical representation. In addition, the close relationship between the axioms and program means that optimizations of the axioms directly reflect improvements in the corresponding imperative program.

Clark and van Emden have used Horn clause logic for verifying flowcharts, which can be interpreted as low level representations of imperative computations [Clark *et al.* 81]. They describe a flowchart's control in terms of Horn clauses, and then use a technique

called consequence verification to prove various properties of the computation. The style of their semantics is very similar to ours. Their low level description of control relates to our semantics in that uniquely named predicates represent unique computational mechanisms. In addition, they use logical variable renaming to describe updates to the store. However, their system is intended for use in program verification, so they make no attempt to associate their axioms with programming language constructs.

Our methodology for partially evaluating imperative programs offers interesting comparisons with work done by Ershov [Ershov 82] [Ershov 85]. Ershov partially evaluates programs by symbolically executing them using an algebraic rewriting system. These rewriting rules are applied to the text of a program, and eventually yield a partially evaluated result. The meta–interpretation of our logical semantics very closely parallels the effects of his transformations. Analogies between the transformation system presented in [Ershov 85] and SLDNF resolution of Horn clause semantics are as follows:

1. *Term reduction:* Ershov reduces expressions by replacing a term by its domain value. In our system, this occurs when an $asgn, callasgn$ or boolean test is resolved and the expression value obtained is carried forward in the computation by the input–output vectors in relations.

2. *Variable reduction:* Ershov uses variable reduction to carry an assigned value of a variable to other statements using that variable. In our system, Prolog's unification automatically "passes" the value of a variable among goals in a clause, and resolvents of a goal.

3. *Assignment reduction:* Ershov removes any assignment whose variable value has been distributed throughout the code. Under SLD resolution, assignments (as with all goals) automatically "disappear" once they are resolved.

4. *If and while loop reduction:* These reductions are used to simulate the control of tests and loops. Again, SLD resolution parallels these operations.

In fact, the traces of program transformations given by Ershov are very analogous to goal traces obtained when executing our semantics in Prolog.

We believe that our approach to the transformation of imperative programs has advantages over Ershov's approach. The main strength of our formalism is that transformations performed on Horn clause semantics have semantic interpretations in logic. Performing program transformations in logic means that the transformations can be verified for correctness in logic itself: *any valid logical manipulation of the axioms is guaranteed of being a correctness–preserving program transformation.* We exploit this fact in this thesis by proving that our partial evaluation of Horn clause semantics is logically sound, since the partial evaluation of logic programs can be considered a pure logical transformation scheme [Lloyd *et al.* 87]. On the other hand, Ershov's transformation system does not have any strong formal semantic interpretation. Consequently, partial evaluation in his system is not formally proved to be correctness preserving.

Our logical semantics offers a substantial improvement to previous techniques for polyvariant partial evaluation, which entails partially evaluating the bodies of tests and loops whose tests have been suspended [Ershov 85] [Bulyonkov 84]. Previous handling of this has been by explicitly maintaining memory states of alternate computations, which is required by the underlying algebraic transformation scheme. In logic, this is handled by simply treating the test as a suspendable goal, which is retained during the meta–interpretation process. Separate memory states are automatically retained during the evaluation of different clauses, since memory states are explicitly stored as terms of the clauses.

7.2 Conclusions and further work

We have illustrated the applicability of Horn clause logic as a tool for transforming imperative programs. We described the semantics of a nontrivial imperative language using a relatively economical set of logical axioms. Using the predicative programming paradigm, direct translation between semantical representations and the source language is possible. This semantics is also ideally suited to many types of transformations, the one we presented

being partial evaluation.

Our semantics are essentially a logical characterization of the operational semantics for Bruce. Our approach to the design of the semantics was to first define Bruce in terms of an abstract machine, and then create logical axioms for different Bruce constructs modelled on the behavior of Bruce's operational semantics. Whether this intermediate abstract definition of Bruce was truly required is debatable, as the Horn clause definition of constructs is just as intuitive as the abstract operational definitions, if not more so. The semantics are also abstract enough that machine–dependent features of the language are minimized, preventing it from being as machine–oriented as other operational semantic systems.

Our handling of branches in Bruce uses both logical semantics and algebraic semantics. The logical semantics comes into play during the initial expansion of the program predicate, since we use the basic axiomatic definitions of program constructs during the expansion process. Distributing terms of the program predicate, as well as removing the control context of the branch, are essentially algebraic operations which are based on logical considerations. Our defense of this approach is that the *formula* of axiom derivation has become more complicated. When no branches exist, the surface structure of program statements directly define the axioms. With branches, we must use a more sophisticated inspection of the program syntax in order to derive the axioms.

One possible extension of this work is to enhance the semantics to handle more complex imperative languages. The inclusion of complex data types would be a relatively straightforward extension, and would be especially useful given the preponderance of imperative programs using arrays and other data structures. A real test would be to derive a Horn clause semantics in the style of this work for an existing programming language such as Pascal.

Worth investigation is whether this style of logical semantics is applicable to other heterogeneous programming paradigms, such as functional programs. Such research would be in effect searching for a general description of computation. It is likely that a more general semantics would have much more complex domains than that of ours, since other

programming paradigms treat programs as data much more easily than do imperative languages. Denotational semantics might therefore offer some insight into the semantic content of the domains of such a representation. In addition, algebraic semantics might play a role in defining a more generally applicable semantic formalism. Much of the textual processing which we do to our semantics, such as factoring, code distributing, and branch processing, are simple algebraic manipulations of the logical formulas. The manipulation of more complex semantic expressions might be more easily defined in terms of algebraic rules. However, it is advantageous to maintain a logical basis to the methodology, as predicate logic permits meaningful interpretations and analysis of the domains being described.

This thesis offers some philosophical views on the disciplines of programming and programming language design. An immediate result of this work is that imperative programs are easily described or simulated by logic programs. One might conjecture from this that logic programming subsumes imperative programming in terms of computational power. Logic programs are certainly capable of computing any Turing–computable function [Tarnlund 77]. However, a logic program interpreted as a logical theory is governed by the same principles of undecidability and effective computability as its imperative brethren [Boolos *et al.* 80]. We should instead surmise that logic programs subsume imperative ones in terms of *descriptive power*, since the logic program interpretation of our semantics uses a relatively limited subset of Prolog's full descriptive and computational capabilities. Prolog programs as written by Prolog programmers use much more complex unification schemes than those used in our semantics. In addition, "real" Prolog programs are often written nondeterministically, which means that arguments to predicates might be either input or output in nature. Our Horn clause semantics are strictly deterministic logic programs, being declarative descriptions of deterministic computations.

Our semantics offers some insight into alternation trees and *and/or* models of programming [Harel 80]. In a logic programming context context, an alternation tree description of computation is just a restatement of the inherent logical semantics of pure Prolog program. Logic programs can be interpreted as *and/or* trees (or alternation trees), in which clauses

of a predicate represent *or*–branches, and the goals of a clause represent *and*–branches. The tree itself is composed of alternate instances of *and*–nodes and *or*–nodes (hence the term *alternation* tree) in which *and*–nodes are the parents of *or*–nodes, and vice versa. A successful computation of the tree requires that all the *and*–nodes under an *or*–node must be successful before the *or*–node is successful; similarly, at least one *or*–node under an *and*–node must be successful before that *and*–node succeeds. Besides having theoretical importance in the semantics of programming, the alternation tree interpretation of logic programs has applications in parallel processing [Shapiro 87]. The parallel execution of an *and/or* tree means that the *or*–nodes and *and*–nodes of the tree all take the form of producers and consumers of data. The data path between the nodes is dependent on the particular variable usage in the Prolog program. Data transfer between the nodes is done through the unification process.

Casting an imperative program into our Horn clause semantics means that an *and/or* model of computation is immediately defined for it. As with all pure logic programs, the alternation tree description of the Horn clause semantics of a Bruce program allows the direct derivation of an *and/or* model of computation. In addition, this alternation tree gives an initial parallel execution model for the program. However, a data flow analysis of the tree would probably be needed in order to derive an efficient model of parallel computation.

Partial evaluation is a scheme which transforms a general program into a more specialized one. However, efficiency is gained at the expense of generality. The ideal goal of program transformation is to transform a program into a more efficient form without any loss of generality. Existing systems which do such transformations are unfortunately not very automatable. Nevertheless, it would be interesting to use our semantics in a more deductive transformational environment. It is likely that heuristics and user control would play a large role in such a transformation system. Another interesting application of our semantics is its use in a transformation system which transforms an input imperative program into a more "stylized" version. Also worth attention is the use of our semantics in a program verification system, which would again rely on heuristic and user control.

Further work in controlling the partial evaluation of imperative programs is needed. Inhibiting recursion on polyvariant computations is difficult to implement in our system. This can be overcome somewhat by partially evaluating each clause in the program predicate individually. However, it is not obvious how constant values can be elegantly propagated to these clauses using this scheme. Controlling the partial evaluation process seems to be a research topic problematic to the field of partial evaluation in general [Lloyd *et al.* 87].

Ershov expresses the need for a systems programming language (SPL) in which program transformation could be based [Ershov 82]. We believe that our technique offers a more solid semantic foundation for program transformation than his algebraic transformations, and claim that Horn clause logic using Prolog's execution model is a prime candidate for Ershov's SPL. First and foremost, pure Prolog has a well–founded underlying mathematical model, namely first–order predicate logic. Using logic immediately leads to the opportunity of using the many theoretical results, tools and techniques which have been established in mathematical logic throughout the years. In addition, using the predicative programming paradigm, there is a strong and natural relationship to the semantics of a language and its syntactic realization. Finally, Prolog has already proven its applicability in many systems programming applications, for example, [Abramson *et al.* 88], [Moss 81], and [Shapiro 83].

References

[Abramson 84] H. Abramson. Definite Clause Translation Grammars. In *Proc. Logic Programming Symp.*, IEEE, Atlantic City, New Jersey, February 1984.

[Abramson *et al.* 88] H. Abramson, M. Crocker, B. Ross, and D. Westcott. A Fifth Generation Translator Writing System: Towards an Expert System for Compiler Development. In *Fifth Generation Comp. Sys. Conf.*, IFIPS, Tokyo, Japan, 1988. submitted.

[Allison 86] Lloyd Allison. *A practical introduction to denotational semantics.* Volume 23 of *Cambridge Computer Science Texts*, Cambridge University Press, 1986.

[Apt 81] K.R. Apt. Ten years of Hoare's logic: a survey. *TOPLAS*, 3:431–483, 1981.

[Ashcroft *et al.* 82] E.A. Ashcroft and W.W. Wadge. R for Semantics. *ACM Transactions on Programming Languages and Systems*, 4(2):283–294, April 1982.

[Backus 78] John Backus. Can Programming Be Liberated from the von Newmann Style? *CACM*, 21(8):613–641, August 1978.

[Beckman *et al.* 76] L. Beckman, A. Haraldson, O. Oskarsson, and E. Sandewall. A partial evaluator and its use as a programming tool. *Artificial Intelligence*, 7:319–357, 1976.

[Bibel 75] W. Bibel. Pradikatives programmieren. In *GI - 2. Fachtagung uber Automatentheorie und Formale Sprachen*, pages 274–283, Springer, 1975.

[Bibel 85] W. Bibel. Predicative programming revisited. In *Mathematical method of specification and synthesis of software systems '85*, pages 25–40, Wendisch-Reitz, E. Germany, April 1985.

[Boolos *et al.* 80] George Boolos and Richard Jeffrey. *Computability and Logic.* Cambridge University Press, 1980.

[Boyer *et al.* 79] R.S. Boyer and J.S. Moore. *A Computational Logic. ACM Monograph Series*, Academic Press, 1979.

[Broy 84] Manfred Broy. Algebraic Methods for Program Construction: The Project CIP. In P. Pepper, editor, *Program Transformation and Programming Environments*, pages 199–222, Springer-Verlag, 1984.

[Bulyonkov 84] M.A. Bulyonkov. Polyvariant Mixed Computation for Analyzer Programs. *Acta Informatica*, 21:473–484, 1984.

[Burstall 69] R.M. Burstall. Formal Description of Program Structure and Semantics in First Order Logic. In Michie Meltzer, editor, *Machine Intelligence 5*, pages 79–98, Elsevier, 1969.

[Clark *et al.* 81] K.L. Clark and M.H. van Emden. Consequence Verification of Flowcharts. *IEEE Transactions on Software Engineering*, SE-7(1):52–60, January 1981.

[Clint *et al.* 72] M. Clint and C.A.R. Hoare. Programming Proving: Jumps and Functions. *Acta Informatica*, 1:214–224, 1972.

[Clocksin *et al.* 81] W.F. Clocksin and C.S. Mellish. *Programming in Prolog.* Springer-Verlag, 1981.

[Cook 78] S.A. Cook. Soundness and completeness of an axiom system for program verification. *SIAM Journal of Computing*, 7(1):70–90, 1978.

[Darlington *et al.* 73] J. Darlington and R.M. Burstall. A System which Automatically Improves Programs. In *IJCAI*, pages 479–485, XXX, 1973.

[de Bruin 81] A. de Bruin. Goto Statements: Semantics and Deduction Systems. *Acta Informatica*, 15:385–424, 1981.

[Ershov 82] Andrei P. Ershov. Mixed Computation: Potential Applications and Problems for Study. *Theoretical Computer Science 18*, 18:41–67, 1982.

[Ershov 85] Andrei P. Ershov. On Mixed Computation: Informal Account of the Strict and Polyvariant Computational Schemes. In M. Broy, editor, *Control Flow and Data Flow: Concepts of Distributed Programming*, pages 107–120, Springer-Verlag, 1985.

[Floyd 67] R.W. Floyd. Assigning Meanings to Programs. In *Proc. Symp. Appl. Math.*, pages 19–32, Amer. Math. Soc., 1967.

[Futamura 71] Y. Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Systems - Comput. - Controls*, 2(5):45–50, 1971.

[Gordon 79] M. Gordon. *The Denotational Description of Programming Languages.* Springer-Verlag, New York, 1979.

[Guessarian 81] Irene Guessarian. *Algebraic Semantics.* Volume 99 of *Lecture Notes in Computer Science*, Springer-Verlag, 1981.

[Harel 80] David Harel. Ans/Or Programs: A New Approach to Structured Programming. *ACM Transactions on Programming Languages and Systems*, 2(1):1–17, January 1980.

[Hehner 84a] Eric C.R. Hehner. Predicative Programming Part I. *CACM*, 27(2):134–143, February 1984.

[Hehner 84b] Eric C.R. Hehner. Predicative Programming Part II. *CACM*, 27(2), February 1984.

[Hehner *et al.* 86] Eric C.R. Hehner, Lorene E. Gupta, and Andrew J. Malton. Predicative Methodology. *Acta Informatica*, 23:487–505, 1986.

[Hoare 69] C.A.R. Hoare. An Axiomatic Basis for Computer Programming. *CACM*, 12(10):576–583, October 1969.

[Hoare 85] C.A.R. Hoare. Programs are Predicates. In C.A.R. Hoare and J.C. Shepherdson, editors, *Mathematical Logic and Programming Languages*, pages 141–155, Prentice-Hall, 1985.

[Hoare *et al.* 73] C.A.R. Hoare and N. Wirth. An axiomatic definition of the programming language PASCAL. *Acta Informatica*, 2:335–355, 1973.

[Jones *et al.* 85] N.D. Jones, P. Sestoft, and H. Sondergaard. An Experiment in Partial Evaluation: the Generation of a Compiler Generator. *SIGPLAN Notices*, 20(8):82–87, August 1985.

[Kernighan *et al.* 78] B.W. Kernighan and D.M. Ritchie. *The C Programming Language.* Prentice-Hall, 1978.

[Knuth 81] Donald E. Knuth. *Seminumerical Algorithms.* Volume 2 of *The Art of Computer Programming*, Addison-Wesley, 2nd edition, 1981.

[Lloyd 84] J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1984.

[Lloyd *et al.* 87] J.W. Lloyd and J.C. Shepherdson. *Partial Evaluation in Logic Programming.* Technical Report CS-87-09, University of Bristol, December 1987.

[Loeckx *et al.* 84] J. Loeckx and K. Sieber. *The Foundations of Program Verification.* Wiley-Teubner, 1984.

[Loveman 77] D.B. Loveman. Program Improvement by Source-to-Source Transformation. *JACM*, 24(1):121–145, January 1977.

[Manna *et al.* 77] Zohar Manna and Richard Waldinger. *Studies in Automatic Programming Logic.* Elsevier North-Holland, 1977.

[Moss 81] Christopher D.S. Moss. *The Formal Description of Programming Languages using Predicate Logic.* PhD thesis, Imperial College, London, U.K., July 1981.

[Moss 82] Christopher D.S. Moss. How to Define a Language Using Prolog. In *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming*, pages 67–73, ACM, 1982.

[O'Donnell 82] M.J. O'Donnell. A Critique of the Foundations of Hoare Style Programming Logics. *CACM*, 25(12):927–935, December 1982.

[Partsch *et al.* 83] H. Partsch and R. Steinbruggen. Program Transformation Systems. *Computing Surveys*, 15(3):199–236, September 1983.

[Pepper 84] P. Pepper, editor. *Program Transformation and Programming Environments.* Springer-Verlag, 1984.

[Pereira *et al.* 84] F. Pereira, D. Warren, D. Bowen, and L. Pereira. *C-Prolog User's Manual.* version 1.5 edition, November 1984.

[Pereira *et al.* 87] Fernando C.N. Pereira and Stuart M. Shieber. *Prolog and Natural-Language Analysis.* Volume 10 of *CSLI Lecture Notes*, CSLI, 1987.

[Robinson 65] J.A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *JACM*, 12(1):23–41, January 1965.

[Shapiro 83] E.Y. Shapiro. *Systems Programming in Concurrent Prolog.* Technical Report TR-034, ICOT, Tokyo, Japan, November 1983.

[Shapiro 87] Ehud Shapiro. *Concurrent Prolog vol. 1 and 2.* MIT Press, 1987.

[Stoy 77] J. Stoy. *Denotational Semantics.* MIT Press, 1977.

[Takeuchi *et al.* 85] A. Takeuchi and K. Furukawa. *Partial Evaluation of Prolog Programs and its Application to Meta Programming.* Technical Report TR-126, ICOT, Tokyo, Japan, July 1985.

[Tarnlund 77] S. Tarnlund. Horn Clause Computability. *BIT*, 17:215–226, 1977.

[Tennent 77] R.D. Tennent. Language Design methods Based on Semantic Principles. *Acta Informatica*, 8:97–112, 1977.

[Tennent 81] R.D. Tennent. *Principles of Programming Languages.* Prentice-Hall, 1981.

[van Emden *et al.* 76] M.H. van Emden and R.A. Kowalski. The Semantics of Predicate Logic as a Programming Language. *JACM*, 23(4):733–742, October 1976.

[Venken 85] Raf Venken. A Prolog Meta-Interpreter for Partial Evaluation and its Application to Source to Source Transformation and Query Optimisation. In T.O'Shea, editor, *Advances in Artificial Intelligence*, pages 347–356, Elsevier Science Publishers B.V. (North Holland), 1985.

[Wirth *et al.* 78] Niklaus Wirth and Kathleen Jensen. *Pascal User Manual and Report.* Springer-Verlag, 2d edition, 1978.

Appendix A

Partial Evaluator Source

```
% ***
% *** Mixed Computation routines
% ***
% ***    - Brian J. Ross
% ***      Dept. of Computer Science
% ***      University of British Columbia
% ***

% ------------------------------------------------------------
% mix(Goal,Mixed):
%       Goal    - top goal to mix (with parameter settings)
%       Mixed   - Mixed Prolog result
%
% mix/2 finds the Prolog result, given a top level goal.
% A Prolog program must be asserted in the Prolog environment before this
% routine can run. The steps taken are:
%       1. Find the mixed result using top_mixer
%       2. Clean up the result from top_mixer.
%       3. Add Prolog predicates to make the resulting program complete.

mix(Goal,Mixed) :-
        setof(Results,Goal^top_mixer(Goal,Results),P),
        add_old_prolog(P,Mixed),
        !.

% ------------------------------------------------------------
% top_mixer(Goal,Result):
```

```
%       Goal    - top goal to mix
%       Result  - resulting mixed Prolog program
%
% top_mixer drive the mixed computation.

top_mixer(Goal,Result) :-
        clause(Goal,Body),
        mix(Body,Result2,[Goal]),
        set_mix_result(Result2,Result3),
        set_mix_ans(Goal,Result3,Result).

% ------------------------------------------------------------
% set_mix_ans(A,B,C):
%       A       - Goal which has been mixed
%       B       - mixed result
%       C       - the form of the result to be passed back to calling program
%
% set_mix_ans simply formats the result of the mixed computation
% the form of a Prolog program.

set_mix_ans(Goal,true,Goal) :- !.
set_mix_ans(Goal,(_:-true),Goal) :- !.
set_mix_ans(Goal,A,(Goal:-A)) :- !.

% ------------------------------------------------------------
% mix(Goals,Result,Recur):
%       Goals   - Goals to be mixed
%       Result  - Mixed result
%       Recur   - recursion list
%
% mix/5 is the core of the mixed computation utility.
% It is a meta-interpreter which executes the Goals under the current Prolog
% environment. The algorithm used simply tries to solve the Goals, halting
% further interpretation when Goal inhibition conditions are satisfied:
%       1. The goal is a builtin predicate, but it has uninstantiated
%               variables which prevent it from being executed.
%       2. The goal has uninstantiated variables which must be
%               instantiated according to the 'var_inst' clause.
%       3. The goal occurs in the recursion list - a loop las been detected.
%       4. No clause exists for solving the goal.
% Whenever one of these conditions arise, the goal is saved as a mixed result.
```

```
mix((Goal,Rest),Result,Recur) :-          % split multiple goals
        !,
        mix(Goal,GResult,Recur),
        mix(Rest,RResult,Recur),
        fuse((GResult,RResult),Result).

mix(Goal,true,_) :-                       % execute builtin's
        builtin_executable(Goal),
        !,
        Goal.

mix(Goal,Goal,Recur) :-                   % halt interpretation
        inhibited(Goal,Recur),
        !.

mix(Goal,Result,Recur) :-                 % interpret goal
        clause(Goal,Body),
        mix(Body,Result2,[Goal|Recur]),
        set_mix_result(Result2,Result).

% ____________________________________________________
% set_mix_result(A,B):
%       A       - Mixed result
%       B       - returned result
%
% set_mix_result reformats the mixed result.

set_mix_result((_:-A),A) :- !.
set_mix_result(A,A) :- !.

% ____________________________________________________
% inhibited(Goal,Recur):
%       Goal    - current Goal being interpreted
%       Recur   - Current recursion list
%
% inhibited checks if any Goal inhibition condition exists.

inhibited(Goal,_) :-        builtin(Goal),!.
inhibited(Goal,_) :-
        functor(Goal,Name,_),
        storematch(Name,_,_,_,_),
        !.
```

```
inhibited(Goal,_) :-        not var_inst(Goal),!.
inhibited(Goal,Recur) :- member3(Goal,Recur),!.
inhibited(Goal,_) :-        not clause(Goal,_),!.

% ____________________________________________________
% unknown_vars(Goal):
%       Goal    - current goal being interpreted
%
% unknown_vars checks if the builtin Goal has any uninstantiated variables
% as part of its executable structure (tests, expressions).

unknown_vars(asgn(_,_,E,_)) :-
        !,
        extract_vars(E,F),
        F \== [].

unknown_vars(callasgn(_,_,E)) :-
        !,
        extract_vars(E,F),
        F \== [].

unknown_vars(Test) :-
        is_a_test(Test),
        !,
        extract_vars(Test,F),
        F \== [].

% ____________________________________________________
% executable_builtin tests if a goal is builtin and executable.
% This may be specialized for individual builtin utilities in the future.

builtin_executable(Goal) :-
        builtin(Goal),
        not unknown_vars(Goal),
        !.
% ____________________________________________________
% builtin goal list:

builtin(true) :- !.
builtin(asgn(_,_,_,_)) :- !.
builtin(callasgn(_,_,_)) :- !.
builtin(Test) :- is_a_test(Test), !.
```

```
% ____________________________________________________
% var_inst(Goal):
%       Goal    - current goal being interpreted
%
% var_inst takes Goal and compares its input vector with the variable
% instantiation flag settings in the Goal's corresponding var_inst clause.
% Each variable in the input vector corresponding to a "i" (instantiated)
% setting must be an instantiated/ground variable; else failure.

var_inst(Goal) :-
        Goal =.. [Name,Vec|_],
        var_inst(Name,Inst),
        !,
        var_inst2(Vec,Inst).
var_inst(_).

var_inst2([],[]).
var_inst2([_|R],[f|S]) :-
        !,
        var_inst2(R,S).
var_inst2([V|R],[i|S]) :-
        !,
        nonvar(V),
        var_inst2(R,S).

% ____________________________________________________
% add_old_prolog(Mixed,New):
%       Mixed   - mixed Prolog program
%       New     - Mixed with added Prolog to make it complete
%
% add_old_prolog/2 completes the mixed prolog program.

add_old_prolog(Mixed,New) :-
        add_old_prolog(Mixed,Mixed,[],New).

% ____________________________________________________
% add_old_prolog(Mixed,Clauses,Done,Ans):
%       Mixed   - Mixed result so far
%       Clauses - clauses yet to check for completeness
%       Done    - names of predicates completed so far
%       Ans     - final result
```

```
%
% add_old_prolog/4 adds to the current Prolog program in Mixed clauses from
% the Prolog environment to make it complete. It calls add_old_prolog2 on the
% list of goals in each clause of Clause, until all of Clause has been
% processed. The names of the predicates completed are stored in Done.

add_old_prolog(Mixed,[],_,Mixed).

add_old_prolog(Mixed,[A|R],Done,S) :-
        split_clause(A,_,Goals),
        add_old_prolog2(Mixed,Goals,Done,Done2,More),
        Mixed \== More,
        add_old_prolog(More,More,Done2,S).

add_old_prolog(Mixed,[A|R],Done,S) :-
        add_old_prolog(Mixed,R,Done,S).

% ____________________________________________________
% add_old_prolog2(Mixed,Goals,OldDone,NewDone,Ans):
%       Mixed   - Current mixed program
%       Goals   - goal list to process
%       OldDone - predicates processed so far
%       NewDone - all the predicates processed after Goals processed
%       Ans     - final result
%
% add_old_prolog2 completes the Prolog program Mixed with respect to the
% goal list Goals. Should a partially ground goal not have a corresponding
% partially ground header in Mixed, the general predicate is copied from
% the Prolog environment and added to Mixed. The name of the predicate is
% saved to avoid duplication later.

add_old_prolog2(Mixed,[],Done,Done,Mixed).

add_old_prolog2(Mixed,[Goal|Rest],Done,Done2,More) :-
        not builtin(Goal),
        functor(Goal,Name,_),
        member(Name,Done),
        !,
        add_old_prolog2(Mixed,Rest,Done,Done2,More).

add_old_prolog2(Mixed,[Goal|Rest],Done,Done2,More) :-
        not builtin(Goal),
```

```
        not head_exists(Goal,Mixed),
        !,
        setof(Clause,get_clause(Goal,Clause),More2),
        append(Mixed,More2,More3),
        functor(Goal,Name,_),
        add_old_prolog2(More3,Rest,[Name|Done],Done2,More).

add_old_prolog2(Mixed,[_|Rest],Done,Done2,More) :-
        add_old_prolog2(Mixed,Rest,Done,Done2,More).

% ------------------------------------------------------------
% get_clause(Goal,Clause):
%       Goal    - (possibly partially) ground goal
%       Clause  - a general clause matching Head
%
% get_clause retrieves a clause to solve Head without unifying anything in
% Head.

get_clause(Goal,(Template:-Body)) :-
        functor(Goal,Name,Arity),
        current_predicate(Name,Template),
        functor(Template,Name,Arity),
        clause(Template,Body).

% ------------------------------------------------------------
% head_exists(Goal,Clauses):
%       Goal    - partially ground goal
%       Clauses - Prolog clauses
%
% head_exists tests if a clause with a partially ground head corresponding to
% Head exists in Clauses.

head_exists(Goal,[(H:-B)|Rest]) :-
        ground_eq(Goal,H).

head_exists(Goal,[_|Rest]) :-
        head_exists(Goal,Rest).
```

Appendix B

Bruce Parser

```
% ***
% *** Bruce parser and predicate generator
% ***
% ***    - Brian J. Ross
% ***      Dept. of Computer Science
% ***      University of British Columbia
% ***

% ____________________________________________________
% This DCTG grammar contains the Bruce syntax.
% (BRUCE: Brian Ross's Unfriendly Computation Environment)
% The semantic portions allow compilation of a Prolog equivalent for the
% Bruce program being parsed.
% This grammar expects a list of lexical tokens, as is generated by  Westcott's
% lexical analyzer.
%
% The semantics produce a list whose elements are Prolog clauses.
% The usual structures being passed through the semantics are:
%       Code  - the code for this construct
%       In    - the initial variable vector
%       Out   - the output variable vector
%       Defns - list of Prolog code created through the parsing of a construct
% Extra arguments are described where appropriate.

env ::= prog^^P, env^^E
<:>
code(Code) ::-
```

```
        P^^code(C),
        E^^code(Defns),
        append(C,Defns,Code).

env ::= []
<:>
code([]).

prog ::= prog_header^^P, ['{'], chain^^Q, ['}']
<:>
code(Code) ::-
        gen_prog(Code,P,Q).

prog_header ::= label^^L, ['('], var_list^^V, [','], var_list^^E, [')',':'],
                                decls^^D
<:>
code(P,VI,EI,Locals,Evars,In,Out) ::-
        gen_headercode(P,VI,EI,Locals,In,Out,Evars,L,V,E,D,Proglabel),
        new_labellist(Proglabel).

decls ::= ['local'], var_list^^L
<:>
code(Code) ::- L^^code(Code).

decls ::= []
<:>
code([]).

chain ::= stat^^S, [';'], chain^^Q
<:>
code(Code,In,Out,Defns) ::-
        gen_chain(Code,In,Out,Defns,S,Q).

chain ::= stat^^S
<:>
code(Code,In,Out,Defns) ::- S^^code(Code,In,Out,Defns).

chain ::= label^^L, [':'], stat^^S, [';'], chain^^Q
<:>
code((label(Label),Code),In,Out,Defns) ::-
        gen_chain(Code,In,Out,Defns,S,Q),
        L^^code(Label),
```

```
        add_curr_labellist(label,Label).

chain ::= label^^L, [':'], stat^^S
<:>
code((label(Label),Code),In,Out,Defns) ::-
        S^^code(Code,In,Out,Defns),
        L^^code(Label),
        add_curr_labellist(label,Label).

% ____________________________________________________________
% The following are calls to the statement constructs

stat ::= test^^S
<:>
code(Code,In,Out,Defns) ::- S^^code(Code,In,Out,Defns).

stat ::= proc_call^^S
<:>
code(Code,In,Out,Defns) ::- S^^code(Code,In,Out,Defns).

stat ::= while^^S
<:>
code(Code,In,Out,Defns) ::- S^^code(Code,In,Out,Defns).

stat ::= repeat^^S
<:>
code(Code,In,Out,Defns) ::- S^^code(Code,In,Out,Defns).

stat ::= branch^^S
<:>
code(Code,In,Out,Defns) ::- S^^code(Code,In,Out,Defns).

stat ::= skip^^S
<:>
code(Code,In,Out,Defns) ::- S^^code(Code,In,Out,Defns).

stat ::= assignment^^A
<:>
code(Code,In,Out,Defns) ::- A^^code(Code,In,Out,Defns).

% ____________________________________________________________
% The following are rules which create Prolog construct.
```

```
skip ::= [skip]
<:>
code(skip(In,In),In,In,[]).

assignment ::= label^^V, [':='], any_expr^^E
<:>
code(Code,In,Out,Defns) ::-
        gen_asgn(Code,In,Out,Defns,V,E).

test ::= ['if','('], bool_expr^^B, [')','{'], chain^^Q1, ['}'],
                                        [else,'{'], chain^^Q2, ['}']
<:>
code(Code,In,Out,Defns) ::-
        gen_testcode(Code,In,Out,Defns,B,Q1,Q2,If_label),
        add_curr_labellist(if,If_label).

test ::= ['if','('], bool_expr^^B, [')','{'], chain^^Q1, ['}']
<:>
code(Code,In,Out,Defns) ::-
        gen_testcode(Code,In,Out,Defns,B,Q1,'',If_label),
        add_curr_labellist(if,If_label).

proc_call ::= [call], label^^P, ['('], var_list^^V, [','], expr_outputs^^E,[')']
<:>
code(Code,In,Out,[]) ::-
        gen_callcode(Code,In,Out,P,V,E).

while ::= [while,'('], bool_expr^^B, [')','{'], chain^^Q, ['}']
<:>
code(Code,In,Out,Defns) ::-
        gen_whilecode(Code,In,Out,Defns,B,Q,Loop_label),
        add_curr_labellist(while,Loop_label).

repeat ::= [repeat,'{'], chain^^Q, ['}',until,'('], bool_expr^^B, [')']
<:>
code(Code,In,Out,Defns) ::-
        gen_repeatcode(Code,In,Out,Defns,B,Q,Loop_label),
        add_curr_labellist(repeat,Loop_label).

branch ::= [goto], label^^L
<:>
```

```
code(Code,In,Out,[]) ::-
        gen_goto(Code,In,Out,L),
        goto_found.

% ____________________________________________________
% Some miscellaneous expression rules ...

label    ::= [id(Name)]
<:>
code(Name).

any_expr ::= expr^^E
<:>
code(Expr,In) ::- E^^code(Expr,In).

any_expr ::= bool_expr^^B
<:>
code(Expr,In) ::- B^^code(Expr,In).

bool_expr ::= [true]
<:>
code(true,_).

bool_expr ::= [false]
<:>
code(false,_).

bool_expr ::= label^^V
<:>
code(Code,In) ::- V^^code(Name),var_match(Name,In,Code).

bool_expr ::= ['('], any_expr^^A, relop^^B, any_expr^^C, [')']
<:>
code((E),In) ::- A^^code(E1,In), B^^code(E2), C^^code(E3,In),
                 E =.. [E2,E1,E3].

bool_expr ::= ['('], bool_expr^^A, binbool_op^^B, bool_expr^^C, [')']
<:>
code((E),In) ::- A^^code(E1,In), B^^code(E2), C^^code(E3,In),
                 E =.. [E2,E1,E3].

bool_expr ::= ['(','~'], bool_expr^^A, [')']
```

```
<:>
code(~(E),In) ::- A^^code(E,In).

expr ::= ['('], expr^^A, math_op^^B, expr^^C, [')']
<:>
code((E),In) ::- A^^code(E1,In), B^^code(E2), C^^code(E3,In),
                 E =.. [E2,E1,E3]. % E2 should be defined as a CProlog operator

expr ::= [const(Num)]
<:>
code(Num,_).

expr ::= label^^V
<:>
code(Code,In) ::- V^^code(Name),var_match(Name,In,Code).

binbool_op ::= [bool(C)]
<:>
code(C).

relop ::= [rel(C)]
<:>
code(C).

math_op ::= [math(C)]
<:>
code(C).

% ____________________________________________________
% Lists...

% A list of variable labels ...

var_list  ::= ['['], label_list^^V, [']']
<:>
code(Code) ::- V^^code(Code).

% A list of expressions, with appropriate logical variable substitutions...

expr_outputs ::= ['['], expr_list^^V, [']']
<:>
code(Code,In) ::- V^^code(Code,In).
```

```
% Generates a label list...

label_list ::= label^^L, [','], label_list^^V
<:>
code([Label|Labels]) ::- L^^code(Label), V^^code(Labels).

label_list ::= label^^L
<:>
code([Label]) ::- L^^code(Label).

label_list ::= []
<:>
code([]).

% Generates an expression list ...

expr_list ::= expr^^E, [','], expr_list^^L
<:>
code(Code,In) ::-
        E^^code(Expr,In),
        L^^code(Elist,In),
        append([Expr],Elist,Code).

expr_list ::= bool_expr^^B, [','], expr_list^^L
<:>
code(Code,In) ::-
        B^^code(Expr,In),
        L^^code(Elist,In),
        append([Expr],Elist,Code).

expr_list ::= expr^^E
<:>
code([Code],In) ::- E^^code(Code,In).

expr_list ::= bool_expr^^B
<:>
code([Code],In) ::- B^^code(Code,In).

expr_list ::= []
<:>
code([],_).
```

```
% ____________________________________________________
% Some semantic primitives

% asgn(In,Var,Expr,Out) is satisfied by evaluating Expr  and saving the value
% in logical variable Var, where In contains input values which may be
% needed by Expr, and Out is the final output value (variant with logical
% variable Var changed).

asgn(In,Var,Expr,Out) :-
        value(Expr,Var).

% callasgn(In,Var,Expr) evaluates Expr, saving the value in Var.
% The In vector is not directly used.
% This is needed to pass expression values in procedure calls.

callasgn(In,Var,Expr) :-
        value(Expr,Var).

% skip always succeeds.

skip(_,_).
skip_branch(_,Dest,_) :- writel(nl,'Ignoring branch to ',Dest,nl).

% label inserts do nothing...

label(_).

% ____________________________________________________
% Prolog code construction ...
% These routines construct Prolog code for the particular constructs
% in question. In doing so, they may call semantic portions of the
% tree. The code from these calls, if any, is passed back in Defns.

% ____________________________________________________
% gen_prog(Code,P,Q):
%       P - points to header portion of tree
%       Q - points to main program chain
%
% Returns in list Code:
%       p(In,Out) :- Body PLUS definitions for rest of program
```

```
gen_prog(Code,P,Q) :-
        P^^code(Header,V_inp,E_inp,Locals,Evars,In,Out),
        Q^^code(Body,In,O2,Defns),
        gen_vec(V_inp,X1),
        append(X1,Evars,Out),
        append(X1,_,O2),
        Pgm =.. [(:-),Header,Body],
        append([Pgm],Defns,Code).

% ------------------------------------------------------------
% gen_headercode(Code,VI,EI,Locals,In,Out,Evars,L,V,E,D,Prog):
% Asserts storematch(Prog,B,VI,EI,LOC)
%     where Prog - program name,
%           B    - label list of var params, val params, and locals resp.
%           VI   - label list of var params
%           EI   - label list of val params
%           Locals - label list of locals

gen_headercode(Code,VI,EI,Locals,In,Out,Evars,L,V,E,D,Prog) :-
        L^^code(Prog),
        V^^code(VI),
        E^^code(EI),
        D^^code(Locals),
        append_list([VI,EI,Locals],Varnames),
        gen_vec(VI,Ivars),
        gen_vec(EI,Evars),
        gen_vec(Locals,Lvars),
        append(Ivars,Evars,HeaderInp),
        append(HeaderInp,Lvars,In),
        Code =.. [Prog,HeaderInp,Out],
        tassert(storematch(Prog,Varnames,VI,EI,Locals)).

% ------------------------------------------------------------

gen_testcode(Code,In,Out,Defns,B,Q1,'',Label) :-
        gen_vec(In,Out),
        B^^code(Booltest,In),
        Q1^^code(Chain1,In,O2,D2),
        gen_label(test,Label),
        Code =.. [Label,In,Out],
        Testhead =.. [Label,In,O2],
        Testhead2 =.. [Label,In,In],
```

```
        copy((Testhead :- (Booltest,Chain1)),Test1),
        copy((Testhead2 :- ~(Booltest)),Test2),
        append([Test1,Test2],D2,Defns).

gen_testcode(Code,In,Out,Defns,B,Q1,Q2,Label) :-
        Q2 \== '',
        gen_vec(In,Out),
        B^^code(Booltest,In),
        Q1^^code(Chain1,In,O2,D2),
        Q2^^code(Chain2,In,O3,D3),
        gen_label(test,Label),
        Code =.. [Label,In,Out],
        Testhead1 =.. [Label,In,O2],
        Testhead2 =.. [Label,In,O3],
        copy((Testhead1 :- (Booltest,Chain1)),Test1),
        copy((Testhead2 :- (~(Booltest),Chain2)),Test2),
        append([Test1,Test2],[D2|D3],Defns).

% ____________________________________________________

gen_whilecode(Code,In,Out,Defns,B,Q,Label) :-
        gen_vec(In,Out),
        gen_vec(In,O3),
        B^^code(Booltest,In),
        Q^^code(Chain1,In,O2,D2),
        gen_label(while,Label),
        Code =.. [Label,In,Out],
        Loophead1 =.. [Label,In,O3],
        Loophead2 =.. [Label,In,In],
        Loopcall =.. [Label,O2,O3],
        copy((Loophead1 :- (Booltest,Chain1,Loopcall)),Loop1),
        copy((Loophead2 :- ~(Booltest)),Loop2),
        gen_var_inst(Label,Booltest,In,Var_inst),
        append([Loop1,Loop2,Var_inst],D2,Defns).

% ____________________________________________________
% gen_repeatcode(Code,In,Out,Defns,B,Q):

gen_repeatcode(Code,In,Out,Defns,B,Q,Label) :-
        gen_vec(In,Out),
        gen_vec(In,O3),
        B^^code(Booltest,O2),
```

```
        Q^^code(Chain1,In,O2,D2),
        gen_label(repeat,Label),
        Code =.. [Label,In,Out],
        Loophead1 =.. [Label,In,O3],
        Loophead2 =.. [Label,In,O2],
        Loopcall =.. [Label,O2,O3],
        copy((Loophead1 :- (Chain1,~(Booltest),Loopcall)),Loop1),
        copy((Loophead2 :- (Chain1,Booltest)),Loop2),
        gen_var_inst(Label,Booltest,In,Var_inst),
        append([Loop1,Loop2,Var_inst],D2,Defns).

% ____________________________________________________

gen_callcode(Code,In,Out,P,V,E) :-
        P^^code(Proc),
        V^^code(Vlist),
        E^^code(Elist,In),
        match_vars(Vlist,In,Call_invars),
        % gen_vec(Vlist,I2),
        % append(I2,I3,In),
        set_expressions(In,Elist,Call_invals,Ecode),
        set_expressions(I3,Elist,Call_invals,Ecode),
        append(Call_invars,Call_invals,In_args),
        gen_vec(In_args,Out_args),
        set_outputs(In,Call_invars,Out_args,Out),
        Call =.. [Proc,In_args,Out_args],
        (Ecode == '' -> Code = Call;
                        fuse((Ecode,Call),Code)).

% ____________________________________________________

gen_asgn(asgn(In,Var_f,Expr,Out),In,Out,[],V,E) :-
        V^^code(Vname),
        E^^code(Expr,In),
        var_match(Vname,In,Var_i),
        vec_variant(In,Var_i,Var_f,Out).

% ____________________________________________________

gen_chain(Code,In,Out,Defns,S,Q) :-
        S^^code(S1,In,X,D1),
        Q^^code(S2,X,Out,D2),
```

```
        fuse((S1,S2),Code),
        append(D1,D2,Defns).

% ____________________________________________________________
% gen_goto(Code,In,Out,L):
%       Code    - goto construct
%       In      - input variable vector
%       Out     - output variable vector
%       L       - points to label

gen_goto(goto(In,Label,Out),In,Out,L) :-
        L^^code(Label),
        gen_vec(In,Out).

% ____________________________________________________________
% set_expressions(In,Elist,Call_invals,Ecode) converts expressions in Elist
% into callasgn statements in Ecode. Call_invals are the variables created
% to assign the expression values to before the call.

set_expressions(_,[],[],'').
set_expressions(In,[VarA|A],[VarB|B],D) :-
        var(VarA),
        set_expressions(In,A,B,C),
        fuse((callasgn(In,VarB,VarA),C),D).
set_expressions(In,[Expr|A],[Var|B],C) :-
        set_expressions(In,A,B,D),
        (D = '' -> C = callasgn(In,Var,Expr);
                   C = (callasgn(In,Var,Expr),D)).


% ____________________________________________________________
% goto_found asserts 'found_a_branch' if a branch is detected during
% parsing.

goto_found :- found_a_branch,!.
goto_found :- tassert(found_a_branch).

% ____________________________________________________________
% gen_var_inst generates an instantiation vector for a construct.
% This vector simply states which variables are used in the construct's
% test, which is used by the mixer later.
```

```
gen_var_inst(Label,Test,Vec,var_inst(Label,Inst)) :-
        extract_vars(Test,Tvars),
        gen_var_inst2(Tvars,Vec,Inst),
        !.

% ____________________________________________________
% gen_var_inst2 creates a Flags vector which consists of letters 'i' and
% 'f'. 'i' corresponds to each variable in Vec which is found in TestVars;
% otherwise 'f' is inserted.

gen_var_inst2(_,[],[]) :- !.
gen_var_inst2(Vars,[Var|Rest],[Flag|Flags]) :-
        (member(Var,Vars) ->
                Flag = i
                ;
                Flag = f),
        gen_var_inst2(Vars,Rest,Flags).


% ***
% *** Syntax definitions for Westcott's lexical analyzer
% ***

white_space_char(32).       % space
white_space_char(10).       % carriage return
white_space_char(9).        % tab
white_space_char(12).       % form feed

?-op(900,fx,'~').
?-op(1100,yfx,'#').
?-op(1000,yfx,'&').


op_chars("+",math('+')).
op_chars("-",math('-')).
op_chars("*",math('*')).
op_chars("/",math('/')).
op_chars("//",math('//')).
op_chars("&",bool('&')).
op_chars("#",bool('#')).
op_chars("~",bool('~')).
op_chars("<",rel('<')).
```

```
op_chars(">",rel('>')).
op_chars("=<",rel('=<')).
op_chars(">=",rel('>=')).
op_chars("=:=",rel('=:=')).
op_chars("=\=",rel('=\=')).

punct_chars("{",'{').
punct_chars("}",'}').
punct_chars(",",',').
punct_chars("(",'(').
punct_chars(")",')').
punct_chars("[",'[').
punct_chars("]",']').
punct_chars(";",';').
punct_chars(":",':').
punct_chars(":=",':=').

reserved(local,local).
reserved(if,if).
reserved(else,else).
reserved(while,while).
reserved(repeat,repeat).
reserved(until,until).
reserved(call,call).
reserved(goto,goto).
reserved(skip,skip).
reserved(true,true).
reserved(false,false).

constant_form(C,const(C)).

% ***
% *** expression evaluation routines...
% ***

~T :- not T.

A # B :-  (A;B).

A & B :-  A,B.

value(Expr,Val) :-
```

```
        Val is Expr.

% ***
% *** label generation and numbering
% ***

% ____________________________________________________
% reset_gen_nums sets gen_nums to 1.

reset_gen_nums :-
        abolish(gen_nums,4),
        tassert(gen_nums(1,1,1,1)).  % test, while, repeat, label

% ____________________________________________________
% get_gen_num(Type,Num) returns a new unique number for the structure
% Type (test,while,repeat,label).

get_gen_num(test,Num) :-
        gen_nums(Num,X,Y,Z),
        Next is Num + 1,
        abolish(gen_nums,4),
        tassert(gen_nums(Next,X,Y,Z)).

get_gen_num(while,Num) :-
        gen_nums(X,Num,Y,Z),
        Next is Num + 1,
        abolish(gen_nums,4),
        tassert(gen_nums(X,Next,Y,Z)).

get_gen_num(repeat,Num) :-
        gen_nums(X,Y,Num,Z),
        Next is Num + 1,
        abolish(gen_nums,4),
        tassert(gen_nums(X,Y,Next,Z)).

get_gen_num(label,Num) :-
        gen_nums(X,Y,Z,Num),
        Next is Num + 1,
        abolish(gen_nums,4),
        tassert(gen_nums(X,Y,Z,Next)).

% ____________________________________________________
```

```
% gen_label(Labeltype,Label) generates a new Label for structure Labeltype.

gen_label(Labeltype,Label) :-
        get_gen_num(Labeltype,Num),
        atom_append(Labeltype,Num,Label),
        !.

% ____________________________________________________
% new_labellist resets the labellist for program Prog, and asserts that
% Prog is the current program being processed.

new_labellist(Prog) :-
        (retract(labellist(Prog,_,_));true),
        tassert(labellist(Prog,[],[])),
        tabolish(curr_prog,1),
        tassert(curr_prog(Prog)),
        !.

% ____________________________________________________
% add_curr_labellist(Type,Label) adds Label to the label list for the current
% program being processed, determined by curr_prog.

add_curr_labellist(label,Label) :-
        curr_prog(Prog),
        retract(labellist(Prog,List1,List2)),
        append([Label],List1,Newlist1),
        append([Label],List2,Newlist2),
        tassert(labellist(Prog,Newlist1,Newlist2)),
        !.

add_curr_labellist(_,Label) :-
        curr_prog(Prog),
        retract(labellist(Prog,List1,List2)),
        append([Label],List1,Newlist),
        tassert(labellist(Prog,Newlist,List2)),
        !.

% ____________________________________________________
% add_approp_labellist(Label,Newlabel):
%
% add_approp_labellist adds Newlabel to the general label list which also
% contains Label.
```

```
add_approp_labellist(Label,Newlabel) :-
        labellist(A,List1,B),
        member2(Label,List1),
        retract(labellist(A,List1,B)),
        append([Newlabel],List1,Newlist),
        tassert(labellist(A,Newlist,B)),
        !.

% ____________________________________________________
% labelpgm(Clause) is satisfied if Clause is a label program.

labelpgm((H:-B)) :-
        functor(H,Name,_),
        labellist(_,_,Labels),
        member2(Name,Labels).
```

Appendix C

Branch Handling

```
% ***
% *** Prolog branch transformation utilities
% ***
% ***    - Brian J. Ross
% ***      Dept. of Computer Science
% ***      University of British Columbia
% ***

% ____________________________________________________
% process_branches(Prolog,Newprolog) transforms the clauses in Prolog so
% that branches are properly handled. First, all chains of goals following
% branches are removed (label programs may exist which contain the code
% being thrown away).
% Then the prolog is transformed to account for the branches within descendent
% predicates. process_branches/3 is called to iterate through all the Prolog
% clauses. For each one with a goto, the goto goal is changed into a call to
% the appropriate label program. Then the parent predicates of the clause
% with this goto are transformed.

process_branches(Prolog,Newprolog) :-
        process_branches(Prolog,Prolog,Prolog2),
        chop_branch_chains(Prolog2,Prolog2,Newprolog),
        !.

process_branches(Prolog,[],Prolog) :- !.
process_branches(Prolog,[Clause|_],Newprolog) :-
        find_goal(Clause,Head,Left,goto(X,Y,Z),Right),
```

```
        create_labelcall(goto(X,Y,Z),Call),
        branch_restruct(Prolog,Clause,Head,Prolog2),
        insert_goal(Prolog2,Clause,Head,Left,[Call],Right,Prolog3),
        process_branches(Prolog3,Prolog3,Newprolog),
        !.
process_branches(Prolog,[_|R],Newprolog) :-
        process_branches(Prolog,R,Newprolog),
        !.

% ____________________________________________________
% branch_restruct restructures Prolog given a clause with a branch call/
% ancestor of a branch call, Goal.
% Case 1: Clause has a parent:
%       Ancestors of Clause are restructured, and code returned is used to
%       restructure the sibling of Clause.
% Case 2: Clause has no parent:
%       Clause is a main program clause, and is left alone for now.

branch_restruct(Prolog,Clause,Head,Newprolog) :-
        restruct_parent(Prolog,Head,Pvec,Pcode,New2),
        restruct_sibling(New2,Clause,Pvec,Pcode,Newprolog),
        !.

branch_restruct(Prolog,_,_,Prolog) :-
        !.

% ____________________________________________________
% restruct_parent finds the non-label-program parent of Head.
% It then pulls goals from the right of the call to Head, resulting in Code.
% The branch_restruct is called on the parent, causing a recursive restructuring
% up the tree.

restruct_parent(Prolog,Head,Ivec,Code,Newprolog) :-
        parent_clause(Prolog,Head,Parent,Phead,Left,Goal,Right),
        not labelpgm(Parent),
        set_code(Phead,Left,Goal,Right,Ivec,Code),
        branch_restruct(Prolog,Parent,Phead,Newprolog),
        !.

% ____________________________________________________
% restruct_sibling adds Code to the end of the sibling of Clause,
% if one exists.
```

```
restruct_sibling(Prolog,Clause,Vec,Code,Newprolog) :-
        sibling_clause(Prolog,Clause,Sibling),
        append_goals(Sibling,Vec,Code,Newsibling),
        remove_item(Sibling,Prolog,New2),
        append([Newsibling],New2,Newprolog),
        !.
restruct_sibling(Prolog,_,_,_,Prolog) :- !.

% ____________________________________________________
% set_code returns the goals to the right of Goal, should they exist, along
% with their input vector.

set_code(Head,Left,Goal,[],_,[]) :-
        !.

set_code(Head,Left,Goal,Right,In,Code) :-
        get_IO([Head],I1,O1),
        append(Left,Goal,G),
        get_prev_IO(G,(I1,I1),I2,O2),
        get_prev_IO(Right,(O2,O2),I3,O3),
        copy((I3,Right),(In,Code)),
        !.

% ____________________________________________________
% restruct_clause restructures a clause by merging Left and Right together.
% restruct_clause2 and restruct_prog do the actual reconstruction.

restruct_clause(Head,Left,Call,Newclause) :-
        functor(Head,Name,_),
        storematch(Name,_,_,_,_),
        !,
        restruct_prog(Head,Left,Call,Newclause).

restruct_clause(Head,Left,Call,Newclause) :-
        restruct_clause2(Head,Left,Call,Newclause),
        !.

restruct_clause2(Head,[],Right,Code) :-
        Head =.. [Name,I1,O1],
        get_prev_IO(Right,(I1,I1),I2,O2),
        Newhead =.. [Name,I1,O2],
```

```
        list_to_body(Right,Body),
        copy((Newhead :- Body),Code),
        !.

restruct_clause2(Head,Left,[],Code) :-
        Head =.. [Name,I1,O1],
        get_prev_IO(Left,(I1,I1),I2,O2),
        Newhead =.. [Name,I1,O2],
        list_to_body(Left,Body),
        copy((Newhead :- Body),Code),
        !.

restruct_clause2(Head,Left,Right,Code) :-
        copy((Head,Left,Right),(Head2,Left2,Right2)),
        Head2 =.. [Name,I1,O1],
        get_prev_IO(Left2,(I1,I1),I2,O2),
        get_prev_IO(Right2,(O2,O2),I3,O3),
        O2 = I3,
        Newhead =.. [Name,I1,O3],
        append(Left2,Right2,B),
        list_to_body(B,Body),
        copy((Newhead :- Body),Code),
        !.

% ____________________________________________________
% restruct_prolog is almost like restruct_clause2.
% It creates a program clause by merging the goals in Left and
% Right, but removing local variables from the header IO vectors.

restruct_prog(Head,Left,Right,Newclause) :-
        Head =.. [Name,I1,O1],
        get_prev_IO(Left,(I1,I1),I2,O2),
        get_prev_IO(Right,(O2,O2),I3,O3),
        gen_vec(O1,O4),
        append(O4,_,O3),
        Newhead =.. [Name,I1,O4],
        append(Left,Right,B),
        list_to_body(B,Body),
        copy((Newhead :- Body),Newclause),
        !.

% ____________________________________________________
```

```
% create_labelcall changes a goal in goto format to a label call.

create_labelcall(goto(In,Label,Out),Call) :-
        Call =.. [Label,In,Out],
        !.

% ____________________________________________________
% chop_branch_chains(Prolog,Clauses,Collect) takes each clause with a
% label program call and throws away any goals following it.

chop_branch_chains(_,[],[]) :- !.

chop_branch_chains(Prolog,[Clause|R],T) :-
        split_clause(Clause,Head,Goals),
        check_for_branches(Prolog,Goals,Goals2),
        restruct_clause(Head,[],Goals2,Newclause),
        chop_branch_chains(Prolog,R,S),
        (member3(Newclause,S) ->
                T = S
                ;
                T = [Newclause|S]),
        !.

chop_branch_chains(Prolog,[Clause|R],[Clause|T]) :-
        chop_branch_chains(Prolog,R,T),
        !.

insert_goal(Prolog,Clause,Head,Left,Goal,Right,Newprolog) :-
        append_list([Left,Goal,Right],Goals),
        list_to_body(Goals,Body),
        Newclause =.. [(:-),Head,Body],
        (member3(Newclause,Prolog) ->
                Newprolog = Prolog
                ;
                remove_item(Clause,Prolog,New2),
                Newprolog = [Newclause|New2]),
        !.


% ***
% *** Label program generation
% ***
```

```
% ____________________________________________________
% gen_labelprogs(Prolog,Labelprogs) creates a new prolog environment which
% contains predicates for the label programs in Prolog. It calls a
% gen_labelprogs/3 version, where the second argument iterates through
% the clauses.
% The goals for the label program are obtained, and a label program is created.
% Then all label references for the program just created are removed from the
% prolog environment.

gen_labelprogs(Prolog,Newprolog) :-
        gen_labelprogs(Prolog,Prolog,Newprolog),
        !.

gen_labelprogs(Prolog,[],Prolog) :- !.

gen_labelprogs(Prolog,[Clause|R],Newprolog) :-
        find_goal(Clause,Head,Left,label(Label),Right),
        get_goals(Prolog,Clause,Head,Left,Label,Right,In,Out,Goals,Defns),
        create_labclause(Label,In,Out,Goals,Labelprog),
        append_list([[Labelprog],Defns,Prolog],Prolog2),
        rem_pred_labels(Prolog2,label(Label),Prolog3),
        gen_labelprogs(Prolog3,Prolog3,Newprolog),
        !.

gen_labelprogs(Prolog,[_|R],Newprolog) :-
        gen_labelprogs(Prolog,R,Newprolog),
        !.

% ____________________________________________________
% get_goals retrieves the goals to be used for a label program.
% If a goal in Right contains a branch, then all the goals
% up to and including that goal are used for the label program.
% Otherwise, the goals of the parent construct are obtained, and the
% result is appended to the goals in Right.

get_goals(Prolog,Clause,Head,Left,_,Right,In,Out,Goals,[]) :-
        branch_under_goals(Prolog,Head,Left,Right,In,Out,Goals),
        !.

get_goals(Prolog,Clause,Head,Left,Label,Right,In,Out,Goals,Defns) :-
        create_construct_control(Prolog,Head,Left,Label,Right,Cin,Cout,Control,Defns1),
```

```
        parent_goals(Prolog,Head,Pin,Pout,Pgoals,Defns2),
        append(Defns1,Defns2,Defns),
        merge_goals(Cin,Cout,Control,Pin,Pout,Pgoals,In,Out,Goals),
        !.

% ____________________________________________________
% parent_goals recursively obtains the parent goals to be used by a
% label program. The successive parent goals to the right of the call to a
% child predicate are obtained and concatenated together. This recursion
% stops when (a) the top predicate is reached, or (b) a branch has been
% detected under a goal. parent_goals2 is used as a subsidiary call.
% Note that label programs are not considered parents.

parent_goals(Prolog,Head,In,Out,Goals,Defns) :-
        parent_clause(Prolog,Head,Parent,Phead,Left,Goal,Right),
        not labelpgm(Parent),
        append(Left,[Goal],L2),
        functor(Goal,Goallabel,_),
        get_goals(Prolog,Parent,Phead,L2,Goallabel,Right,In,Out,Goals,Defns).

parent_goals(_,_,_,_,[],[]).

% ____________________________________________________
% branch_under_goals obtains all the goals up to and including a goal in Right
% which contains a branch. It fails should no branches be found under any goal
% in Right.

branch_under_goals(Prolog,Head,Left,Right,In,Out,Goals) :-
        check_for_branches(Prolog,Right,Goals2),
        get_IO([Head],I1,_),
        get_prev_IO(Left,(I1,I1),_,In2),
        get_prev_IO(Goals2,(In,In),_,Out2),
        copy((In2,Out2,Goals2),(In,Out,Goals)).

% ____________________________________________________
% check_for_branches does a search for branches under Goals.
% It keeps all the goals in Goals up to and including the first one with a
% descendent branch. It fails should no descendent branches be found.

check_for_branches(_,[],[]) :- !,fail.

check_for_branches(Prolog,[goto(I,L,O)|_],[goto(I,L,O)]).
```

```
check_for_branches(Prolog,[Goal|_],[Goal]) :-
        functor(Goal,Name,_),
        labellist(_,_,Labels),
        member(Name,Labels).

check_for_branches(Prolog,[Goal|R],[Goal]) :-
        descendent_branches(Prolog,Goal).

check_for_branches(Prolog,[Goal|R],[Goal|S]) :-
        check_for_branches(Prolog,R,S).

% ____________________________________________________
% descendent_branches performs a recursive search for branches under goal G.
% For efficency, G is analyzed for it's goal type so that needless
% search is not done on goals with no descendents (assignments, program calls,
% etc). descendent_branches2 is called to iterate through new goal lists
% obtained during the search.

descendent_branches(Prolog,G) :-
        member2(G,[asgn(_,_,_,_),callasgn(_,_,_),label(_),goto(_,_,_)]),
        !,
        fail.

descendent_branches(Prolog,T) :-
        is_a_test(T),
        !,
        fail.

descendent_branches(Prolog,Call) :-
        functor(Call,Prog,_),
        storematch(Prog,_,_,_,_),
        !,
        fail.

descendent_branches(Prolog,G) :-
        functor(G,Lab,_),
        labellist(_,_,Labels),
        member2(Lab,Labels),
        !.

descendent_branches(Prolog,Goal) :-
```

```
        find_clause(Prolog,Goal,Clause),
        remove_item(Clause,Prolog,Prolog2),
        split_clause(Clause,_,Body),
        descendent_branches2(Prolog2,Body).

descendent_branches2(_,[]) :- !,fail.

descendent_branches2(_,[goto(_,_,_)|_]) :- !.

descendent_branches2(Prolog,[Goal|_]) :-
        descendent_branches(Prolog,Goal),
        !.
descendent_branches2(Prolog,[_|R]) :-
        descendent_branches2(Prolog,R),
        !.

% ____________________________________________________
% create_labclause creates a label program.

create_labclause(Lab,In,Out,[],Clause) :-
        Header =.. [Lab,In,Out],
        copy(Header,Clause).

create_labclause(Lab,In,Out,Goals,(Header :- Body)) :-
        copy((In,Out,Goals),(I2,O2,Goals2)),
        Header =.. [Lab,I2,O2],
        list_to_body(Goals2,Body).

% ____________________________________________________
% create_construct_control creates a list of goals, Control, which comprise
% the particular control sequence needed when junping into the construct
% at the point Right. With bruce, there are two cases:
%       (1) jumping into a repeat - since repeat tests are done at the end of
%               the loop, the repeat test must be simulated. This requires
%               calling create_repeat_control to create this special control.
%       (2) jumping into any other mechanism - simply return the goals to the
%               right of the jump point.

create_construct_control(Prolog,Head,Left,Label,Right,In,Out,Control,Defns) :-
        functor(Head,Name,_),
        get_label_refs(Prolog,Name,[],[A,B]),
        test_if_repeat(A,B,OutT,Test,Body),
```

```
        create_repeat_control(OutT,Test,Body,Label,Head,Left,Right,In,Out,Control,Defns).

create_construct_control(Prolog,Head,Left,_,Right,In,Out,Control,[]) :-
        get_IO([Head],I1,_),
        get_prev_IO(Left,(I1,I1),_,O2),
        get_prev_IO(Right,(O2,O2),I3,O3),
        copy((Right,I3,O3),(Control,In,Out)).

% ____________________________________________________
% create_repeat_control creates the control environment for a repeat construct.
% The output is
%                 Control = [Body,Test]
%                 Defns = [(Test1 :- ~Test,Head),(Test1 :- Test)]
% all suitably tidied.
% The name for Test is externally saved through calling add_approp_labellist.

create_repeat_control(Vec,Test,Body,Label,Head,Left,Right,In,Out,Control,[D1,D2]) :-
        copy((Vec,Test,Body),(Vec2,Test2,Body2)),
        copy((Head,Left,Right),(Head2,Left2,Right2)),
        append(Left2,Rightgoals,Body2),  % get goals after jump point
        functor(Head,Name,_),
        get_IO([Head2],I1,_),
        get_prev_IO(Left2,(I1,I1),_,O2),
        get_prev_IO(Rightgoals,(O2,O2),I3,O3),
        gen_label(test,Testlabel),
        gen_vec(Vec2,Vout),
        HeadA =.. [Testlabel,Vec2,Vout],
        HeadB =.. [Testlabel,Vec2,Vec2],
        Call  =.. [Name,Vec2,Vout],
        list_to_body([~(Test2),Call],BodyA),
        copy((HeadA:-BodyA),D1),
        copy((HeadB:-Test2),D2),
        add_approp_labellist(Label,Testlabel),
        merge_goals(I3,O3,Rightgoals,Vec2,Vout,[HeadA],In,Out,Control).

% ____________________________________________________
% rem_pred_labels simply removes all goals of the form Label from Clauses.
% This is done so that we don't construct duplicate label programs.

rem_pred_labels([],_,[]) :- !.

rem_pred_labels([Clause|R],Label,[Newclause|S]) :-
```

```
        find_goal(Clause,Head,Left,Label,Right),
        make_clause(Head,Left,Right,Newclause),
        rem_pred_labels(R,Label,S),
        !.

rem_pred_labels([Clause|R],Label,[Clause|S]) :-
        rem_pred_labels(R,Label,S),
        !.
```

Appendix D

Bruce Generation

```
% ***
% *** Bruce code generation facility
% ***
% ***    - Brian J. Ross
% ***      Dept. of Computer Science
% ***      University of British Columbia
% ***


% ____________________________________________________________
% brucify(Env) converts the entire Prolog environment in list Env
% into Bruce code.

brucify(Env,Bruce) :-
        bagof([P|Q],X^labellist(P,Q,X),Labels),
        copy(Env,Env2),
        program_separate(Env2,Labels,[],Prolog),
        brucify_programs(Prolog,[],Bruce).

% ____________________________________________________________
% brucify_programs(Programs,Collect,Result) converts each prolog program
% in Programs to Bruce code.

brucify_programs([],Bruce,Bruce).
brucify_programs([Progname,Prog|Rest],Collect,Bruce) :-
        get_label_refs(Prog,Progname,[],Clauses),
        get_IO(Clauses,In,_),
```

```
        template_match(Progname,In,Prog,Clauses,B,[],_),
        rem_unused_labels(B,B2),
        create_outval_asgns(Progname,Clauses,Asgns),
        chain_append(B2,Asgns,Body),
        bruce_header(Progname,Header),
        append_list([Header,['{'],Body,['}']],Bruce_prog),
        brucify_programs(Rest,[Bruce_prog|Collect],Bruce).

% ____________________________________________________
% template_match(Progname,Prolog,Clauses,Bruce,Labels,Newlabels):
% Matches Clauses to known bruce construct definitions.
% NOTE:  Should the dataflow of goals be altered, include the two lines noted
%        below.

template_match(_,_,_,[],_,_,_) :- !,fail.

template_match(Progname,_,Prolog,[Clause],Bruce,Labels,Newlabels) :-
        % split_clause(Clause,Head,Goals),          % <-- see note above
        % order_clause_goals(Goals,Ordered),        % <-- see note above
        split_clause(Clause,Head,Ordered),
        get_IO([Head],In,_),
        template_match_goals(Progname,In,Prolog,Ordered,Bruce,Labels,Newlabels).

% repeat:

template_match(Progname,Vec,Prolog,[A,B],Bruce,Labels,Newlabels) :-
        is_a_repeat(Progname,A,B,Test,Body),
        get_IO(A,_,Out),
        template_match_goals(Progname,Out,Prolog,Body,Bruce_body,Labels,Newlabels),
        append_list([[repeat,'{'],Bruce_body,['}',until,'(',Test,')']],Bruce).

% while:

template_match(Progname,Vec,Prolog,[A,B],Bruce,Labels,Newlabels) :-
        is_a_while(Progname,A,B,Test,Body),
        get_IO(A,_,Out),
        template_match_goals(Progname,Out,Prolog,Body,Bruce_body,Labels,Newlabels),
        append_list([['while (',Test,')','{'],Bruce_body,['}']],Bruce).

% if:

template_match(Progname,Vec,Prolog,[A,B],Bruce,Labels,Newlabels) :-
```

```
        is_an_if(Progname,A,B,Test,Thengoals,Elsegoals),
        get_IO(A,_,Out),
        template_match_goals(Progname,Out,Prolog,Thengoals,Thencode,Labels,Newlabels2),
        (Elsegoals == [] ->
            append_list([['if (',Test,')','{'],Thencode,['}']],Bruce),
            Newlabels = Newlabels2
            ;
            template_match_goals(Progname,Out,Prolog,Elsegoals,Elsecode,Newlabels2,Newlabels),
            append_list([['if (',Test,')','{'],Thencode,['}',else,'{'],
                                                    Elsecode,['}']],Bruce)).

template_match(Progname,Vec,Prolog,Clauses,Bruce,Labels,Newlabels) :-
        factor(Clauses,Prolog,NewClauses,NewProlog),
        template_match(Progname,Vec,NewProlog,NewClauses,Bruce,Labels,Newlabels).

% ____________________________________________________
% template_match_goals(Progname,Prolog,Goallist,Bruce,Labels,Newlabels):
% Matches each goal in Goallist to a construct.
% One of three situations is possible:
%       (a) Finished – empty Goal list
%       (b) Goal is an assignment: a Bruce assignment is created
%       (c) Goal is a call assignment: it is temporarily ignored until the
%               corresponding call is found.
%       (d) Goal is a procedure call: a Bruce proc. call is created.
%       (e) Goal refers to a structure previously created: create a
%           branch to it.
%       (f) Goal refers to a structure previously created, but step (e) failed:
%           then fail (unknown structure).
%       (g) none of the above: then the goal must be a call to a more complex
%               mechanism. All the clauses relevant to that goal are
%               extracted, and a template match is performed on them.

% case (a)
template_match_goals(_,_,_,[],[],L,L).

% case (b)
template_match_goals(Progname,_,Prolog,[asgn(In,Var,Expr,Out)|Rest],Bruce,Labels,Newlabels) :-
        !,
        template_match_goals(Progname,Out,Prolog,Rest,B,Labels,Newlabels),
        convert_bruce_expr(Progname,In,Expr,Expr2),
        convert_bruce_expr(Progname,Out,Var,Var2),
        chain_append([Var2,':=',Expr2],B,Bruce).
```

```
% case (c)
template_match_goals(Progname,Vec,Prolog,[callasgn(A,B,C)|Rest],Bruce,Label,Newlabels) :-
        !,
        append(Rest,[callasgn(A,B,C)],Rest2),
        template_match_goals(Progname,Vec,Prolog,Rest2,Bruce,Label,Newlabels).

% case (d)
template_match_goals(Progname,Vec,Prolog,[Goal|Rest],Bruce,Labels,Newlabels) :-
        Goal =.. [Prog,In,Out],
        storematch(Prog,Vars,Ref,Val,_),
        !,
        collect_callasgns(Rest,In,Calls),
        gen_vec(Val,V1),
        append(Refvec,V1,In),
        create_val_vector(V1,Calls,Ivec,Valvec),
        Call =.. [Prog,Refvec,Valvec],
        remove_items2(Calls,Rest,Rest2),
        template_match_goals(Progname,Out,Prolog,Rest2,B,Labels,Newlabels),
        convert_bruce_expr(Progname,Vec,Call,Call2),
        chain_append(['call',Call2],B,Bruce).

% case (e)
template_match_goals(Progname,Vec,Prolog,[Goal|Rest],[goto,Name],Labels,Labels) :-
        functor(Goal,Name,_),
        member2(Name,Labels).
        % gen_impl_asgns(Progname, Vec, Goal, Asgns),        % remove??
        % chain_append(Asgns,[goto,Name],Bruce).                % remove??

% case (f)
template_match_goals(_,_,_,[Goal|_],_,Labels,_) :-
        functor(Goal,Name,_),
        member2(Name,Labels),
        !,
        fail.

% case (g)
template_match_goals(Progname,Vec,Prolog,[Goal|Rest],Bruce,Labels,Newlabels) :-
        functor(Goal,Name,_),
        get_label_refs(Prolog,Name,[],Clauses),
        template_match(Progname,Vec,Prolog,Clauses,BruceA,[Name|Labels],Newlabels2),
        get_IO([Goal],In,Out),
```

```
        gen_impl_asgns(Progname,Vec,In,Asgns),
        template_match_goals(Progname,Out,Prolog,Rest,BruceB,Newlabels2,Newlabels),
        append([Name,':'],BruceA,X1),
        chain_append(Asgns,X1,X2),
        chain_append(X2,BruceB,Bruce).

% ____________________________________________________________
% is_an_if(Progname,A,B,Brucetest,Then,Else)
% Tests if goals A,B can be matched with a Bruce test construct, and
% creates Bruce code if they match.

is_an_if(Progname,A,B,Brucetest,Then,Else) :-
        test_if_if(A,B,I,Test,Then,Else),
        convert_bruce_expr(Progname,I,Test,Brucetest).

% ____________________________________________________________
% is_a_while(Progname,A,B,Brucetest,Body):
% Tests A and B for a match with a Bruce while construct, and
% creates Bruce code if they match.

is_a_while(Progname,A,B,Brucetest,Body) :-
        test_if_while(A,B,I,Test,Body),
        convert_bruce_expr(Progname,I,Test,Brucetest).

% ____________________________________________________________
% is_a_repeat(Progname,A,B,Brucetest,Body):
% Tests A and B for a match with a Bruce repeat construct,
% creates Bruce code if they match.

is_a_repeat(Progname,A,B,Brucetest,Body) :-
        test_if_repeat(A,B,O,Test,Body),
        convert_bruce_expr(Progname,O,Test,Brucetest).

% ____________________________________________________________
% complementary_tests(BodyA,BodyB,TA,TB) finds an instance of complementary
%       tests TA and TB in BodyA and BodyB (ie. TA = ~TB). Repeated calls
%       will return new complementary tests, if any.

complementary_tests(BodyA,BodyB,TA,TB) :-
        member2(TA,BodyA),
        is_a_test(TA),
        member2(TB,BodyB),
```

```
        is_a_test(TB),
        complementary(TA,TB).

% ____________________________________________________________
% convert_bruce_expr(Progname,In,Expr,Conv) created a copy of Expr having
%       the appropriate Bruce variable names, based on ordering in In.
%       The result is in Conv, as we don't want to bind the actual logical
%       variables (lose dataflow information).

convert_bruce_expr(Progname,In,Expr,Conv) :-
        copy((In,Expr),(In2,Conv)),
        set_all_to_vars(In2,In3),
        storematch(Progname,In3,_,_,_),
        !.

% ____________________________________________________________
% A bruce program header is created...

bruce_header(Progname,[A,':',local,Locals]) :-
        storematch(Progname,Vars,Ref,Val,Locals),
        A =.. [Progname,Ref,Val].

% ____________________________________________________________
% recursive_call(Call,Goals,C) is satisfied if a recursive call C to header
%       Call exists in the Goals list.

recursive_call(Call,Goals,C) :-
        copy(Call,C),
        member2(C,Goals),
        same_out_vars(Call,C).

% ____________________________________________________________
% collect_callasgns(Goals,Vars,Calls) finds instances of callasgn goals in Goals
% which assign values to variables listed in Vars. The results are in Calls.

collect_callasgns([],_,[]) :- !.
collect_callasgns([callasgn(A,Var,B)|R],Vars,[callasgn(A,Var,B)|D]) :-
        member(Var,Vars),
        collect_callasgns(R,Vars,D),
        !.
collect_callasgns([_|R],Vars,D) :-
        collect_callasgns(R,Vars,D),
```

```
        !.

% ____________________________________________________________
% create_val_vector(Vars,Calls,Valvec) creates a value parameter list
% for a Bruce procedure call.
% Variables in Vars which have a corresponding callasgn in Calls are replaced
% with the expression in that callasgn. Otherwise that variable is used.

create_val_vector([],_,_,[]) :- !.
create_val_vector([V|R],Callasgns,Ivec,[T|W]) :-
        (collect_callasgns(Callasgns,[V],[callasgn(Ivec,_,Expr)]) ->
                create_val_vector(R,Callasgns,_,W),
                T = Expr
                ;
                create_val_vector(R,Callasgns,Ivec,W),
                T = V),
        !.

% ____________________________________________________________
% chain_append(A,B,C) appends lists A and B together, such that if B is
% nonempty, a semicolon is inserted between them.

chain_append(A,[],A) :- !.
chain_append([],A,A) :- !.
chain_append(A,B,C) :-
        append_list([A,[';'],B],C),
        !.

% ____________________________________________________________
% rem_unused_labels(Bruce,Newbruce) removes labels from the bruce source
% which have no branches onto them.

rem_unused_labels(Bruce,Newbruce) :-
        rem_unused_labels(Bruce,Bruce,Newbruce),
        !.

rem_unused_labels(_,[],[]).
rem_unused_labels(U,[L,':'|R],[L,':'|A]) :-
        find_goto(L,U),
        rem_unused_labels(U,R,A).
rem_unused_labels(U,[L,':'|R],A) :-
        rem_unused_labels(U,R,A).
```

```
rem_unused_labels(U,[S|R],[S|A]) :-
        rem_unused_labels(U,R,A).

% find_goto(Label,Bruce) finds a branch to label Label in the Bruce code.

find_goto(Label,[goto,Label|_]) :- !.
find_goto(Label,[S|R]) :-
        find_goto(Label,R),
        !.

% ____________________________________________________
% test_if_if checks if A and B match with an if construct of Bruce.
% The matching criteria are:
%       (a) complementary tests in A and B
%       (b) Both tests are dependent on their header input variables

test_if_if(A,B,I,Test,Then,Else) :-
        split_clause(A,HeadA,BodyA),
        split_clause(B,HeadB,BodyB),
        complementary_tests(BodyA,BodyB,TA,TB),
        dep_in_vars(TA,HeadA),
        dep_in_vars(TB,HeadB),
        remove_item(TA,BodyA,NewA),
        remove_item(TB,BodyB,NewB),
        set_then_else(TA,HeadA,NewA,TB,HeadB,NewB,Test,Then,Else,Head),
        Head =.. [_,I,_].

% ____________________________________________________
% test_if_while checks if A and B match with a while construct of Bruce.
% The matching criteria are:
%       (a) complementary tests in A and B.
%       (b) Both tests are dependent on their header input variables
%       (c) One of the clauses has a body consisting only of the test.
%       (d) The other clause contains a recursive call matching the head.

test_if_while(A,B,I,Test,Body) :-
        split_clause(A,HeadA,BodyA),
        split_clause(B,HeadB,BodyB),
        complementary_tests(BodyA,BodyB,TA,TB),
        dep_in_vars(TA,HeadA),
        dep_in_vars(TB,HeadB),
        (BodyA == [TA],
```

```
                (Test,Head,Goals) = (TB,HeadB,BodyB)
                ;
                BodyB == [TB],
                (Test,Head,Goals) = (TA,HeadA,BodyA)),
        recursive_call(Head,Goals,Call),
        remove_items2([Test,Call],Goals,Body),
        Head =.. [_,I,_].

% ____________________________________________________
% test_if_while checks if A and B match with a repeat construct of Bruce.
% The matching criteria are:
%       (a) complementary tests in A and B.
%       (b) One clause contains a recursive call matching the head.
%       (c) The test in that clause is dependent on the headers input variables.
%       (d) The test in the other clause uses variables in the headers output
%               variables list.
%       (e) The remaining goals which should compose the repeat body are
%               the same (permutation).

test_if_repeat(A,B,O,Test,Body) :-
        split_clause(A,HeadA,BodyA),
        split_clause(B,HeadB,BodyB),
        complementary_tests(BodyA,BodyB,TA,TB),
        (recursive_call(HeadA,BodyA,Call),
                (T1,B1,Test,H2,B2) = (TA,BodyA,TB,HeadB,BodyB)
                ;
                recursive_call(HeadB,BodyB,Call),
                (T1,B1,Test,H2,B2) = (TB,BodyB,TA,HeadA,BodyA)),
        dep_in_vars(T1,Call),
        dep_out_vars(Test,H2),
        remove_item(Test,B2,Body),
        remove_items2([T1,Call],B1,New2),
        permutation(Body,New2),
        H2 =.. [_,_,O].

% ____________________________________________________
% set_then_else(TA,HeadA,NewA,TB,HeadB,NewB,Test,Then,Else,Head) sets
% Test, Then, Else, and Head appropriately.

set_then_else(TA,HeadA,[],TB,HeadB,NewB,TB,NewB,[],HeadB) :-
        NewB \== [],
        !.
```

```
set_then_else(TA,HeadA,NewA,TB,HeadB,[],TA,NewA,[],HeadA) :-
        NewA \== [],
        !.
set_then_else(TA,HeadA,NewA,TB,HeadB,NewB,TB,NewB,NewA,HeadB) :-
        functor(TA,'~',_),
        !.
set_then_else(TA,HeadA,NewA,TB,HeadB,NewB,TA,NewA,NewB,HeadA) :- !.

% ____________________________________________________
% set_all_to_vars copies A to B, replacing nonvar items with new var ones.

set_all_to_vars([],[]) :- !.
set_all_to_vars([A|B],[A|D]) :-
        var(A),
        !,
        set_all_to_vars(B,D).
set_all_to_vars([A|B],[C|D]) :-
        set_all_to_vars(B,D),
        !.

% ____________________________________________________
% create_outval_asgns creates an assignment for each call-by-value
% parameter for program Progname if that parameter has a ground value in the
% programs output vector (which can occur through mixed computation).

create_outval_asgns(Progname,[(H:-B)],Asgns) :-
        get_IO([H],_,Out),
        storematch(Progname,_,Ref,_,_),
        gen_vec(Ref,Refvec),
        append(Refvec,_,Out),
        set_asgns(Progname,Out,Ref,Refvec,Asgns),
        !.

% ____________________________________________________
% gen_impl_asgns checks the In vector: for each ground item in In,
% an implicit assignment is created. These
% assignments represent the effects of the mixed computation.

gen_impl_asgns(Progname,Vec,In,Asgns) :-
        !,
        storematch(Progname,Vars,_,_,_),
        set_asgns(Progname,In,Vars,In,Asgns).
```

```
% ____________________________________________________
% set_asgns creates an implicit assignment for each item in Vec having
% a ground value.

set_asgns(_,_,_,[],[]) :- !.

set_asgns(Progname,Vec,[Var|R],[S|T],A) :-
        nonvar(S),
        !,
        set_asgns(Progname,Vec,R,T,Q),
        convert_bruce_expr(Progname,Vec,S,S2),
        chain_append([Var,':=',S2],Q,A).

set_asgns(Progname,Vec,[_|R],[_|T],A) :-
        !,
        set_asgns(Progname,Vec,R,T,A).

% ***
% *** Prolog factoring utilities
% ***

% ____________________________________________________
% factor(Clauses,Prolog,[NewClause],NewProlog) tries to factor out common
% goals in Clauses. Goals may be factored if:
%       (a) the factorable goals are common between two clauses
%       (b) dataflow among them is the same
% factor extracts two different clauses from Clauses, and calls factor_goals to
% factor out the common goals, if any.
% If factor_goals succeeds, then the old clauses are removed from the Prolog
% environment, and the new restructured clause NewClause plus the additional
% definitions are inserted into the environment, resulting in NewProlog.

factor(Clauses,Prolog,[NewClause],NewProlog) :-
        member2(A,Clauses),
        member2(B,Clauses),
        A \== B,
        factor_goals(A,B,NewClause,NewDefns),
        remove_items([A,B],Prolog,X1),
        append([NewClause|NewDefns],X1,NewProlog),
        !.
```

```
% ____________________________________________________
% factor_goals(A,B,NewClause,NewDefns) attempts to factor out common goals in
% clauses A and B. If common goals are found, the disjoint goals are
% removed from the clauses, and are put into a new predicate defined in
% NewDefns. Clauses A and B are merged, resulting in NewClause.
%
% Factor_goals first orders the goals in A and B w.r.t. dataflow.
% Then common goals on the left side of the goal body are found.
% If not successful, common goals on the right side are searched for.
% If common goals can be extracted, spawn_clause is called to do the
% Prolog code maintenance.

factor_goals(A,B,NewClause,NewDefns) :-
        split_clause(A,HeadA,BodyA),
        split_clause(B,HeadB,BodyB),
        order_clause_goals(BodyA,OrdA),
        order_clause_goals(BodyB,OrdB),
        (Side = left,
                duo_split(left,OrdA,CommonA,RestA,OrdB,CommonB,RestB)
                ;
                Side = right,
                duo_split(right,OrdA,RestA,CommonA,OrdB,RestB,CommonB)),
        permutation(CommonA,CommonB),
        spawn_clause(Side,HeadA,CommonA,RestA,HeadB,CommonB,RestB,NewClause,NewDefns).

% ____________________________________________________
% Spawn_clause creates new Prolog code, given common goal sets CommonA and
% CommonB. The two cases are when common goals are found on the left and
% right sides of the clauses. The Prolog code creation involves
% merging the two clauses into one containing the common goals plus call to
% the new predicate, and creating a new predicate containing the disjoint
% goals. spawn_left_main, spawn_left_aux, etc., do the actual prolog
% transformation, dividing the Prolog clause on its left or right side.

spawn_clause(left,HeadA,CommonA,RestA,HeadB,CommonB,RestB,New1,[New2,New3]) :-
        spawn_left_main(HeadA,CommonA,RestA,Newlabel,New1),
        spawn_left_aux(Newlabel,HeadA,CommonA,RestA,HeadB,CommonB,RestB,New2,New3),
        !.

spawn_clause(right,HeadA,CommonA,RestA,HeadB,CommonB,RestB,New1,[New2,New3]) :-
        spawn_right_main(HeadA,CommonA,RestA,Newlabel,New1),
        spawn_right_aux(Newlabel,HeadA,CommonA,RestA,HeadB,CommonB,RestB,New2,New3),
```

```
        !.

spawn_left_main(Head,Common,Rest,Newlabel,(Head3 :- Body)) :-
        copy((Head,Common,Rest),(Head2,Common2,Rest2)),
        get_prev_IO(Common2,Head2,I2,O2),
        gen_label(label,Newlabel),
        gen_vec(I2,NewOut),
        HeadA =.. [Name,In,_],
        Call =..  [Newlabel,O2,NewOut],
        Head3 =.. [Name,In,NewOut],
        list_to_body(CommonA,X1),
        fuse((X1,Call),Body),
        !.

spawn_left_aux(Newlabel,HeadA,CommonA,RestA,HeadB,CommonB,RestB,(Head2 :- Body2),(Head3
        copy((HeadA,CommonA,RestA,HeadB,CommonB,RestB),
             (HeadA2,CommonA2,RestA2,HeadB2,CommonB2,RestB2)),
        get_prev_IO(CommonA2,HeadA2,_,OA),
        get_prev_IO(CommonB2,HeadB2,_,OB),
        get_prev_IO(RestA2,(OA,OA),I2,O2),
        get_prev_IO(RestB2,(OB,OB),I3,O3),
        Head2 =.. [Newlabel,I2,O2],
        Head3 =.. [Newlabel,I3,O3],
        list_to_body(RestA2,Body2),
        list_to_body(RestB2,Body3),
        !.

spawn_right_main(Head,Common,Rest,Newlabel,(Head2 :- Body)) :-
        copy((Head,Common,Rest),(Head2,Common2,Rest2)),
        get_prev_IO(Rest2,Head2,I2,O2),
        get_prev_IO(Common2,(O2,O2),I3,O3),
        gen_label(label,Newlabel),
        gen_vec(I2,NewOut),
        Head2 =.. [Name,In,Out],
        Call =.. [Newlabel,I2,I3],
        list_to_body(Common2,X1),
        fuse((Call,X1),Body),
        !.

spawn_right_aux(Newlabel,HeadA,CommonA,RestA,HeadB,CommonB,RestB,(Head2 :- Body2),(Head
        copy((HeadA,CommonA,RestA,HeadB,CommonB,RestB),
             (HeadA2,CommonA2,RestA2,HeadB2,CommonB2,RestB2)),
```

```
get_prev_IO(RestA2,HeadA2,IA,OA),
get_prev_IO(RestB2,HeadB2,IB,OB),
Head2 =.. [Newlabel,IA,OA],
Head3 =.. [Newlabel,IB,OB],
list_to_body(RestA2,Body2),
list_to_body(RestB2,Body3),
!.
```

Appendix E

Program Printing Utilities

```
% ***
% *** Bruce program pretty printer
% ***
% ***   - Brian J. Ross
% ***     Dept. of Computer Science
% ***     University of British Columbia
% ***

print_bruce(File,B) :-
        telling(Old),
        tell(File),
        print_bruce(B),
        told,
        telling(Old),
        !.

print_bruce([]) :- !.
print_bruce([A|B]) :-
        nl,
        print_bruce2(A,0),
        nl,
        print_bruce(B),
        !.

print_bruce2([],_).
print_bruce2(['\t'|R],Tab) :-
        Nexttab is Tab + 3,
```

```
        writel(['{',nl,tab(Nexttab)]),
        print_bruce2(R,Nexttab).

print_bruce2(['}',';'|R],Tab) :-
        Nexttab is Tab - 3,
        writel([nl,tab(Nexttab),'};',nl,tab(Nexttab)]),
        print_bruce2(R,Nexttab).

print_bruce2(['}','until'|R],Tab) :-
        Nexttab is Tab - 3,
        writel([nl,tab(Nexttab),'} until']),
        print_bruce2(R,Nexttab).

print_bruce2(['}','}'|R],Tab) :-
        Nexttab is Tab - 3,
        Nexttab2 is Nexttab - 3,
        writel([nl,tab(Nexttab),'}',nl,tab(Nexttab2),'}',nl,tab(Nexttab2)]),
        print_bruce2(R,Nexttab2).

print_bruce2(['}'|R],Tab) :-
        Nexttab is Tab - 3,
        writel([nl,tab(Nexttab),'}',nl,tab(Nexttab)]),
        print_bruce2(R,Nexttab).

print_bruce2([S,';'|R],Tab) :-
        writel([S,';',nl,tab(Tab)]),
        print_bruce2(R,Tab).

print_bruce2([local|R],Tab) :-
        writel([nl,local,' ']),
        print_bruce2(R,Tab).

print_bruce2([S|R],Tab) :-
        writel([S,' ']),
        print_bruce2(R,Tab).

% ***
% *** Prolog program pretty printer ("Peter Piper picked...")
% ***

% ____________________________________________________________
% print_prolog/2 prints a beautified version of P into file File.
```

```
% It calls print_prolog/1 to do the actual printing.

print_prolog(File,P) :-
        telling(Old),
        tell(File),
        print_prolog(P),
        told,
        telling(Old),
        !.

print_prolog(P) :-
        copy(P,Q),
        print_prolog2(Q),
        !.

print_prolog2([]).

print_prolog2([A|B]) :-
        print_clause(A),
        nl,
        print_prolog2(B).

% ____________________________________________________
% print_clause(C) beautifies clause C, and then writes it.

print_clause([]).

print_clause(C) :-
        beautify(C),
        C =.. [(:-)|[Head|Body]],
        writel(Head,' :- ',nl),
        print_goals(Body).

print_clause(C) :-
        writel(C,'.',nl).

% ____________________________________________________
% print_goals(G) prints the goals in list G, 1 per line.

print_goals([]).

print_goals([(A,B)|R]) :-
```

```
        print_goals([A,B|R]).

print_goals([G]) :-
        tab(5),
        writel(G,'.',nl).

print_goals([G|R]) :-
        tab(5),
        writel(G,',',nl),
        print_goals(R).

% ____________________________________________________
% beautify(C) sets all the logical variables in clause C to atoms which
% represent their Bruce names, which were saved during parsing.
% The name numbering is reset each time beautify is invoked.

beautify((H:-B)) :-
        H =.. [Name|Args],
        find_names(Name,Vnames),
        reset_name_numbering,
        args_to_list(B,Goals),
        set_names([H|Goals],Vnames),
        !.
beautify(_).

% ____________________________________________________
% set_names(Goallist,Varnames) sets the logical variables in the list of goals
% to their corr. names in Varnames.
% Temporary variables found in 'callasgn' goals are names "tmp".
% Goals which represent subprogram calls are skipped (vars are set elsewhere).
% Otherwise, each list in a goal's argument list, which is an IO vector,
% is processed.

set_names([],_).

set_names([callasgn(_,X,_)|Rest],Vnames) :-
        set_name(X,tmp),
        set_names(Rest,Vnames).

set_names([A|Rest],Vnames) :-
        A =..[Name|_],
        storematch(Name,Vnames2,_,_),
```

```
        Vnames2 \== Vnames,
        set_names(Rest,Vnames).

set_names([A|Rest],Vnames) :-
        A =.. [_|Args],
        save_lists(Args,Vecs),
        set_vec_names(Vecs,Vnames),
        set_names(Rest,Vnames).

set_names([_|Rest],Vnames) :-
        set_names(Rest,Vnames).

% ____________________________________________________________
% save_lists(L,Vecs) saves each list argument in list L into Vecs.

save_lists([],[]).

save_lists([A|B],Vecs) :-
        var(A),
        save_lists(B,Vecs).

save_lists([A|B],[A|Vecs]) :-
        functor(A,'.',_),
        save_lists(B,Vecs).

save_lists([_|B],Vecs) :-
        save_lists(B,Vecs).

% ____________________________________________________________
% find_names(Name,Varnames) finds out what type of mechanism Name represents
% and returns the appropriate variable name list.

find_names(Name,Vnames) :-
        storematch(Name,Vnames,_,_,_).

find_names(Name,Vnames) :-
        labellist(N,L,_),
        member(Name,L),
        storematch(N,Vnames,_,_,_).

% ____________________________________________________________
% set_vec_names(Varlist,Namelist) sets the logical variables in each vector in
```

```
% Veclist to atoms. set_vec_names2(Vec,Namelist) sets each logical variable in
% Vec to an atom created from the corr. Bruce variable Namelist.

set_vec_names([],_).

set_vec_names([V|Rest],Vnames) :-
        set_vec_names2(V,Vnames),
        set_vec_names(Rest,Vnames).

set_vec_names2([],_).

set_vec_names2([V|R],[N|S]) :-
        set_name(V,N),
        set_vec_names2(R,S).

% ____________________________________________________
% set_name(V,Name) unifies logical variable V to an atom representing the
% Prolog variable name. The  atom is based on the Bruce variable name Name.
% A counter is kept for each variable name, which is updated for each
% successful unification.

set_name(V,Name) :-
        not var(V),
        !.

set_name(V,Name) :-
        retract(save_no(Name,Num)),
        Next is Num + 1,
        atom_append(Name,Next,New),
        capitolize(New,Newname),
        V = Newname,
        tassert(save_no(Name,Next)),
        !.

set_name(V,Name) :-
        atom_append(Name,1,New),
        capitolize(New,Newname),
        V = Newname,
        tassert(save_no(Name,1)).

% ____________________________________________________
% Reset the variable counters...
```

```
reset_name_numbering :-
        abolish(save_no,2).
reset_name_numbering.
```

Appendix F

Miscellaneous Code

```
% ***
% *** Main driver
% ***
% ***    - Brian J. Ross
% ***      Dept. of Computer Science
% ***      University of British Columbia
% ***

mixbruce(File,Call) :-
        translate(File,Prolog),
        mixcall(File,Call),
        !.

mixcall(File,Call) :-
        writel('* Mixing program ... *',nl),
        mix(Call,Mixed),
        atom_append(File,'.pmix',PFile),
        writel('* Mixing complete (see "',PFile,'") *',nl),
        print_prolog(PFile,Mixed),
        %
        brucify(Mixed,Bruce),
        atom_append(File,'.bmix',BFile),
        writel('* Bruce generated (see "',BFile,'") *',nl),
        print_bruce(BFile,Bruce),
        writel(nl,'* DONE. *',nl),
        !.
```

```
mixprolog(File,Call) :-
        writel('* Reading Prolog in "',File,'" *',nl),
        [File],
        %
        mixcall(Call),
        !.

parse(File) :-
        translate(File,Prolog),
        writel(nl,'* DONE. *',nl),
        !.

translate(File,Prolog) :-
        start_up,
        alt_read_file(File,Data),
        writel('* "',File,'" read *',nl),
        %
        lexical(Lex,Data),
        writel('* Lexical succeeds *',nl),
        %
        env(Tree,Lex,[]),
        writel('* Parse succeeds *',nl),
        % pretty(Tree),
        %
        gen_prolog(File,Tree,Prolog),
        atom_append(File,'.pro',PFile),
        writel('* Prolog generated (see "',PFile,'") *',nl),
        print_prolog(PFile,Prolog),
        assert_prolog(Prolog),
        %
        brucify(Prolog,B),
        atom_append(File,'.bru',BFile),
        writel('* Bruce generated (see "',BFile,'") *',nl),
        print_bruce(BFile,B),
        !.


% lexical(Lex,Data) converts the characters Data into lexical tokens
% in Lex.

lexical(Lex,Data) :-
        lex(Lex,_,Data,[]).
```

```
% some cleaning up...

start_up :-
        tabolish,
        reset_gen_nums,
        !.

gen_prolog(File,Tree,Prolog) :-
        Tree^^code(Code),
        flatten(Code,Pro1),
        writel('* Basic semantics created *',nl),
        atom_append(File,'.bas',BFile),
        writel('* ... (see "',BFile,'") *',nl),
        print_prolog(BFile,Pro1),
        (found_a_branch ->
                gen_labelprogs(Pro1,Pro2),
                writel('* Label programs created *',nl),
                atom_append(File,'.lab',LFile),
                writel('* ... (see "',LFile,'") *',nl),
                print_prolog(LFile,Pro2),
                %
                process_branches(Pro2,Prolog),
                writel('* Branches processed *',nl)
                ;
                Prolog = Pro1),
        !.

% ***
% *** Prolog dataflow checking utilities
% ***

% ____________________________________________________
% order(Goals,A,B) is satisfied if B is dependent on A wrt data flow
% One of five tests is performed to determine dataflow ordering:
%       (a) B's input matches A's output - succeeds.
%       (b) A's input matches B's output - failure.
%       (c) variables in test B are a subset of mechanism A's output - succeed.
%       (d) like (c), but A and B reversed
%       (e) No direct correlation with A and B, so the list of remaining goals
%               is used for a recursive dataflow check.
```

```
order(Goals,A,B) :-
        vars_used(A,VAI,VAO),
        vars_used(B,VBI,VBO),
        VAO == VBI,
        !.
order(Goals,A,B) :-
        vars_used(A,VAI,VAO),
        vars_used(B,VBI,VBO),
        VAI == VBO,
        !,
        fail.
order(Goals,A,B) :-
        is_a_test(B),
        extract_vars(B,VB),
        vars_used(A,VAI,VAO),
        subset(VB,VAO),
        !.
order(Goals,A,B) :-
        is_a_test(A),
        extract_vars(A,VA),
        vars_used(B,VBI,VBO),
        subset(VA,VBO),
        !,
        fail.
order(Goals,A,B) :-
        remove_items([A,B],Goals,G2),
        member2(C,G2),
        remove_item(C,G2,G3),
        order(G3,A,C),
        order(G3,C,B),
        !.

% ------------------------------------------------------------
% dep_in_vars(A,B) succeeds if the (input) variables in A are dependent
%       or a subset of the input variables in B.

dep_in_vars(Test,B) :-
        is_a_test(Test),
        extract_vars(Test,TV),
        vars_used(B,In,_),
        subset(TV,In),
        !.
```

```
dep_in_vars(A,B) :-
        vars_used(A,InA,_),
        vars_used(B,InB,_),
        subset(InA,InB),
        !.

% ____________________________________________________________
% dep_out_vars(A,B) succeeds if the (input) variables in A are dependent
%       or a subset of the output variables in B.

dep_out_vars(Test,B) :-
        is_a_test(Test),
        extract_vars(Test,TV),
        vars_used(B,_,Out),
        subset(TV,Out),
        !.
dep_in_vars(A,B) :-
        vars_used(A,InA,_),
        vars_used(B,_,OutB),
        subset(InA,OutB),
        !.

% ____________________________________________________________
% same_out_vars(A,B) succeeds if A and B have the same output variables.

same_out_vars(A,B) :-
        vars_used(A,_,OutA),
        vars_used(B,_,OutB),
        OutA == OutB,
        !.

% ____________________________________________________________
% order_clause_goals(L,S) orders goals in L, resulting in S.

order_clause_goals(L,S) :-
        insort(L,S).

% ____________________________________________________________
% from Clocksin & Mellish, pp.146 ...

insort([],[]) :- !.
insort([X|L],M) :-
```

```
        insort(L,N),
        insortx(X,N,M),
        !.

insortx(X,[A|L],[A|M]) :-
        order([A|L],A,X),
        !,
        insortx(X,L,M).
insortx(X,L,[X|L]).

% ***
% *** some variable vector utilities
% ***

% ____________________________________________________________
% gen_vec(A,B) creates a list of 'fresh' variables of the same size as list A.

gen_vec([],[]).
gen_vec([Name|A],[New|B]) :-
        gen_vec(A,B).

% ____________________________________________________________
% var_match(Name,Input,Var) matches variable Name with the variable names
% in storematch for the current program. The corresponding logical variable
% in Input is returned in Var.

var_match(Name,Input,Var) :-
        curr_prog(Prog),
        storematch(Prog,Vars,_,_,_),
        append(A,[Name|B],Vars),
        count(A,S),
        T is S + 1,
        nmember(T,Input,Var),
        !.

% ____________________________________________________________
% A variant of Initial is created, in which Var_i is replaced by Var_f.

vec_variant(Initial,Var_i,Var_f,Final) :-
        break(Var_i,A,[B|C],Initial),
        append(A,[Var_f|C],Final),
        !.
```

```
% ____________________________________________________
% match_vars(A,B,C) matches variable names in A with their logical
% variable values, as are listed in B, resulting in C.

match_vars([],_,[]).
match_vars([A|L],Input,[V|R]) :-
        var_match(A,Input,V),
        match_vars(L,Input,R).

% ____________________________________________________
% set_outputs(Vars,In,Out,Ans) matches each variable in Vars
% with its corresponding output variable, as determined by In and Out.

set_outputs([],_,_,[]).
set_outputs([Var|A],In,Out,[Outvar|D]) :-
        set_output(Var,In,Out,Outvar),
        set_outputs(A,In,Out,D).

set_output(Var,[],_,Var).
set_output(Var,[I|A],[O|B],Ans) :-
        (Var == I -> Ans = O;
                set_output(Var,A,B,Ans)).

% ***
% ***  List utilities
% ***

% ____________________________________________________
% program_separate(Env,Labels,Collect,Prolog) separates predicates into
% the sections used by the original Bruce source programs.
% The Labels list these predicate names.

program_separate(Env,[],Prolog,Prolog).
program_separate(Env,[[Prog|Labels]|Rest],Collect,Prolog) :-
        get_pgms(Env,[Prog|Labels],Some),
        (Some == [] ->
                program_separate(Env,Rest,Collect,Prolog)
                ;
                program_separate(Env,Rest,[Prog,Some|Collect],Prolog)).

% ____________________________________________________
```

```
% get_pgms(Env,Labels,Collect,Result) collects all the clauses in Env
% which are listed in Labels list.

get_pgms(Env,[],[]).
get_pgms(Env,[Label|Rest],A) :-
        get_label_refs(Env,Label,[],Refs),
        (Refs == [] ->
                get_pgms(Env,Rest,A)
                ;
                get_pgms(Env,Rest,B),
                append(Refs,B,A)).


% ____________________________________________________
% get_label_refs(Env,Label,Collect,Result) collects all the clauses in Env
% named by Label.

get_label_refs([],Label,Refs,Refs).
get_label_refs([(Head :- Body)|Rest],Label,Collect,Refs) :-
        !,
        (functor(Head,Label,_) ->
                get_label_refs(Rest,Label,[(Head :- Body)|Collect],Refs)
                ;
                get_label_refs(Rest,Label,Collect,Refs)).
get_label_refs([Clause|Rest],Label,Collect,Refs) :-
        !,
        (functor(Clause,Label,_) ->
                get_label_refs(Rest,Label,[Clause|Collect],Refs)
                ;
                get_label_refs(Rest,Label,Collect,Refs)).

% ***
% *** Miscellaneous Prolog utilities...
% ***

% ____________________________________________________
% member routine in which the Items may be uninstantiated
% (logical variable matching).

member(_,[]) :- !,fail.
member(Item,[X|_]) :- Item == X.
member(Ans,[_|Rest]) :-
```

```
        member(Ans,Rest).

% member2 doesn't handle logical variable objects.

member2(Item,[Item|_]).
member2(Ans,[_|Rest]) :-
        member2(Ans,Rest).

% member3 tests if the ground parts of terms are equivalent.
% (corresponding variables may be different)

member3(Goal,[]) :- !,fail.
member3(Goal,[Item|_]) :-
        ground_eq(Goal,Item).
member3(Goal,[_|R]) :-
        member3(Goal,R).

% ____________________________________________________
% ground_eq(A,B) tests if the ground portions of A and B are equivalent.
% In other words, terms A and B must be equal, except that variables may be
% renamed.

ground_eq(A,B) :-
        var(A),
        var(B),
        !.

ground_eq(A,B) :-
        (var(A) ; var(B)),
        !,
        fail.

ground_eq(A,B) :-
        A == B,
        !.

ground_eq([A1|A2],[B1|B2]) :-
        !,
        ground_eq(A1,B1),
        ground_eq(A2,B2).

ground_eq(A,B) :-
```

```
        !,
        A =.. [Name|ArgsA],
        B =.. [Name|ArgsB],
        ground_eq(ArgsA,ArgsB).

% ____________________________________________________________
% atom_append appends atom names together.

atom_append(Atom1,Atom2,Newatom) :-
        name(Atom1,A),
        name(Atom2,B),
        append(A,B,C),
        name(Newatom,C),
        !.

% ____________________________________________________________
% Count the number of elements in a list

count([],0) :- !.
count([_|L],N) :-
        count(L,M),
        N is M + 1,
        !.

% ____________________________________________________________
% nmember(N,L,I) retrieves the Nth element I of list L

nmember(Num,List,Item) :-
        nmember(1,Num,List,Item),
        !.
nmember(N,N,[Item|_],Item).
nmember(Curr,Num,[_|Rest],Ans) :-
        Next is Curr + 1,
        nmember(Next,Num,Rest,Ans).

% ____________________________________________________________
% break(Item,Front,End,List) breaks List up into Front and End, such
% that Item is the first item in End.

break(Item,[],[A|B],[A|B]) :-
        Item == A.
break(Item,[B|R],E,[B|C]) :-
```

```
        Item \== B,
        break(Item,R,E,C).

% ____________________________________________________________
% args_to_list(A,B) converts the argument structure A to a list B.

args_to_list(A,[A]) :-
        var(A),
        !.
args_to_list(A,T1) :-
        A =.. [',',X,A1],
        args_to_list(X,X1),
        args_to_list(A1,T),
        append(X1,T,T1),
        !.
args_to_list(A,[A]) :- !.

% ____________________________________________________________
% fuse a multi-layered goal list into one flat goal list.

fuse((A,true),B) :-
        fuse(A,B),
        !.
fuse((true,A),B) :-
        fuse(A,B),
        !.
fuse((',',','),',') :- !.
fuse((A,','),E) :-
        fuse(A,E),
        !.
fuse((',',A),E) :-
        fuse(A,E),
        !.
fuse(((',',B),C),E) :-
        fuse((B,C),E),
        !.
fuse(((A,B),C),(D,E)) :-
        fuse(A,D),
        fuse((B,C),E),
        !.
fuse((A,B),(D,E)) :-
        fuse(A,D),
```

```
        fuse(B,E),
        !.
fuse(E,E).

% ____________________________________________________
% get the input and output vectors for an ORDERED list of goals...

get_IO((H:-B),In,Out) :-
        !,
        vars_used(H,In,Out).

get_IO(Goallist,In,Out) :-
        member2(First,Goallist),
        vars_used(First,In,_),
        rev(Goallist,Rev),
        member2(Last,Rev),
        vars_used(Last,_,Out),
        !.

% ____________________________________________________
% get_prev_IO(Goals,Prev,In,Out) determines the In and Out vectors
% for the list of Goals. Should Goals be composed of calls which do not
% have intrinsic IO vectors, the IO vectors of Prev are returned instead.

get_prev_IO(Goals,_,In,Out) :-
        get_IO(Goals,In,Out),
        !.
get_prev_IO(_,Prev,In,Out) :-
        get_IO([Prev],In,Out),
        !.

% ____________________________________________________
% get a term's input and output variable vectors ...

vars_used(A,_,_) :-
        is_a_test(A),
        !,
        fail.
vars_used(A,In,Out) :-
        A =.. [_,In,Out],
        !.
vars_used(asgn(In,_,_,Out),In,Out) :-
```

```
        !.
vars_used(goto(In,_,Out),In,Out) :-
        !.

% ____________________________________________________
% extract_vars(V,A) removes all the variables used in V into list A,
% without duplicates.

extract_vars(V,A) :-
        extract_vars(V,[],A),
        !.

extract_vars(V,Collect,A) :-
        var(V),
        (member(V,Collect) ->
                A = Collect
                ;
                A = [V|Collect]).
extract_vars(X,A,A) :-
        (X == [] ; atom(X)).
extract_vars([V|R],Collect,A) :-
        extract_vars(V,Collect,X),
        extract_vars(R,X,A).
extract_vars(X,Collect,A) :-
        X =.. [_|L],
        extract_vars(L,Collect,A).

% ____________________________________________________
% test if a term is an expression (namely, a boolean test) ...

is_a_test(Thing) :-
        functor(Thing,Name,_),
        name(Name,String),
        (op_chars(String,bool(_));op_chars(String,rel(_))),
        !.
is_a_test(Thing) :-
        member(Thing,[true,false]),
        !.

% ____________________________________________________
% test if two expressions are complementary ...
```

```
complementary(A,B) :- not(not(A =.. ['~',B])).
complementary(A,B) :- not(not(B =.. ['~',A])).

% ____________________________________________________
% remove_item(Item,List,Newlist) removes the first instance of Item in List,
% resulting in Newlist.

remove_item(_,[],[]) :- !.
remove_item(Item,[Item|X],X) :-
        !.
remove_item(Item,[X|Y],[X|Ans2]) :-
        remove_item(Item,Y,Ans2),
        !.

% ____________________________________________________
% remove_item2(Item,List,Newlist) removes the first instance of Item in List,
% resulting in Newlist. It differs from remove_item in that no unification
% is done when matching items.

remove_item2(_,[],[]) :- !.
remove_item2(ItemA,[ItemB|X],X) :-
        ItemA == ItemB,
        !.
remove_item2(Item,[X|Y],[X|Ans2]) :-
        remove_item2(Item,Y,Ans2),
        !.

% ____________________________________________________
% split the clause into its Head, and a list of the goals in the Body.

split_clause((Head :- Body),Head,Bodylist) :-
        args_to_list(Body,Bodylist),
        !.

% ____________________________________________________
% Copy structure A into B, so that it has unique variables

copy(A,B) :-
        assert(copied(A)),
        retract(copied(B)),
        !.
```

```
% ____________________________________________________________
% flatten a multi-layered list into a one-dimension list.

flatten([],[]) :- !.
flatten([[]|A],B) :-
        flatten(A,B),
        !.
flatten([[A|B]|C],D) :-
        flatten([A|[B|C]],D),
        !.
flatten([A|B],[A|C]) :-
        A \== [],
        A \== [_|_],
        flatten(B,C),
        !.

% ____________________________________________________________
% permutation(A,B) checks if the items in list A are a variation of those
% in list B, in terms of ordering.

permutation([],[]).
permutation([A|B],C) :-
        C \== [],
        remove_item(A,C,D),
        permutation(B,D),
        !.

% ____________________________________________________________
% append2 is a deterministic append.

append2([],L,L) :- !.
append2([X|R],L,[X|R1]) :- append2(R,L,R1).

% ____________________________________________________________
% append_list(L,R) appends the lists found in list L together, resulting
% in R.

append_list([],[]) :- !.
append_list([A],A) :- !.
append_list([A,[]],A) :- !.
append_list([A,B|C],D) :-
        append(A,B,E),
```

```
        append_list([E|C],D),
        !.

% ____________________________________________________
% subset(A,B) succeeds if the elements in A are all in B.

subset([],_).
subset([X|R],U) :-
        member(X,U),
        subset(R,U),
        !.

% ____________________________________________________
% remove_items(A,B,C) removes the items in list A from list B, resulting in C.

remove_items([],A,A).
remove_items([A|B],C,D) :-
        remove_item(A,C,E),
        remove_items(B,E,D),
        !.

% ____________________________________________________
% remove_items2(A,B,C) removes the items in list A from list B, resulting in C.
% Unlike remove_items, no unification is done when item matching.

remove_items2([],A,A).
remove_items2([A|B],C,D) :-
        remove_item2(A,C,E),
        remove_items2(B,E,D),
        !.

% ____________________________________________________
% write_list(Direction,Write,List) writes the List in the desired
% direction(vert,horz), using the goven write routine Write.

write_list(_,_,[]) :- !.
write_list(vert,WriteRtn,[A|B]) :-
        Write =.. [WriteRtn,A],
        Write,nl,
        write_list(vert,WriteRtn,B),
        !.
write_list(horz,WriteRtn,[A]) :-
```

```
        !,
        Write =.. [WriteRtn,A],
        Write.
write_list(horz,WriteRtn,[A|B]) :-
        Write =.. [WriteRtn,A],
        Write,
        write(','),
        write_list(horz,WriteRtn,B).

% ____________________________________________________
% convert a list of goals to a body of goals

list_to_body([],'') :- !.
list_to_body([A],A) :- !.
list_to_body([A|C],(A,D)) :-
        list_to_body(C,D),
        !.


% ____________________________________________________
% duo_split(Side,L1,Left1,Right1,L2,Left2,Right2) splits L1 and L2 into
% equal size, non-null parts, depending if Side is 'left' or 'right'.

duo_split(right,L1,Left1,Right1,L2,Left2,Right2) :-
        append(Left1,Right1,L1),
        Left1 \== [],
        Right1 \== [],
        count(Right1,N),
        append(Left2,Right2,L2),
        count(Right2,N).

duo_split(left,L1,Left1,Right1,L2,Left2,Right2) :-
        rev(L1,RL1),
        rev(L2,RL2),
        append(RLeft1,RRight1,RL1),
        RLeft1 \== [],
        RRight1 \== [],
        count(RRight1,N),
        append(RLeft2,RRight2,RL2),
        count(RRight2,N),
        rev(RLeft1,Right1),
        rev(RLeft2,Right2),
```

```
        rev(RRight1,Left1),
        rev(RRight2,Left2).

% ____________________________________________________________
% list reverser...

rev(X,RX) :-
        revx(X,[],RX),!.

revx([],R,R) :-
        !.
revx([X|Y],Z,R) :-
        revx(Y,[X|Z],R).

% ____________________________________________________________
% find_clause(Prolog,Head,Clause) finds a clause in Prolog with head Head.
% Repeated calls return new results.

find_clause(Prolog,Head,Clause) :-
        member2(Clause,Prolog),
        split_clause(Clause,Head2,_),
        not not Head = Head2.


% ____________________________________________________________
% find_goal(Clause,Head,Left,Goal,Right) finds Goal within the body of
% Clause. Left and Right are the goals on each side of the goal.

find_goal(Clause,Head,Left,Goal,Right) :-
        split_clause(Clause,Head,Goals),
        append(Left,[Goal|Right],Goals),
        !.

% ____________________________________________________________
% make_clause(Head,Left,Right,Clause) appends lists Left and Right together,
% converts them into a body structure, and forms a clause with head Head.
% Note: There must already exists proper input-output between the constructs

make_clause(Head,[],[],Head).

make_clause(Head,Left,[],(Head :- Body)) :-
        list_to_body(Left,Body).
```

```
make_clause(Head,[],Right,(Head :- Body)) :-
        list_to_body(Right,Body).

make_clause(Head,Left,Right,(Head :- Body)) :-
        append(Left,Right,Goals),
        list_to_body(Goals,Body).

% ____________________________________________________
% merge_goals(IA,OA,A,IB,OB,B,In,Out,G) appends goal lists A and B into G,
% while assuring that the IO of the two lists is merged. In and Out contain
% the IO of the result.

merge_goals(_,_,[],_,_,[],_,_,[]) :- !.

merge_goals(_,_,[],IA,OA,A,In,Out,G) :-
        copy((IA,OA,A),(In,Out,G)),
        !.

merge_goals(IA,OA,A,_,_,[],In,Out,G) :-
        copy((IA,OA,A),(In,Out,G)),
        !.

merge_goals(IA,OA,A,IB,OB,B,In,Out,G) :-
        copy((IA,OA,A,IB,OB,B),(In,OA2,A2,IB2,Out,B2)),
        OA2 = IB2,
        append(A2,B2,G),
        !.

% ____________________________________________________
% append_goals(Clause,Vec,Code,Newclause) adds the goals in Code to the goals
% of Clause, resulting in Newclause.

append_goals(Clause,Vec,Code,(Newhead :- Body)) :-
        copy((Clause,Vec,Code),(Clause2,Vec2,Code2)),
        split_clause(Clause2,Head,Goals),
        Head =.. [Name,I1,O1],
        get_prev_IO(Goals,(I1,I1),I2,Vec2),
        append(Goals,Code2,G),
        get_IO(G,I3,O3),
        Newhead =.. [Name,I1,O3],
        list_to_body(G,Body),
```

```
        !.

% ____________________________________________________________
% parent_clause(Prolog,Head,Parent,Phead,Left,Goal,Right):
%       Prolog  – current prolog environment
%       Head    – clause to find a parent for
%       Parent  – the found parent clause
%       Phead, Left, Goal, Right – the breaking up of Parent such that
%               Phead is the head, Goal is the call to Head, and Left/Right
%               are goals to the left and right of Goal
%
% A parent clause is found. Recursive calls are not used -- a clause cannot
% be its own parent.

parent_clause(Prolog,Head,Parent,Phead,Left,Goal,Right) :-
        copy(Head,Goal),
        member2(Parent,Prolog),
        split_clause(Parent,Phead,Goals),
        not Phead = Head,
        append(Left,[Goal|Right],Goals).

% sibling_clause(Prolog,Clause,Sibling) finds a sibling for Clause from Prolog.

sibling_clause(Prolog,Clause,Sibling) :-
        split_clause(Clause,Head,_),
        copy(Head,Item),
        member2(Sibling,Prolog),
        split_clause(Sibling,Item,_),
        Sibling \== Clause.

% ____________________________________________________________
% assert_prolog(P) tasserts all the code in list P.

assert_prolog([]) :- !.
assert_prolog([Clause|Rest]) :-
        tassertz(Clause),
        assert_prolog(Rest),
        !.

% ____________________________________________________________
% Make an atom like Name into one which is capitolized.
```

```
capitolize(Name,Caps) :-
        name(Name,[A|L]),
        A >= 97,
        A =< 122,
        A2 is A - 32,
        name(Caps,[A2|L]),
        !.

capitolize(Name,Name).


%_____________________________________________________
% Read: (written by Doug Westcott)
%
%       The following routines allow the reading of a file's contents into
%       prolog in the form of a list of characters' ASCII numbers (the usual
%       prolog representation of a string).
%

alt_read_file(File,Contents) :-
        read_file(File,Contents1),
        reduce_white_space(Contents1,Contents),
        !.

read_file(File,Contents) :-
        seeing(OldFile),
        see(File),
        get0(Char),
        read_file1(Char,[],RContents),
        seen,
        seeing(OldFile),
        reverse(RContents,Contents),
        !.

read_file1(Char,Sofar,Sofar) :-
        end_of_file(Char),
        !.
read_file1(Char,Sofar,Contents) :-
        get0(Newchar),
        read_file1(Newchar,[Char|Sofar],Contents),
        !.
```

```
end_of_file(-1) :-
        !.

reduce_white_space(Old,New) :-
        change_to_spaces(Old,Mid),
        reduce_spaces(Mid,New),
        !.

change_to_spaces([Char|Chars],[NewChar|NewChars]) :-
        change_to_space(Char,NewChar),
        !,
        change_to_spaces(Chars,NewChars),
        !.
change_to_spaces([],[]) :-
        !.

change_to_space(09,32).
change_to_space(10,32).
change_to_space(11,32).
change_to_space(12,32).
change_to_space(X,X).

reduce_spaces([32,32|Chars],NewChars) :-
        reduce_spaces([32|Chars],NewChars),
        !.
reduce_spaces([Char|Chars],[Char|NewChars]) :-
        reduce_spaces(Chars,NewChars),
        !.
reduce_spaces([],[]) :-
        !.

% ***
% ***  Tagged assertion utilities
% ***

% ____________________________________________________________
% tassert routines assert clauses, plus update a tag to be used for
% later removal.

tassert(Clause) :-
        set_rem_clause(Clause),
        assert(Clause),
```

```
        !.
tasserta(Clause) :-
        set_rem_clause(Clause),
        asserta(Clause),
        !.
tassertz(Clause) :-
        set_rem_clause(Clause),
        assertz(Clause),
        !.

% ____________________________________________________________
% set rem_clause tag. If predicate exists already before a tag has been set,
% then do not set a remove tag, as this predicate is used elsewhere.

set_rem_clause((Head :- Goals)) :-
        functor(Head,Name,Arity),
        set_rem_clause2(Name,Arity),
        !.
set_rem_clause(Clause) :-
        functor(Clause,Name,Arity),
        set_rem_clause2(Name,Arity),
        !.

set_rem_clause2(Name,Arity) :-
        rem_clause(Name,Arity),
        !.
set_rem_clause2(Name,Arity) :-
        current_predicate(Name,Head),
        functor(Head,Name,Arity),
        !.
set_rem_clause2(Name,Arity) :-
        !,
        assertz(rem_clause(Name,Arity)).

% ____________________________________________________________
% remove a predicate asserted with tassert  utility

tabolish(Pred,Arity) :-
        rem_clause(Pred,Arity),
        abolish(Pred,Arity),
        retract(rem_clause(Pred,Arity)).
tabolish(_,_).
```

```
% ____________________________________________________
% remove all clauses asserted by tassert utilities

tabolish :-
        rem_clause(Name,Arity),
        abolish(Name,Arity),
        fail.
tabolish :-
        abolish(rem_clause,2).


% ____________________________________________________
% list writing routine (written by Doug Westcott)

writel([A|B]) :-
        write_item(A),
        writel(B),
        !.
writel([]) :-
        !.

writel(A) :-
        writel([A]),
        !.
writel(A,B) :-
        writel([A,B]),
        !.
writel(A,B,C) :-
        writel([A,B,C]),
        !.
writel(A,B,C,D) :-
        writel([A,B,C,D]),
        !.
writel(A,B,C,D,E) :-
        writel([A,B,C,D,E]),
        !.
writel(A,B,C,D,E,F) :-
        writel([A,B,C,D,E,F]),
        !.
writel(A,B,C,D,E,F,G) :-
        writel([A,B,C,D,E,F,G]),
```

```
        !.
writel(A,B,C,D,E,F,G,H) :-
        writel([A,B,C,D,E,F,G,H]),
        !.
writel(A,B,C,D,E,F,G,H,I) :-
        writel([A,B,C,D,E,F,G,H,I]),
        !.
writel(A,B,C,D,E,F,G,H,I,J) :-
        writel([A,B,C,D,E,F,G,H,I,J]),
        !.

write_item(X) :-
        var(X),
        write(X),
        !.
write_item(nl) :-
        nl,
        !.
write_item(tab(N)) :-
        tab(N),
        !.
write_item(X) :-
        write(X),
        !.
```

Appendix G

Binary GCD Basic Semantics

```
bingcd([Gcd1,U1,V1],[Gcd2,U2,V2]) :-
      asgn([Gcd1,U1,V1,K1,P1,T1],K2,0,[Gcd1,U1,V1,K2,P1,T1]),
      test1([Gcd1,U1,V1,K2,P1,T1],[Gcd2,U2,V2,K3,P2,T2]).

test1([Gcd1,U1,V1,K1,P1,T1],[Gcd2,U2,V2,K2,P2,T2]) :-
      U1//2*2=\=U1,
      b2([Gcd1,U1,V1,K1,P1,T1],[Gcd2,U2,V2,K2,P2,T2]).

test1([Gcd1,U1,V1,K1,P1,T1],[Gcd2,U2,V2,K2,P2,T2]) :-
      ~U1//2*2=\=U1,
      test2([Gcd1,U1,V1,K1,P1,T1],[Gcd2,U2,V2,K2,P2,T2]).

test2([Gcd1,U1,V1,K1,P1,T1],[Gcd2,U2,V2,K2,P2,T2]) :-
      V1//2*2=\=V1,
      b2([Gcd1,U1,V1,K1,P1,T1],[Gcd2,U2,V2,K2,P2,T2]).

test2([Gcd1,U1,V1,K1,P1,T1],[Gcd2,U2,V2,K2,P2,T2]) :-
      ~V1//2*2=\=V1,
      asgn([Gcd1,U1,V1,K1,P1,T1],K3,K1+1,[Gcd1,U1,V1,K3,P1,T1]),
      asgn([Gcd1,U1,V1,K3,P1,T1],U3,U1//2,[Gcd1,U3,V1,K3,P1,T1]),
      asgn([Gcd1,U3,V1,K3,P1,T1],V3,V1//2,[Gcd1,U3,V3,K3,P1,T1]),
      b1([Gcd1,U3,V3,K3,P1,T1],[Gcd2,U2,V2,K2,P2,T2]).

test3([Gcd1,U1,V1,K1,P1,T1],[Gcd2,U2,V2,K2,P2,T2]) :-
      U1//2*2=\=U1,
      asgn([Gcd1,U1,V1,K1,P1,T1],T3,0-V1,[Gcd1,U1,V1,K1,P1,T3]),
      b4([Gcd1,U1,V1,K1,P1,T3],[Gcd2,U2,V2,K2,P2,T2]).
```

```
test3([Gcd1,U1,V1,K1,P1,T1],[Gcd2,U2,V2,K2,P2,T2]) :-
      ~U1//2*2=\=U1,
      asgn([Gcd1,U1,V1,K1,P1,T1],T3,U1,[Gcd1,U1,V1,K1,P1,T3]),
      asgn([Gcd1,U1,V1,K1,P1,T3],T4,T3//2,[Gcd1,U1,V1,K1,P1,T4]),
      test4([Gcd1,U1,V1,K1,P1,T4],[Gcd2,U2,V2,K2,P2,T2]).

test4([Gcd1,U1,V1,K1,P1,T1],[Gcd2,U2,V2,K2,P2,T2]) :-
      T1//2*2=:=T1,
      b3([Gcd1,U1,V1,K1,P1,T1],[Gcd2,U2,V2,K2,P2,T2]).

test4([Gcd1,U1,V1,K1,P1,T1],[Gcd2,U2,V2,K2,P2,T2]) :-
      ~T1//2*2=:=T1,
      test5([Gcd1,U1,V1,K1,P1,T1],[Gcd3,U3,V3,K3,P3,T3]),
      asgn([Gcd3,U3,V3,K3,P3,T3],T4,U3-V3,[Gcd3,U3,V3,K3,P3,T4]),
      test6([Gcd3,U3,V3,K3,P3,T4],[Gcd2,U2,V2,K2,P2,T2]).

test5([Gcd1,U1,V1,K1,P1,T1],[Gcd1,U2,V1,K1,P1,T1]) :-
      T1>0,
      asgn([Gcd1,U1,V1,K1,P1,T1],U2,T1,[Gcd1,U2,V1,K1,P1,T1]).

test5([Gcd1,U1,V1,K1,P1,T1],[Gcd1,U1,V2,K1,P1,T1]) :-
      ~T1>0,
      asgn([Gcd1,U1,V1,K1,P1,T1],V2,0-T1,[Gcd1,U1,V2,K1,P1,T1]).

test6([Gcd1,U1,V1,K1,P1,T1],[Gcd2,U2,V2,K2,P2,T2]) :-
      T1=\=0,
      b3([Gcd1,U1,V1,K1,P1,T1],[Gcd2,U2,V2,K2,P2,T2]).

test6([Gcd1,U1,V1,K1,P1,T1],[Gcd2,U1,V1,K2,P2,T1]) :-
      ~T1=\=0,
      asgn([Gcd1,U1,V1,K1,P1,T1],K2,K1,[Gcd1,U1,V1,K2,P1,T1]),
      callasgn([Gcd1,U1,V1,K2,P1,T1],Tmp1,2),
      callasgn([Gcd1,U1,V1,K2,P1,T1],Tmp2,K2),
      power([P1,Tmp1,Tmp2],[P2,_137447,_137448]),
      asgn([Gcd1,U1,V1,K2,P2,T1],Gcd2,U1*P2,[Gcd2,U1,V1,K2,P2,T1]).

b1([Gcd1,U1,V1,K1,P1,T1],[Gcd2,U2,V2,K2,P2,T2]) :-
      test1([Gcd1,U1,V1,K1,P1,T1],[Gcd2,U2,V2,K2,P2,T2]).

b2([Gcd1,U1,V1,K1,P1,T1],[Gcd2,U2,V2,K2,P2,T2]) :-
      test3([Gcd1,U1,V1,K1,P1,T1],[Gcd2,U2,V2,K2,P2,T2]).
```

```
b3([Gcd1,U1,V1,K1,P1,T1],[Gcd2,U2,V2,K2,P2,T2]) :-
      asgn([Gcd1,U1,V1,K1,P1,T1],T3,T1//2,[Gcd1,U1,V1,K1,P1,T3]),
      test4([Gcd1,U1,V1,K1,P1,T3],[Gcd2,U2,V2,K2,P2,T2]).

b4([Gcd1,U1,V1,K1,P1,T1],[Gcd2,U2,V2,K2,P2,T2]) :-
      test4([Gcd1,U1,V1,K1,P1,T1],[Gcd2,U2,V2,K2,P2,T2]).
```

Appendix H

Partially Evaluated Binary GCD Semantics

```
bingcd([Gcd1,7,V1],[Gcd2,U2,V2]) :-
      asgn([Gcd1,7,V1,0,P1,T1],T3,0-V1,[Gcd1,7,V1,0,P1,T3]),
      b4([Gcd1,7,V1,0,P1,T3],[Gcd2,U2,V2,D7,P2,T2]).

test4([Gcd1,U1,V1,K1,P1,T1],[Gcd2,U2,V2,K2,P2,T2]) :-
      T1//2*2=:=T1,
      b3([Gcd1,U1,V1,K1,P1,T1],[Gcd2,U2,V2,K2,P2,T2]).

test4([Gcd1,U1,V1,K1,P1,T1],[Gcd2,U2,V2,K2,P2,T2]) :-
      ~T1//2*2=:=T1,
      test5([Gcd1,U1,V1,K1,P1,T1],[Gcd3,U3,V3,K4,P4,T3]),
      asgn([Gcd3,U3,V3,K3,P3,T3],T3,U3-V3,[Gcd3,U3,V3,K3,P3,T4]),
      test6([Gcd3,U3,V3,K3,P3,T4],[Gcd2,U2,V2,K2,P2,T2]).

test5([Gcd1,U1,V1,K1,P1,T1],[Gcd1,U2,V1,K1,P1,T1]) :-
      T1>0,
      asgn([Gcd1,U1,V1,K1,P1,T1],U2,T1,[Gcd1,U2,V1,K1,P1,T1]).

test5([Gcd1,U1,V1,K1,P1,T1],[Gcd1,U1,V2,K1,P1,T1]) :-
      ~T1>0,
      asgn([Gcd1,U1,V1,K1,P1,T1],V2,0-T1,[Gcd1,U1,V2,K1,P1,T1]).

test6([Gcd1,U1,V1,K1,P1,T1],[Gcd2,U2,V2,K2,P2,T2]) :-
      T1=\=0,
      b3([Gcd1,U1,V1,K1,P1,T1],[Gcd2,U2,V2,K2,P2,T2]).
```

```
test6([Gcd1,U1,V1,K1,P1,T1],[Gcd2,U1,V1,K2,P2,T1]) :-
      ~T1=\=0,
      asgn([Gcd1,U1,V1,K1,P1,T1],K2,K1,[Gcd1,U1,V1,K2,P1,T1]),
      callasgn([Gcd1,U1,V1,K2,P1,T1],Tmp1,2),
      callasgn([Gcd1,U1,V1,K2,P1,T1],Tmp2,K2),
      power([P1,Tmp1,Tmp2],[P2,Tmp3,Tmp4]),
      asgn([Gcd1,U1,V1,K2,P2,T1],Gcd2,U1*P2,[Gcd2,U1,V1,K2,P2,T1]).

b3([Gcd1,U1,V1,K1,P1,T1],[Gcd2,U2,V2,K2,P2,T2]) :-
      asgn([Gcd1,U1,V1,K1,P1,T1],T3,T1//2,[Gcd1,U1,V1,K1,P1,T3]),
      test4([Gcd1,U1,V1,K1,P1,T3],[Gcd2,U2,V2,K2,P2,T2]).

b4([Gcd1,U1,V1,K1,P1,T1],[Gcd2,U2,V2,K2,P2,T2]) :-
      test4([Gcd1,U1,V1,K1,P1,T1],[Gcd2,U2,V2,K2,P2,T2]).
```