# Analysis and Specialisation of Imperative Programs: An approach using CLP

Julio C. Peralta Estrada



A thesis submitted to the University of Bristol in accordance with the requirements for the degree of Doctor in Philosophy in the Faculty of Engineering,

Department of Computer Science.

#### Abstract

Program analysis and specialisation for declarative languages have been subjects of active research during the last 2 decades or so. Imperative languages could benefit from the advances made in declarative languages but most research on analysis and specialisation is language-specific. Thus, it would be desirable to build connections between both worlds, enabling the transfer of results from one to the other.

The semantics of an imperative programming language can be expressed as a program in a declarative constraint language. Not only does this render the semantics (as potentially) executable, but it opens up the possibility of applying to imperative languages the advances made in program analysis and transformation of declarative languages.

This thesis presents results on analysis and specialisation of imperative programs through analysis and specialisation of constraint logic programs. The cornerstone of this connection is a constraint logic program capturing the structural operational semantics of the subject imperative language whose programs are to be specialised and analysed. To provide a connection between imperative programs and constraint logic programs an existing specialiser for logic programs is extended and modified. The extension consists of adding constraints to increase the accuracy of the specialiser. The modification of the specialiser enables the systematic reconstruction of a large class of imperative programs from the specialised constraint logic programs. A key advantage of this approach to analysis and specialisation of imperative programs is that the same tools can be applied to programs in different languages, via their semantics. Also, the correctness of the resulting analysers and specialisers comes from correctness of specialisation and analysis in CLP. The tools needed are a specialiser for CLP and the imperative language semantics as a CLP program. Semantics is provided for a programming language resembling Pascal, though other languages could be used too (e.g. Java or JVM). Through experimentation the generality of the method is shown.

A mi papá, a mi mamá, a todos mis hermanos, y a Bibi, gracias

#### Acknowledgements

First and foremost I would like to thank my supervisor John P. Gallagher without whose help and advice this thesis would not exist. He provided all the opportunities for the appropriate development of this piece of research. His friendship, patience and support made this thesis worth pursuing.

I am also indebted to the National University of Mexico (UNAM), in particular to DGAPA and IIMAS for providing the grant that made this thesis possible. I would like to thank David Rosenblueth for his support and friendship during these years. He introduced me to logic programming. Special thanks to Hüseyin Sağlam for explaining me his work and the fruitful discussions we had. Laura Lafave's friendship, optimism and cheerful character made the perfect officemate during these years. To Patricia Russeau, Rafael Ramirez, Polly Nethersole, Jorge Buenabad, Julio Gallardo and to all my friends and colleagues for the pleasant memories I have of these years in Bristol. To the examiners Andy King and Steve Gregory whose comments helped in improving the presentation of this work.

To my family for their infinite support and encouragement.

### Declaration

The work in this dissertation was carried out in the Department of Computer Science of the University of Bristol, and has not been submitted for any other degree or diploma of any examining body. Except where acknowledged in the text, all work is the original work of the author.

Julio C. Peralta Estrada July 2000

# Contents

1	Intr	troduction				
	1.1	1 Thesis outline				
	1.2	Contri	ibutions of the Thesis	15		
2	Ana	alysis of CLP				
	2.1	The framework for CLP program analysis				
		2.1.1	CLP programs	19		
		2.1.2	The Method	19		
	2.2	2 The Approximation				
		2.2.1	Regular Unary Logic Programs	23		
		2.2.2	Linear Arithmetic Expressions	29		
		2.2.3	Constrained Regular Unary Logic Programs	30		
	2.3	Query-answer transformations				
	2.4	Summary				
3	$\mathbf{Spe}$	cialisa	tion of CLP	37		
	3.1			37		
	3.2			44		
		3.2.1	On-line specialisation	44		
		3.2.2	Local Control	56		
		3.2.3	Global Control	57		
	3 3	Summ	arv	64		

4	Sen	nantics in CLP	65		
	4.1 Imperative Program Semantics				
		4.1.1 Operational Semantics	66		
	4.2	A Simple Semantics-based Interpreter	72		
		4.2.1 Structural Operational Semantics	73		
		4.2.2 Procedures	77		
		4.2.3 Other Data Types	82		
		4.2.4 Block declarations	86		
	4.3	Summary	87		
5	Rel	ating CLP to Imperative Programs	89		
	5.1	Specialisation using Program Points	90		
		5.1.1 Representation of Program Points	92		
		5.1.2 Control of Specialisation	95		
		5.1.3 Monovariant Imperative Program Specialisation	96		
		5.1.4 Polyvariant Imperative Program Specialisation	99		
	5.2 Program recovery		100		
			108		
	5.3	Summary	116		
6	Applications in Imperative Languages 1				
	6.1	Program Specialisation	117		
	6.2	Program Analysis			
	6.3	Some possible analyses and transformations	147		
		6.3.1 Program Specialisation	147		
		6.3.2 Program Analysis	148		
	6.4	Summary	149		
7	Rel	Related Work			
	7.1	Program Analysis			
	7.2	Program Specialisation	155		

8	Conclusions			
	8.1	CLP Analysis	159	
	8.2	Writing semantics as CLP programs	160	
	8.3	Specialisation of semantics-based interpreters	161	
		8.3.1 Languages with multiple procedures	162	
	8.4	Limitations and Possible directions for Future Work	164	
A	Sen	nantics of three imperative languages	169	
A		nantics of three imperative languages  A small language		
A		2	169	
$\mathbf{A}$	A.1	A small language	169 171	
A	A.1	A small language	169 171 172	
A	A.1 A.2	A small language	169 171 172 174	

# List of Figures

1.1	General framework	14
3.1	Specialisation for compilation	39
3.2	The specialisation algorithm	53
5.1	The monovariant generalisation algorithm	98
5.2	The polyvariant generalisation algorithm	101
5.3	Imperative program recovery	106
5.4	Example callgraph	109
5.5	Example reducing a callgraph	109
6.1	Minimal callgraph of residual program	121
6.2	Minimal callgraph of powers residual program	124

### Chapter 1

### Introduction

In the quest for better programming environments and faster programs new methods are proposed to solve specific tasks in program analysis and specialisation. In most cases, their applicability is restricted to a specific language. Hence, to reuse these methods in a different language some generalisation has to be made otherwise a new method is developed from scratch. In the case that one wants to reuse an existing tool, one is faced with several tasks such as understanding the programming language, i.e. its semantics, as well as understanding the method, in order to produce the desired results (in analysis or specialisation). In addition, the correctness of the method for the new programming language has to be established. In the case that a new tool is developed, the new programming language semantics has to be understood too and everything has to be constructed from the beginning (the method and proofs). Hence, it would be desirable to have a general framework where results in analysis and specialisation could be transferred or exchanged with less effort. In this thesis a general framework is proposed for transferring results in analysis and specialisation of constraint logic programs to imperative programs. The advantage is that existing analysers and specialisers in the logic programming world can be reused with the ensuing benefits associated with declarative languages. The cornerstone of this connection is a constraint logic program capturing the semantics of the subject imperative language whose programs are to be specialised and analysed.

As already noted, language semantics is the starting point of any program analysis

and specialisation. The current wide spectrum of different program semantics makes it difficult to decide which semantics is appropriate for the task at hand [Cou97]. The usual choice is between operational and denotational semantics. We chose structural operational semantics as the departure point for analysis and specialisation of imperative programs [MS96]. Later, through abstract interpretation we could refine our semantics to suit the needs of analysis and specialisation. Abstract compilation [HWD92] of constraint logic programs, combined with other techniques, may be used to achieve the refinement referred to above. To reduce the complexity of analysis and specialisation we use a slightly modified form of structural operational semantics which we call one-state semantics.

One aim of our framework is to provide specialisation of imperative programs by specialisation of constraint logic programs. In order to achieve this aim we must note that structural operational semantics definitions can be relatively easily made to coincide with constraint logic programs. This makes it possible to obtain interpreters of imperative programs by writing down the semantic definitions of an imperative language as a constraint logic program. Next, using an existing on-line specialiser for constraint logic programs and an input imperative program it is possible to derive a specialised equivalent program in CLP. This specialised interpreter may be rendered as the specialisation of the input imperative program. However, specialisation is a source-to-source program transformation, and returning a CLP program as the specialisation of an imperative program is not an example of this. Because specialisation is an aggressive technique for transforming programs in general it may not be possible to recognise the different imperative components of the specialised imperative program. If this method is to be used for specialisation of imperative programs, we need a way of systematically relating the results of this specialisation to the imperative source program. Program points in combination with other properties of residual CLP programs are exploited to recover imperative programs from specialised CLP programs.

The other aim of our framework is to analyse imperative programs by analysing CLP programs. Specialisation with program points may also be used to obtain resid-

ual CLP programs whose analysis may be automatically related to imperative programs. Provided we start from a sufficiently general semantics we could reduce the complexity of analysis by a technique known as abstract compilation. We use a general framework for analysing constraint logic programs based on this technique. In our case, abstract compilation is achieved by defining a preinterpretation of the program and performing a systematic replacement of each function symbol appearing in the program by its preinterpreted form. In addition, some knowledge, in the form of another CLP program, about these new function symbols has to be present in the compiled program. Through this technique only declarative properties of the subject constraint logic program can be derived, though this includes some information typically thought of as procedural. Moreover, abstract compilation defines a semi-morphic transformation [GDL95] on the 'function' computed by the source program. By doing this (abstract compilation), the complexity of the analysis is reduced. Our one-state structural operational semantics also aids the analysis framework used by reducing the complexity of the programs to be analysed.

Briefly, the framework proposed (see Fig. 1.1) takes the semantics of an imperative language described as a CLP program in one-state structural operational form, and an imperative program to be analysed and/or specialised.

It specialises the interpreter with respect to the imperative program (and possibly a partial input) producing a CLP residual program. Later there are two paths which could be followed by this residual program. One option is to analyse it using an existing analyser for CLP programs. The other one is to recover from such a residual program an appropriate imperative program. In both cases the results are presented in the imperative language, either using the original imperative program (for program analysis) or constructing a specialised imperative program (for program specialisation). A key advantage of this approach to analysis and specialisation of imperative programs is that the same tools can be applied to programs in different languages, via their semantics. The tools needed are a partial evaluator for CLP and the imperative language semantics as a CLP program.

Semantics is provided for a programming language resembling Pascal, though

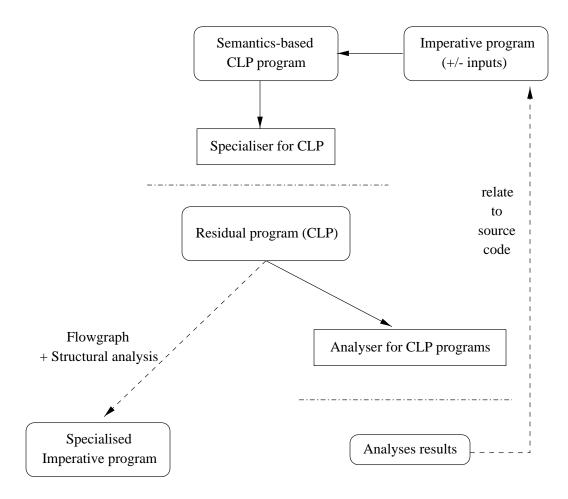


Figure 1.1: General framework

other languages could be used as well (e.g. Java or JVM, see Appendix).

Through experimentation the generality of the method is shown, for analysis (inferring linear relationships between variables of a program) and specialisation (using an on-line polyvariant specialiser for CLP programs).

### 1.1 Thesis outline

Chapter 2 describes the analysis framework used and pinpoints possible uses of analysis domains in CLP that could be useful for imperative programs. Using an existing specialiser for specialisation is not adequate for this application. In Chapter 3 a standard specialisation algorithm is extended with constraints for several domains. The

new specialisation algorithm is presented. Representing the semantics of an imperative program deserves particular attention because poor representations may preclude some opportunities for specialisation and reduce the accuracy of the analyses. In Chapter 4 we discuss the major issues in implementing semantics for programming languages with several features: procedures with parameters, different control structures, and data types. Program points are introduced into the semantics and the imperative programs are used during specialisation to allow the systematic recovery of the program analysis results and/or specialisation. Next in Chapter 5, some properties of the residual programs generated are identified which allow the recovery of imperative programs from CLP programs. Also, the polyvariance of the resulting imperative specialisers is discussed there. Program points induce a monovariant specialisation. Program points combined with trace terms<sup>1</sup> or other dynamic information open the possibility of producing a polyvariant specialisation. In Chapter 6 some examples of analysis and specialisation of imperative languages are given. Semantics definitions are given for a language with procedures and parameters. In Chapter 7 we go through a review of related work in program analysis and specialisation for imperative programs. Finally, in the last chapter some discussion of the contributions is given as well as pointers for possible extensions. In the appendix we show the results of some other experiments.

### 1.2 Contributions of the Thesis

In the following we briefly list the contributions made by this thesis. Some of the contributions of this work have been published as [PGS98, PG99, GP00].

- A new widening operator for constrained regular unary logic programs.
- Some steps towards integrating analysis and specialisation by integrating three analysis domains into an existing specialisation algorithm.

<sup>&</sup>lt;sup>1</sup>Trace terms are the abstraction used by the specialiser of constraint logic programs, and allow polyvariance.

- A style of writing semantics of imperative programs in CLP using a modified form of structural operational semantics. This style of writing the semanticsbased CLP interpreters aids specialisation and analysis as is shown in this work.
- Some techniques to specialise semantics-based CLP interpreters while being able to reconstruct specialised imperative programs.
- The use of the same techniques to analyse imperative programs by analysis of CLP programs.
- A general framework for systematic transfer of results from analysis and specialisation of CLP to analysis and specialisation of imperative programs.

## Chapter 2

# **Analysis of CLP**

The aim of program analysis is to find as much information as possible about the runtime behaviour of a program for all or some possible inputs, without actually running it. It is well known that automatic program analysis is unable to obtain (exact descriptions of) most of the interesting properties associated with programs (e.g. termination, failure, determinism, reachable definitions, etc.). In spite of this hard restriction, program analysis can often be achieved with a great deal of precision using safe approximations of properties instead of the direct approach (exact descriptions). The problem is then how to obtain correct approximations. There are two main approaches to program analysis: one is to obtain the analysis by successive approximations of a semantics (usually derived from a structural operational or denotational semantics [NN97]) of the language; and the other one is by direct implementation of an approximation using a particular semantics for the language. In the first case, a major contribution towards providing a systematic development of program analysers is abstract interpretation [CC77]. In the latter case, individual analysers and ad-hoc semantics are provided for specific applications. The analyser we used [Sag98] is based on abstract interpretation in the sense that the approximation to the semantics has a general purpose (core) semantics. It differs from 'traditional' abstract interpretation in that the application-dependent part of the approximation to the collecting semantics is not a constant of the semantics. Instead, this application-dependent part is a parameter to the core semantics, a preinterpretation. In this chapter we recall such a framework for analysis of CLP programs. Particular attention is given to three approximation domains, namely linear arithmetic expressions, regular programs and a combination of both.

### 2.1 The framework for CLP program analysis

The declarative semantics of a definite logic program is its minimal model. Models are interpretations for which the logic program denotes true statements. Interpretations, in turn, assign meanings to the different symbols appearing in the language (function symbols<sup>1</sup> and predicate symbols). Preinterpretations provide the building blocks for constructing interpretations. That is, preinterpretations assign meanings to function symbols. One may say that an interpretation is built based on a preinterpretation. This view of constructing models provides a way of constructing different analysers for the same program by providing the appropriate preinterpretation. Ideally, such a preinterpretation abstracts away some information and highlights the properties sought by the analysis.

Methods for constructing and checking models exist. They may be roughly classified as top-down or bottom-up depending on the inference rule underlying the method. The most common inference rules known for logic programming are SLD-resolution and the  $T_P$  operator on programs. SLD-resolution is a correct and complete inference rule for definite logic programs, for all elements of the minimal model have an SLD-refutation [Llo87]. Potentially a combination of a search strategy, the underlying ideas behind SLD-resolution and other techniques may be used to construct models. Another way of computing models for definite logic programs is by computing fix-points of an operator,  $T_P$ , on logic programs. An important result in the semantics of logic programs is the fact that the least fix-point of  $T_P$  corresponds to the minimal model of the program [EK76]. With respect to the way proof trees are constructed, methods based on SLD-resolution may be considered as top-down as opposed to the bottom-up approach of finding a fix-point on  $T_P$ . The program analysis

<sup>&</sup>lt;sup>1</sup>Constants are regarded as 0-ary function symbols.

framework [Sağ98] used in this thesis is based on  $T_P$ , thus it provides a bottom-up analysis tool. This choice does not prevent the analysis from obtaining information which may be obtained with top-down analysis methods. One may think that is the case partly because top-down analysers are goal-directed whereas bottom-up analysers are goal-independent. However, query-answer transformations (Section 2.3) on the program to be analysed simulate goal-directed (e.g. top-down) analysis. Nonetheless, some procedural information (e.g. definite freeness of variables, or possible sharing) in the program may not be derivable using this analysis approach since it is based on a declarative semantics. This is because procedural semantics are more detailed than the declarative semantics.

### 2.1.1 CLP programs

CLP programs may be regarded as a declarative extension to logic programs [JM94]. Hence, their declarative semantics is constructed in a similar way. Definite CLP programs are regarded as first order constraint theories and the concept of substitution for definite programs is generalised to a constraint. Interpretations and preinterpretations are built in a similar manner as with definite logic programs. Function symbols and predicate symbols are given meanings by a preinterpretation and interpretation, respectively. In CLP program interpretations, a difference is made between user and constraint symbols. Constraint function symbols and predicates have a fixed interpretation whereas user function symbols and predicates are given their interpretation as it is done with definite logic programs.

#### 2.1.2 The Method

Abstract interpretation may be regarded as a nonstandard interpretation of programs. It is nonstandard in the sense that program statements are interpreted using values from the abstract domain (an abstraction/approximation to the concrete values) and the deduced information comes from the resulting abstract descriptions of stores, values, etc. [JN95]. In this vein, analysis based on abstract interpretation may be

regarded as interpreter-based. Another approach to abstract interpretation would be to compile-in the abstract domains and functions into the subject program and use standard execution. This compiler approach is known as abstract compilation, and provides the well-known efficiency benefits associated with compilation as opposed to the interpreter-based approach. Abstract compilation in the logic programming setting may be roughly summarised as coding the functions and/or operations on the abstract domain as logic program clauses and introducing them directly into the subject program. Then, find models using standard methods<sup>2</sup> (top-down, or bottomup). The starting point to describe abstract interpretations of CLP programs, with abstract compilation, is to define a nonstandard preinterpretation of the program. Then, through abstract compilation derive an abstract program capturing this preinterpretation. Finally, the analysis consists of the results of computing the minimal model of this CLP program. Abstract interpretation with the interpreter-based approach may be used to compute approximations instead of abstractions. In this vein, we differentiate approximations from abstractions. For instance, the values taken by an integer variable ranging between 1 and 3 may be abstracted as positive and approximated by an open interval between 0 and 4. It may be said that abstractions change or introduce a (new) domain whereas approximations stay within the same domain.

### 2.2 The Approximation

Up to now we have mentioned that for most interesting properties of programs program analysers are unable to find the exact information arising at run-time. This has led to program analysers which compute with approximations (or abstractions) instead of exact descriptions of such information. Informally, assume that all computations of a program can be described by a set of objects  $S_P$ . Any superset,  $S_P'$ , of  $S_P$  is an (over) approximation of this set. Similarly, a program P' is called an approximation of a program P if P' has a larger set of computations than P does.

<sup>&</sup>lt;sup>2</sup>A further pass through a program specialiser may remove some interpretation overhead introduced by the new clauses.

Interestingly, certain properties of P may be proved by studying P', instead of P. In this respect, the analysis results of P' are safe and correct since any behaviour that is not in P' cannot be achieved by P.

The properties derived from the analysis depend on the properties captured by the preinterpretation (as well as the capabilities of the analysis algorithm). Preinterpretation definitions, in turn, use domains as their basic building blocks. Domains may be classified according to different criteria. Here they are grouped into two sets: finite and infinite domains. Within finite domains consider modes, groundness, and sharing. Infinite domains are like those used in type analysis with regular approximations and inter-argument relationships with arithmetic approximations. Due to their infinite nature, and to increase precision, widening and narrowing [CC92b] are employed. This exposition presents several domains for approximating constraint logic programs, namely linear arithmetic expressions, regular programs and a combination of both.

These approximations make use of the concrete values of the original program as opposed to the abstractions of such values captured by preinterpretations with finite domains. Also, with finite domains and preinterpretations the analysis result is the least fix-point of the immediate consequences operator,  $T_P$ , which is the minimal model of a domain program,  $P^3$ . By contrast, using approximations, rather than abstractions, a fix-point is computed on successive programs each defining an approximation to the concrete values used in the source/concrete program. One may regard this generation of approximations as computing interpretations with the core semantics operator  $T_P$ , where programs that represent interpretations replace interpretations.

The sequence of programs  $P_0, P_1, \ldots, P_i$  approximating a program P is defined as

<sup>&</sup>lt;sup>3</sup>A domain program comes from abstract compilation of a nonstandard preinterpretation into the source program thus yielding a program whose symbols belong to the domain of the preinterpretation. It differs from abstract programs in that execution remains unchanged with domain programs while execution of abstract programs may involve an abstraction on execution.

follows.

$$P_0 = \emptyset$$

$$P_{i+1} = P_i \nabla T(P_i)$$

where

$$T(P_i) = \{ q(\bar{x}) \leftarrow \mathtt{solve}(q, P_i) \mid q \in \mathtt{preds}(P) \} \quad (i \ge 0)$$

and

 $\nabla$  is a widening operator.

preds(P) stands for the set of user-defined predicates in program P.  $solve(q, P_i)$  returns the formula obtained by computing an upper bound of the solutions for the clauses of q in program  $P_i$ . The next equation summarises this description.

$$\mathtt{solve}(q, P_i) = \sqcup \{\mathtt{proj}^{vars(H)}\mathtt{simpl}(\mathtt{unfold}(B, P_i)) \mid H \leftarrow B \in \mathtt{proc}(q)\}$$
 (2.2)

 $\operatorname{proc}(q)$  above stands for the set of clauses in P having predicate symbol q. For a clause body B and a constraint approximation  $P_i$ ,  $\operatorname{unfold}(B,P_i)$  denotes the set of formulas obtained by unfolding each user-defined predicate in B using its defining clause in  $P_i$  (if any). The result of unfolding may return a set of constraints which may be simplified.  $\operatorname{simpl}(F)$  produces a simplified (solved) form of the formula F. Temporary constraints generated during unfolding and simplification may be removed by projecting the result of simplification onto the variables of interest.  $\operatorname{proj}^V(F)$  projects formula F onto set of variables V yielding another formula. Finally, the upper bound operation,  $\sqcup$  constructs a definition of the required form for each predicate from a set of individual clauses.

Each program  $P_i$  consists of clauses of the form:

$$p(X_1, X_2, \dots, X_n) \leftarrow \mathcal{C} \quad (n > 0)$$
(2.3)

where C is some constraint. C may take different forms depending on the domain of approximation. Next, we define three domains of approximation and their associated

operations (upper bound, widening, simpl and proj) for their use in the core semantics function equation 2.1 for T above. In a similar work, logic programs are analysed using finite height [RRS95] (or more generally, possessing the finite ascending chain property [GDL95]) constraint domains.

#### 2.2.1 Regular Unary Logic Programs

Types for logic programs have been studied by many researchers [Mis84, Zob87, YS91, JB92, HCC94, GdW94]. Any information about the possible types taken by a program at runtime is considered as descriptive typing as opposed to prescriptive typing where programs are checked against type definitions for compliance. The former typing mechanism is closer to the concept of approximation and thus is the one adopted for the analyses shown here.

Regular types were first introduced by Mishra in [Mis84]. Further developments to this work were proposed by Yardeni et al. in [YS91]. Lately, a similar form of regular programs (with little modifications) was introduced in [GdW94] for program analysis. Informally, regular approximations attempt to describe with regular term grammars (or regular tree automata) the possible values taken by each argument-place occurring in a predicate definition. For the domain of regular programs the clause 2.3 takes the following form, for distinct variables  $X_1, X_2, \ldots, X_n$  (n > 0).

$$p(X_1, X_2, \dots, X_n) \leftarrow t_1(X_1), t_2(X_2), \dots, t_n(X_n) \quad (n > 0)$$
 (2.4)

The body of the clause is a conjunction of regular constraints whose definitions are called regular unary logic clauses. The regular definition of a predicate p is made of a clause such as 2.4 above and the definitions for the  $t_i$   $(1 \le i \le n)$  in its body.

**Definition 2.1** (RUL clause) A regular unary logic (RUL) clause is a clause of the form

$$t_0(f(x_1, x_2, \dots, x_n)) \leftarrow t_1(x_1), t_2(x_2), \dots, t_n(x_n) \quad (n \ge 0)$$

where  $x_1, x_2, \ldots, x_n$  are distinct variables and  $t_i$  are constraints on  $x_i$   $(0 < i \le n)$ .

Regular clauses as defined here are called *canonical* in [GdW94]. Note that all variables  $x_i$  ( $0 \le i \le n$ ) must be pairwise different. During approximation (particularly solve above) it may occur that two subgoals with predicates  $t_i, t_j$  ( $i \ne j$ ) have the same variable as argument. They are replaced by their intersection<sup>4</sup> thus guaranteeing that the form required by the definition above is kept at all times.

**Definition 2.2 (RUL program)** An RUL program is a finite set of RUL clauses in which no two clause heads have a common instance.

Such (canonical) regular programs, also described in [GdW94], define top-down deterministic regular sets of terms. Unlike the word case for regular languages [HU79], deterministic term grammars are less expressive than nondeterministic term grammars, for top-down representations [CDG<sup>+</sup>99].

Furthermore, intersection of regular definitions guarantees the canonical form in the clauses of the approximation program,  $P_i$ , as well as providing part of the definition for the simpl operation (in Eq. 2.2) needed for the core semantics (Equation 2.1) as defined by T above. The other part of the definition of simpl unfolds (SLD-resolution) with their RUL clauses until no constant or function symbol appear in the constraints, and then checks for non-emptiness of the intersection of any sets of unary atoms containing the same variable. This amounts to simplification and consistency checking in the jargon of constraints [JM94].

Projection of regular constraints, proj, consists of removing the constraints (and their associated definitions) on variables that do not appear in the term provided, the clause head in this case.

Widening is that of [SG98a] adapted from a shortening operation [GdW94] where precision is traded for speed. Also containment between regular definitions is used as well as dependency between predicates.

Definition 2.3 (Inclusion of regular predicates,  $\subseteq$ ) Let p and q be two regular predicates defined in RUL program R. Then p is included in q, written as  $p \subseteq q$ 4 An empty intersection is equivalent to failure of solve (Equation 2.2) to obtain any information

<sup>&</sup>lt;sup>4</sup>An empty intersection is equivalent to failure of solve (Equation 2.2) to obtain any information from a given program's clause.

iff for all clauses  $p(t) \leftarrow p_1(x_1), p_2(x_2), \dots, p_n(x_n) \in R$  there exists a clause  $q(r) \leftarrow q_1(y_1), q_2(y_2), \dots, q_n(y_n) \in R$  for  $n \geq 0$  and  $\theta = \{y_1/x_1, y_2, /x_2, \dots, y_n/x_n\}$  a variable renaming such that

- $t = r\theta$  and
- $p_1 \subseteq q_1, p_2 \subseteq q_2, \dots, p_n \subseteq q_n$

**Definition 2.4 (calls, call-chains, depends on)** Let R be a program containing predicates t and s ( $t \neq s$ ). Then t calls s if s occurs in the body of a clause defining t in program R. A sequence of predicate names  $t_1, t_1, \ldots, t_n$  where  $t_i$  calls  $t_{i+1}$  ( $i \geq 1$ ) is a call-chain from  $t_1$  to  $t_n$ . We say t depends on s if there is a call-chain from t to s.

**Definition 2.5 (Shortening of an RUL program)** Let R be an RUL program containing predicates t and s ( $t \neq s$ ). Let an occurrence of predicate t depend on predicate s, and let t and s have the same function symbols in their clause heads. Then the occurrence of s is replaced by t if  $s \subseteq t$ .

Iterative applications of this shortening starting from the original RUL program results in a widened RUL program. Unfortunately it is not guaranteed to terminate [Mil99] and thus the term 'widening' is not strictly justified<sup>5</sup>. In the worst case, it could be combined with a depth-bound, or a similar finite restriction, to ensure termination. In practice, it terminates for all programs encountered. Unlike other widening operators, the widening on RUL programs is applied to one program, rather than successive approximations, and produces a widened program.

The upper bound is the tuple distributive upper-bound used in [GdW94]. Because top-down deterministic regular term languages are not closed under union, upper bound of RUL programs does not define the union but in general it includes the union. Next, we quote the upper bound of regular predicates as defined in [GdW92] and give some strategies to improve the accuracy of the upper bound.

<sup>&</sup>lt;sup>5</sup>But we continue to (mis)use the term since.

**Definition 2.6 (Upper bound of RUL predicates)** Let R be an RUL program, and let  $t_1$  and  $t_2$  be unary predicates defined in R. Then the upper bound of  $t_1$  and  $t_2$ , denoted  $t_1 \sqcup_{\text{rul}} t_2$  is defined as predicate t with definition R' computed as follows.

- $t = t_2$  if  $t_1 \subseteq t_2$ , or
- $t = t_1$  if  $t_2 \subseteq t_1$ , or
- otherwise t is defined by the following clauses.
  - if clause  $t_1(f(x_1, ..., x_n)) \leftarrow s_1(x_1), ..., s_n(x_n) \in R$ and clause  $t_2(f(x_1, ..., x_n)) \leftarrow r_1(x_1), ..., r_n(x_n) \in R$ then clause  $t(f(x_1, ..., x_n)) \leftarrow q_1(x_1), ..., q_n(x_n) \in R'$ where  $q_i = s_i \sqcup_{\text{rul}} r_i \ (0 \le i \le n);$
  - if clause  $t_1(f(x_1, ..., x_n)) \leftarrow s_1(x_1), ..., s_n(x_n) \in R$ and there is no clause  $t_2(f(x_1, ..., x_n)) \leftarrow B \in R$ then clause  $t(f(x_1, ..., x_n)) \leftarrow s_1(x_1), ..., s_n(x_n) \in R'$ ;
  - if clause  $t_2(f(x_1, ..., x_n)) \leftarrow r_1(x_1), ..., r_n(x_n) \in R$ and there is no clause  $t_1(f(x_1, ..., x_n)) \leftarrow B \in R$ then clause  $t(f(x_1, ..., x_n)) \leftarrow r_1(x_1), ..., r_n(x_n) \in R'$

Example 1 (Upper bound of regular predicates) Consider the following two RUL programs defining the lists [a,b] and [c,d], respectively.

```
% First program
s([X|Y]) <- s1(X),s2(Y)
s1(a) <-
s2([X|Y]) <- s3(X),s4(Y)
s3(b) <-
s4([]) <-
% Second program</pre>
```

 $r([X|Y]) \leftarrow r1(X), r2(Y)$ 

```
r1(c) <-
r2([X|Y]) <- r3(X),r4(Y)
r3(d) <-
r4([]) <-
```

The upper bound of RUL predicates s and r is predicate t defined by the following RUL program.

```
t([X|Y]) <- t1(X),t2(Y)
t1(a) <-
t1(c) <-
t2([X|Y]) <- t3(X),t4(Y)
t3(b) <-
t3(d) <-
t4([]) <-</pre>
```

This new program describes lists [a,b], and [c,d] as well as lists [a,d], and [c,b] due to the tuple distributivity of the upper bound. In an application where the order of the elements and their context is important this result is clearly not good enough.

If we knew that the number of possible distinct terms (lists here) were finite, and we knew which terms they were then we could 'wrap' those terms in different function symbols prior to computing their regular definitions, their upper bound would keep them separate. Hence, resulting in a more precise approximation.

**Example 2 (Cont.)** For the example above, it suffices to 'wrap' the two distinct lists in a different function symbol not appearing in the term (to avoid confusing the widening), say f1 and f2. Thus the lists become terms f1([a,b]) and f2([c,d]) whose regular definition follows.

```
% First program
s0(f1(X)) <- s(X)
s([X|Y]) <- s1(X),s2(Y)</pre>
```

The upper bound of RUL predicates s0 and r0 is predicate t0 defined by the following RUL program.

Note that the two clauses for t0 represent the terms 'wrapped' in f1 and f2 respec-

tively. Now the upper bound describes the union of the terms instead of including it, as it generally happens with tuple distributive upper bound.

### 2.2.2 Linear Arithmetic Expressions

Consider a set of linear arithmetic expressions as constraints on the possible values of some variables, provide the operations projection, union, intersection and widening and then we have a domain for approximate analysis. For instance, the convex hull [CH78, BK96, Sağ98] of two sets of linear arithmetic expressions may be used as the upper bound operator. The elements of this domain are sets of vectors in an *n*-dimensional space. Linear arithmetic expressions define sets of vectors. Such linear arithmetic expressions are composed of linear equalities and inequalities<sup>6</sup>. When used for approximate analysis linear arithmetic expressions are referred to as *linear constraints*, for short.

Informally, in a two dimensional space a set of points could be approximated by a convex polygon which encloses all of them. This idea can be generalised to N-dimensional space. In this sense, finding the least convex polyhedra to approximate the points in two given sets of linear constraints may be regarded as finding an upper bound of those sets of linear constraints. Furthermore, the constraint part  $\mathcal{C}$  of approximation clause 2.3 is composed of conjunctions of linear constraints. Thus, the upper bound operation is defined for two sets of linear constraints.

There are several benefits in using arithmetic constraints to approximate constraint logic programs where the variables of the program range over numbers. One of them is that projection (proj) is already provided by the implementation of the language used to program the algorithm for approximation. So is the case for intersection of linear constraints and simplification (e.g. the simpl operator of 2.1). Widening, by contrast, has to be provided. We quote a definition given by H. Sağlam in [Sağ98].

<sup>&</sup>lt;sup>6</sup>Possibly disequalities too, when the upper bound is not required (e.g. determinate nonlooping predicates).

<sup>&</sup>lt;sup>7</sup>Arithmetic constraints in SICStus Prolog [Hol95].

**Definition 2.7 (Widening of linear constraints,**  $\nabla_{ac}$ ) Let  $S_1$  and  $S_2$  be two sets of linear inequalities  $S_1 = \{\beta_1, \beta_2, \dots, \beta_k\}$ ,  $S_2 = \{\gamma_1, \gamma_2, \dots, \gamma_m\}$  defining two polyhedra in  $\mathbb{R}^n$ . Then

$$S_1 \nabla_{ac} S_2 = \{ \beta_i \in S_1 \mid S_2 \Rightarrow \beta_i \} \bigcup$$
$$\{ \gamma_i \in S_2 \mid S_1 \Rightarrow \gamma_i \bigwedge \exists_i (((S_1 - \{\beta_i\}) \cup \{\gamma_i\}) \Rightarrow \beta_i) \}$$

#### 2.2.3 Constrained Regular Unary Logic Programs

RUL programs provide a domain for approximate program analysis just like linear constraints do. The program properties obtained with either domain alone has some associated benefits. Hence, it is desirable to produce a combination of both domains conveying the benefits of both domains. Such a combination reuses some of the definitions for the operations above and produces a combination of both as needed.

One way of achieving this integration [SG98a] of regular constraints and linear constraints is to extend RUL clauses to include linear constraints into their bodies.

**Definition 2.8 (CRUL clause)** A constrained regular unary logic (RUL) clause is a clause of the form

$$t_0(f(x_1, x_2, \dots, x_n)) \leftarrow C_{ac}, t_1(x_1), t_2(x_2), \dots, t_n(x_n) \quad (n \ge 0)$$

where  $x_1, x_2, ..., x_n$  are distinct variables and  $C_{ac}$  is a conjunction of linear constraints projected onto  $x_1, x_2, ..., x_n$ .

CRUL programs are constructed with (canonical) CRUL clauses as above. In addition, canonical CRUL programs are such that no two clause heads in a predicate definition have a common instance. Furthermore, the constraint  $\mathcal{C}$  of clause 2.3 is instantiated to conjunctions of linear and regular constraints. Inclusion of CRUL predicates is defined accordingly.

**Definition 2.9 (Inclusion of constrained regular predicates )** Let p and q be two regular predicates defined in CRUL program R. Then p is included in q, written as  $p \subseteq q$  iff for all clauses  $p(t) \leftarrow C_p, p_1(x_1), p(x_2), \ldots, p_n(x_n) \in R$  there exists

a clause  $q(r) \leftarrow C_q, q_1(y_1), q(y_2), \dots, q_n(y_n) \in R$  for  $n \geq 0$  and variable renaming  $\theta = \{y_1/x_1, y_2/x_2, \dots, y_n/x_n\}$  such that

- $t = r\theta$  and  $C_p \Rightarrow C_q\theta$
- $p_1 \subseteq q_1, p_2 \subseteq q_2, \ldots, p_n \subseteq q_n$

The definition of upper bound for CRUL clauses is adapted from [SG98a] because we do not consider the Herbrand constraints.

**Definition 2.10 (Upper bound of CRUL clauses)** Let a CRUL clause in the definition for predicate p be  $p(f(x_1, ..., x_n)) \leftarrow \mathcal{A}_1, t_1(x_1), ..., t_n(x_n)$  and let CRUL clause  $q(f(y_1, ..., y_n)) \leftarrow \mathcal{A}_2, r_1(y_1), ..., r_n(y_n)$  be in the definition for unary predicate q. The upper bound of unary predicates p and q for these clauses is clause  $pq(f(y_1, ..., y_n)) \leftarrow \mathcal{A}, s_1(y_1), ..., s_n(y_n)$  having  $\mathcal{A} = CH(\mathcal{A}_1\theta, \mathcal{A}_2)$ , where  $\theta = \{x_1/y_1, ..., x_n/y_n\}$ , and  $CH(\mathcal{A}_1\theta, \mathcal{A}_2)$  is the convex hull computed from the conjunctions of arithmetic constraints  $(\mathcal{A}_1\theta \text{ and } \mathcal{A}_2)$ . Also the upper bound between RUL constraints  $t_1(x_1), ..., t_n(x_n)$  and  $r_1(y_1), ..., r_n(y_n)$  is  $s_1(y_1), ..., s_n(y_n)$  respectively.

Next we define a refined version of the widening proposed in [Sağ98]. Basically, the widening is the result of applying the shortening (Definition 2.5) on CRUL programs, instead of RUL programs, until shortening cannot be applied. Besides renaming, this shortening operation computes the widening of the arithmetic constraints associated with the clauses where loops are introduced. Let us first consider the pairs of clauses involved in verifying the inclusions of two CRUL predicates.

**Definition 2.11 (Inclusion Clause Pairs)** Let s and t be two CRUL predicates defined in CRUL program P where  $s \subseteq t$ . The clause pairs considered for checking inclusion of s in t is inductively defined as the smallest set p(s,t) where

$$(cl_s, cl_t) \in p(s, t)$$
 if  $cl_s = s(f(x_1, \dots, x_n)) \leftarrow \mathcal{A}_s, s_1(x_1), \dots, s_n(x_n) \in P,$  
$$cl_t = t(f(y_1, \dots, y_n)) \leftarrow \mathcal{A}_t, t_1(y_1), \dots, t_n(y_n) \in P,$$
 
$$n > 0$$

$$(cl_{s_{i}}, cl_{t_{i}}) \in p(s, t) \quad \text{if} \quad (cl_{s}, cl_{t}) \in p(s, t)$$

$$cl_{s} = s(h(x_{1}, \dots, x_{n})) \leftarrow \mathcal{A}_{s}, s_{1}(x_{1}), \dots, s_{n}(x_{n}) \in P,$$

$$cl_{t} = t(h(y_{1}, \dots, y_{n})) \leftarrow \mathcal{A}_{t}, t_{1}(y_{1}), \dots, t_{n}(y_{n}) \in P,$$

$$cl_{s_{i}} = s_{i}(g(x_{1}, \dots, x_{m})) \leftarrow \mathcal{A}_{s_{i}}, s_{i_{1}}(x_{1}), \dots, s_{i_{m}}(x_{m}) \in P,$$

$$cl_{t_{i}} = t_{i}(g(y_{1}, \dots, y_{m})) \leftarrow \mathcal{A}_{t_{i}}, t_{i_{1}}(y_{1}), \dots, t_{i_{m}}(y_{m}) \in P,$$

$$0 < i < n, \ m, n > 0$$

**Definition 2.12 (Shortening of CRUL programs)** Let t and s ( $t \neq s$ ) be two predicates in CRUL program P. Let an occurrence of predicate t depend on predicate s, and let t and s have the same set of function symbols in their clause heads. If  $s \subseteq t$  all occurrences of s in call-chains from t to s are replaced by t. Also, replace arithmetic constraints  $A_t$  by  $A_k = A_s \nabla_{ac} A_t$  in clause  $cl_t = t(f(y_1, \ldots, y_n)) \leftarrow A_t, t_1(y_1), \ldots, t_n(y_n) \in P$  for every  $(cl_s, cl_t) \in p(s, t)$  where  $cl_s = s(f(x_1, \ldots, x_n)) \leftarrow A_s, s_1(x_1), \ldots, s_n(x_n) \in P$ .

Successive applications of the shortening operation on a CRUL program produce the widening of that CRUL program.

**Example 3** Consider the following CRUL program.

- 1) t1(btm) <-
- 2) t1(f(X,Y,Z)) <-X>=Y, t2(Z)
- 3) t2(btm) <-
- 4) t2(f(X,Y,Z)) <-X>Y, t3(Z)
- 5) t3(btm) <-

There is dependency between t1 and t2 and they have the same set of function symbols in their clause heads. Next, to verify that  $t2 \subseteq t1$  according to the definition of

inclusion of CRUL predicates we have that it holds trivially for clauses 3 and clause 1. Inclusion for clauses 4 and 2 holds since  $X>Y \Rightarrow X>=Y$  and  $t3 \subseteq t2$  (which is verified by clauses 5 and 3). Then the shortened CRUL program is obtained by replacing t2 with t1 in clause 2 where the new constraint is  $X \geq Y = \{X > Y\}\nabla_{ac}\{X \geq Y\}$  (which are the constraints compared in checking inclusion of 3 and clause 1). The shortened CRUL program follows.

```
t1(btm) <-
t1(f(X,Y,Z)) <-
X>=Y,
t1(Z)
```

Note that no further shortening is possible, hence this is a widened CRUL program.

### 2.3 Query-answer transformations

Query-answer transformations provide a way of simulating goal-directed top-down analysis using the bottom-up analysis tool of [Sağ98]. In addition, query-answer transformations provide ways of making the analysis sensitive to program points. This flexibility is achieved by a systematic renaming of the predicates in the program to be analysed. Analysis using the core semantics function 2.1 produces the most general results for every predicate in the program to be analysed. By contrast, analysis for a program which has been transformed with query-answer transformations would produce results for calls and answers for every predicate in the program occurring in a computation of a given initial goal. For instance, given a clause  $p_0 \leftarrow p_1, \ldots, p_n$  (n > 0) its query-answer transformed form has clauses of the form  $call_p \leftarrow call_p \leftarrow call_p$ ,  $answer_p \leftarrow call_p$ ,  $answer_p \leftarrow call_p$  for the first subgoal of the original clause, and clause  $answer_p \leftarrow call_p \leftarrow call_p$ ,  $answer_p \leftarrow call_p \leftarrow call_p$ 

directed analysis. There are several variations to this transformation which can produce the call and answer division considering the position and the predicate name for every subgoal of a clause, instead of the more general call and answer division per predicate name. That is, a different name is given to goals with the same predicate name depending on their clause body where they occur. All these transformations on programs may be regarded as a generalisation of similar techniques used in deductive databases for answering queries to recursive programs, namely [RLK86, BMSU86].

### 2.4 Summary

An analysis framework based on [Sağ98] for CLP programs was described. This analyser is based on the general framework of abstract interpretation [CC92a]. Unlike 'traditional' abstract interpretation where the application-dependent part of the approximation to the collecting semantics is a constant in the semantics, this approach takes it as a parameter in the form of a preinterpretation. Using approximations, rather than abstractions, a fix-point is computed on successive programs each defining an approximation to the concrete values used in the concrete program. Accordingly, an approximation operator is provided to generate the approximations in the sequence (produced during fix-point computation). Such an approximation operator is parameterised by a constraint domain. That is, some constraint language and operations for projecting, simplifying, upper-bound and widening are provided. Next, three constraint domains are described, namely the domain of linear arithmetic expressions, the domain of regular unary logic programs (regular term grammars) and a combination of both, constrained regular unary logic programs. The result of the above integration of constraint domains into a core semantics function is a bottom-up analyser for CLP programs. Also, this framework for approximating programs using infinite domains may be coupled with abstract compilation and finite domains (as well as finite height domains) to increase the number of possible analyses computed. Query-answer transformations on the program to be analysed simulate top-down analysis using a bottom-up analyser. In addition, query-answer transformations may be

2.4. SUMMARY 35

used where the analysis requires information at different program points in the CLP program, as opposed to the conventional analysis approach in logic programming of generating analysis information per predicate.

# Chapter 3

# Specialisation of CLP

The purpose of this chapter is to review specialisation with emphasis on specialisation of CLP programs. By no means do we intend to survey the whole field. For a survey of the field, the interested reader should consult [JGS93, Leu97, HMT98] and the references therein. First, we present specialisation in general, regardless of any specific programming language. Then, we identify the main approaches to specialisation describing their features and main phases. Next, specialisation for CLP programs is discussed. The core algorithm underlying several specialisers [PG99, GP00] used in this thesis is presented. The parameters for such an algorithm are a constraint domain (such as those used for analysis in Chapter 2) as well as other operations commonly used in on-line specialisers for logic programs [Gal91, Leu97]. Finally, we give some remarks on the extensibility and generality of this specialisation algorithm.

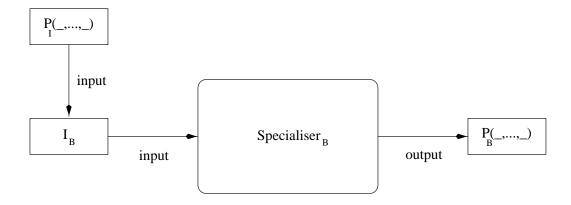
### 3.1 A General View

Specialisation and partial evaluation are terms often used interchangeably to refer to the same concept. Here we argue that they are different concepts and thus should be used differently. Partial evaluation is a program transformation achieved mainly though unfolding and/or procedure inlining. Also, it suggests a specific execution mechanism (evaluation<sup>1</sup>) and thus excludes other transformation techniques [dWG93]. Specialisation, by contrast, does not suggest any transformation strategy. Moreover, it declaratively states the purpose of the transformation without specifying the techniques used. Therefore, we will use specialisation throughout this exposition because it better suits the transformation techniques here described.

The Concept. Specialisation generates a residual program which denotes all computations of a given source program restricted to some constraints on the input variable values. Typically these constraints on some of the input variables are given as equality constraints. That is, the constraints fix the values of some variables upon start of program execution. Often, these equality constraints are called partial input. The main objective of specialisation is to generate residual programs which run faster. Also, for the specialisation to be correct, the transformations performed must preserve some specified program semantics, such as operational semantics, for computations with the input constraints.

Specialisation and Compilation. There are similarities and differences between specialisation and compilation. We argue that specialisation subsumes compilation in many situations. It may be thought that they are inherently different because specialisation is a source-to-source program transformation whereas compilation is not. However, the effect produced by compilation may be achieved with a specialiser through the so-called first Futamura projection [JGS93, pages 75–76] (See Figure 3.1). Given an interpreter  $I_B$  for language I written in language B, and a program  $P_I$  in language I, specialisation of this interpreter with respect to a program and empty input constraints produces a residual program in language B. This amounts to translating a program in language I ( $P_I$ , for instance) to a program in language B (residual program  $P_B$ ), which is commonly associated with compilation. In this regard, specialisation may be thought of as 'removing' layers of interpretation while generating code for the remaining parts. By removing layers of interpretation is meant that some form

<sup>&</sup>lt;sup>1</sup>Evaluation in the standard way. Otherwise it should not be called partial evaluation, see for instance the description of [HDL<sup>+</sup>98].



 $P_{I}(\_,...,\_)$ : A program written in language I (no input constraints).

I<sub>B</sub>: An interpreter wirtten in language B, for programs in language I.

Specialiser B : A specialiser for programs in language B.

 $P_{\underline{B}}(\underline{\ },...,\underline{\ })$ : The residual program (in language B) of specialising an interpreter wrt a program.

Figure 3.1: Specialisation for compilation

of execution on the program reduces the size of some computation paths replacing them by smaller ones. Those computation steps removed correspond to *compile-time* elements of the program, while the remaining computation steps are associated with the *run-time* elements. For this reason the aim of program specialisation could be described as to identify the compile-time layers of a program and thus eliminate them as well as generate code for the remaining code (run-time layers). Then, a specialiser may be conceptually decomposed into two parts: an interpreter part and a compiler part. The former is used to evaluate compile-time expressions and the latter to generate code for the run-time expressions. A distinctive feature is that specialisers attempt to propagate input constraints as well as simplify the source program while compilers scan the input program with no explicit constraints on the input variables.

Main components. In general, program specialisation comprises three main phases: a preprocessing of the source program; an analysis of the program with input con-

straints; and a postprocessing phase (residual program generation). In the first phase the input source program is typically brought into a suitable representation for analysis. In addition, some data-flow and control-flow information is extracted to aid the analysis. During the second phase the source program (or its equivalent intermediate representation) along with the input constraints (or a representation of them) is analysed thus yielding a set of facts conveying information about compile-time and run-time components. Finally, at postprocessing time the residual program is generated according to the results of the analysis.

Conceptually, a program specialiser may be decomposed into the three phases described above. However, existing specialisers need not be in stages. This leads to two main classes of specialisers. Because the preprocessing stage of specialisation has little or nothing to do with the choice of approach to specialisation, but the programming language itself, then the classification does not account for this stage. By contrast, the analysis and postprocessing stages may be altered by the choice made. Hence, these are the specialisation phases that determine the class to which a specialiser belongs to. In particular, the analysis phase determines most of the adjectives attached to a specialiser.

Off-line and On-line Specialisation. Existing specialisers may be roughly classified as either off-line or on-line<sup>2</sup>. Whether the analysis phase is done using concrete values or abstract ones determines the class to which a specialiser belongs to. During program analysis, off-line specialisers compute an approximation to the possible abstract values that the source program variables may take at every program point. The best-known abstract values are static and dynamic for compile-time and runtime components, respectively. Later, the residual program is generated using the input constraints, evaluating components tagged as static and generating code for those tagged as dynamic. On-line specialisers produce instances of some source program constructs (program point specialisation) which represent an approximation to part of the computation state which arises every time execution reaches that pro-

<sup>&</sup>lt;sup>2</sup>A combination of both is possible too.

gram construct. Program generation consists of composing the appropriate instances from the analysis results. The main difference between these two approaches is that off-line specialisation uses an abstraction of the values that program variables may take whereas on-line specialisers handle concrete values. Also, for this reason on-line specialisers are said to take decisions on the spot (according to some predefined rules) whether to continue or stop specialisation, whereas off-line specialisers take those decisions beforehand. The actual program specialisation (for off-line specialisers) occurs during program generation not during the analysis, since the actual values are not available at analysis time<sup>3</sup>.

During analysis a specialiser may find that an n-argument procedure is called with two different patterns. That is, in some places the procedure is called with a constraint on some arguments and no constraint on the rest, and in other places is the converse. A similar situation may arise for individual program statements, when regarded as procedures themselves. Thus, the decision as to how many program statement versions are generated as a result of specialisation leads to a classification of specialisation analyses, and yet another classification of specialisers.

Polyvariance in specialisation. Polyvariance in specialisation may be found in two forms thus rendering the analysis phase results as monovariant or polyvariant. A monovariant analysis produces a single binding<sup>4</sup> for each variable and program point <sup>5</sup> in the program. A polyvariant analysis may find and thus produce more than one binding for the same occurrence (program point) of a variable in the program. Different versions of the same program statement could be generated when its variables are bound to different values during specialisation analysis.

Producing only one version for each program statement, if any, to be included in the residual program seems to be trouble-free, regarding termination. Yet, a monovariant analysis may not terminate for several reasons. Polyvariant analyses, in turn,

<sup>&</sup>lt;sup>3</sup>Attempts to overcome this limitation are [Asa99, HNC97].

<sup>&</sup>lt;sup>4</sup>An abstract description of the contents of a variable.

<sup>&</sup>lt;sup>5</sup>A simpler form of monovariant analysis produces the same binding for a variable throughout the program.

are prone to loop because, for most interesting programs, it is undecidable to know how many different values a variable may take. For this reason, most specialisers borrow techniques from abstract interpretation [CC77] and approximation methods to achieve their goals [GB91, Gal93, Jon97, Leu98]. In a sense, a polyvariant analysis subsumes a monovariant analysis. Because monovariant analysis groups together all versions of a program statement into a single representative one, whereas polyvariant analysis aims to devise a finite partition of the set of versions arising in the program, during specialisation. Each partition defines a possibly different version of a program statement. For each partition, the polyvariant analysis, performs a similar operation to that performed by monovariant analysis in order to find a representative statement of that set.

**Termination of Specialisation.** There are two main sources of nontermination of a specialiser: failure to produce a finite number of versions for each program statement, and failure to stop the evaluation during specialisation. However, these problems can be overcome in different ways throughout specialisation, either during preprocessing of the source program, at analysis time, or during program generation. Each phase may contribute by keeping its results finite so that other phases can proceed and terminate. Nontermination of the preprocessing stage is not considered as a source of nontermination of the specialiser because the preprocessing stage normally performs simple transformations on the source program to aid the following stages (analysis and program generation). This is an ad-hoc transformation or change of representation and has little or nothing to do with the specialisation itself. It depends on the input program. However, a specialiser with a nonterminating preprocessing stage leads to nontermination of the whole specialiser. Nontermination of specialisation occurs when the analysis fails to terminate. As a result, the program generation may not even start. A postprocessing stage that does not terminate or fails is a sign of an analysis phase that is incorrect, since the program generation, in its simplest form, could be regarded as a verification of the results of the analysis phase. In this sense, the program generation phase blindly follows the results of the analysis phase whether

they are safe or not. Termination of the analysis phase (and thus of specialisation) follows from the finiteness of the set computed. That is, a suitable abstraction of the possible values that program variables may take, for every program point, ensures termination (off-line analysis). Otherwise some means of finding a representative statement (e.g. abstraction) for some program points is required (on-line analysis). For analysis methods based on abstract interpretation it means that some ordering on the abstract domain is required as well as an upper bound operation and possibly widening and/or narrowing operators.

Termination and soundness are the two main properties required<sup>6</sup> in any specialiser. The more aggressive a specialiser is in its transformations, the harder it is to ensure and prove termination. Soundness, in turn, is no different since correctness of every transformation rule and operation used must be provided as well as some compliance with some global criteria on the results produced by the combination of the operations involved in specialisation.

Soundness of Specialisation. For residual programs to be semantically equivalent to its source program with input constraints the specialiser must be sound. A sound off-line specialiser must compute a safe approximation in its analysis phase. That is, every possible step of a computation arising when running the source program (with the given input constraints) is correctly abstracted by the bindings computed during analysis. An on-line specialiser is sound if all transformation rules and operations used are sound with respect to the language semantics to be preserved. Also, all possible statement bindings arising during execution of the residual program are instances of the set of facts computed in the analysis phase. Thus, programming languages with clear semantics are amenable to sound specialisation. Usually, the semantics to be preserved are implicit in the analysis and program generation phases of the specialiser.

<sup>&</sup>lt;sup>6</sup>However, some researchers do not consider nontermination of specialisation in the presence of static infinite loops as a failure of the specialiser but a 'poor programming technique' [Jon88, Das98].

### 3.2 Logic Programming Approach

In what follows we describe an extension, from normal logic programs to CLP programs, of an existing on-line specialiser [Gal91]. With respect to preprocessing and postprocessing phases, the specialiser remains the same as it was originally proposed [Gal91]. In this light, the present section is devoted to the analysis phase and the other phases will be explained where required. The underlying core algorithm for analysis is adapted and parameterised with respect to a constraint domain, such as those used for analysis in Chapter 2. The modification consists of extending the algorithm to compute a set (see Section 3.2.3) of constrained atoms, as opposed to just atoms. Some definitions are introduced to keep the presentation self-contained. Terminology without definition is part of the standard background knowledge of the field of logic programming and constraint logic programming. Their definitions could be found in [Llo87, JM94, MS98].

Although there exist some off-line specialisers for logic programs [MB92, LJ99], their automation is still an issue. Perhaps it is because the so-called static-dynamic distinction is hardly present in logic programs or rather because it appears in a different form. To our knowledge, existing automatic specialisers for logic programs are on-line [Sah91, Gal91, Leu97].

### 3.2.1 On-line specialisation

Unlike off-line specialisation which analyses programs by abstracting their values replacing them by elements in a finite domain (namely, static and dynamic), on-line specialisation analyses programs using the concrete values, of which in principle there could be an infinite number. As a result, the on-line specialisation may not terminate if the analysis continues generating new instances of a given variable which potentially has an unbounded number of different possible values. For this reason, analysis for on-line specialisation adopts some strategies, during analysis, to ensure termination. Such strategies are grouped into two main classes: global control and local control. If soundness of the specialiser is to be proved according to the framework of Lloyd and

Shepherdson [LS91], then finding a set of atoms is the target of the analysis phase. In this regard, the global control amounts to finding a finite set of atoms and the local control with constructing finite SLDNF-trees for every atom in the given set [Leu97, Chapter 2]. Here we replace atoms by pairs consisting of an atom and a constraint, and use the same framework to show soundness of our specialisers. Hence, the set of atoms is extended to a set of constrained atoms and the finite SLDNF-trees are replaced by CLP-computation trees. In addition, resolution steps are extended to constraint solving.

We refer to unfolding as the transformation whose core is constraint solving (formerly SLDNF-resolution) coupled with other transformation techniques (e.g. more specific programs [MNL88]). In order to construct finite computation trees some rules state when to stop unfolding<sup>7</sup>. The set of rules which determine the unfolding strategy is called the unfolding rule<sup>8</sup>. Intuitively, given a program and a constrained atom the unfolding rule produces a definition for that atom. Every clause in such a definition is called a resultant. A partial evaluation of a program and an atomic constrained goal is made of the definitions obtained from unfolding using the program clauses starting from the constrained atom. Next, more constrained atoms may be extracted from such definitions thus forming a set of constrained atoms. One of the main problems of correct specialisation is to compute a set of constrained atoms which is closed and independent, according to Lloyd and Shepherdson [LS91]. The following definitions, adapted from [Gal93] to CLP, formalise the concepts mentioned above.

**Definition 3.1 (Constrained Atom)** Let A be an atom,  $\overline{x}$  its set of variables, and c constraints on some set of variables  $\overline{y}$  ( $\overline{y} \subseteq \overline{x}$ ) over constraint domain  $\mathcal{D}$ . The pair  $\langle A, c \rangle$  is a constrained atom.

Note that this definition should be extended for the domain of linear arithmetic constraints in order to get a correct definition of the instance relation (definition 3.5) and more precise results of the upper bound operation (definition 3.8). For instance,

<sup>&</sup>lt;sup>7</sup>Other approaches besides unfolding exist for the same purpose e.g. BU specialisation [VMSV98].

<sup>&</sup>lt;sup>8</sup>The unfolding rule may be viewed as a combination of several rules.

given the atom p(2) with no constraint (true), the associated constraint atom is  $\langle p(X), X=2 \rangle$ . A new fresh variable is introduced and an equality constraint is appended to the existing constraints (true). A similar reasoning would apply to other constraint domains not considered in this thesis, such as the string constraint domain. Throughout the thesis we assume that the appropriate constrained atom is used according to the constraint domains involved (either adding new constraints and variables to the already existing ones or just applying the definition above). An empty constraint is represented as true (for the domain of linear constraints) or as conjunctions of any(X) for the appropriate variables X in the domain of regular programs (Section 2.2.1). Thus, to avoid confusion, we shall adopt the convention of writing empty to denote an empty constraint in any constraint domain. Similarly, the associated goal with constrained atom  $\langle A, c \rangle$  is  $\leftarrow A, c$ . In the following we will use CLP( $\mathcal{X}$ ) to denote a constraint logic programming language over any domain (here represented as  $\mathcal{X}$ ). Also, we will omit the domain name where it is implicit in the context or it is not needed.

**Definition 3.2 (CLP Resultant)** A  $CLP(\mathcal{X})$  resultant is a first order constraint formula  $Q_1 \leftarrow Q_2$  where  $Q_1$  is either absent or a conjunction of user-atoms,  $Q_2$  is a conjunction of constraint-atoms c, and user-atoms  $Q_u$ , and any variables in  $Q_1$  or  $Q_2$  are assumed to be universally quantified, over their respective domain, at the front of the resultant.

**Definition 3.3 (CLP Resultant of a Derivation)** Let P be a  $CLP(\mathcal{X})$  program and  $G_0 = \leftarrow G$ , c be the goal associated with constrained atom  $\langle G, c \rangle$ . Let  $G_0, \ldots, G_n$  be a derivation of  $P \cup \{G_0\}$ , where  $G_n = \leftarrow Q$ ,  $c_n$  and  $c, c_1, c_2, \ldots, c_n$  is the sequence of accumulated constraints associated with the steps of the derivation over constraint domain  $\mathcal{X}$ . Then the derivation has length n, accumulated constraint  $c_n$ , and  $CLP(\mathcal{X})$  resultant  $G \leftarrow Q$ ,  $c_n$ .

Remember that in the extension of the theory of logic programs to the theory of CLP programs substitutions are replaced by constraints. Since constraints may be used to express the query as well as the answer in a CLP scheme [JM94], we

could think of constraint  $c_n$  as containing equality constraints for the possible bindings of the variables in the head (the answer substitution) and some of the as yet unsolved constraints collected during the derivation. Constraints are either generated by collecting the arithmetic expressions found during unfolding [PG99] or alternatively regular approximations are computed for those terms that are removed during the msg operation [GP00]. Then during specialisation the unfolding rule (Definition 3.7) checks that the substitutions applied during unfolding are consistent with the constraints.

Example 4 (Derivation using arithmetic constraints) Consider the following CLP program.

4) 
$$r(A) \leftarrow A1 = 2*A,$$
  
 $p(A1,A)$ 

5) 
$$r(A) \leftarrow A1 = A-2$$
,  
 $p(A1,A)$ 

Given the query <-p(X,Y) and input constraints X>Y+3 a CLP derivation could be the following.

```
<- p(X,Y), X>Y+3
|
<- X>2,q(X,Y),X>Y+3
|&
<-q(X,Y),X>2,X>Y+3
|
<-r(X),Y=1,X>2,X>1+3
|&
```

```
<-r(X),Y=1,X>4

|
<-A1 = 2*X,p(A1,X),Y=1,X>4

|&
<-p(A1,X),A1>8,A1 = 2*X,Y=1
```

In the first derivation step clause 1 is used<sup>9</sup>; then constraints are placed together (and consistency check takes place) before resolving with clause 3; the next step shows the constraints before constraint satisfaction is attempted. The result of consistency checking<sup>10</sup> is displayed in the next step before resolving with clause 4; and finally constraints are collected at the end and constraint satisfaction is attempted resulting in the last triple of constraints for the last goal of the derivation. The resultant of this derivation is

$$p(X,Y) \leftarrow p(A1,X), A1>8, A1 = 2*X, Y=1$$

The accumulated constraints associated with the derivation are those which occur in the goal just before every SLD-resolution step (after those indented and marked with &) and the constraints in the last goal. That is, (X>Y+3), (X>2,X>Y+3), (Y=1,X>4), (A1>8,A1 = 2\*X,Y=1).

In the following example constraints are over the domain of regular programs. A program defining the unary predicates (and auxiliary predicates) used as constraints is provided as opposed to arithmetic constraints where no program has to be given explicitly.

Example 5 (Derivation using regular constraints) Given the following CLP program.

- 1) st(A,[fin|L],St) <-</pre>
- 2) st(A,[plus|L],St) <-</pre>

<sup>&</sup>lt;sup>9</sup>Using clause 2 yields an inconsistent set of constraints; hence this clause cannot be used in a successful derivation.

<sup>&</sup>lt;sup>10</sup>For some constraint solvers checking consistency amounts to simplification as a by-product.

```
st(s(A),L,St)
3) st(s(A),[minus|L],St) <-
      st(A,L,St)
4) st(A,[rtn|L],fr(T,St)) <-
      cont(T,NL),
      st(A,NL,St)
5) cont(1,[plus,plus,plus,fin]) <-</pre>
6) cont(2,[plus,minus,rtn,plus]) <-</pre>
7) cont(3,[minus,plus,plus,rtn]) <-</pre>
Consider the following atomic goal <- st(A,B,St) with any(A),t1(B),t3(St) as
input constraint and associated RUL program as follows.
t1([]) <-
t1([X|Y]) <-
      t2(X),
      t1(Y)
t2(plus) <-
t2(rtn) <-
t3(bt) <-
t3(fr(X,Nxt)) <-
      t4(X),
      t3(Nxt)
t4(1) <-
t4(2) <-
A CLP derivation for the above goal could be the following.
\langle -st(A,B,St),t1(B),t3(St) \rangle
 1
\leftarrow st(s(A),L,St),t1([plus|L]),t3(St)
 | &
 <-st(s(A),L,St),t2(plus),t1(L),t3(St)
```

As in the previous example we used & to mark those steps where operations on constraints are taking place. Now, the previous derivation is achieved by first resolving with clause 2 and after constraint simplification, resolving with clause 4. Then constraint simplification takes place. Finally the last resolution step takes clause 6 and some simplification of the constraints occur. The resultant of this derivation is

```
st(A,[plus,rtn|L],fr(1,St1)) <-
    st(s(A),NL,St1),
    t1(L),t3(St1).</pre>
```

Respectively, the accumulated constraints associated with this derivation are 11 (t1(B),t3(St)),(t1(L),t3(St)),(t1(L1),t4(T),t3(St1)),(t1(L1),t3(St1)).

**Definition 3.4 (Partial Evaluation)** Let P be a  $CLP(\mathcal{X})$  program and  $\langle A, c \rangle$  a constrained atom. Let T be a computation tree for  $P \cup \{\leftarrow A, c\}$ , and let  $\leftarrow G_1, c_1, \ldots$ ,

<sup>&</sup>lt;sup>11</sup>We have purposely omitted all any constraints which hold trivially and are given to all free variables.

 $\leftarrow G_n, c_n$  be the goals chosen from the non-root nodes of T such that there is exactly one goal from each non-failing branch of T. Let  $c_1, \ldots, c_n$  be the accumulated constraints of the derivations from  $\leftarrow A, c$  to  $\leftarrow G_1, c_1, \ldots, \leftarrow G_n, c_n$  respectively. Then the set of resultants  $\{A \leftarrow G_1, c_1, \ldots, A \leftarrow G_n, c_n\}$  is called a partial evaluation of  $\langle A, c \rangle$  in P. If A is a finite set of constrained atoms, then the partial evaluation of A in A is the union of the partial evaluation of the elements in A.

We will see in Chapter 5 that the decision of which non-root nodes are chosen (definition above) to define a partial evaluation may produce residual programs with some desired properties.

**Definition 3.5 (Instance relation)** Given two constrained atoms  $CA = \langle A, c_A \rangle$  and  $CB = \langle B, c_B \rangle$  over constraint domains  $\mathcal{D}$ . instance(CA, CB) holds iff  $B = A\theta$  and implies( $c_B, c_A$ ).

Care should be taken to use the appropriate form of the constrained atoms in the constraint domain of linear arithmetic expressions. Also, the *implies* relation is defined in the constraint domain chosen. For instance, it may be entailment in the domain of linear arithmetic expressions (Section 2.2.2), or inclusion for the domain of regular approximations (Section 2.2.1).

**Definition 3.6 (Closedness)** Let S be a set of constrained first order formulas and A a finite set of constrained atoms. Then S is A-closed if for each constrained atom CB occurring in S,  $\exists CA \in A$ : instance (CA, CB) holds.

Independence is the other requirement for soundness of a specialiser, according to [LS91]. This condition requires that no two constrained atoms in the set of constrained atoms have a common instance. We will not test for independence. However, this condition can be met if renaming is properly applied to constrained atoms.

Clearly, given a constrained atom A and a CLP program P there are infinitely many partial evaluations for A and P. Some fixed rule for generating resultants is used, called an *unfolding rule*.

**Definition 3.7 (Unfolding Rule)** An unfolding rule U is a function which given a  $CLP(\mathcal{X})$  program P and an constrained atom A, returns exactly one finite set of CLP resultants that is a partial evaluation of A in P. If A is a finite set of constrained atoms and P a  $CLP(\mathcal{X})$  program, then the set of resultants obtained by applying U to each constrained atom in A is called a partial evaluation of A in P using U.

Once the unfolding rule is set it may be used to generate new constrained atoms. Next, another rule is needed to decide which atoms are to be unfolded using the unfolding rule. Moreover, this extra rule is responsible for the polyvariance. We shall refer to this rule as the *global control* because it decides which atoms will compose the set of atoms. By contrast, we refer to the unfolding rule as the *local control* since it determines what the definitions for the constrained atoms look like. Termination at this level is referred to as *local termination* while *global termination* is given at the other level.

Consider the algorithm of Figure 3.2 for computing a set of constrained atoms (parameterised by an unfolding rule Cr and a generalisation operation, **generalise**). The unfolding rule Cr constructs clauses from a set of atoms and a program. This rule decides the shape of the computation trees and its responsibility is to keep them finite. Given some computation trees (P'), their leaves (constrained atoms) are collected into a set of possibly new constrained atoms (R). Next, through the **generalise** $(R, A_i)$  operation these new constrained atoms and the old ones  $(A_i)$  are combined according to some similarity criteria and a new set of (possibly new) constrained atoms is produced,  $A_{i+1}$ . These three steps are repeated until the set of constrained atoms is closed in the sense that other pass through the loop results in no new constrained atoms, according to the similarity criteria, and constraint equivalence. The **generalise** $(R, A_i)$  operation involves the msg operator, upper bound and widening. Upper bound and widening are reused from those defined in Chapter 2 depending on the domain used.

Note that constraints are collected from derivations and approximated as described in Chapter 2. During generalisation upper bound and widening provide a safe approximation to the possible values that the constrained variables may take during specialisation (see Section 3.2.3). Compared with analysis (Chapter 2) the operations on

```
INPUT: a (CLP) program P, and atomic goal \leftarrow D with input constraints C_0 GIVEN: An unfolding rule {\tt Cr} and a generalisation operator {\tt generalise}(\cdot,\cdot) OUTPUT: a set of pairs \langle {\tt atom}, {\tt constraints} \rangle begin A_0 := \{\langle D, C_0 \rangle \} i := 0 repeat P' := {\tt a partial evaluation of } A_i {\tt in } P {\tt using } {\tt Cr}. R := \{\langle p(\overline{t}), C_1 \rangle \mid (\langle B \leftarrow Q, p(\overline{t}), Q', C \rangle \in P' \lor (B \leftarrow Q, {\tt not}(p(\overline{t})), Q', C) \in P' ) \land C_1 {\tt is } C {\tt projected onto } vars(\overline{t}) \} A_{i+1} := {\tt generalise}(R, A_i) i := i+1 until A_i = A_{i-1} (modulo variable renaming and constraint equivalence) end
```

Figure 3.2: The specialisation algorithm

the constraint domain remain the same, the difference is that analysis would compute approximations per predicate name, and (polyvariant) specialisation would compute an approximation per predicate in a partition (defining a different version for that predicate), which means that there could be several versions of the same predicate<sup>12</sup>. Roughly speaking, constrained atoms could be regarded as another way of representing the clauses used (e.g. clause 2.3) to express the program analysis. In this sense, constraints are used to increase precision by providing some 'memory' where some invariants of the history of specialisation are recorded. Also, constraints provide a way of pruning through consistency checks during unfolding. Elsewhere [LM99] the need for constraint reasoning to enhance specialisation was suggested.

Anticipating some of the operations that will be explained later, let us see how information is preserved when constraints are used.

Example 6 (Upper bound using RUL constraints) Consider the next two atoms whose upper bound is required.

```
m(X,Y,[b,c])
m(U,V,[c])
```

Using the msg operator the result is the following atom.

```
m(R,S,[B|C])
```

The information about the lists [b,c] or [c] is lost, only some structure is preserved, namely a list with at least one element. By contrast, the upper bound, using RUL constraints combined with the msg operator, is the constrained atom (m(R,S,Q), t(Q)) where constraint predicate t is defined by the following RUL program.

```
t([X1|X2]) <-

t1(X1),

t2(X2)

t1(b) <-
```

<sup>&</sup>lt;sup>12</sup>Arguably, monovariant specialisation would compute the same results as program analysis, since there would be only one version per predicate, less those eliminated by specialisation.

```
t1(c) <-
t2([]) <-
t2([X1|X2]) <-
t3(X1),
t4(X2)
t3(c) <-
t4([]) <-
```

Note that the constraint restricts the possible function symbols appearing in the argument Q as opposed to using the msg operator.

Soundness of specialisation. Upon termination of the algorithm a closed set of constrained atoms  $A_i$  (upon exit from the repeat loop) is produced. Using the same unfolding rule we may construct a residual program, P', for the set of constrained atoms output from the algorithm, namely a partial evaluation of  $A_i$  in P. Next, the soundness of specialisation based on the above algorithm could be stated in the style of Lloyd-Shepherdson [LS91] as follows. For CLP program P and goal G (the associated goal with constrained atom  $\langle D, C_0 \rangle$ )  $P \cup \{G\}$  has an accumulated constraint c if  $P' \cup \{G'\}$  has accumulated constraint c' (included in c), if instance (G', G) holds. Also,  $P \cup G$  finitely fails if  $P' \cup G'$  does. The only difference with standard specialisation is that we extend SLDNF-derivations to derivations in CLP, which is sound. Thus the existing results for soundness of specialisation [LS91] could be readily extended to specialisation of CLP programs.

Note that the above algorithm is flexible with respect to the unfolding rule (local control), the generalisation operation (global control) and a constraint domain to express constraints. When the constraint domain is given by the Herbrand interpretations, this algorithm reduces to an existing algorithm for specialisation [Gal93]. As a result, the quality of the residual programs produced by specialisation depends on the choice for each parameter. Conceptually, their design may be independent but in practice local control and global control are strongly related<sup>13</sup>. The unfolding rule

<sup>&</sup>lt;sup>13</sup>During analysis information is exchanged between these two thus determining their precision.

is associated with the local control, while the generalisation operation is associated with the global control. Next, we discuss some issues on the design of every parameter. For the discussion about the constraint domain design we refer the reader to Chapter 2.

#### 3.2.2 Local Control

For constraint logic programs the core component of the unfolding rule is constraint solving. Some other heuristics are attached to this rule to increase precision. For instance, more specific transformations, look-ahead, etc. Hence, unfolding is synonymous with several operations on constraints such as simpl and proj (discussed in Chapter 2) for constraint atoms and some form of resolution for user atoms. In the case of nondeterministic choices, a common situation arising during specialisation, some decision should be made whether to continue or stop altogether. Here we identify some desirable characteristics [Gal93] which an unfolding rule must possess.

- Unfolding must not 'compile-in' poor control choices, inadvertently increasing the run-time search space. That is, it should avoid search space explosion and code duplication.
- It should not remove indexing information from the program, which is needed to get fast clause selection possibly removing some unnecessary choice points.
- Unfolding tends to propagate data structures through substitutions at compile time. This can have a harmful effect on space efficiency, and hence time efficiency, in the specialised program. Hence, it is desirable to remove redundant structure from the original program.

The first two items above could be encoded within the unfolding rule. The last item is part of another phase during specialisation, namely program generation. Some techniques to achieve the above results could be to have an unfolding rule incorporating one or more of the following elements. Determinacy. Except once, only select atoms that unify a single clause head. A variant of this notion includes a so-called "look-ahead" to detect failure at deeper levels.

Well-founded orders. Impose some well-founded order in selected atoms. It guarantees termination but fails to be a general rule for better specialisation/partial-evaluation.

Homeomorphic embedding. Same as above but using well-quasi orders instead. It may capture cases where the above techniques are conservative but suffers from the same problems, it is application dependent as well as being expensive.

An application of the unfolding rule may be depicted as a tree. We say that an unfolding rule is determinate if the trees constructed using such a rule have all but one node with only one successor. Depending on the place of the tree where this nondeterministic choice occurs the rule adopts different names such as shower, fork, beam, and pure after the shape of the tree constructed [Leu97, page 40]. Our specialisers use determinate unfolding with shower shape. That is, the root of the tree is the only node allowed to have more than one child. Clearly, this has important implications for the quality of the specialised programs produced by specialisation. That is, if the program to be specialised contains too much nondeterminism our specialisers would produce little or no specialisation at all. Programs whose structure or data (constraints) help in resolving nondeterministic choices during unfolding are good for specialisation. Because this is hardly the general case several strategies have been developed to guide the programmer in writing programs for specialisation [Jon96].

#### 3.2.3 Global Control

The global control determines the polyvariance (number of different versions of predicates) in the specialised program. It also ensures global termination, since it incorporates some similarity criterion which is used for deciding how many versions of each program point appear in the specialised program. Termination may be achieved

if the number of versions is finite. One way to have a finite number of versions is through an appropriate abstraction of the constrained atoms<sup>14</sup>. Some contributions into the solution of this problem are *characteristic trees* [GB91], *trace terms* [GL96] and other related methods [MG95, LM96]. Our specialisers use trace terms to set the polyvariance of specialisation.

Roughly speaking, a trace term is an abstraction of a constrained atom. A trace term describes some 'shape' information of the partial evaluation tree induced by the constraint atom it represents. The shape of the tree is represented as nested terms whose names come from the names of the clauses used to construct the partial evaluation tree; their arity corresponds to the number of subgoals in the clause they label; every clause is given a different term name prior to specialisation; and the leaves of this possibly incomplete tree are systematically labelled with different constants. Hence, the set of constrained atoms in algorithm 3.2 is partitioned into nonempty subsets, equivalence classes, each having the same trace term as abstraction. That is, all elements in a equivalence class have the same trace term. Each equivalence class defines a version of a predicate in the specialised program. A version, in turn, is a constrained atom itself. Thus, all constrained atoms in the same equivalence class contribute to form a single constrained atom representing the information contained in all of them. This is achieved firstly by computing the upper bound of the current atom representing the partition and any incoming atom arising during specialisation whose trace term corresponds to that partition. Secondly, the widening is applied to the result of the upper bound and the new incoming constrained atom. For instance, if  $S = \{S_1, S_2, \ldots, S_n\}$  is the current set of constrained atoms at *i-th* iteration of the specialisation algorithm, and  $\{A_1, A_2, \ldots, A_n\}$  are the corresponding upper bounds of every partition  $S_i$  (0 <  $i \leq n$ ) of S, and the elements of S' (newly incoming atoms sorted by trace term), then **generalise** $(S', S) = \mathbf{widen}(S', \{A_1, A_2, \dots, A_n\}),$ where **widen** is the widening of the two sets applied to the elements position-wise. Appropriate widening operators (such as those used in Chapter 2) are used depending on the constraint domain employed. The upper bound here is a combination of upper

<sup>&</sup>lt;sup>14</sup>Clearly, techniques based on abstract interpretation [CC92a] play an important role here.

bound from the constraint domains of Chapter 2 and the *msg* operator. The following definitions show how to compute the upper bound of constrained atoms depending on the constrain domain used. Firstly we state the upper bound of constrained atoms using linear arithmetic expressions (adapted from an algorithm presented in [SG98a]).

**Definition 3.8** ( $\sqcup_{a-ac}$ ) Let  $CA_1 = \langle p(\overline{t_1}), A_1 \rangle$  and  $CA_2 = \langle p(\overline{t_2}), A_2 \rangle$  be two constrained atoms in the domain of linear arithmetic expressions. The upper bound,  $\sqcup_{a-ac}$ , of these two constrained atoms,  $CA_1 \sqcup_{a-ac} CA_2$ , is defined as constrained atom  $CA = \langle p(\overline{t}), A \rangle$  where

#### begin

$$\overline{t} = \operatorname{msg}(\overline{t_1}, \overline{t_2})$$

$$\theta_1 = \operatorname{mgu}(\overline{t}, \overline{t_1})$$

$$\theta_2 = \operatorname{mgu}(\overline{t}, \overline{t_2})$$

$$\mathbf{while} \ \exists x_i/v, x_k/v \in \theta_1, i \neq k, v \ occurs \ in \ \mathcal{A}_1$$

$$\theta_1 = \theta_1 - \{x_k/v\}$$

$$\mathcal{A}_1 = \mathcal{A}_1 \cup \{x_i = x_k\}$$

$$\mathbf{end}$$

$$\mathbf{while} \ \exists y_j/w, y_l/w \in \theta_2, j \neq l, v \ occurs \ in \ \mathcal{A}_2$$

$$\theta_2 = \theta_2 - \{y_l/w\}$$

$$\mathcal{A}_2 = \mathcal{A}_2 \cup \{y_j = y_l\}$$

$$\mathbf{end}$$

$$\rho_1 = \{v/x_i \mid x_i/v \in \theta_1, v \ is \ a \ variable\}$$

$$\rho_2 = \{w/y_j \mid y_j/w \in \theta_2, w \ is \ a \ variable\}$$

end

 $\mathcal{A} = CH(\mathcal{A}_1 \rho_1, \mathcal{A}_2 \rho_2)$ 

and  $CH(A_1\rho_1, A_2\rho_2)$  is the convex hull computed from the conjunctions of arithmetic constraints  $(A_1\rho_1 \text{ and } A_2\rho_2)$ .

Secondly, for the domain of regular unary logic programs the upper bound [GP00] of constrained atoms is defined as follows.

**Definition 3.9** ( $\sqcup_{a-\text{rul}}$ ) Let  $CA_1 = \langle p(\overline{t_1}), \mathcal{A}_1 \rangle$  and  $CA_2 = \langle p(\overline{t_2}), \mathcal{A}_2 \rangle$  be two constrained atoms in the domain of regular unary logic programs. The upper bound,  $\sqcup_{a-\text{rul}}$ , of these two constrained atoms,  $CA_1 \sqcup_{a-\text{rul}} CA_2$ , is defined as constrained atom  $CA = \langle p(\overline{t}), \mathcal{A} \rangle$  where

$$\overline{t} = \text{msg}(\overline{t_1}, \overline{t_2})$$

$$\theta_1 = \text{mgu}(\overline{t}, \overline{t_1})$$

$$\theta_2 = \mathrm{mgu}(\overline{t}, \overline{t_2})$$

Let  $\{x_1, \ldots, x_n\}$  be the distinct variables occurring in term  $\bar{t}$  and  $\mathcal{A} = r_1(x_1), \ldots, r(x_n)$ where  $x_i/\bar{t}' \in \theta_1, x_i/\bar{t}'' \in \theta_2$ , construct a definition for predicate  $r_i(0 \le i \le n)$  as follows.

- 1. Construct an RUL definition for t' according to  $A_1$  whose top unary predicate is r'.
- 2. Construct an RUL definition for t'' according to  $A_2$  whose top unary predicate is r''.
- 3. Let  $r_i$  be the upper bound of RUL predicates r' and r''.

Finally, the upper bound of constrained atoms in the domain of CRUL programs is similar to the upper bound with RUL programs exchanging RUL for CRUL throughout the definition. The arithmetic constraints in the CRUL definitions come from the constraints on the variables of terms t' and t'' as well as from computing the upper bound of CRUL predicates. Next, an example showing the use of widening.

**Example 7 (Widening with RUL constraints)** Consider the constrained atom result of upper bound in Example 6.  $\langle m(R,S,Q), t(Q) \rangle$  where predicate t is defined by the following RUL program.

```
t([X1|X2]) <-
t1(X1),
t2(X2)

t1(b) <-
t1(c) <-
t2([]) <-
t2([X1|X2]) <-
t3(X1),
t4(X2)

t3(c) <-
t4([]) <-
```

If we have to compute the upper bound of this constrained atom and constrained atom (m(X,Y,[b|Z]), any(X), any(Y), t(Z)) the result is the constrained atom (m(R,S,Q), r(Q)) whose RUL program is.

```
r([X1|X2]) <-
r1(X1),
rt(X2)
r1(b) <-
r1(c) <-
rt([]) <-
rt([X1|X2]) <-
rt1(X1),
rt2(X2)
rt1(b) <-
rt1(c) <-
rt2([]) <-
rt2([X1|X2]) <-
rt3(X1),
rt4(X2)
```

```
rt3(c) <-
rt4([]) <-
```

Now, this RUL program could be shortened (once, thus producing a widened RUL program). There is dependency between predicates  $\mathtt{rt}$  and  $\mathtt{rt2}$  and they have the same set of function symbols in their clause heads. Also  $\mathtt{rt} \subseteq \mathtt{rt2}$ . Finally, the widened RUL program is depicted next.

```
r([X1|X2]) < -
    r1(X1),
    rt(X2)
r1(b) <-
r1(c) <-
rt([]) <-
rt([X1|X2]) <-
    rt1(X1),
    rt(X2)
rt1(b) <-
rt1(c) <-
rt2([]) <-
rt2([X1|X2]) <-
    rt3(X1),
    rt4(X2)
rt3(c) <-
rt4([]) <-
```

Now this RUL program succeeds with any list whose elements are b or c. Particularly the lists which have any number of b's and the last element is a c.

Note that programs with accumulators would benefit from RUL constraints. In particular, if the list of elements ([b,b,c]) above denotes the stack of activation records in an interpreter for an imperative language (see Chapter 4) this information would prevent return to an undefined program point in the imperative program. RUL

approximations of the stack of activations records would restrict the possible places where a procedure call may return. We will see (Section 4.2.2) that the representation of the elements in such a stack determines the precision of the approximation (mainly because the upper bound is tuple-distributive, Section 2.2.1).

Related work. Consel [CK91] sets out the theory for generalising functional program specialisation using other domains besides the usual domain with concrete values for on-line specialisation and the static and dynamic values for off-line specialisation. It achieves some form of specialisation aided by program analysis domains. Constraints were not considered though. In a different approach, Hickey and Smith [HS91] use five transformation rules, given in the style of unfold-fold transformation [PP94], to specialise CLP programs. Their algorithm for specialisation traverses the depth first spanning tree of the CLP program callgraph applying these rules to every predicate associated with the visited nodes. Apparently, their generalisation operation is based on syntactic equality (and probably the msq operator). This makes it difficult to assess the generality of their method. However, their techniques were successfully applied [Smi91] to derive the Knuth-Morris-Pratt pattern matcher from a semi-naive program using finite domain constraints. By contrast we can accommodate finite as well as infinite domains provided some satisfiability procedures are provided. Recently Howe and King [HK99] use the results of analysis based on convex polyhedra to reduce the search space in CLP programs over finite domains, thus specialising them. In positive supercompilation [SG98b] driving refers to the constraint propagation from guards in conditionals to their true branch (perfect driving propagates the constraints to both branches of conditionals). Similar ideas were applied for functional logic programming languages by Lafave in [LG97, Laf99]. There the use of linear arithmetic approximations with convex hull was only suggested.

### 3.3 Summary

Specialisation is a program transformation technique for improving program execution. Conceptually, a specialiser can be thought of as comprising three main parts: a preprocessor, an analyser and a program generator. During preprocessing some information from the input program is gathered to aid the analysis. At analysis time the specialiser aims to propagate as much information as possible throughout the whole program. Whether the analysis is made using concrete values or an abstraction of them renders the specialiser as on-line or off-line, respectively. Analyses which produce more than one version, for a program point, are called polyvariant as opposed to monovariant analyses producing only one. Finally, during program generation the results of the analysis phase are translated into code for the residual program. Further processing of this program is common to remove some redundant information.

According to this view of a specialiser its core is the analysis phase. Thus, proving soundness and termination for this phase could ensure that they hold for the specialiser altogether. In [LS91] two conditions (closedness and independence) on a set of atoms are given for soundness of partial evaluation, given a program, and a query. Here we extended the underlying algorithm of an existing on-line specialiser [Gal91] to include constraints; yet the soundness results of partial evaluation can be applied. In particular, the extension consists of replacing atoms by constrained atoms, where different constraint domains could be accommodated. Accordingly, unfolding combines resolution and constraint satisfaction. Generalisation, in turn, is a combination of the upper bound operation used in program analysis (see Chapter 2), the msg operator and widening. Constraints provide a 'record' of the invariants which hold upon generalisation in a similar manner to what is obtained during program analysis. Hence, we obtained a specialiser which may incorporate the constraint domains used for program analysis.

# Chapter 4

## Semantics in CLP

As already noted, language semantics is the starting point of program analysis and specialisation [Nie97]. The current wide spectrum of different program semantics makes it difficult to decide which semantics is appropriate for the task at hand [Cou97, Mos91, Sch86]. Before describing our experiments in detail let us consider some critical points concerning representing semantics in a form suitable for analysis. There are several styles of semantics for imperative languages to be found in textbooks [Sch86, Hen90, NN92, SK95]. These may all be translated more or less directly into declarative programming languages [Har99], but it is necessary to consider carefully the choice of semantics and the style in which the semantics is represented. The choice is influenced by two questions: firstly, what kind of analysis is to be performed on the imperative program, and secondly, how can the complexity of the analysis and transformation be minimised? We chose structural operational semantics as the departure point for analysis and specialisation of imperative programs [MS96]. To reduce the complexity of analysis and specialisation we use a slightly modified form of structural operational semantics which we call one-state small-step operational semantics. Here we show how operational semantics may be represented as CLP programs.

### 4.1 Imperative Program Semantics

Program semantics are mathematical representations to specify the meanings of programs. Also, they are the departure point for any attempt to obtain correct analysis or transformation of programs. Despite this strong dependency between tools for manipulating programs and their semantics, semantics and programs are not expressed in the same formalism, thus leading to misrepresentations (so-called bugs). As a result, several program semantics have been proposed to reduce this gap. Action semantics, evolving algebras, denotational semantics, operational semantics, are examples of program semantics for imperative programs, to name a few. Those semantics are different in the way they reflect program behaviour as well as the program features they abstract and/or highlight. For the purposes of program analysis it may be possible to choose a semantics which better reflects a certain class of program properties, namely those sought by the analysis. Alternatively, a general semantics may be chosen and abstractions on this semantics could lead to the desired semantics. In this chapter we will use operational semantics as the general semantics.

### 4.1.1 Operational Semantics

In operational semantics the meaning of a program construct is specified by the computation it induces when it is executed on a real or abstract machine [NN92]. It is of particular interest how the effect of a computation is produced. Also, operational semantics, and program semantics in general, may be regarded as methods for specifying programming languages or abstract machines. In this regard, the specification method of the operational semantics uses abstract syntax trees, and so-called semantics functions. Abstract syntax trees are a generalisation of parse trees, where ambiguity has been thrown away. Different parse trees may be associated with the same abstract syntax tree but not vice-versa. Semantics functions, in turn, take syntactic objects (e.g. abstract syntax trees) and return their meaning. Next, we describe how these two may be implemented in CLP.

**Abstract syntax trees.** Abstract syntax definitions describe structure. Often these definitions are ambiguous, if used for parsing. However, their main use is not for parsing but to describe at a sufficiently high level the structure of a language.

Usually, semantics definitions use strings as a substitute for abstract syntax trees. Trees are more naturally represented (as nested terms) in CLP than strings. Thus we use nested terms to represent abstract syntax trees instead of their string representation counterpart<sup>1</sup>.

Starting from the standard textual representation of imperative programs the first step is to produce a list of tokens. Next the term representation is produced using a conventional LALR parser [ASU86]. The parser takes for input the tokenised representation of the imperative program and produces as output a term representing the program. For instance, consider the following program fragment representing an if-then-else statement with one assignment in each conditional branch.

```
[if,a,>,b,then,
    x,:=,a,+,2,;,
else,
    y,:=,b,-,1]
```

By parsing the above code we obtain the following abstract syntax tree.

Semantics functions. In order to give meanings to syntactic objects semantics specifications use functions. Semantics functions define mappings from syntactic objects to their meanings, mathematical objects. Those syntactic objects are programs or more generally referred to as expressions. The meaning of an expression depends

<sup>&</sup>lt;sup>1</sup>Though semantics definitions will be expressed using strings where abstract syntax trees are intended.

on the values bound to its variables. For example, if  $\mathbf{x}$  is bound to  $\mathbf{2}$  then the meaning of expression  $\mathbf{x+4}$  is  $\mathbf{6}$ . The context surrounding the execution of an expression could be represented as another function that maps variables to their values, the *state*. Then, semantics functions take as arguments other functions describing the execution context of a given expression, the *environment*. For instance, given semantics functions  $\mathcal{N}, \mathcal{A}$  giving meaning to numerals and arithmetic expressions, respectively, their definitions may look as follows<sup>2</sup> for a state function s.

$$\mathcal{A}[\![n]\!]s = \mathcal{N}[\![n]\!]$$

$$\mathcal{A}[\![x]\!]s = s x$$

$$\mathcal{A}[\![a_1 + a_2]\!]s = \mathcal{A}[\![a_1]\!]s + \mathcal{A}[\![a_2]\!]s$$

$$\mathcal{A}[\![a_1 * a_2]\!]s = \mathcal{A}[\![a_1]\!]s * \mathcal{A}[\![a_2]\!]s$$

$$\mathcal{A}[\![a_1 - a_2]\!]s = \mathcal{A}[\![a_1]\!]s - \mathcal{A}[\![a_2]\!]s$$

In CLP, a semantics function (with the exception of exec, below) is associated with a single predicate definition (or a program with other predicate definitions as required). The above semantics definitions could be described in the following CLP program.

```
a_expr(N,_,Vn) <-
    numeric(N),
    {Vn = N}
a_expr(X,S,Vx) <-
    variable(X),
    lookup(X,S,Vx)
a_expr(plus(A1,A2),S,V) <-
    a_expr(A1,S,V1),
    a_expr(A2,S,V2),
    {V = V1 + V2}
a_expr(times(A1,A2),S,V) <-</pre>
```

<sup>&</sup>lt;sup>2</sup>The brackets [ and ] are commonly used to enclose syntactic arguments to semantics functions.

```
a_expr(A1,S,V1),
a_expr(A2,S,V2),
{V = V1 * V2}
a_expr(minus(A1,A2),S,V) <-
a_expr(A1,S,V1),
a_expr(A2,S,V2),
{V = V1 - V2}</pre>
```

Here the semantics function  $\mathcal{A}$  was represented by predicate  $a_{\texttt{expr}}(A,S,V)$  which holds iff A is an arithmetic expression whose meaning is V in state S. numeric(N) succeeds for N a numeral, and variable(X) succeeds iff X represents a valid variable name. Finally, lookup(X,S,V) holds iff V is the value given by state S to variable name X. Curly brackets are used to denote arithmetic constraints over the rationals or reals, in the syntax of SICStus Prolog. A simple definition for the lookup predicate could be

where the program variable environment is represented as an association list (a list of ground terms denoting the names of the imperative variables and another list representing their contents in logic variables). This is the variable environment structure assumed throughout this chapter, unless otherwise stated (e.g. for locations).

#### Big-Step and Small-Step Semantics

The usual distinction between different kinds of semantics is between the compositional and operational styles. However for our purpose, the most relevant division is between big-step and small-step semantics. Note that operational semantics can be either big-step (natural semantics) or small-step (structural operational semantics).

Let us represent program statements by S and computation states by E. Big-step semantics is modelled using a relation of the form  $bigstep(S, E_1, E_2)$ , which means that the statement S transforms the initial state  $E_1$  to final state  $E_2$ . The effect of each program construct is defined independently; compound statements are modelled by composing together the effects of their parts.

On the other hand, small-step semantics is typically modelled by a transition relation of the form  $smallstep(\langle S_1, E_1 \rangle, \langle S_2, E_2 \rangle)$ . This states that execution of statement  $S_1$  in state  $E_1$  is followed by the execution of statement  $S_2$  in state  $E_2$ . Small-step semantics models a computation as a sequence of computation states, and the effect of a program construct is defined as its effect on the computation state in the context of the program in which it occurs.

Big steps can be derived from small steps by defining a special terminating statement, say halt, and expressing big-step relations as a sequence of small steps.

$$bigstep(S, E_1, E_2) \leftarrow smallstep(\langle S, E_1 \rangle, \langle halt, E_2 \rangle).$$
  
 $bigstep(S, E_1, E_3) \leftarrow smallstep(\langle S, E_1 \rangle, \langle S_1, E_2 \rangle), bigstep(S_1, E_2, E_3).$ 

For the purposes of program analysis, the two styles of semantics have significant differences. Analysis of the *bigstep* relation allows direct comparison of the initial and final states of a computation. As shown above, big steps are derivable from small steps but the analysis becomes more complex. If the purpose of analysis is to derive relationships between initial and final states, then big-step semantics would be recommended, as is the case of reasoning about programs in terms of basic blocks, interprocedurally and so on.

On the other hand, the *smallstep* relation represents directly the relation between one computation state and the next, information which would be awkward (though not impossible) to extract from the big-step semantics. Small-step semantics is more appropriate for analyses [MS96] where local information about states is required, such as the relationship of variables within the same state, or between successive states.

#### One-State and Two-State Semantics

There is another option for representing small-step semantics, which leads to programs significantly simpler to analyse. Replace the  $small step(\langle S_1, E_1 \rangle, \langle S_2, E_2 \rangle)$  relation by a clause  $exec(S_1, E_1) \leftarrow exec(S_2, E_2)$ . Accordingly, predicate exec(S, E) holds if the execution of program P terminates for initial environment E1. Clearly, this is not expressive enough because the observable semantics of an imperative program is assumed to be the relation between the initial and the final states. That is, this relation is not explicitly included in the declarative semantics of the logic program expressed with predicate exec(S, E). The reason is that the operational semantics has been left implicit in the clause meaning. In this respect, the predicate exec(S, E) corresponds to a configuration (see Section 4.2). We also have to add a special terminal statement<sup>3</sup> called halt which is placed at the exit of every program, and a statement  $exec(halt, E) \leftarrow true$ .

We call this style of small-step semantics a *one-state* semantics since the relation *exec* represents only one state, in contrast to *two-state* semantics given by the *bigstep* and *smallstep* relations.

As an aside, the one-state and two-state styles of representation follow a pattern identified by Kowalski in [Kow79], when discussing graph-searching algorithms in logic programming. Kowalski noted that there were two ways to formalise the task of searching for a path from node a to node z in a directed graph. One way is to represent the graph as a set of facts of the form go(X,Y) representing arcs from node X to node Y. A relation path(X,Y) could then be recursively defined on the arcs. The search task is to solve the goal  $\leftarrow path(a,z)$ . Alternatively, an arc could be represented as a clause of form  $go(Y) \leftarrow go(X)$ . In this case, to perform the task of searching for a path from node a to node a, the fact  $go(a) \leftarrow true$  is added to the program, and computation is performed by solving the goal  $\leftarrow go(z)$ . There is no need for a recursively defined path relation, since the underlying logic of the implication relation fulfils the need. Conversely, an arc go(X,Y) could be represented

<sup>&</sup>lt;sup>3</sup>Corresponding to the special configuration denoted by a state (Section 4.2).

as a clause  $go'(X) \leftarrow go'(Y)$  and adding the fact  $go'(z) \leftarrow true$  for the goal  $\leftarrow go'(a)$ . These two descriptions could be related to the forward and backward description of a flowgraph [CE81], respectively

One-state semantics corresponds to the use of the relation go'(X) while two-state semantics corresponds to using the relation go(X,Y). Our experiments show that the one-state semantics is considerably simpler to analyse and specialise than the two-state semantics.

### Analysis of the Semantics

It may be asked whether one-state semantics is expressive enough, since no output state can be computed. That is, given a program P and initial state E, the computation is simulated by running the goal  $\leftarrow exec(P, E)$  and the final state is not observable. This is certainly inadequate if we are using our semantics to simulate the full effect of computations.

However, during program analysis of  $\leftarrow exec(P, E)$  more things are observable than during normal execution of the same goal. In particular we can use a program analysis algorithm that gives information about calls to different program subgoals. In one-state semantics with a relation exec(S, E), the analysis can determine (an approximation of) every instance of exec(S, E) that arises in the computation. We can even derive information about the relation of successive states since the analysis can derive information about instances of a clause  $exec(S_1, E_1) \leftarrow exec(S_2, E_2)$ .

In the specifications to be described below, we start from a structural operational semantics in a textbook style, and derive a one-state small-step semantics.

# 4.2 A Simple Semantics-based Interpreter

As already mentioned a desirable property of structural operational semantics is the way it reflects every change in the computation state. Here we present briefly a way of systematically translating formal operational semantics (adapted from [NN92]) into a constraint logic program.

We shall first provide some elements of the syntax of the imperative language and the metavariables used in the semantics descriptions. Assume we have an imperative language L with assignments, arithmetic expressions, while statements, if-then-else conditionals, empty statement, statement composition, and boolean expressions. Let S be a statement, a be an arithmetic expression, b a boolean expression, e a state function (mapping variables to their value), and x a program variable. All these variables may occur subscripted. A pair  $\langle S, e \rangle$  is a configuration. Also a state on its own is a special terminal configuration. The operational semantics below give meaning to programs by defining an appropriate transition relation holding between configurations.

### 4.2.1 Structural Operational Semantics

Using structural operational semantics [NN92] a transition relation ⇒ defines the relationship between successive configurations. There are different kinds of transition corresponding to different kinds of statement. Accordingly, the meaning of empty statement, assignment statement, if-then-else statement, while-do statement and composition of statements are:

$$\langle skip, e \rangle \Rightarrow e$$
 (4.1)

$$\langle x := a, e \rangle \Rightarrow e[x \mapsto \mathcal{A}[a]e]$$
 (4.2)

$$\langle if \ b \ then \ S_1 \ else \ S_2, e \rangle \ \Rightarrow \ \langle S_1, e \rangle \ if \ \mathcal{B}[\![b]\!]e = \mathbf{tt}$$
 (4.3)

$$\langle if \ b \ then \ S_1 \ else \ S_2, e \rangle \ \Rightarrow \ \langle S_2, e \rangle \ if \ \mathcal{B}[\![b]\!]e = \mathbf{ff}$$
 (4.4)

$$\langle while \ b \ do \ S, e \rangle \Rightarrow \langle (S; while \ b \ do \ S), e \rangle \text{ if } \mathcal{B} \llbracket b \rrbracket e = \mathbf{tt}$$
 (4.5)

$$\langle while \ b \ do \ S, e \rangle \Rightarrow e \ if \ \mathcal{B}[\![b]\!]e = \mathbf{ff}$$
 (4.6)

$$\langle S_1; S_2, e \rangle \Rightarrow \langle S_1'; S_2, e' \rangle \text{ if } \langle S_1, e \rangle \Rightarrow \langle S_1', e' \rangle$$
 (4.7)

$$\langle S_1; S_2, e \rangle \Rightarrow \langle S_2, e' \rangle \text{ if } \langle S_1, e \rangle \Rightarrow e'$$
 (4.8)

where  $\mathcal{A}$  is the semantics function for arithmetic expressions and  $\mathcal{B}$  is the semantics function for boolean expressions<sup>4</sup>. Intuitively, the assignment axiom schema above says that in a state e, x := a is executed to yield a state  $e[x \mapsto \mathcal{A}[a]e]$  which is as e except that x has the value  $\mathcal{A}[a]e$ , for the original e. Moreover, transition 4.7 expresses the fact that if  $S_1$  is not a primitive statement of the language then execution won't proceed to  $S_2$  until the rest of  $S_1$ ,  $S'_1$ , has been fully executed. Transition 4.8 considers the case when execution of  $S_1$  has been completed thus yielding state e', hence execution of  $S_2$  starts from this new state.

We may specialise transitions 4.7 and 4.8 by unfolding their conditions with respect to transitions 4.1, 4.2, 4.3, 4.4, 4.5 and 4.6 above to transitions 4.10, 4.11, 4.12, 4.13, 4.14, and 4.15 below. Transition 4.9 below is obtained by unfolding the condition of transition 4.7 with respect to itself and applying the associativity property of the ';' operator. We assume that cases 4.2, 4.3, 4.4, 4.5 and 4.6 do not occur since all programs are terminated by *halt*. Hence the new semantics:

$$\langle halt, e \rangle \Rightarrow e$$

$$\langle (S_1; S_2); S_3, e \rangle \Rightarrow \langle S_1; (S_2; S_3), e \rangle$$

$$(4.9)$$

$$\langle skip; S_2, e \rangle \Rightarrow \langle S_2, e \rangle$$
 (4.10)

$$\langle x := a; S_2, e \rangle \Rightarrow \langle S_2, e[x \mapsto \mathcal{A}[\![a]\!]e] \rangle$$
 (4.11)

$$\langle (if \ b \ then \ S_1 \ else \ S_2); S_3, e \rangle \Rightarrow \langle S_1; S_3, e \rangle \text{ if } \mathcal{B}[\![b]\!]e = \mathbf{tt}$$
 (4.12)

$$\langle (if \ b \ then \ S_1 \ else \ S_2); S_3, e \rangle \Rightarrow \langle S_2; S_3, e \rangle \text{ if } \mathcal{B}[\![b]\!]e = \mathbf{ff}$$
 (4.13)

$$\langle (\textit{while b do S}); S_2, e \rangle \ \Rightarrow \ \langle (S; \textit{while b do S}); S_2, e \rangle \text{ if } \mathcal{B}[\![b]\!]e = \mathbf{tt}(4.14)$$

$$\langle (while \ b \ do \ S); S_2, e \rangle \Rightarrow \langle S_2, e \rangle \text{ if } \mathcal{B}[\![b]\!]e = \mathbf{ff}$$
 (4.15)

Next, we express the semantics as a constraint logic program below. It is worth noting that this representation aids the analysis and specialisation phases by carrying a single environment instead of double environment (i.e, the one-state small-step semantics).

<sup>&</sup>lt;sup>4</sup>The semantics definitions for boolean expressions, and their associated CLP program may be constructed in a similar way as the function for arithmetic expressions.

```
exec(halt,E) <-
exec(compose(compose(S1,S2),S3),E) <-
       exec(compose(S1,compose(S2,S3)),E)
exec(compose(skip,S2),E) <-
       exec(S2,E)
exec(compose(assign(X,Ae),S2),E) <-</pre>
       a_expr(E, Ae, V),
       update(E,E2,X,V),
       exec(S2,E2)
exec(compose(ifte(B,S1,S2),S3),E) <-</pre>
       b_expr(E,B,true),
       exec(compose(S1,S3),E)
exec(compose(ifte(B,S1,S2),S3),E) <-</pre>
       b_expr(E,B,false),
       exec(compose(S2,S3),E)
exec(compose(while(B,S1),S2),E) <-</pre>
       b_expr(E,B,true),
       exec(compose(compose(S1, while(B,S1)),S2),E)
exec(compose(while(B,S1),S2),E) <-
       b_expr(E,B,false),
       exec(S2,E)
```

In this program the term assign(X,A) represents an assignment statement X := A, and the term compose(S1,S2) represents the composition of statement S1 with statement S2. The predicate exec(P,E) holds iff program P can be completely executed from state E (see Section 5.1.1). The predicate update(E1,E2,X,V) models the change of state from E1 to E2 induced by the assignment of V to X. The value of the assigned variable X is changed to V but all other values of variables are passed unchanged from E1 to E2. Observe that the clause for composition of statements in this implementation has been specialised to the individual cases of our program constructs,

namely assign, skip, ifte and while in this example. Therefore, we do not use the clauses corresponding to composition of statements in their general form. By construction of the abstract syntax tree and by inspection of the clauses above it is correct to remove the clause expressing the right associativity of the compose operator. As a result, the semantics definition of programming languages in the style of one-state small-step operational semantics has a specific form. Specifically, the form of the left-hand side of semantics definitions follows the scheme

$$\mathcal{Z}_L \llbracket command; S \rrbracket \overline{v} \Rightarrow \dots$$

where semantics function  $\mathcal{Z}$  assign meanings to programs in language L. command is an instance of any program construct in L. Composition of statements is denoted as ';', and the functions used as environment are  $\overline{v}$ . Accordingly, the associated CLP program reflects this form in clauses of the following form.

These are tail recursive clauses.

Up to now we have defined a CLP program that represents the one-state small-step operational semantics of a single procedure language with arithmetic expressions, boolean expressions, assignments, if-then-else conditionals and while loops. Though it has not been explicitly stated variables take their values from numeric domains (e.g. rationals or reals). Next, several extensions to this language are described. The extensions may be classified into two main groups: procedures and data structures. The group of procedures encompasses extensions such as adding procedures, adding parameters, and adding a combination of both, and local variables with block declarations. Different scopes for parameters and procedures are explained. In the group of data structures we will explain how to add other data types to the language, such as arrays, and pointers.

### 4.2.2 Procedures

Operational semantics provides several new environment functions which allow the description of recursive procedures with parameters, with different scopes for their definition. Locations and scope for variables and procedures are some of the new concepts introduced, in order to be able to give meanings to programs with procedures and parameters. We will not clutter with notation the existing semantics definitions above. Instead we will describe the CLP programs that implement the new features of the language, and refer the interested reader to [NN92] for a discussion of some of the concepts here described.

#### Locations

A structure used by operational semantics as well as denotational semantics [Sch86] to split the state is called *locations*. A simple modelling of the state is to have a mapping from variables to the value they are bound to. Consider, for instance, the names of program variables x,y,z, and their values 0.4,7,-1. This state may be described in a CLP term by a list of two lists as [[x,y,z],[0.4,7,-1]]. Using locations the mapping is decomposed into two mappings, one from variable names to their locations and other from locations to their content. Hence, the previous state with three variables becomes [[(x,1),(y,2),(z,3)],[(1,0.4),(2,7),(3,-1)]], where the list of variables was replaced by list of pairs variable-location, and the list of values by a list of pairs location-value. Such representation resembles the way program variables are commonly implemented in compiler design, where a name is associated with an address which contains the value bound to that name<sup>5</sup>.

### Scopes

A *scope* is associated with a name (either a variable or a procedure/function) to determine the places within a program where it may be referenced. The scope of a variable or procedure may be either *dynamic* or *static*. Current programming languages

<sup>&</sup>lt;sup>5</sup>Pointers introduce a new meaning of what an address may contain.

support complex combinations of these scopes for their variables and procedures. Nonetheless, this simple classification is useful to illustrate the concept. Consider the following program.

```
0)
    begin
      var x := 0;
1)
2)
      proc p { x := x*2 };
      proc q { call p };
3)
4)
        begin
           var x := 5;
5)
6)
           proc p { x := x+1 };
7)
           call q;
8)
           y := x
9)
         end
10) end
```

Execution of this program under  $dynamic\ scope$  for variables and procedures binds variable y to 6. The reason is that call q will use the local procedure p (line 6) which will update the local variable x (line 5). Alternatively, if the program is executed under dynamic scope for variables and static scope for procedures y is bound to 10. Now call q will use the global procedure p (line 2) which will update the local variable x (line 5). However, if static scope is used for variables and procedures, then y gets the value 5. This time procedure q calls global procedure p (line 2) whose execution updates the global variable x (line 1) and the local variable x (line 5) retains its value. Finally, static scope for variables and dynamic scope for procedures binds y to 6 because procedure q calls local procedure p (line 6) which updates x in line 5.

Note that to resolve a procedure call there must be a function that provides the connection between procedure names and their associated code and execution environment<sup>6</sup>. For updates or lookups of program variables names are related to

<sup>&</sup>lt;sup>6</sup>In the simplest case procedure names are associated with their code and the execution environment is given elsewhere.

their contents by a function called *state*. For procedures, the function that provides the context for resolving a procedure call is a *procedure environment*. The state and the procedure environment provide some of the *program execution environment* used by semantics functions to assign meanings to programs.

The way that the procedure environment is built and accessed determines the scope used for procedure names. A similar reasoning applies to the variable environment (or state). For this example, the ordering in which variable and procedure declarations appear, in a top-down traversal/reading of the program's text, determines their context.

During execution, the program environment is either built or looked up. For instance, when a name (a variable or a procedure) definition is met during execution, the current program environment changes to one where this new definition is visible, and thus their contents may be accessed. Alternatively, if the program environment already contains descriptions of the names visible (procedures for instance) from different points in the program, then name declarations are ignored during execution. Depending on the scope used for names, parts of the program environment may change, after a procedure call, then restored upon exit from to the called procedure. That is, the names visible are different to those visible before the procedure call and visibility is restored upon exit from the called procedure. Under dynamic scope for names, a procedure call leaves the environment unchanged whereas with static scope for names the environment changes. Under dynamic scope for procedures a procedure call is resolved by inlining the code associated with the procedure called (similar to a macro-expansion) and then resuming execution where the code was inlined. Static scope for procedures, on the other hand, amounts to updating the procedure environment to that where the called procedure is defined. The implementation in CLP of static scope for procedures gives the following definition.

```
update_pe(E2,E3,NPE1,OPE),
exec(S,E3)
```

In addition, the definitions for the environment handling predicates lookup\_p and update\_pe must be provided. The intended meaning of lookup\_p(E,Pn,C,NPE) is that given execution environment E procedure name Pn has body C and the new procedure environment, reflecting the names visible from procedure Pn, is NPE1. The predicate update\_pe(E,E1,OPE,NPE) holds iff E1 is the same execution environment as E with the exception that the procedure environment is NPE1 and was OPE. Note that the fourth subgoal in this clause update\_pe(E2,E3,NPE1,OPE) would normally restore the old procedure environment OPE upon exit from call to procedure Pname thus updating state E2. However, E2 is not 'returned' from the procedure call and as a result the update is on an empty (a fresh logic variable) execution environment, which is clearly not the intended meaning of executing a procedure call. Moreover, the above clause is not in the style of one-state small-step operational semantics discussed in Section 4.2.1, since we need access to the terminal state (E2) after executing Code, which is why bigstep was used there.

An extra function in the semantics definitions may store the information necessary to resume execution (last two subgoals) appropriately, after procedure exit. Accordingly, the predicate exec is modified to take another argument that represents this new function. Hence, the semantics definition of a procedure call may be written in CLP as follows.

```
exec(compose(call(Pname),S),E,St) <-
    lookup_p(E,Pname,Code,NPE1),
    update_pe(E,E1,OPE,NPE1),
    push_pop(S,OPE,St,St1),
    exec(compose(Code,pop_fr),E1,St1)</pre>
```

Now, this definition complies with the form of one-step small-step operational semantics. Observe that a new subgoal (push\_pop), a 'new' command (pop\_fr) in the imperative language, and a new argument (St) to the semantics predicate were

added. The new argument represents an extra function to the semantics function. Its use resembles the stack of activation records in conventional program execution because it stores the code, where execution would resume upon exit from the called procedure, and the variables which must be restored upon exit from a procedure call. Then, the definition for predicate push\_pop may be

```
push_pop(S,PE,St,fr(PE,S,St)) <-</pre>
```

That is, a new frame (fr(PE,S,\_)) is pushed onto the current stack, St. Finally, the instruction pop\_fr serves as a special marker to distinguish between program termination and procedure exit<sup>7</sup>. Moreover, a new program point should be added to this special marker. When the language contains a return instruction the program point is added at parsing time, when that is not the case, the instruction as well as the program point should be added appropriately prior to specialisation. The CLP definition for this pseudo-instruction<sup>8</sup> is as follows.

```
exec(pop_fr,E,St1) <-
    push_pop(S,OPE,St,St1),
    update_pe(E,E1,NPE1,OPE),
    exec(S,E1,St)</pre>
```

Observe that the last subgoal, exec(compose(Code,pop\_fr),E1,St1) of the definition for procedure call above cannot be unfolded using this clause, since they do not unify<sup>9</sup>.

Additionally, the initial contents of the stack of activation records is empty\_st. Accordingly, the clause for the base case of the definition of predicate exec is

```
exec(halt, E, empty_st) <-
```

assuming that the program is started with an empty stack, empty\_st.

<sup>&</sup>lt;sup>7</sup>In case the language has a return instruction the semantics definition is similar, with the difference that no new statement is added to Code.

<sup>&</sup>lt;sup>8</sup>It never occurs in user programs.

<sup>&</sup>lt;sup>9</sup>Even when the called procedure's body is empty, Code=compose(skip,pop\_fr).

### 4.2.3 Other Data Types

There are several modifications necessary to extend the semantics-based CLP interpreter with other data types, such as arrays, records and pointers. The abstract syntax is extended so that the new data types are given a distinctive representation, although this is not necessarily the case in the concrete syntax. The semantics functions are modified accordingly.

**Arrays and records.** Consider the following assignment statement referencing two elements of a unidimensional array.

```
i := a[i] - a[i+2]
```

The associated abstract syntax tree in CLP is the term

```
assign(j, minus(array(a,index(i)),array(a,index(plus(i,2)))))
```

Clearly there would be two changes to the semantics-based CLP interpreter, an extension in the definition for the semantics function for arithmetic expressions and modification of the state manipulation predicates (e.g. update and lookup). Several clauses would be added to the CLP program interpreting arithmetic expressions, i.e. new semantics definitions.

```
a_expr(array(A,I),E,Ae) <-
        a_expr(I,E,Iv),
        lookup_a(E,A,Iv,Ae)
a_expr(index(I),E,Iv) <-
        a_expr(I,E,Iv)

exec(compose(assign(X,Ae),S),E,St) <-
        a_expr(Ae,E,Ve),
        update_s(E,E1,X,Ve)
        exec(S,E1,St)</pre>
```

Here we have added the two new cases for the  $\mathcal{A}$  semantics function. Also, the definition for assignments was modified to cater for the two possible cases arising in the left-hand side of the assignment, and finally some auxiliary predicates were introduced. Observe that the extension to multidimensional arrays could be easily handled by nesting the function symbol index (making it a two argument function symbol) as necessary, to denote the indexing. Also, a new semantics function (a\_expr\_array) is required to evaluate nested index<sup>10</sup> function symbols, as follows.

The clauses for predicates update\_s/5 and a\_expr(array(A,I),E,Ae) are redefined with respect to unidimensional arrays above. These same techniques for implementing arrays map over to represent records (i.e. symbolically named fields).

<sup>&</sup>lt;sup>10</sup>For instance, index(I,index(J,last)) denotes the array element in cell [I,J].

Pointers. Locations open the possibility of describing the semantics of programs with pointers and/or procedures with parameters 11. A limited form of pointers is implicit in languages with procedures and parameters by reference. For the programmer to get the full expressiveness of pointers the language makes them explicit commonly providing two operators for their use (e.g. '\*' for points-to and '&' for indirection, in C [KR88]). Recall that locations split the definition of the function for the program state into two functions, one which maps variable names to their location, and another one which maps locations to their contents, variable environment and store respectively. As a result, two variable names may have the same location and thus one points to the other one. In procedures with parameters by reference this is the case, the corresponding actual parameters and the formal parameters, in the procedure call and procedure definition, are made to share the same location. In CLP this may be expressed with little modifications to the clause already provided for procedure calls without parameters, as follows.

```
exec(compose(call(Pname, Actuals), S), E, St) <-
        lookup_p(E,prc(Pname, Actuals), Code, NPE1),
        update_pe(E,E1,OPE,NPE1),
        push_pop(S,OPE,St,St1),
        exec(compose(Code,pop_fr),E1,St1)</pre>
```

Now, the call is made with an extra argument, Actuals representing the actual parameters of the procedure. Moreover, lookup\_p is extended to carry this new argument as well, and thus produce information of the new procedure environment into NPE1, which will be reflected as a new execution environment E1 by update\_pe. Then some information about the old environment is pushed onto the stack of activation records, St. Procedure exit is similar to what we already described (Section 4.2.2).

When pointers are explicit in the language, i.e. indirection '&' and points-to '\*' operators are provided, the semantics definitions must be extended. Firstly, variables which are of pointer type are so tagged in the variable environment. Secondly, the

<sup>&</sup>lt;sup>11</sup>A language with procedures and parameters by value without pointers does not require locations.

abstract syntax is extended to reflect the association of operators '\*' and '&' to variable names in a similar way as in [Rep98] for program analysis. Finally, the semantics functions are extended accordingly. Consequently, for variable name p, the abstract syntax trees of &p and \*p are address\_of(p) and points\_to(p) respectively. The new semantics-based CLP clauses extend and replace the definition of different predicates. On the one hand, some definitions for arithmetic expressions change. Next, the first two clauses extend the definition of a\_expr (semantics function  $\mathcal A$  respectively) and the last two modify the definition for the meaning of a variable name as an arithmetic expression.

On the other hand, the meaning of an assignment (predicate exec) to a pointer is modified. Two cases must be considered for the definition of an assignment and the update of a variable of pointer type as shown below.

```
exec(compose(assign(points_to(P),Ae),S),E,St) <-
    a_expr(Ae,E,V),
    lookup_l(P,E,L),
    lookup_st(L,E,L1),</pre>
```

Notice the use of two mappings (those predicates with suffix \_1 for locations and \_st for store) to access the contents of a normal variable, as opposed to one mapping in programs without locations. Also, a new concept was introduced, the assignment between pointers, update\_pt for a variable of pointer type (the pointer variable in the right-hand side of the assignment and the pointer variable in the left-hand side are made to share the same location, i.e. they point to the same value).

### 4.2.4 Block declarations

Extending the language with procedures with parameters to handle block declarations is straightforward. Block declarations, block(Dv,Code), consist of a set of variable declarations, Dv, and code, Code, where these variables (so-called locals) are used in conjunction with the remaining visible variables from the context where they appear. Their semantics definition is very close to the definition of procedures with parameters since some information is pushed onto the stack argument related to which variables should be restored and what is the next program statement to be executed upon exit from the block. The list of variable declarations works very much the same

4.3. SUMMARY 87

as the formal parameters in a procedure and their initial value might be that of a variable external to the block thus seen as an actual parameter in the comparison with procedures. Hence the CLP program associated with the semantics of blocks is as follows.

```
exec(end_block,E,St1) <-
     push_pop(S,OE,St,St1),
     restore_e(E,E1,OE),
     exec(S,E1,St)

exec(compose(block(Dv,Code),S),E,St) <-
     update_e(E,E1,Dv),
     push_pop(S,E,St,St1),
     exec(compose(Code,end_block),E1,St1)</pre>
```

This time there is no change in the scope of procedures hence the predicates update\_e(E,E1,Dv) and restore\_e(E,E1,OE). The first one updates the execution environment according to Dv thus yielding new execution environment E. The second one restores the visible variables and values from E to E1 according to the old execution environment OE. The instruction end\_block is introduced in a similar way as the new instruction marking the end of a procedure, to denote the end of the current scope.

# 4.3 Summary

Program semantics specify the meanings of programs though mathematical representations. They are used as a basis for any attempt to obtain correct analysis and transformation tools. In this chapter we have shown how to code semantics as constraint logic programs (CLP programs) in order to provide a common representation for executable language semantics useful for analysis and specialisation.

We used a special representation of structural operational semantics which we call one-state small-step semantics. The latter is obtained from the former by specialising the definitions for composing statements to the different instructions provided

by the programming language. As a result we obtained semantics definitions in the style of small-step semantics where the definition of composition of statements has been specialised to the composition of the particular language instructions with any trailing program statement. We showed how this specialised semantics could be implemented as a CLP program, where configurations are modelled with a predicate and the transitions are given by the logic of the CLP program.

Here, we provided the semantics definitions for a small imperative programming language resembling Pascal. The language contains programs with while-loops, if-then-else conditionals, assignment statements and simple arithmetic and boolean expressions. This is expressive enough for single procedure programming; however most interesting programs are not commonly expressed in this small language thus we described some extensions to the language, namely procedures with parameters and complex data structures. Operational semantics may cope with this extensions by introducing other auxiliary functions and concepts to cover the semantics of a bigger programming language. Locations is one of these concepts and allows the introduction of procedures with parameters by reference and pointers. Also, different scopes (static and dynamic) for variables and procedures may be described by appropriately updating the program execution environment to reflect the set of names (variables and procedures) available for access at any one point during program execution. In order to handle procedures with parameters and yet remain in the style of one-state small-step operational semantics the concept of a stack was introduced in the program semantics function. As a result the predicate associated with this program execution function takes an extra argument denoting this new function. In compiler jargon we could refer to this new function as the stack of activation records since that is the use made of this new argument. A similar description of operational semantics appears in [JM82].

Finally, the details of extending this language with arrays, records and pointers is discussed. It is important to note the relevance of the abstract syntax here, since it provides the necessary information needed to identify some of the occurrences of these new types within a program.

# Chapter 5

# Relating CLP to Imperative Programs

Constraint logic programs appear to be a suitable vehicle for imperative language semantics implementation. When written carefully [Jon96], the semantics can be regarded as an *interpreter* for the imperative language<sup>1</sup>. Specialisation of the interpreter with respect to an imperative program yields an equivalent declarative program. Moreover, if some input to the imperative program is provided, specialisation will produce a specialised CLP program, as opposed to simply 'compiling' imperative programs to CLP. By so doing we open up the possibility of applying well-developed techniques for analysis and transformation of constraint logic programs to imperative programs as well. Nevertheless, it is not clear how to relate the results of such analysis and/or transformation back to an imperative program. In this chapter we show how to modify the specialisers presented (Chapter 3) in order to allow the systematic mapping of the results of analysis of CLP programs to the source imperative program, on the one hand, and the reconstruction of specialised imperative programs from residual CLP programs, on the other hand.

<sup>&</sup>lt;sup>1</sup>Appropriate techniques for implementing semantics are discussed in Chapter 4.

# 5.1 Specialisation using Program Points

As we already noted in Section 3.1 a specialiser may be used to compile programs from one language to another language, when coupled with the appropriate interpreter (see Chapter 4). This may be useful for transferring results of program analysis between programs in the two languages involved, where no input constraints to the imperative program are provided (i.e. analysis of imperative programs by analysis of CLP programs [PGS98]). When some input constraints to the imperative program are available, specialising the semantics-based interpreter with respect to such an imperative program and input constraints results in a specialised interpreter and possibly a specialised imperative program, represented as a CLP program (i.e. imperative program specialisation by specialisation of CLP programs [PG99]). In this case, not only analysis but also specialisation results could be transferred to imperative programs. The analysis results would be with respect to some input constraints to the program, as opposed to the more common approach to program analysis where no input is provided. Yet, some specialisation at the imperative level could be achieved when no input to the imperative program is provided. However, we will not consider this case<sup>2</sup> for transferring results of specialisation of imperative programs by specialisation of CLP programs, since this is more the subject of dedicated compilers<sup>3</sup> [BGS94] and would be difficult to achieve with a more general tool (e.g. by specialising interpreters). Hence, if the aim of specialising a semantics-based interpreter is imperative program specialisation some input constraints are assumed and if the aim is program analysis the input constraints may or may not be present.

In summary, (see Fig. 1.1) our specialiser is applied to our semantics-based interpreter and some input (an imperative program with/without input constraints). The result is a residual CLP program which is a version of the semantics interpreter specialised with respect to that input. Then, such a residual program could be analysed, and the analysis results related to the original imperative program, and/or

<sup>&</sup>lt;sup>2</sup>Unless the input is specified as assignments within the program.

<sup>&</sup>lt;sup>3</sup>Other tools for generating efficient compilers from semantics definitions exist [Die96, Pet98].

the residual program could be used for reconstructing a residual imperative program.

In order to obtain a correspondence between the imperative program and its corresponding CLP program some tuning of the specialiser is needed. Otherwise, the specialiser may remove important information needed to relate imperative statements and variables with their declarative counterpart. Such tuning involves selecting among the predicates of the semantics-based interpreter those we want to be defined in the residual program. Hence, we choose predicates from the semantics-based interpreter that relate directly to the meaning of the statements in the imperative program to be specialised. As a result we get at least one predicate for each statement of the imperative program, thus highlighting the correspondence between imperative statements and predicates in the residual program.

The role of specialising a semantics-based interpreter is two-fold. One is to increase the efficiency and precision of the analysis of CLP programs<sup>4</sup>. The other is a combination of two somewhat conflicting functions. Firstly it should yield a residual program which reflects the structure of the original imperative program; simultaneously it should propagate information through the successive environments/states generated by the semantics, so as to obtain the source code specialisation, thus changing the original structure.

Analysis and specialisation of imperative programs commonly associate assertions with program points. By contrast, our analyser for constraint logic programs produces a separate analysis per program predicate and the specialiser only associates assertions with some predicates in the program (a set of constrained atoms for which a definition will be given in the residual program). This suggests having at least one predicate in the residual program for each program point in the imperative program.

Also, some special purpose control over specialisation is required, in order to produce the program definitions for the desired predicates. This will be described below, but first we need to describe the data structure which is used to represent program points within imperative programs.

<sup>&</sup>lt;sup>4</sup>Rather than analysing interpreters with respect to a goal, the interpretation layer is reduced by specialisation with respect to the goal.

### 5.1.1 Representation of Program Points

Starting from the abstract syntax of the imperative language the first step is to make program points explicit in the program. As the next step in producing the desired specialisation we add a unique label to each statement of the input imperative program. These labels are used in the control of specialisation to keep track of the structure of the imperative program. They appear as an extra argument in the syntactic structures denoting statements<sup>5</sup>. For instance, consider the following abstract syntax tree representing an if-then-else statement with two assignments and one assignment in each conditional branch, respectively.

When decorated with program points we get the following statement.

We use the function symbol p to enclose a program point. The first argument of p is a program point label and the second its associated program statement.

Accordingly, the semantics-based interpreter is modified to handle the newly extended representation of statements. For example, instead of

```
exec(compose(assign(X,Ae),S),E) <-
    a_expr(E,Ae,V),</pre>
```

<sup>&</sup>lt;sup>5</sup>Only imperative instructions are labelled, compose is not considered as an instruction of the language.

```
update(E,E2,X,V),
    exec(S,E2)

we write

exec(compose(p(La,assign(X,Ae)),S),E) <-
    a_expr(E,Ae,V),
    update(E,E2,X,V),
    exec(S,E2)</pre>
```

As another example of how program points are made explicit in the semantics consider the semantics definitions for the while statement.

These are replaced by the following two clauses.

```
exec(compose(p(L,while(B,S1)),S),E) <-
        b_expr(E,B,true),
        exec(compose(S1,compose(p(L,while(B,S1)),S2)),E)
exec(compose(p(L,while(B,S1)),S),E) <-
        b_expr(E,B,false),
        exec(S,E)</pre>
```

A similar modification is made for the remaining semantics clauses defining a program statement and its program point.

### Intended meaning of exec(St,Env)

Previously (Section 4.1.1) we argued about the benefits of representing a transformed form of structural operational semantics as CLP programs in a style we called one-

state small-step operational semantics. With this style of writing the semantics the relation between consecutive states in the semantics was implicit in the clauses' meaning. There was no predicate associated with the semantics function for programs. The only predicate which could be related to this function was that denoting a configuration, namely exec(St,Env). Accordingly, the meaning of this predicate is that program St terminates starting from execution environment Env. We assume that the observable semantics of an imperative program is the relation between the initial and final states in any computation. However this relation is not captured in the declarative meaning of the logic program using predicate exec.

As an aid in proving that this relation can be captured by our one-state small-step semantics it suffices to add an extra argument to the predicate exec. This new argument denotes the 'final state', hence the new predicate exec' (St,Env,Env\_final). For all clauses, but one, of predicate exec' we simply copy the 'final state' argument from head to tail, as the following clause schema shows.

```
exec'(St,Env_initial,Env_final) <-
...,
...,
exec'(St1,Env_int,Env_final)</pre>
```

The final state is bound when the following clause is used.

```
exec'(halt,Env,Env) <-
```

The intended imperative semantics is that the program P maps initial execution environment Env\_initial to final state Env\_final exactly when the following clause is derivable from the semantics-based CLP program.

```
exec'(P,Env_initial,Env_final) <-</pre>
```

Thus the input-output state relation of the imperative program is included in the declarative semantics of the semantics-based CLP program, and hence the correctness of specialisation and other transformations ensures that the imperative program semantics is preserved during specialisation. However, for most practical analysis and

specialisation applications the relation between the initial and the final states is irrelevant. Thus the use of predicate exec instead of predicate exec' is justified. This prevents constraints linking Env\_initial and Env\_final from being made explicit, which would increase the complexity of the specialised programs.

### 5.1.2 Control of Specialisation

Standard partial evaluation of logic programs [LS91], given a query and a program, seeks a set of atoms for which two properties hold, closedness and independence. In Chapter 3 we extended these results by considering constrained atoms instead of single atoms and the aim remains the same: to seek a set of constrained atoms which is closed<sup>6</sup>. To achieve this goal (closedness) our specialisers distinguish two levels of control:

- the *global control*, in which one decides which constrained atoms will be specialised, and
- the *local control*, in which one constructs the finite (possibly incomplete) computation trees for each individual constrained atom in the set and thus determines what the definitions of the specialised constrained atoms look like.

The global control determines whether the set of atoms contains more than one version of each predicate (polyvariant control) or whether only one version of every predicate is kept (monovariant control). We require polyvariant control since each program point should result in at least a distinct predicate in the residual program; for instance, at least one version of the semantics transition dealing with assignment statements should be produced for each assignment in the input program.

In the algorithm (see Fig. 3.2) we use for specialisation we define the unfolding rule Cr and the global control operation **generalise** to cater for program points. Our CLP specialisers use trace terms [GL96] as an aid in controlling the polyvariance and keeping the set of constrained atoms finite at the global control level (Section 3.2.3).

<sup>&</sup>lt;sup>6</sup>Independence of the atoms is guaranteed by predicate renaming.

Accordingly, global control based on trace terms is modified to include the program point labels. The unfolding rule, Cr, remains determinate with shower shape<sup>7</sup> stopping specialisation when a subgoal containing a program point is met. This is achieved by detecting when a subgoal has the predicate associated with the semantics definition of an imperative construct and a program point label.

Notice that when specialising imperative programs by specialising CLP programs there is only one specialiser, namely the specialiser for CLP programs. The specialiser for imperative programs is a particular 'rendering' of the result of specialising semantics-based interpreters. Yet we will talk about specialisation of CLP and specialisation of imperative programs bearing in mind their relationship. Moreover, the polyvariance of the imperative program specialiser depends on the polyvariance of the CLP specialiser. However, a polyvariant CLP specialiser does not necessarily yield a polyvariant specialiser for imperative programs. Any decision affecting the polyvariance of the specialised CLP programs may affect the polyvariance of the specialised imperative programs recovered from them. Hence, we may say that polyvariance is given at two levels: at the CLP level and at the imperative level, depending which specialiser we are talking about, the CLP specialiser or the imperative programs specialiser. To avoid confusion we will refer to CLP polyvariance as the polyvariance obtained at the CLP level, and imperative polyvariance to the polyvariance obtained at the imperative level. We will see that to obtain a monovariant imperative specialiser we need some polyvariance of the CLP specialiser and a simple modification of this CLP specialiser results in a polyvariant imperative specialiser.

## 5.1.3 Monovariant Imperative Program Specialisation

In specialisation with trace terms [GL96] atoms having the same trace term are merged using the most specific generalisation operator [Plo70, Rey70], msg. In this sense, trace terms are an abstraction of the information contained in the atoms they represent. When replacing atoms for constrained atoms (Section 3.2.1) trace terms

<sup>&</sup>lt;sup>7</sup>Only the root of the tree constructed is allowed to have a branching structure.

may still be used as abstraction, however, the msg operator is replaced by an operation,  $\stackrel{\nabla}{\sqcup}$ , which combines upper bound  $\sqcup$  and widening  $\nabla$  for the appropriate constraint domain and the msg operation. If instead of trace terms we abstract some constrained atoms by their program point label we obtain polyvariant CLP specialisation with respect to program points and the effect at the imperative level is monovariant imperative program specialisation. That is, each program point has a distinct predicate in the residual program provided every program point occurring in the source imperative program has a unique label. For instance, one version of the semantics transition dealing with assignment statements should be produced for each assignment in the input program.

The global control rule (**generalise** operator in algorithm of Fig. 3.2) is made to follow the algorithm shown in Figure 5.1, where  $\stackrel{\nabla}{\sqcup}$  is an operation using the upper bound and widening defined in Chapter 2 for the constraint domain chosen.

This operation,  $\stackrel{\nabla}{\sqcup}$ , is similar to an approximation step in the analysis of CLP programs (Equations 2.1 and 2.2) where the clauses of the program  $P_i$  are obtained from the constrained atoms using an appropriate renaming of the predicates according to the partition they belong to. That is, given a constrained atom  $\langle p(\overline{t}), \mathcal{C} \rangle$  in partition  $R_i$ , its associated clause for approximation is as follows.

$$p_i(\overline{t}) \leftarrow \mathcal{C}$$

At any one time during specialisation there is only one such clause per partition, and the incoming constrained atom is used for computing the upper bound and the result of this is widened with the incoming constrained atom. For instance, if the incoming constrained atom is  $CA_2 = \langle A_2, c_2 \rangle$  and the partition is described by clause  $A_1 \leftarrow c_1$ , namely constrained atom  $CA_1 = \langle A_1, c_1 \rangle$ , then first compute the msg of  $A_1$  and  $A_2$ , say  $A_3$ , and find the appropriate aliasing information between  $A_1$  and  $A_3$  and between  $A_2$  and  $A_3$ . Next apply the substitutions, say  $\rho_1$  and  $\rho_2$  respectively, to  $c_1$ , and  $c_2$  resulting in constraints  $c'_1$  and  $c'_2$ . Then the final result is constrained atom,  $CA_3 = \langle A_3, c'_2 \nabla (c'_2 \sqcup c'_1) \rangle$ , or clause  $A_3 \leftarrow c'_2 \nabla (c'_2 \sqcup c'_1)$ . Notice that the underlying algorithm behind this description was given in Section 3.2.3.

- 1. Let R be a finite set of constrained atoms.
  - Let  $R_{label}$  be the set of constrained atoms in R whose atom component contain an argument of the form p(N,Stm), where Stm is one of the program constructs of the imperative language, (e.g. assign, ifte or while) and N is a program point label.
  - Let  $R_{traceterm}$  be the remaining atoms in R.
- 2. Let  $\{R_{N_1}, \ldots, R_{N_k}\}$  be the finite partition of  $R_{label}$  where all constrained atoms in  $R_{N_i}$  contain the argument p(Nj,Stm).
- 3. Let  $\{R_{c_1}, \ldots, R_{c_m}\}$  be the finite partition of  $R_{traceterm}$  where  $R_{c_i}$  is the set of constrained atoms in  $R_{traceterm}$  having trace term  $c_i$ .
- 4. Let **generalise** $(R) = \{ \overset{\nabla}{\sqcup} (R_{N_1}), \ldots, \overset{\nabla}{\sqcup} (R_{N_k}), \overset{\nabla}{\sqcup} (R_{c_1}), \ldots, \overset{\nabla}{\sqcup} (R_{c_m}) \}.$

Figure 5.1: The monovariant generalisation algorithm

This is a correct abstraction according to the conditions in [Gal93]. The purpose of the abstraction is to preserve a separate atom for each predicate that manipulates a program point, and to use trace term abstraction for the other atoms. Note that the operation  $(\stackrel{\nabla}{\sqcup})$  preserves the program point argument. Some polyvariance and thus specialisation can be lost compared to an abstraction based on trace terms alone, but the result is appropriate for our needs.

**Soundness.** It is easy to see that the modification to the resulting CLP specialiser is correct. At the local control level, the partial evaluation of a program with respect to a query (as defined in Section 3.2.1) is taken as the computation trees from the root to the nearest nodes containing a program point. Clearly this is correct since the computation tree is still constructed using resolution and constraint solving. At the global control level, the operation  $\stackrel{\nabla}{\sqcup}$  guarantees correctness as it does for the algorithm previously presented (Fig. 3.2) and for analysis (Chapter 2).

**Termination.** The operation  $\Box$  terminates in the same way as it does for program analysis (Chapter 2). Also, there is a finite number of program points in the program. Hence, the unfolding rule also terminates because there are no static infinite loops in the semantics-based interpreter and unfolding always reaches a program point thus rendering every branch in the computation tree finite, provided the term denoting the imperative program code is ground and so is the list of imperative variables in the execution environment. A ground term representing the imperative program guarantees that program points are not variables when used for abstraction. A ground list of imperative program variables<sup>8</sup> ensures that the predicates updating elements of the execution environment terminate.

### 5.1.4 Polyvariant Imperative Program Specialisation

Note that trace terms already produce polyvariance for our CLP specialisers, and polyvariance at the imperative level is achieved when there is more than one version per program point. In order to obtain monovariant imperative specialisation we replaced the trace term of certain constrained atoms by their program point thus ignoring/overriding the possible polyvariance provided by trace terms. Unlike trace terms the program point label is a syntactic component of some constrained atoms independent of the shape of the computation trees they induce. This suggests combining trace terms and program points to obtain polyvariant imperative program specialisation and yet preserve some structure of the original imperative program, allowing the reconstruction of imperative programs from residual CLP programs and/or the mapping of program analysis of specialised CLP programs so produced. The combination consists in appending the trace term induced by a constrained atom to its program point label, if any. By so doing we now obtain at least a different predicate in specialised CLP programs for every different program point in the source imperative program. Next, the modified algorithm for polyvariant imperative program specialisation is depicted in Figure 5.2. In this algorithm we used program points but

<sup>&</sup>lt;sup>8</sup>Regardless of their contents, unless locations are used.

additional program information could be used as well e.g. the list of variables within scope at every program point.

**Soundness.** It is clear that the specialiser is correct provided the monovariant specialiser is. Unfolding is not affected by the polyvariance, it remains as constraint solving. The abstraction is correct because trace terms provide a correct abstraction and so do program points, thus their combination is correct.

**Termination.** At the global control the operation  $\Box$  terminates in a similar way as it does for program analysis (Section 2), i.e. based on the termination properties of widening [CC92b] there cannot be an infinite sequence of the constrained atoms associated with a program point. The combination of trace terms with program points also terminates because there is a finite number of program points in any imperative program and the combination (a subset of the Cartesian product) produces a finite number of different pairs, a program-point and a trace term, provided specialisation with trace terms terminates. Local termination is inherited from monovariant imperative program specialisation.

# 5.2 Program recovery

From now on we may recover specialised imperative programs (in our imperative language with assignments, conditionals and loops) from specialised CLP programs. We may assert that the residual programs generated by specialisation have a certain form as stated by the following definition.

**Definition 5.1** A flowgraph constraint logic program is a constraint logic program with clauses of the following form:

• 
$$H \leftarrow C, B$$

where H and B are user atoms or true (B only), and C is a constraint atom (possibly true too).

- 1. Let R be a finite set of constrained atoms.
  - Let  $R_{label-tt}$  be the set of constrained atoms in R whose atom component contain an argument of the form p(N,Stm), where Stm is one of the program constructs of the imperative language, (e.g. assign, ifte or while) and N is a program point label. Let  $R_{traceterm}$  be the remaining atoms in R.
- 2. Let  $\{R_{N_1T_1}, \ldots, R_{N_kT_l}\}$  be the finite partition of  $R_{label-tt}$  where all constrained atoms in  $R_{N_iT_j}$  contain the argument p(Ni,Stm) and have trace term  $T_j$ .
- 3. Let  $\{R_{c_1}, \ldots, R_{c_m}\}$  be the finite partition of  $R_{traceterm}$  where  $R_{c_i}$  is the set of constrained atoms in  $R_{traceterm}$  having trace term  $c_i$ .
- 4. Let **generalise** $(R) = \{ \overset{\nabla}{\sqcup} (R_{N_1T_1}), \ldots, \overset{\nabla}{\sqcup} (R_{N_kT_l}), \overset{\nabla}{\sqcup} (R_{c_1}), \ldots, \overset{\nabla}{\sqcup} (R_{c_m}) \}.$

Figure 5.2: The polyvariant generalisation algorithm

Claim. Given an imperative program and an environment in which the program and all variables used in the program are known<sup>9</sup> (i.e. listed) during unfolding specialisation generates only flowgraph constraint logic programs.

**Justification.** Because the one-state small-step semantics yields a tail recursive program, for each clause in the semantics program the specialiser will be able to evaluate completely all subgoals (deterministic<sup>10</sup>), leaving at most a single constraint (after elimination of local variables) and the last subgoal we chose not to expand so as to preserve the structure of the imperative program.

In the following we shall show how to extract an imperative program from a constraint logic program. Notice that using a left to right top-down computation rule to execute flowgraph constraint logic programs closely resembles the execution of a program with conditionals, assignments and goto statements. We will exploit this information to obtain the specialised imperative program from the residual constraint logic program.

Besides extracting some control information from the constraint logic program we need to know which logic variables are associated with which imperative variables. The specialiser is modified to record the correspondence of the filtered  $^{11}$  atoms with the original atoms. Thus we obtain residual programs where each logic variable can be associated with its corresponding imperative variable using the structure of the state (e.g. in state [[u,v,w],[X,Y,Z]] logic variables X,Y and Z correspond to imperative variables u,v and w, respectively). In addition, some further manipulation of the constraints is necessary to enable the reconstruction of legal imperative assignments as well as to distinguish between assignments and equality tests (in CLP their syntax)

<sup>&</sup>lt;sup>9</sup>Because the program and their variables are represented as ground terms, an unknown program or variable is a nonground term.

<sup>&</sup>lt;sup>10</sup>By definition of operational semantics of imperative programs the semantics-based interpreter cannot have repeated clauses or multiple answers to the same goal thus the interpreter may be written deterministic, as needed for specialisation, given the appropriate information (e.g. the program statement and the list of variables in scope).

<sup>&</sup>lt;sup>11</sup>During specialisation a stage called *argument filtering* removes some term structure from atoms, thus leaving only variables as arguments.

is the same). All this information is systematically extracted from the clause that contains the constraint.

**Definition 5.2** The callgraph G of a flowgraph constraint logic program P is a set of nodes N and arcs A. There is a mapping from the atoms in P to N such that two atoms  $B_1$  and  $B_2$  map to the same node iff  $B_1$ ,  $B_2$  have a common instance. There is an arc, with no label, from node  $N_1$  to node  $N_2$  iff P contains clause  $B_1 \leftarrow B_2$  and  $B_1$ ,  $B_2$  map to  $N_1$ ,  $N_2$  respectively; or P contains a clause  $B_1 \leftarrow C'$ ,  $B_2$  and the arc is labelled by  $C'\theta$ , where substitution  $\theta$  maps logic variables to imperative ones as determined by the structure of the imperative program state in  $B_1$  and  $B_2$ .

Callgraphs so constructed have some arcs with no label. Those arcs may denote transfer of control<sup>12</sup> in an imperative language with goto statements in which case the values of the program variables remain unchanged from the state prior to executing this command as after its execution. Alternatively, in an imperative language with assignments it may happen that an assignment is fully evaluated during specialisation thus the associated CLP clause for such an arc would have the form  $H \leftarrow B$  and the change of value for a variable would be reflected as a change in the contents of the states in H and B. Other transitions in the callgraph of a CLP program which have a special treatment during program recovery are those which arise as a result of specialisation of branching structure in the original imperative program where the branching has disappeared due to pruning, thus becoming deterministic. Such transitions of the callgraph have a boolean test as a label and there is no branching, thus they may be considered as an unconditional jump and the test may be thrown away as well as the corresponding arcs and nodes, with the appropriate redirecting as described in the next definition. This compressing of the callgraph corresponds to some imperative program specialisation.

**Definition 5.3** The minimal callgraph of callgraph G is graph G' obtained by removing from G every node  $S_n$  and arc  $(S_n, T_n)$  with no label, provided  $S_n$  has no  $\overline{}^{12}$ Procedure calls may have a similar effect, in some special cases, but in the general case a procedure call does not result in a simple transfer of control.

other directed-out arcs. Also, remove from G all nodes  $S_b$  and arcs  $(S_b, T_b)$  labelled by a boolean test, whose source node  $S_b$  has no branching structure. All arcs  $(P_x, S_n)$ or  $(P_x, S_b)$  entering the source node of such an arc are replaced by arcs  $(P_x, T_n)$  and  $(P_x, T_b)$  respectively, and node  $T_n$  is labelled with explicator(v), where v is the variable changing value from state in node  $S_n$  to state in node  $T_n$ .

Nodes with labels explicator [Mey91] in addition to their program point label are used to generate statements of fully evaluated<sup>13</sup> statements (especially assignments) in the residual program. Such labels represent additional information for some nodes in the graph, they do not replace the atoms associated with nodes.

Note that both definitions (callgraph and minimal callgraph) apply to residual CLP programs irrespective of the use made of them (i.e. monovariant or polyvariant imperative program specialisation). However, the program recovery may change if goto statements and procedures with parameters are allowed into the imperative language. One such a modification would be in computing the minimal callgraph, since goto statements lead to transitions in the callgraph that are removed for the minimal callgraph. A similar effect is produced by procedure calls without parameters. We will not cover this here but some discussion will be provided in Section 5.2.1.

### Reconstructing an imperative program

For a simple imperative language with assignments, conditionals and loop constructs a combination of a method for structural analysis [Muc97] with information from the flowgraph CLP program is enough to recover imperative programs. An algorithm is given in Fig. 5.3. In summary, structural analysis associates some graph shapes with program constructs among other things<sup>14</sup>. Because there is some overlapping between the graph shapes associated with different imperative program constructs, a common case arising in structural analysis, we use data contained in the source flowgraph CLP program. This information aids in resolving ambiguities by deciding which program

<sup>&</sup>lt;sup>13</sup>That is, all their expressions are evaluable at compile-time.

<sup>&</sup>lt;sup>14</sup>Knowing the programming language constructs allows the definition of the graph shapes and this information is used to perform efficient bit-vector data-flow analysis.

construct corresponds to the (sub)graph considered. For instance, the following two clauses correspond to residual code for an if-then-else construct

```
ifte(1,[R,W,X,Y,2]) <- Y<2, assign(2,[R,W,X,Y,2])
ifte(1,[R,W,X,Y,2]) <- Y>=2, assign(4,[R,W,X,Y,2])
```

Recall that some function symbols have been removed during specialisation; however, some were restored for presentation. Also, remember that this may be done automatically. In order to decide which assignment statement corresponds to the then branch and which corresponds to the else branch it is enough to see the non-filtered version of these two clauses (where the truth value should be added as an extra argument of the head of each clause defining the semantics of the if-then-else program construct in page 75). There the truth value given to each branch is shown. For instance, the head of the first clause could be

```
ifte(1,[[r,w,x,y,z],[R,W,X,Y,2]],le(y,2),true,\\ compose(p(2,assign(x,plus(x,2))),...),\\ compose(p(4,assign(y,minus(y,1))),...))
```

where the given truth value is true for true. Hence, the first clause corresponds to the then branch. The end of each branch is determined when they meet a common program point. A missing branch results in a transition of the callgraph whose associated program construct (if-then-else or while in the language considered) has been evaluated by the specialiser, thus yielding the kind of transitions eliminated when computing the minimal callgraph.

It may be thought<sup>15</sup> that this algorithm for program recovery could be related to classic parsing algorithms. We argue that the connection could exist had we suggested which is the representation of the graph. The algorithm is bottom-up and SLR or LALR parsing algorithms could bear some resemblance, however parsing proceeds from left to right and our graphs have no orientation as such. Devising a proper ordering and representation of the graph for parsing algorithms to apply could be

<sup>&</sup>lt;sup>15</sup>Andy King suggested this connection.

```
INPUT: A residual CLP program P and its associated minimal callgraph G
OUTPUT: An abstract syntax tree for an imperative program
G_0 := \mathbf{compress}(G);
g_0 := \mathbf{subgraph}(G_0);
i := 0;
repeat
        if graph(if B then S1 else S2, g_i) then
              G_{i+1} := \mathbf{replace}(g_i, G_i, \mathsf{if} \; \mathsf{B} \; \mathsf{then} \; \mathsf{S1} \; \mathsf{else} \; \mathsf{S2})
        else
              if graph(while B do S, g_i) then
                     G_{i+1} := \mathbf{replace}(g_i, G_i, \text{ while B do S})
              else
                     improper-region(q_i,G_i,G_{i+1},P)
        i := i + 2;
        G_i := \mathbf{compress}(G_{i-1});
        g_i := \mathbf{subgraph}(G_i)
until g_i = G_i
```

Figure 5.3: Imperative program recovery

very expensive<sup>16</sup> and thus we disregarded it as a possibility to improve the algorithm. Nonetheless some ideas from bottom-up parsing could possibly be adapted for their use in program recovery. This is a subject of future research.

The algorithm of Fig. 5.3 describes the reconstruction of an abstract syntax tree from the residual program P and its associated minimal callgraph G. Function **subgraph** takes a directed graph and returns a minimal subgraph having a graph pattern as those associated with the language instructions [Muc97, page 203] (e.g. while,

<sup>&</sup>lt;sup>16</sup>Or equivalent to solving the problem by finding an appropriate ordering for parsing.

if-then-else, etc.). When more than one pattern can be associated with the same subgraph this function returns the same graph. Also, it returns the same graph if the input graph has only one arc with an input and an output node only (all nodes of the initial callgraph represented as single node and the halt node). The predicate graph(S,g) succeeds when S (a program statement) has graph representation g with the same edge and node labelings. Statement if-then-else is a generalisation of several forms of conditionals, hence predicate graph introduces the redundant information where appropriate (e.g., add skip to obtain an if-then statement). Replacing a subgraph by a single node is done by several functions depending on the case. compress takes a graph and replaces any non-branching sequence (excluding the node labelled by halt) of nodes by a single node whose label is the sequential composition of the nodes compressed<sup>17</sup>, returning another graph;  $\mathbf{replace}(g,G,S)$  replaces subgraph g of G by a single node whose label is S; finally, **improper-region** $(g_i,G_i,G_{i+1},P)$ recursively traverses the depth-first spanning tree of  $g_i$  bottom up until it finds an if-then-else or a while (through its program point and P) label on a node and replaces the appropriate subgraph of  $g_i$  (and  $G_i$  too) by the matched construct, until it reduces all  $g_i$  to a single node with an input and output edge only (graph  $G_{i+1}$ ).

Example 8 (Program recovery) Consider the minimal callgraph depicted in Figure 5.4a. The algorithm described above (Figure 5.3) applied to this graph performs the following steps. Note that nodes 5 and 6 could be compressed (see subgraph enclosed in dotted rectangle in Figure 5.4b) thus yielding the composition of statements h=h+1 and k=i\*h+k further referred to as code1. Then this pair of nodes becomes a single node whose outgoing arc is labelled by the composition of assignments, and this node together with node 4 form a loop subgraph, as shown in Figure 5.4b in the nodes enclosed with dotted lines. These nodes and the corresponding arcs and labels are updated according to the definition of a while loop (see Figure 5.5a, where the new node name is 456). No compression of this graph is possible, but two subgraphs match the graph pattern of an if-then-else (subgraphs encircled in Figure 5.5b).

<sup>&</sup>lt;sup>17</sup>When a node with an explicator label is compressed an assignment for the explicator is prefixed to the corresponding statement.

The result of both updates result in the graph shown in Figure 5.5b, where B1 = eq(plus(1,1),h), code2 = l=k+10, code3 = l=k-10, code4 = h=h-1 and code5 = while(gt(h,i),code1). Then compression reduces nodes 1 and 23456 forming the composition of the corresponding statements. Next, a subgraph enclosing nodes 7 and 8910 matches a graph pattern for a loop and is reduced accordingly resulting in a graph with two nodes (not shown) whose compression gives a graph with only two nodes whose connecting arc, in turn, is labelled by the imperative program recovered from the original callgraph. The resulting imperative program reads as follows.

```
j = j+h;
if i>j then
   h = h-1
 else
   while h>i do
     {
      h = h+1;
      k = i*h+k
     }
while k>l do
   {
   if 1+1=h then
      1 = k+10
   else
      1 = k-10
   }
halt
```

### 5.2.1 Extending the imperative language

The simple imperative language assumed in this chapter corresponds to the WHILE language of [NN92]. It contains if-then-else, while, and assignment statements

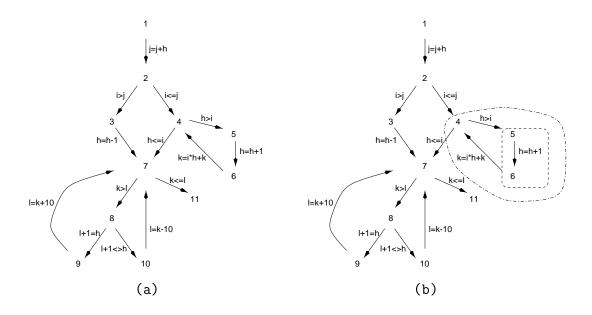


Figure 5.4: Example callgraph

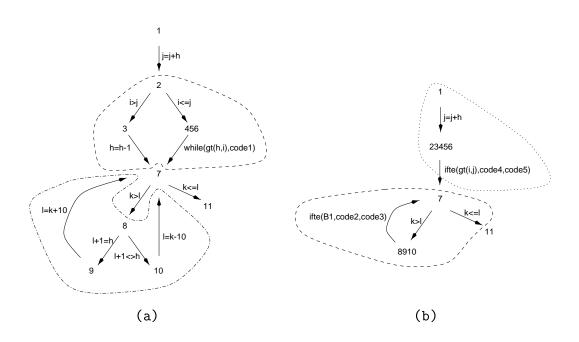


Figure 5.5: Example reducing a callgraph

with simple arithmetic and boolean expressions over integers. We envisage three modifications to the current proposal to cope with extensions in the imperative language treated.

#### Procedures with parameters

The semantics is modified in two ways in order to stay within the one-state small-step operational semantics form. Firstly, the state is extended to have locations (Section 4.2.2). Secondly, the semantics predicate includes a third argument in addition to the variable environment and the program statements. This new argument provides the functionality of the stack of activation records in traditional compiler implementations for imperative languages. A problem arises when computing the upper bound<sup>18</sup> of two atoms with the same program point but different stack of activation records. A promising solution is the use of regular approximations during generalisation in specialisation, as shown in [GP00]. Next we elaborate more on this proposal to increase the precision of regular approximations and specialisation of semantics-based interpreters (see the Appendix for examples of specialisation of interpreters that have the stack of activation records using this proposal).

The proposal consists in solving several inherent limitations associated with having a determinate unfolding rule, program points as abstraction to guide the local control of our specialisers, and tuple-distributivity of the upper bound of RUL constraints. All these features combined together prevent specialisation of interpreters for languages with parameters and block declarations to parse away [VM97] the interpretation layer thus producing small programs useful for analysis and specialisation as we have shown. The problems can be summarised in one example. Assume we have an imperative program with procedures. A quick look at the list of visible variables during execution of this program with our semantics-based interpreters reveals that the variables within scope (e.g. visible variables) at the program points before entering a procedure may be different depending on the place where this procedure was called. Moreover, after entering the procedure the frame with the return address and the list of variables to

<sup>&</sup>lt;sup>18</sup>Within the **generalise** operation of Fig. 3.2.

be restored upon exit from the procedure are context dependent. That is, for the same program point (e.g. before entering a procedure or a block) there may be more than one possible list of variables within scope<sup>19</sup>. Using monovariant imperative program specialisation forces atoms containing different list of visible variables and contents in the stack of activation records to be generalised. Here the tuple distributivity of RUL constraints would lose the contextual information, thus accepting lists of variables which are a combination of the original lists, prior to upper bound; hence the predicates handling the execution environment become too general thus precluding some specialisation opportunities. Polyvariant imperative program specialisation may not force program points with different execution contexts (list of visible variables and stack of activation records) to be generalised, but there is no guarantee that this will be the case. This suggests to use the list of visible variables as a guide in finding the right polyvariance. Yet, program points are needed for the reasons already mentioned (Section 5.1). Thus a combination of these two ensures that at least one predicate for every different execution context of the same program point is produced. Nonetheless, tuple distributivity affects the approximation of the stack. This problem can be overcome using 'function wrappers' as described in Section 2.2.1 (Example 2). Thus we trade tuple distributivity for union which is more precise. In particular, the clause for procedure calls with parameters (Section 4.2.2, page 84) becomes the following clause.

Now, the subgoal code\_id(E,F,S,S\_id) is defined in such a way that it picks the list

<sup>&</sup>lt;sup>19</sup>The contents of the activation records varies in a similar way.

of variables from E and together with S forms a term S\_id using a function symbol which doesn't occur in the semantics-based interpreter. For instance, if the first element of E is the list of visible variables before entering the procedure's body, and the function symbol used to create S\_id takes the program point of the first element in S (predicate first\_pp(S,F)) then the definition of predicate code\_id is as follows.

```
code_id([V|Es],1,S,[pp1(V,S)|Es]) <-
code_id([V|Es],2,S,[pp2(V,S)|Es]) <-
code_id([V|Es],3,S,[pp3(V,S)|Es]) <-
code_id([V|Es],4,S,[pp4(V,S)|Es]) <-
...</pre>
```

There is a clause as above for every program point in the source imperative program because potentially the program may return to any point in the program, though this may not be the general case. The new function symbol isolates the information which is vital to obtain a good specialisation (an unknown imperative program during specialisation results in no specialisation, and an unknown list of visible variables results in not being able to eliminate most of the environment handling predicates in the semantics-based interpreter). Respectively, upon return from the procedure a frame is popped from the stack and the contents unwrapped accordingly. That is, the clause for return of execution from a procedure call (page 81) must be modified as follows.

```
exec(pop_fr,E,St1) <-
    push_pop(S_id,OPE,St,St1),
    code_id_cont(S_id,E,OPE,St)</pre>
```

The definition of predicate code\_id\_cont(S\_id,E,OPE,St) is again a sequence for every possible program point of the source imperative program.

Similar clauses should be produced for every program point in the source imperative program. In addition, through a separate analysis (to determine the set of names visible from a given program point) we could write more specific definitions for predicate code\_id\_cont, thus generating smaller residual programs. That is, if such an analysis determines that the possible lists of visible variables at program point 3 are [a,b,d] and [g,a,b,h] then the clauses defining predicate code\_id\_cont for function symbol pp3 become

By so doing we avoid tuple distributivity in the upper bound. Similar ideas could be applied to the procedure environment (when it changes throughout execution). Here we have presented the more common case of a changing set of visible variables. Some interpreters for imperative languages with procedures with/without parameters and examples of specialised programs using the techniques described above are shown in Appendix A.1.

Regarding imperative program specialisation the combination of the domains linear arithmetic expressions and regular unary logic programs makes it possible to have some form of interprocedural constant propagation as shown in [JM82] and potentially more since constants are a special case of linear expressions. Imperative program recovery has to be extended with the cases of the new program statements (call, procedure definition, and possibly a return statement). Also, the extensions discussed above for arrays and records apply. Care should be taken when inserting explicators because there are nonliftable values, e.g. pointers. For program analysis these results may be useful since most of the interpreter layer has been removed using these techniques for improving the precision of upper bound of regular predicates.

#### Compound data types

It is straightforward to extend the current semantics to include arrays and records yet preserving the one-state small-step operational semantics form (see Chapter 4). The ensuing modification to the specialiser is to allow flowgraph CLP programs to have a conjunction of constraints where there is only one constraint in the body of a clause. Such a conjunction of constraints would be composed of arithmetic constraints (as before) and certain user atoms. Arithmetic constraints are translated as explained above. The user atoms in the conjunction represent unevaluated selector predicates for accessing parts of compound structures, e.g. a lookup in the variable environment (see, for instance example 12 and 13). Those subgoals could be regarded as constraints on the logical variable representing the output of such an update. Recovering a variable name (array or record) is straightforward from the meaning of the selector predicate symbol according to the semantics. For example, consider the following program fragment with an assignment using arrays.

- 1) j := a[i]+2;
- 2) while ....

If the contents of array a and the values of the other variables are unknown, then [[i,j,array(a)],[I,J,A]] could be used to describe the state before executing

the assignment. Specialisation of an imperative program containing this program fragment would result in the next CLP program.

```
assign(I,J,A) <-
lookup_a(I,A,Vi),

J1 = Vi+2,
while(I,J1,A)
while(I,J,A) <- ....</pre>
```

Here the subgoal lookup\_a(I,A,Vi) acts as a 'constraint' on the value Vi of the I-th cell of array A. Moreover, for imperative program recovery subgoal lookup\_a(I,A,Vi) is equivalent to replacing expression a[i] wherever variable Vi occurs in the body of the same clause.

#### Other program constructs

Extending the language with repeat-until, for, case and other program constructs could be done with a price to pay in program recovery and preprocessing of the imperative program. It is well known that such program constructs could be expressed in terms of while and if-then-else constructs. As a result, their semantic definitions would have some overlapping with those of the basic program constructs (while and if-then-else). Hence, recovering the appropriate program statement raises potential problems of nondeterminism in reconstructing the imperative program. In addition, some contextual information is required to produce the appropriate abstract syntax trees for input to the semantics-based CLP program. For these reasons, such constructs are normally avoided for program analysis during specialisation, commonly regarded as "syntactic sugar" [GMS98]. A possible solution is to allow the inclusion of the above program constructs at the expense of recovering programs where only goto, while and if-then-else constructs occur.

### 5.3 Summary

When specialising semantics-based interpreters in CLP the input imperative program is guaranteed to correspond to the residual CLP program; however, the connection between both is unclear as regards where the operations in the CLP program take place with respect to the operations in the original imperative program. In order to obtain a correspondence between the imperative programs and its corresponding declarative program some tuning of the specialiser is needed. Otherwise, the specialiser may remove important information needed to relate imperative statements and variables with their declarative counterpart. Such tuning involves selecting among the predicates of the semantics-based interpreter those we want to be defined in the residual program. Hence, we choose predicates from the semantics-based interpreter that relate directly to the meaning of the statements in the imperative program to be specialised. This is achieved by adding program points to the imperative program. As a result we get at least one predicate for each statement of the imperative program, thus highlighting the correspondence between imperative statements and predicates in the residual program. In this way analysis results of the residual CLP programs can be systematically mapped back to the source imperative program.

Alternatively, if the semantics-based CLP interpreter is specialised with respect to an imperative program with input constraints, the residual CLP program may be regarded as a specialisation of the source imperative program. We identified a class of CLP programs, flowgraph CLP programs, from which imperative programs may be recovered. Flowgraph CLP programs coincide with the class of programs produced by specialising our semantics-based CLP interpreters, thus we have a method for specialising imperative programs using CLP as an intermediate language. Finally, trace terms were combined with program points to obtain polyvariance in the resulting imperative program specialisers.

## Chapter 6

## Applications in Imperative

# Languages

Decorating statements in imperative programs with unique identifiers leads to ways of relating imperative programs to residual programs (see Chapter 5). That is, for every program point of the imperative program we generate at least one clause head in the corresponding residual program thus providing information of how the specialised CLP program relates to an imperative program (either the original imperative program or a specialisation of it). In this chapter some examples of analysis and specialisation of imperative programs are provided.

## 6.1 Program Specialisation

In our application the program to be specialised is the CLP program defining the semantics of an imperative language, while the partial input consists of a term representing the target imperative program, and a partially given environment<sup>1</sup>, both of which appear as arguments of the semantics predicate.

The effect of these techniques is to give specialisations in which a different predicate is generated for each reachable program point in the source imperative program.

<sup>&</sup>lt;sup>1</sup>A partially given environment means that the variable names within scope are known whereas their contents may not be known.

Note that more than one predicate per program point may be generated if we use information (e.g. trace terms [GL96] or characteristic trees [GB91] or the structure of the execution environment) in addition to the program points for the **generalise** operation.

In order to specialise a given imperative program several steps should be followed. Given the text of the program, program points are attached to every distinguishable construct of the imperative program (e.g. assignments, while and if-then-else conditionals). Next, its associated abstract syntax tree is produced in the form of a term. Then, specialisation may proceed given some input constraints and the semantics-based interpreter in CLP.

**Example 9** Consider the following imperative program (whose program points appear at the left of every program command).

```
1) while y<2 do
2)    x := x - 3*z;
3)    y := y + x
    endwhile;
4) w := y + 1;
5) if w<z then
6)    r := 1
    else
7)    r := -1
8) halt</pre>
```

The associated abstract syntax tree term follows.

Now, specialising this program with respect to input constraint z=2 (that is, environment [[r,w,x,y,z],[\_,\_,\_,2]]) we obtain the following CLP program.

```
exec([R,W,X,Y,2]) < -
    while (1, [R, W, X, Y, 2])
while (1, [R, W, X, Y, 2]) \leftarrow
    Y<2,
    assign(2,[R,W,X,Y,2])
while(1,[R,W,X,Y,2]) \leftarrow
    Y > = 2,
    assign(4,[R,W,X,Y,2])
assign(2,[R,W,X,Y,2]) <-
    X1=X-6,
    assign(3,[R,W,X1,Y,2])
assign(3,[R,W,X,Y,2]) \leftarrow
    Y1=Y+X,
    while(1,[R,W,X,Y1,2])
assign(4,[R,W,X,Y,2]) <-
    W1=Y+1,
    ifte(5,[R,W1,X,Y,2])
ifte(5,[R,W,X,Y,2]) < -
```

```
W>=2,
    assign(7,[R,W,X,Y,2])
assign(7,[R,W,X,Y,2]) <-
    halt(8,[-1,W,X,Y,2]).
halt(8,[-1,W,X,Y,2]) <-</pre>
```

Note that, for conciseness, some information was omitted<sup>2</sup>. The missing information corresponds to the imperative program argument as well as the list of variable names<sup>3</sup>. Later on, this information is used to recover the associated imperative program. The first numeral argument to some predicates above denote the program point associated with that predicate. The list corresponds to half of the contents of the variable environment. The half not shown is [r, w, x, y, z].

For instance, the clause with head assign(4,[R,W,X,Y,2]) above would read, prior to argument filtering and renaming (exec to assign), as:

```
assign(4, [[r, w, x, y, z], [R, W, X, Y, 2]], \\ compose(p(4, assign(w, plus(y, 1))), ...)) <- \\ W1=Y+1, \\ ifte(5, [[r, w, x, y, z], [R, W1, X, Y, 2]], compose(p(5, if(less(w, z), \\ p(6, assign(r, 1)), \\ p(7, assign(r, -1)))), \\ p(8, halt))))
```

We have adopted the convention of using names starting with a capital letter to denote variable names, and the rest are constants of the program. From now on we may recover the specialised imperative program from this specialised logic program.

Besides constructing the callgraph of this program we need to know which logic variables are associated with which imperative variables. For this purpose, argument filtering and flattening [GB90] are suppressed from the program generation phase of

<sup>&</sup>lt;sup>2</sup>Also, exec predicates have been renamed to have the name of the first element (program construct) in their argument denoting the imperative program.

<sup>&</sup>lt;sup>3</sup>We intentionally named the logic variables as their imperative counterpart for ease of reading.

our specialisers because they obscure the correspondence between logic variables and their imperative counterpart. Also the correspondence between the predicates in the CLP program and the program points is extracted from the verbose version of the specialised CLP program. The minimal callgraph of the above CLP program is depicted in Figure 6.1.

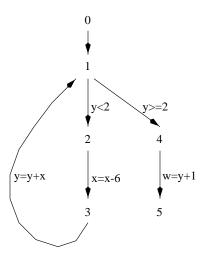


Figure 6.1: Minimal callgraph of residual program.

The atom mapping is as shown in the next table connecting the callgraph, the CLP program, the original imperative program and its specialised program.

Predicate	Program point	Explicator	Node
exec			0
while_1	1		1
assign_2	2		2
assign_3	3		3
assign_4	4		4
halt_8	8	r = -1	5

From the information above and using the algorithm of Fig. 5.3 we obtain the following residual imperative program.

```
while y<2 do
    x := x-6;
    y := y=X;
endwhile
w := y+1;
r := -1;</pre>
```

Note that a constant was propagated inside of the loop (similar results could be achieved with 'conditional constant propagation' [WZ85]) and one of the branches was pruned from a former if-conditional, using constraint propagation.

It may be thought that because our unfolding rule 'stops when a program point is met' that no specialisation at the imperative level is achieved. In the following example we will see that using polyvariant imperative program specialisation (see Section 5.2) some 'traditional' specialisation, namely loop unrolling, is achieved.

**Example 10** Consider the program for computing powers  $(z = x^y, y > 0)$ .

)

```
),
p(5,halt)
)
)
```

By specialising our semantics-based interpreter with respect to y=3 we obtain the next residual CLP program.

```
powers(X1,3) :-
                                  exec_7(X1,X2):-
      exec_1(X1,X2).
                                        exec_8(X1,X2).
exec_1(X1,X2):-
                                  exec_8(X1,X2):-
                                        exec_9(X1,X2).
      exec_2(X1).
exec_2(X1) : -
                                  exec_9(X1,X2):-
      exec_3(X1).
                                        X3 = X2*X1,
exec_3(X1) : -
                                        exec_10(X1,X3).
      X1 = 1*X1,
                                  exec_10(X1,X2):-
      exec_4(X1).
                                        exec_11(X1,X2).
exec_4(X1) : -
                                  exec_{11}(X1,X2):-
      exec_5(X1).
                                        exec_12(X1,X2).
exec_5(X1) :-
                                  exec_12(X1,X2) :-
      exec_6(X1).
                                        true.
exec_6(X1):-
      X2 = X1*X1
      exec_7(X1,X2).
```

The associated minimal callgraph of this program is depicted in Figure 6.2, and the corresponding atom mapping is given by the following table.

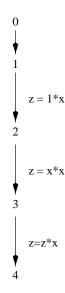


Figure 6.2: Minimal callgraph of powers residual program

Predicate	Program point	Explicator	Node
powers			0
exec_3	(3)	z = 1; y = 3	1
exec_6	(3)	y = 2	2
exec_9	(3)	y = 1	3
exec_12	(5)	y = 0	4

The specialised program extracted from this callgraph and table of nodes is

```
z := 1;

y := 3;

z := 1*x;

y := 2;

z := x*x;

y := 1;

z := z*x;

y := 0;
```

Execution of this program is trivial and explicators are not needed. Furthermore, optimising compilers [Muc97] would remove them by simple def-use analysis.

### 6.2 Program Analysis

A specialiser is applied to our semantics-based interpreters with respect to an input imperative program. The result is a residual CLP program which is a version of the interpreter specialised with respect to the input program. The residual program is then analysed, and the analysis results are related to the original imperative program, as we will discuss.

Thus, for the purposes of analysis, specialisation is not required to produce the maximum possible specialisation. It suffices to have a translator from imperative to constraint logic programs where imperative statements and variables can be explicitly related to predicates and their arguments. We already saw that it suffices to have program points to relate residual CLP programs to imperative programs; thus the specialiser may also produce specialisation besides translating.

Example 11 Consider the contrived imperative program

```
(1)
     i := 2;
(2)
     i := 0;
(3)
     while (n*n > 1) do
       if (n*n = 2) then
(4)
            i := i+4;
(5)
       else
(6)
            i := i+2;
(7)
            j := j+1;
       endif;
      endwhile;
```

By specialisation with respect to the above program and unknown initial environment we obtain:

```
exec_2(X2,X3).
exec_2(X1,X2) :-
      exec_3(2,0,X2).
exec_3(X1, X2, X3) :-
      X4 = X3*X3,
      X4 > 1,
      exec_4(X1,X2,X3).
exec_3(X1, X2, X3) :-
      X4 = X3*X3,
      X4 = <1,
      halt(X1, X2, X3).
exec_4(X1,X2,X3):-
      X4 = X3*X3,
      X4 = 2,
      exec_5(X1,X2,X3).
exec_4(X1,X2,X3):-
      X4 = X3*X3,
      X4=\=2,
      exec_6(X1,X2,X3).
exec_5(X1, X2, X3) :-
      X4 = X1+4
      exec_3(X4,X2,X3).
exec_6(X1, X2, X3) :-
      X4 = X1+2
      exec_7(X4,X2,X3).
exec_{7}(X1, X2, X3) :-
      X4 = X2+1,
      exec_3(X1,X4,X3).
halt(X1,X2,X3).
```

Here the correspondence between predicates and program points is as follows; (this

information	mas	extracted	automatically	from	the	specialiser)
ung on muuuuuu	$\omega u \sigma$	Callactea	aatontattattattg	JIOIII	0100	spectation.

Predicate	Program point
ch	
exec_1	1
exec_2	2
exec_3	3
exec_4	4
exec_5	5
exec_6	6
exec_7	7
halt	8

Once we have an appropriate constraint logic program representing an imperative program with the desired clause heads the next step is to produce a query answer transformed form of such a program in order to simulate goal directed analysis (see Section 2.3). The result is input to the analyser for constraint logic programs [Sağ98]. By analysing the residual program shown above we obtain the following results:

```
ch_query(X1,X2,X3) :-true.
statements_1_query(X1,X2,X3) :-true.
statements_2_query(X1,X2) :-true.
statements_5_query(X1,X2,X3) :-X2-0.5*X1=< -1.0,X2>=0.0.
statements_3_query(X1,X2,X3) :-X2-0.5*X1=< -1.0,X2>=0.0.
statements_4_query(X1,X2,X3) :-X2-0.5*X1=< -1.0,X2>=0.0.
statements_6_query(X1,X2,X3) :-X2-0.5*X1=< -1.0,X2>=0.0.
statements_6_query(X1,X2,X3) :-X2-0.5*X1=< -1.0,X2>=0.0.
```

where we have a set of constraints associated with each predicate in the analysed program<sup>4</sup>. The suffix \_query in every predicate name indicates that the constraints hold for the variables named every time we call that predicate during program execution. The query predicates are produced by query-answer transformation [GdW94]

<sup>&</sup>lt;sup>4</sup>We show the results only for the relevant predicates

(1)

applied to the specialised program and the initial goal. Since we have a clause defining a predicate associated with a program point we may then say that the constraints associated with a predicate query are those that hold before the program point to which that predicate refers.

Once we have identified which is the relationship holding between the predicates and the program points it remains to determine how variables relate. Given the variable environment we may readily obtain the correspondence between logic variables and imperative variables, and between clause heads and program points. Interpreting the above results yields:

```
i := 2;
(2)
     j := 0;
                  {j>=0, 2j+2=<i}
     while (n*n > 1) do
(3)
                  {j>=0, 2j+2=<i}
(4)
       if (n*n = 2) then
                  {j>=0, 2j+2=<i}
(5)
           i := i+4;
       else
                  {j>=0, 2j+2=<i}
(6)
           i := i+2;
                  {j>=0, 2j+4=<i}
(7)
           j := j+1;
       endif;
      endwhile;
```

Next we present another example in which arrays are represented as lists.

**Example 12** This code sorts an array of size n using the bubblesort algorithm [Knu73]. This example was adapted from one used by Cousot and Halbwachs [CH78].

(1) b := n;

```
(2)
    while (b>=1) do
(3)
          j := 1;
          t := 0;
(4)
(5)
          while (j = < b-1)
(6)
             if (k[j] > k[j+1]) then
(7)
                tm := k[j];
                k[j] := k[j+1];
(8)
(9)
                k[j+1] := tm;
(10)
                t := j;
             endif;
(12)
             j := j+1
          endwhile;
(13)
          if (t == 0) then
(14)
             b := -1;
          else
             b := t;
(15)
          endif;
      endwhile;
```

Using our specialiser we obtain the following equivalent CLP program:

```
bbsrt(X1,[X2,X3|X4]) :-
        exec_1(X5,X1,X2,X3,X4,X6,X7,X8).

exec_1(X1,X2,X3,X4,X5,X6,X7,X8) :-
        exec_2(X2,X2,[X3,X4|X5],X6,X7,X8).

exec_2(X1,X2,X3,X4,X5,X6) :-
        1=<X1,
        exec_3(X1,X2,X3,X4,X5,X6).

exec_2(X1,X2,X3,X4,X5,X6) :-
        1>X1,
        halt(X1,X2,X3,X4,X5,X6).
```

```
exec_3(X1, X2, X3, X4, X5, X6) :-
       exec_4(X1, X2, X3, X4, X5).
exec_4(X1, X2, X3, X4, X5):-
       exec_5(X1, X2, X3, 0, X5, 1).
exec_5(X1, X2, X3, X4, X5, X6):-
      X7 = X1-1
      X6 = < X7,
       exec_6(X1, X2, X3, X4, X5, X6).
exec_5(X1, X2, X3, X4, X5, X6):-
      X7 = X1-1,
      X6>X7,
       exec_7(X1, X2, X3, X4, X5, X6).
exec_6(X1, X2, X3, X4, X5, X6):-
       look_up1_15(X3,X6,X7,1),
      X8 = X6+1,
       look_up1_15(X3,X8,X9,1),
      X7 > X9,
       exec_8(X1, X2, X3, X4, X5, X6).
exec_6(X1, X2, X3, X4, X5, X6) :-
      look_up1_15(X3,X6,X7,1),
      X8 = X6+1,
       look_up1_15(X3,X8,X9,1),
      X7 = < X9
       exec_9(X1, X2, X3, X4, X5, X6).
exec_7(X1, X2, X3, X4, X5, X6):-
      X4 = 0,
       exec_{10}(X1, X2, X3, X4, X5, X6).
exec_7(X1, X2, X3, X4, X5, X6):-
      X4=\=0,
       exec_{17}(X1, X2, X3, X4, X5, X6).
```

```
exec_8(X1,X2,X3,X4,X5,X6):-
      look_up1_15(X3,X6,X7,1),
      exec_{11}(X1, X2, X3, X4, X7, X6).
exec_9(X1, X2, X3, X4, X5, X6) :-
      exec_{12}(X1, X2, X3, X4, X5, X6).
exec_{10}(X1,X2,X3,X4,X5,X6):-
      exec_2(-1, X2, X3, X4, X5, X6).
exec_{11}(X1,X2,X3,X4,X5,X6):-
      X7 = X6+1,
      look_up1_15(X3,X7,X8,1),
      look_up1_16(X3,X9,X6,X8,1),
      exec_{13}(X1, X2, X9, X4, X5, X6).
exec_{12}(X1, X2, X3, X4, X5, X6) :-
      X7 = X6+1,
      exec_5(X1, X2, X3, X4, X5, X7).
exec_{13}(X1, X2, X3, X4, X5, X6) :-
      X7 = X6+1,
      look_up1_16(X3,X8,X7,X5,1),
      exec_{14}(X1, X2, X8, X4, X5, X6).
exec_{14}(X1,X2,X3,X4,X5,X6):-
      exec_{12}(X1, X2, X3, X6, X5, X6).
look_up1_15([X1|X2],X3,X1,X4) :-
      X3=X4.
look_up1_15([X1|X2],X3,X4,X5) :-
      X5<X3,
      X6 = X5+1,
      look_up1_15(X2,X3,X4,X6).
look_up1_16([X1|X2],[X3|X2],X4,X3,X5) :-
      X4 = X5.
look_up1_16([X1|X2],[X1|X3],X4,X5,X6) :-
```

```
X6<X4,
X7 = X6+1,
look_up1_16(X2,X3,X4,X5,X7).
exec_17(X1,X2,X3,X4,X5,X6) :-
        exec_2(X4,X2,X3,X4,X5,X6).
halt(X1,X2,X3,X4,X5,X6) :-
        true.</pre>
```

Accordingly, the relationship between predicate names and program points is given in the following table.

Predicate	Program point	Predicate	Program point
bbsrt		exec_8	7
exec_1	1	exec_9	11
exec_2	2	exec_10	14
exec_3	3	exec_11	8
exec_4	4	exec_12	12
exec_5	5	exec_13	9
exec_6	6	exec_14	10
exec_7	13	exec_17	15
halt	16		

The analysis results of the query-answer transformed form of this program follow.

```
bbsrt_query(X1,X2) :-
    true.
exec_1_query(X1,X2,X3,X4,X5,X6,X7,X8) :-
    true.
exec_10_query(X1,X2,X3,X4,X5,X6) :-
    X4=0.0,X1-X6<1.0,X6>=1.0,X1-X6>= -0.0.
exec_2_query(X1,X2,X3,X4,X5,X6) :-
    true.
exec_3_query(X1,X2,X3,X4,X5,X6) :-
```

X1 >= 1.0.

$$exec_7_query(X1,X2,X3,X4,X5,X6)$$
:-
$$X1-X6<1.0,X6>=1.0,X4>=0.0,X1-X6>= -0.0.$$

Interpretation of these results is depicted in the next decorated program.

(1) 
$$b := n;$$

(2) while 
$$(b>=1)$$
 do

$$\{b>=1\}$$

(3) 
$$j := 1;$$

$$(4)$$
 t := 0;

```
(5)
           while (j = < b-1)
                            \{t>=0, j>=1, b>=j+1\}
(6)
               if (k[j] > k[j+1]) then
                             \{t>=0, j>=1, b>=j+1\}
(7)
                  tm := k[j];
                  k[j] := k[j+1];
(8)
                  k[j+1] := tm;
(9)
(10)
                  t := j;
                            \{t>=0, j>=1, b>=j+1\}
(12)
              j := j+1
(13)
           if (t == 0) then
                            \{t>=0, j>=1, b>=j, b< j+1\}
              b := -1;
(14)
           else
                             \{t>0, j>=1, b>=j, b<j+1\}
(15)
              b := t;
```

These results are different to those obtained in [PGS98] because we used a weaker analyser, hence some constraints are weaker (e.g. it should have been possible to obtain the result that n>=b hold throughout the whole program). By adding this information our results can be compared to those obtained in [CH78], where it was noted that projecting the constraints onto the variables indexing the array would show that all array accesses are within bounds.

**Example 13** Consider the following program for sorting the elements of an array of size n using the heapsort algorithm.

```
else
6)
   k := t[r];
      t[r] := t[1];
7)
8)
      r := r-1;
   endif;
9) while r \ge 2 do
10)
      i := 1;
11)
     j := 2*i;
12)
     tm := 1;
13)
     while (j<=r and tm=1) do
14)
          if j \le r-1 then
             if t[j] < t[j+1] then
15)
16)
                j := j+1;
             endif;
          endif;
17)
          if k \ge t[j] then
18)
            tm := 0
          else
            t[i] := t[j];
19)
20)
            i := j;
21)
             j := 2*j
           endif;
        endwhile;
        t[i] := k
22)
23)
        if 1 \ge 2 then
24)
           1 := 1-1;
25)
          k := t[1];
        else
26)
           k := t[r];
27)
           t[r]:= t[1];
```

```
28)
            r := r-1;
         endif;
         t[1] := k;
29)
    endwhile;
30) halt
The result of specialisation is the next CLP program.
hpsrt(X1,[X2,X3|X4]) :-
      exec_1(X5,X6,X7,X8,X9,X10,X1,X2,X3,X4).
exec_1(X1, X2, X3, X4, X5, X6, X7, X8, X9, X10) :-
      X11 = X7/2
      X12 = X11+1
      exec_2(X1, X2, X12, X4, X5, X6, X7, X8, X9, X10).
exec_2(X1, X2, X3, X4, X5, X6, X7, X8, X9, X10) :-
      exec_3(X1, X2, X3, X7, X5, X6, X8, X9, X10).
exec_3(X1, X2, X3, X4, X5, X6, X7, X8, X9) :-
      2 = < X3,
      exec_4(X1, X2, X3, X4, X5, X6, X7, X8, X9).
exec_3(X1,X2,X3,X4,X5,X6,X7,X8,X9) :-
      2>X3,
      exec_5(X1, X2, X3, X4, X5, X6, X7, X8, X9).
exec_4(X1, X2, X3, X4, X5, X6, X7, X8, X9) :-
      X10 = X3-1,
      exec_{6}(X1,X2,X10,X4,X5,X6,X7,X8,X9).
exec_5(X1, X2, X3, X4, X5, X6, X7, X8, X9) :-
      look_up1_7([X7,X8|X9],X4,X10,1),
      exec_8(X1, X2, X3, X4, X10, X6, X7, X8, X9).
exec_6(X1, X2, X3, X4, X5, X6, X7, X8, X9) :-
      look_up1_7([X7,X8|X9],X3,X10,1),
```

 $exec_9(X1,X2,X3,X4,X10,X6,X4,[X7,X8|X9])$ .

```
look_up1_7([X1|X2],X3,X1,X4) :-
      X3=X4.
look_up1_7([X1|X2],X3,X4,X5) :-
      X5<X3,
      X6 = X5+1,
      look_up1_7(X2, X3, X4, X6).
exec_8(X1,X2,X3,X4,X5,X6,X7,X8,X9) :-
      look_up1_10([X7,X8|X9],X10,X4,X7,1),
       exec_{11}(X1, X2, X3, X4, X5, X6, X10).
exec_9(X1, X2, X3, X4, X5, X6, X7, X8) :-
      2 = < X4,
       exec_{12}(X1, X2, X3, X4, X5, X6, X7, X8).
exec_9(X1,X2,X3,X4,X5,X6,X7,X8):-
      2>X4,
      halt(X1, X2, X3, X4, X5, X6, X7, X8).
halt(X1,X2,X3,X4,X5,X6,X7,X8).
look_up1_10([X1|X2],[X3|X2],X4,X3,X5) :-
      X4=X5.
look_up1_10([X1|X2],[X1|X3],X4,X5,X6) :-
      X6 < X4,
      X7 = X6+1
      look_up1_10(X2,X3,X4,X5,X7).
exec_{11}(X1,X2,X3,X4,X5,X6,X7) :-
      X8 = X4-1,
      exec_9(X1, X2, X3, X8, X5, X6, X4, X7).
exec_{12}(X1, X2, X3, X4, X5, X6, X7, X8) :-
       exec_{13}(X3, X2, X4, X5, X6, X7, X8).
exec_{13}(X1, X2, X3, X4, X5, X6, X7) :-
      X8 = 2*X1,
      exec_{14}(X1, X8, X3, X4, X5, X6, X7).
```

```
exec_14(X1,X2,X3,X4,X5,X6,X7) :-
       exec_{15}(X1, X2, X1, X3, X4, 1, X6, X7).
exec_{15}(X1, X2, X3, X4, X5, X6, X7, X8) :-
       X2 = < X4,
       X6=1.
       exec_{16}(X1, X2, X3, X4, X5, X6, X7, X8).
exec_{15}(X1,X2,X3,X4,X5,X6,X7,X8):-
       le_test_33(X2,X4,X9),
       eq_test_34(X6,X10),
       andi_35(X9,X10),
       exec_17(X1, X2, X3, X4, X5, X6, X7, X8).
exec_{16}(X1, X2, X3, X4, X5, X6, X7, X8) :-
       X9 = X4-1,
       X2 = \langle X9,
       exec_{18}(X1, X2, X3, X4, X5, X6, X7, X8).
exec_{16}(X1, X2, X3, X4, X5, X6, X7, X8) :-
       X9 = X4-1,
       X2>X9
       exec_{23}(X1, X2, X3, X4, X5, X6, X7, X8).
exec_{17}(X1, X2, X3, X4, X5, X6, X7, X8) :-
       look_up1_10(X8,X9,X1,X5,1),
       exec_{20}(X1, X2, X3, X4, X5, X6, X7, X9).
exec_{18}(X1, X2, X3, X4, X5, X6, X7, X8) :-
       X9 = X2+1
       look_up1_7(X8, X9, X10, 1),
       look_up1_7(X8,X2,X11,1),
       X10 > X11,
       exec_21(X1, X2, X3, X4, X5, X6, X7, X8).
exec_{18}(X1,X2,X3,X4,X5,X6,X7,X8):-
       X9 = X2+1,
```

```
look_up1_7(X8,X9,X10,1),
      look_up1_7(X8,X2,X11,1),
      X10 = < X11,
      exec_{23}(X1, X2, X3, X4, X5, X6, X7, X8).
exec_20(X1,X2,X3,X4,X5,X6,X7,X8):-
       2 = < X3,
       exec_{24}(X1, X2, X3, X4, X5, X6, X7, X8).
exec_20(X1,X2,X3,X4,X5,X6,X7,X8) :-
       2>X3,
       exec_{25}(X1, X2, X3, X4, X5, X6, X7, X8).
exec_21(X1,X2,X3,X4,X5,X6,X7,X8):-
      X9 = X2+1,
      exec_{23}(X1, X9, X3, X4, X5, X6, X7, X8).
exec_23(X1,X2,X3,X4,X5,X6,X7,X8):-
      look_up1_7(X8,X2,X9,1),
      X9 = < X5
       exec_26(X1, X2, X3, X4, X5, X6, X7, X8).
exec_23(X1,X2,X3,X4,X5,X6,X7,X8):-
      look_up1_7(X8,X2,X9,1),
      X9>X5,
      exec_{27}(X1, X2, X3, X4, X5, X6, X7, X8).
exec_24(X1,X2,X3,X4,X5,X6,X7,X8):-
      X9 = X3-1,
      exec_{28}(X1, X2, X9, X4, X5, X6, X7, X8).
exec_{25}(X1, X2, X3, X4, X5, X6, X7, X8) :-
      look_up1_7(X8, X4, X9, 1),
       exec_{29}(X1, X2, X3, X4, X9, X6, X7, X8).
exec_26(X1,X2,X3,X4,X5,X6,X7,X8):-
       exec_{15}(X1, X2, X3, X4, X5, 0, X7, X8).
exec_27(X1,X2,X3,X4,X5,X6,X7,X8):-
```

```
look_up1_7(X8,X2,X9,1),
      look_up1_10(X8,X10,X1,X9,1),
      exec_30(X1, X2, X3, X4, X5, X6, X7, X10).
exec_{28}(X1, X2, X3, X4, X5, X6, X7, X8) :-
      look_up1_7(X8,X3,X9,1),
      exec_31(X1, X2, X3, X4, X9, X6, X7, X8).
exec_{29}(X1,X2,X3,X4,X5,X6,X7,[X8|X9]) :-
      look_up1_10([X8|X9],X10,X4,X8,1),
      exec_32(X1, X2, X3, X4, X5, X6, X7, X10).
exec_30(X1,X2,X3,X4,X5,X6,X7,X8):-
      exec_36(X2,X3,X4,X5,X6,X7,X8).
exec_{31}(X1, X2, X3, X4, X5, X6, X7, [X8|X9]) :-
      exec_9(X1, X2, X3, X4, X5, X6, X7, [X5|X9]).
exec_32(X1,X2,X3,X4,X5,X6,X7,X8):-
      X9 = X4-1
      exec_31(X1, X2, X3, X9, X5, X6, X7, X8).
le_test_33(X1,X2,1) :-
      X1 = < X2.
le_test_33(X1,X2,0) :-
      X1>X2.
eq_test_34(X1,1) :-
      X1=1.
eq_test_34(X1,0) :-
      X1=\=1.
andi_35(1,0):-
      true.
andi_35(0,1):-
      true.
andi_35(0,0):-
      true.
```

```
exec_36(X1,X2,X3,X4,X5,X6,X7) :-
X8 = 2*X1,
exec_15(X1,X8,X2,X3,X4,X5,X6,X7).
```

Respectively the analysis results are

- hpsrt\_query(X1,X2) :true.
- exec\_1\_query(X1,X2,X3,X4,X5,X6,X7,X8,X9,X10) :true.
- exec\_2\_query(X1,X2,X3,X4,X5,X6,X7,X8,X9,X10) :-X7= -2.0+2.0\*X3.
- exec\_3\_query(X1, X2, X3, X4, X5, X6, X7, X8, X9) :-X4 = -2.0 + 2.0 \* X3.
- exec\_5\_query(X1,X2,X3,X4,X5,X6,X7,X8,X9) :-X3<2.0,X4= -2.0+2.0\*X3.
- exec\_4\_query(X1, X2, X3, X4, X5, X6, X7, X8, X9) :-X4= -2.0+2.0\*X3, X3>=2.0.
- exec\_6\_query(X1,X2,X3,X4,X5,X6,X7,X8,X9):-X4=2.0\*X3,X3>=1.0.
- exec\_25\_query(X1,X2,X3,X4,X5,X6,X7,X8) :
  X3<2.0,X6=<1.0,X3-X1=<0.0,X1-0.5\*X2=< -0.0,

  X3+X4-1.5\*X7=< -0.0,X4>=2.0.
- exec\_8\_query(X1,X2,X3,X4,X5,X6,X7,X8,X9) :-X3=1.0+0.5\*X4,X4>=1.0.
- exec\_11\_query(X1,X2,X3,X4,X5,X6,X7) :-X3=1.0+0.5\*X4,X4>=1.0.
- exec\_9\_query(X1,X2,X3,X4,X5,X6,X7,X8) :-X7-0.666\*X3-0.666\*X4>=0.0.
- exec\_12\_query(X1,X2,X3,X4,X5,X6,X7,X8) :-X4-1.5\*X7+X3=< -0.0,X4>=2.0.

- exec\_13\_query(X1,X2,X3,X4,X5,X6,X7) :-X3-1.5\*X6+X1=< -0.0,X3>=2.0.
- exec\_14\_query(X1,X2,X3,X4,X5,X6,X7) :X2=2.0\*X1,X1+X3-1.5\*X6=< -0.0,X3>=2.0.
- exec\_26\_query(X1,X2,X3,X4,X5,X6,X7,X8) :-X6=1.0,X1-0.5\*X2=< -0.0,X2-X4=<0.0,
  - X4+X3-1.5\*X7=<-0.0, X4>=2.0, X1-X3>=0.0.
- exec\_21\_query(X1,X2,X3,X4,X5,X6,X7,X8) :-X6=1.0,X1-0.5\*X2=< -0.0,X2-X4=< -1.0,
- exec\_16\_query(X1,X2,X3,X4,X5,X6,X7,X8) :-X6=1.0,X1-0.5\*X2=< -0.0,X2-X4=< -0.0,

X4+X3-1.5\*X7=< -0.0, X2>=1.0, X1-X3>=0.0.

- X3+X4-1.5\*X7=<-0.0, X4>=2.0, X1-X3>=-0.0.
- exec\_18\_query(X1,X2,X3,X4,X5,X6,X7,X8) :-X6=1.0,X1-0.5\*X2=< -0.0,X2-X4=< -1.0,
  - X3+X4-1.5\*X7=<-0.0, X4>=2.0, X1-X3>=0.0.
- exec\_23\_query(X1,X2,X3,X4,X5,X6,X7,X8) :-X6=1.0,X1-0.5\*X2=< -0.0,X2-X4=< -0.0,
  - X4+X3-1.5\*X7=< -0.0, X4>=2.0, X1-X3>=0.0.
- exec\_27\_query(X1,X2,X3,X4,X5,X6,X7,X8) :-X6=1.0,X1-0.5\*X2=< -0.0,X2-X4=<0.0,
  - X4+X3-1.5\*X7=< -0.0, X4>=2.0, X1-X3>=0.0.
- ${\tt exec\_30\_query(X1,X2,X3,X4,X5,X6,X7,X8)}$  :-
  - X6=1.0, X1-0.5\*X2=<-0.0, X2-X4=<0.0,
  - X4+X3-1.5\*X7=< -0.0, X1>=1.0, X1-X3>=0.0.
- ${\tt exec\_36\_query(X1,X2,X3,X4,X5,X6,X7)}: -$ 
  - X5=1.0, X1-X3=< -0.0, X3+X2-1.5\*X6=< -0.0,
  - X1>=2.0, X1-2.0\*X2>=0.0.
- exec\_15\_query(X1,X2,X3,X4,X5,X6,X7,X8) :-X6=<1.0,X1-0.5\*X2=< -0.0,

```
X3+X4-1.5*X7=<-0.0, X4>=2.0, X1-X3>=0.0.
exec_17_query(X1,X2,X3,X4,X5,X6,X7,X8) :-
    X6 = <1.0, X1 - 0.5 * X2 = < -0.0,
    X3+X4-1.5*X7=<-0.0, X4>=2.0, X1-X3>=0.0.
exec_20_query(X1,X2,X3,X4,X5,X6,X7,X8) :-
    X6 = <1.0, X1 - 0.5 * X2 = < -0.0,
    X3+X4-1.5*X7=< -0.0, X4>=2.0, X1-X3>=0.0.
exec_24_query(X1,X2,X3,X4,X5,X6,X7,X8) :-
    X6 = <1.0, X1 - 0.5 * X2 = < -0.0,
    X3+X4-1.5*X7=<-0.0,X3>=2.0,X4>=2.0,X1-X3>=0.0.
exec_{28}query(X1,X2,X3,X4,X5,X6,X7,X8):-
    X6 = <1.0, X4 - 1.5 * X7 + X3 = < -1.0,
    X1-0.5*X2=<-0.0, X4-1.5*X7=<-2.0,
    X4>=2.0, X1-X3>=1.0.
exec_29_query(X1,X2,X3,X4,X5,X6,X7,X8) :-
    X3<2.0, X6=<1.0, X3-X1=<0.0, X1-0.5*X2=<-0.0,
    X3+X4-1.5*X7=<-0.0,X4>=2.0.
exec_32_query(X1,X2,X3,X4,X5,X6,X7,X8) :-
    X3<2.0, X6=<1.0, X3-X1=<0.0, X1-0.5*X2=<-0.0,
    X3+X4-1.5*X7=<-0.0,X4>=2.0.
exec_31_query(X1,X2,X3,X4,X5,X6,X7,X8) :-
    X6 = <1.0, X1 - 0.5 * X2 = < -0.0,
```

Next the mapping between predicates and program points is shown.

X7-0.666\*X3-0.666\*X4>=0.666.

Predicate	Program point	Predicate	Program point
hpsrt		exec_17	22
exec_1	1	exec_18	15
exec_2	2	exec_20	23
exec_3	3	exec_21	16
exec_4	4	exec_23	17
exec_5	6	exec_24	24
exec_6	5	exec_25	26
exec_8	$\gamma$	exec_26	18
exec_9	9	exec_27	19
exec_11	8	exec_28	25
exec_12	10	exec_29	27
exec_13	11	exec_30	20
exec_14	12	exec_31	29
exec_15	13	exec_32	28
exec_16	14	exec_36	21
halt	30		

Finally, the annotated imperative program is shown below.

{1<2,r=2\*1-2}

else

```
6)
        k := t[r];
                  \{2*1=2+r,r>=1\}
7)
        t[r] := t[1];
                  \{2*1=2+r,r>=1\}
8)
        r := r-1;
   endif;
                  {3*n-2*1-2*r>=0}
9) while r \ge 2 do
                  \{r \ge 2, 2 \le r - 3 \le n + 2 \le 1 \le 0\}
10)
      i := 1;
                  \{1>=2,2*1-3*tm+2*i<=0\}
11)
        j := 2*i;
                  {j=2*1,2*1-3*tm+2*i<=0}
12)
        tm := 1;
                  \{tm <= 1, 2*i-j <= 0, 2*1+2*r-3*n <= 0, r>= 2, i>= 1\}
        while (j<=r and tm=1) do
13)
                  \{tm=1,2*i-j<=0,j<=r,2*l+2*r-3*n<=0,r>=2,i>=1\}
14)
            if j \le r-1 then
                  \{tm=1, 2*i-j \le 0, j \le r-1, 2*1+2*r-3*n \le 0, r \ge 2, i \ge 1\}
                if t[j] < t[j+1] then
15)
                  \{tm=1, 2*i-j \le 0, j \le r-1, j \ge 1, 2*1+2*r-3*n \le 0, i \ge 1\}
16)
                   j := j+1;
               endif;
            endif:
                  \{tm=1,2*i-j \le 0, j \le r,2*l+2*r-3*n \le 0,r \ge 2, i \ge 1\}
17)
            if k>=t[j] then
                  \{tm=1,2*i-j<=0,j<=r,2*l+2*r-3*n<=0,r>=2,i>=1\}
18)
               tm := 0
            else
                  \{tm=1,2*i-j \le 0, j \le r,2*l+2*r-3*n \le 0,r \ge 2, i \ge 1\}
```

```
19)
                  t[i] := t[j];
                     \{tm=1, 2*i-j \le 0, j \le r, 2*1+2*r-3*n \le 0, i \ge 1, i \ge 1\}
20)
                  i := j;
                     \{k=1, i \le 1, 2*1+2*j-3*tm \le 0, i \ge 2, i \ge 2*j\}
21)
                  j := 2*j
               endif;
           endwhile;
                     \{tm \le 1, 2*i-j \le 0, 2*1+2*r-3*n \le 0, r \ge 2, i \ge 1\}
22)
           t[i] := k
                     \{tm <= 1, 2*i-j <= 0, 2*1+2*r-3*n <= 0, r>= 2, i>= 1\}
23)
           if 1>=2 then
                     \{tm \le 1, 2*i-j \le 0, 2*1+2*r-3*n \le 0, r \ge 2, 1 \ge 2, i \ge 1\}
24)
               1 := 1-1;
                     \{tm \le 1, 2*i-j \le 0, 2*1+2*r-3*n \le 0, 2*r \le 3*n-4, r \ge 2, i \ge 1\}
               k := t[1];
25)
           else
                     \{tm <= 1, 1 < 2, 1 <= i, 2 <= i, 2 <= i, 2 <= 1, 2 <= 2, r >= 2\}
26)
               k := t[r];
                     \{tm \le 1, 1 \le 2, 1 \le i, 2 \le i \le j, 2 \le 1 + 2 \le r - 3 \le n \le 0, r \ge 2\}
               t[r]:= t[1];
27)
                     \{tm <= 1, 1 < 2, 1 <= i, 2 + i <= j, 2 + 1 + 2 + r - 3 + n <= 0, r >= 2\}
28)
               r := r-1;
           endif:
                     \{tm <= 1, i <= 2*j, 3*n-2*l-2*r>= 2\}
           t[1] := k;
29)
     endwhile;
30) halt
```

In spite of the fact that these are not the same results as those obtained in [CH78] some array accesses could be verified within bounds by projection on the constraints of the annotations and some information from the source program.

Up to now the examples given used the domain of linear arithmetic expressions. However, this is not the only direction that could be taken for program analysis and specialisation. Next, we elaborate on some possible directions that analysis and specialisation of CLP programs could take, with the ensuing transference of this results to imperative programs.

### 6.3 Some possible analyses and transformations

The discussion and examples shown in this chapter serve to establish our basic principles and framework. Having done this, we now sketch a number of applications which differ only in the abstractions employed. Some of the directions that are here proposed may overlap, as regards program transformation and analysis; however we will present them as different application fields, namely program analysis and program specialisation.

#### 6.3.1 Program Specialisation

It is well-known that 'standard' transformation techniques for program specialisation only provide linear speedups. There are other transformation strategies that could improve the residual CLP programs, namely, those used in unfold-fold program transformation [PPR97, BCD90]. More aggressive argument filtering in the residual programs could remove unnecessary variables more like the programs obtained from SSA form [App98].

At preprocessing time, data-flow and control-flow from the imperative program could be gathered to aid specialisation (e.g. live variables, points-to analysis, visible variables, dependence graphs, etc.)

SSA form. Single-Static Assignment form was developed to improve the representation of def-use chains and thus make single procedure program optimisations clean and efficient. To achieve single-assignment a new variable name is created for each assignment to the variable. That is, in SSA form, each variable in the program is

assigned to only once; so single-assignment is a *static* property of the program text, not a dynamic property of program execution. For a program with no jumps this is easy. But where two control-flow edges join together, carrying potentially different values of some variable, we must somehow merge the two values. In SSA-form this is done by a notational trick, the  $\phi$ -function. Rather than assigning a value from one of the incoming control-flow edges both values are passed as arguments to a  $\phi$ -function which in turn will produce the appropriate result. In practice  $\phi$ -function definitions need not be known; they are an aid in showing a semantics preserving transformation (from non-SSA-form to SSA form). SSA form is used in compiler optimisations for dead-code elimination, conditional constant propagation, construction of the register interference graphs and in control dependence. Part of this applications are tasks aimed at by specialisers hence their potential use in specialisation.

#### 6.3.2 Program Analysis

In the same way as it was suggested in Chapter 2 abstract compilation could be used to obtain other analyses. The important point here is the choice of the domain. Some possible domains and uses are listed next.

- By abstracting lists by their length (assuming arrays are represented using lists in the semantics) in conjunction with CRUL approximations, programs using matrix calculations could be verified with respect to the compatibility of their dimensions.
- Using Herbrand constraints in conjunction with the domain of CRUL approximations [SG98a] for sorting programs opens the possibility of verifying the 'sortedness' of the program's output.
- For imperative programs using pointers and user-defined data structures residual CLP programs could be analysed using RUL or CRUL approximations to obtain some shape information.
- Allocating memory (in a semantics with locations) using integers and analysing

6.4. SUMMARY 149

the residual CLP programs using linear arithmetic constraints could reveal information about boundedness of the memory required (if the counter to the next free location kept by the semantics is found to be bound on the program exit points).

- Sharing analysis<sup>5</sup> of residual CLP programs (using locations) may be related to alias analysis of pointers in imperative programs.
- Termination is an area of intensive research in CLP. There are some tools for automatic termination analysis of CLP programs and thus these results could be readily transferred to imperative programs.

In summary, existing domains for program analysis could be reused or new ones could be constructed for the application required.

#### 6.4 Summary

The analyses here shown are restricted to the domain of linear arithmetic expressions, though any abstract domain could be used. Program points are used to map specialisation results, on the one hand, and analysis results, on the other hand, to the original source program. For program analysis there is flexibility with respect to where information should be gathered by the respective analyser. That is, program points serve as tags to which we want the analysis to display information in the imperative program. In this regard, program points could be placed and removed anywhere within the program thus leading to potentially smaller programs as a result of specialisation and as input to analysis. Also, through abstractions several features of the residual programs could be made to 'disappear' for the analysis, thus simplifying its complexity. In the examples of sorting given, arrays could be systematically replaced by an appropriate term disregarded by the analysis.

<sup>&</sup>lt;sup>5</sup>In the form of determining which variables can possibly be bound to terms containing a common variable.

# Chapter 7

## Related Work

One contribution [CH78] on analysis of imperative programs and another two [Mou97, Ros89] in specialisation bear a resemblance to what we achieve in this thesis. Here we will describe some features of these contributions and some comparisons are provided in chapters 2 and 3. Other methods in the literature are related, in different ways, to our framework and will be mentioned here too.

#### 7.1 Program Analysis

The first practical results on imperative languages for deriving linear equality or disequality relations among the variables of a program is due to Cousot and Halbwachs [CH78]. Their analysis method is based on abstract interpretation and a prototype system was implemented in Pascal. Flow-charts are the execution models that assign meaning to programs. The analysis associates a set of restraints with every outgoing arc in the flow-chart. For every node in the flow-chart a transformation specifies the assertion associated with the output arc(s) of the node in terms of the assertions in the input arcs. A different transformation is given for every different statement in the program, i.e. a transformation is associated with assign statements, another one with if-then-else conditionals and another one is associated with loops. Starting from no assertion in all arcs of the flow-chart the analysis consists in propagating the input assertions through all the possible paths in the flow-chart until a

fix-point is reached. For multiple entry nodes their output assertion is given by the upper bound of the assertions coming from every input arc. A convex hull of the input linear restraints approximates the upper bound. In addition to applying its associated transformation widening is computed for loop nodes. Widening is introduced to guarantee termination in the presence of loops. Also, it is only computed at one distinguished node of every loop in the flow-chart. Examples are given where statements involving accesses to array elements are ignored. Any nonlinear restraint is ignored too.

Later on in [Cou97] the author poses the possibility of deriving different static analysers, through abstract interpretation, parameterised by the programming language as well as its semantics (e.g. operational, axiomatic, denotational, etc.). This approach goes a step further than our proposal since it is parametric with respect to the programming language semantics.

The next two contributions could be expressed as instances of the general proposal above, where the language semantics is natural operational semantics and denotational semantics, respectively. By contrast, our proposal uses structural operational semantics and the analysers derived depend on the existing tools for analysing CLP programs rather than the choice of semantics style.

In a slightly different way to abstract interpretation above the authors of [GM97] show how to obtain a static analyser for a non strict functional language. Such a static analyser is derived by successive refinements of the original language specification, natural operational semantics. The possible analyses obtained by the analyser derived with this method depend on the program property sought. This program property should be provided in advance. It appears that this technique has been applied to obtain some classical compiler analyses of programs in the sense of [ASU86]. The main goal of this work is to put forward software engineering techniques to support the design of new analyses.

Using denotational semantics as a departing point, in [Nie89] the author lays out the theory of abstract interpretation using two-level semantics. Two-level semantics had been previously used in [NN88] to describe code generation. A brief summary of both can be found in [Nie96]. Denotational definitions for the semantics of Pascal-like languages are given making explicit the distinction between compile time computations and run time computations, hence the two levels of the metalanguage. Here the semantic definitions provide a means to translate statements in the desired language to a two level metalanguage. Furthermore, the resulting code describes the run time computations specified by the semantics whereas compile time computations have been carried out during translation. For program analysis, an appropriate interpretation of the run time metalanguage aids the analysis by giving a nonstandard semantics to run time constructs.

The following contributions are proposals for specific analysers rather than general techniques, as above. Along each description we describe how they are related to our work.

Deriving linear relationships between the variables of a program is a more general problem than finding ranges of variables throughout a program. For this reason the following description is related to our work.

In [VCH96] the authors describe a technique based on the style of abstract interpretation to statically estimate the ranges of variables throughout a program. It generalises the constant propagation problem. Their implementation has been realised in the context of an optimising/parallelising compiler for C. This is an example of using a variant of operational semantics to describe the abstract interpreters for static analysis of imperative programs. Independently, an implementation based on natural semantics was developed by [Gau97] for static pointer analysis in a subset of the C programming language.

Our contribution to program analysis may be thought of as a problem reduction because we translate a problem in one domain (imperative programs) to another domain (CLP programs). However, the experiments here shown do not change the analysis problem. We aim to find linear relationships between the variables of a program (either the imperative program of its CLP equivalent). Nonetheless, we believe that our framework could be used for problem reduction as described in the following piece of research.

Analysis by problem reduction is provided in [RS98]. The authors convert the problem of identifying dependencies among program elements to a problem of identifying may-aliases. The transformation output is a program in the same language as the input program where may-aliases in the transformed program yield information directly translatable to control-flow dependencies between statements in the original program. In a similar way, the authors claim that control-flow dependencies in the transformed program have a direct reading as may-aliases in the corresponding program.

In [SFRW98] it is shown how to use logic programs to aid the analysis of imperative programs with pointers. The formalism is shown for the case of the pointer equality problem in Pascal. During analysis a set of assertions, represented as unary clauses, is updated according to the meaning of the program statement evaluated, the update operation designated, and a set of consistency rules. The update operations resemble operations in deductive databases.

Presumably this technique could be used in our CLP programs to obtain the same results. A possible advantage over doing it in the imperative domain is that the program is already in a logic language and the assert and retract operations could be inserted in the appropriate places into the CLP program. Also some abstraction of the CLP program may be needed. The feasibility of this connection is a subject of our current research.

Recently, Gupta [Gup99] uses denotational semantics as the specification method to write CLP interpreters. He argues that program verification could be achieved and compilers could be obtained from CLP interpreters based on denotational semantics. Arguably compilers could be generated by specialising CLP interpreters using a specialiser (Mixtus [Sah91]) on his interpreters. In this regard, the 'compiled code' would be in CLP rather than a low-level language. By contrast, we address the problems of relating the results back to the source language for specialisation and analysis rather than compiling. Potentially our CLP specialisers could be used for the applications he describes.

#### 7.2 Program Specialisation

Here we will consider work on partial evaluation or specialisation of imperative languages. There has been important work [CD91, JGS93] on the specialisation of imperative languages expressed as denotational definitions, to functional languages. Nevertheless, we will only discuss some of the work which focuses on source-to-source specialisation of imperative languages.

A similar approach to ours is presented by Ross [Ros89]. Logical semantics provide definitions of imperative program constructs associating a relation with each program construct. Imperative statement composition (sequencing) corresponds to the composition of relations. Thus, a logic program is derived from an imperative program by rewriting each program construct as a predicate, whose definition is made of Horn clauses, and composing such relations according to the control flow of the imperative program. Such a logic program describes the natural operational semantics of the imperative program with predicates denoting the input-output relation of variable values for each program construct. A preprocessing of the resulting logic program allows the removal of dead code in the presence of unconditional jumps (gotos). Later that logic program is partially evaluated and the resulting program analysed to reconstruct a partially evaluated imperative program. During reconstruction some literal reordering may be necessary in order to preserve the operational semantics of the residual imperative program.

An example of the same approach using functional programs instead of logic programs is the work of B. Moura [Mou97]. An existing off-line partial evaluator is used together with other existing tools to obtain specialisation. Functional programs are derived from denotational specifications of the semantics of the subject imperative language. This translation assumes that single procedure imperative programs converted to SSA form [CFR<sup>+</sup>91] can be regarded as a functional program. Later, Appel [App98] used the same argument to show that SSA form is equivalent to functional programs with nested scope of variables. By staging SSA with a continuation passing style transformation (CPS) the author is able to produce functional programs thus

reducing the complexity of specialisation. Specialisation is carried out by an existing specialiser [Con93] for a higher order functional language. Imperative specialised programs may be recovered from specialised functional programs by applying the inverse transformations of CPS and SSA. Pointers and mutable data structures are represented by closures obtained by a closure analysis. The programming language is an slightly adapted version of a subset of the C programming language [KR88] with if-then-else conditionals, assignments, and subroutines with recursion.

Because SSA form was originally developed for single procedure programs it is arguable whether the imperative language treated allows programs with multiple procedures. No mention is made to any modification to the Schism specialiser to allow the systematic recovery of imperative programs from residual functional programs. However, the systematic labelling as runtime or compile-time of the functions used to specify the transformations (SSA and CPS) suggest an induced binding time provided from the definitions. This is similar to the specialisation of imperative programs to functional programs shown in [CD91] by splitting the semantic functions into static and dynamic. It appears that the method to obtain the specialiser for imperative programs is not general enough since the transformations required before and after specialisation are specific to the language. To compute the SSA form of a program some control flow and data flow information embedded in the program is required. In addition the closure analysis provides ad-hoc information about certain functions in the program.

Appel [App98] first obtains a functional representation of the SSA form of an imperative program where all variables of the program are within scope at all times. Later on, he rewrites such a functional program to one where not all variables are syntactically present at every program point but left implicit in the semantics of the newly restructured program (nested scope of variables). After specialisation of our one-state operational semantics we obtain constraint logic programs that resemble the intermediate functional programs obtained by Appel. In addition, some of the variables may be eliminated because they are useless and would be eliminated through the argument filtering of the specialiser. In logic programs nested scope cannot be

represented because the scope of a variable is the clause where it occurs.

Analysis and transformation of programs in ANSI C was the target of Andersen's work in [And94], whose present form is briefly discussed in Glenstrup et al. [GMS98]. The specialiser is automatic and generates generating extensions in C.

C program specialisation is achieved in several steps. Using a context-insensitive binding-time analysis the source C program is analysed. The analysed C program is transformed thus producing a generating extension program. Attaching the appropriate libraries and the partial input, the generating extension is executed. The output of such an execution is a specialised C program. A set of C libraries describe the transformations that the generated extension contains.

Recent work on specialisation of C programs is given in [HNC97]. The specialiser handles a considerable subset of the C language. Their binding time analysis is polyvariant. Recursion has not been considered into their programs. Constraints have not been integrated into their analysis phase though.

In [KKZG95] a partial evaluator for a subset of Fortran 77 is described. Partial evaluation is a 4-step process: Translation from the Fortran program into a low level code (Core Fortran); monovariant binding-time analysis and annotation of the program statements as static or dynamic; specialisation of the annotated low level code; and translation into the source language, Fortran 77. The partial evaluator was written in Fortran. The monovariant binding-time analysis used results in little constant propagation.

In a slightly different approach to partial evaluation Blazy et al. [BF96] adapt a partial evaluator for Fortran to aid program understanding through partial evaluation. A partial evaluator is modified so that loops are never unfolded neither procedure calls inlined thus residual programs preserve most of the structure of the source program. Through inference rules coded in Prolog the authors specify propagation of constants inter-procedurally. Memoing allows them to obtain some control of polyvariance, but they fail to provide a terminating strategy for their online partial evaluator.

In [Mey91] a technique for partial evaluation of programs in a subset of the Pascal programming language is presented. Their specialisation method consists of a set of

rules for transforming each construct in the language considered. It is a combination of off-line and on-line methods for specialisation.

There is no clear separation between the semantics representation (semantics-based interpretation/execution), and the partial evaluation.

Using an on-line approach called symbolic execution Coen-Porisini, De Paoli, Carlo Ghezzi and Mandrioli [CPPGM91] present a method for specialising programs in a subset of the Ada programming language. Their method divides specialisation into two functions, whose definition is given for each program construct. One for symbolic execution and the other one for constructing the specialised program. Symbolic execution is achieved using a pair  $\langle State, PC \rangle$ : in State variable names and their contents are stored; PC contains a first order formula regarded as a constraint on the variables of State. Conditionals are symbolically executed using a theorem prover to decide whether the current formula implies the guard in the conditional, or its complement. The quality of the specialised code depends on the user helping the specialiser (e.g. providing invariants, helping the loop folder or helping the theorem prover). Standard data-flow analysis techniques [ASU86] are applied to the imperative program to aid the specialisation. Both prototypes above were implemented in Lisp.

In [NP92] the authors describe an off-line technique to partially evaluate imperative programs for hard real-time applications. Hard real-time programs must obey several restrictions on the program behaviour such as predictability. The authors propose a language as well as a partial evaluator.

Although most of the contributions are labelled as techniques for specialisation of imperative languages, only one language is explored in each paper, and it is not clear how to extend the techniques to other languages. Besides, we believe the methods above do not commit to a semantics-based approach. None of them take the semantics of the language studied as a basis for transforming the language programs, except [Ros89]. Most of the methods above are off-line with the exception of [BF96, CPPGM91, Mey91, Ros89]. Our method is based on an on-line partial evaluator, which in principle allows greater specialisation.

# Chapter 8

## Conclusions

In this thesis it has been argued that tools for analysis and specialisation of CLP can be applied to imperative programs. The main technique for achieving this was by encoding imperative program semantics as CLP programs. Next, the following discussion intends to show to what extent the methods and results shown here demonstrate the feasibility of this approach to analysis and specialisation of imperative programs. In the last section we assess future directions for research in this area.

#### 8.1 CLP Analysis

Analysis of CLP programs is a subject of active research. As a result, some generic tools for analysing CLP programs exist, and better tools are constantly under development. Here we have reviewed analysis for CLP programs using an existing general framework for bottom-up analysis. The discussion focussed on approximation techniques using constraints of various kinds, though the possibilities of using abstract domains also exist. An advantage of using a general framework for program analysis, such as that of [Sağ98], is that new domains could be readily integrated and thus a new analyser produced. A disadvantage of this particular framework is that it is based on declarative semantics and there are some program properties (e.g. properties associated with the procedural semantics of a program) which would be difficult if not impossible to model in this apparently restricted framework. However, other

frameworks and tools could be used if non-declarative semantics were needed.

Besides using standard CLP techniques for analysis we have made some advances in this area. A new widening for the domain of constrained regular approximations was proposed. The novelty of this widening is that arithmetic constraints are considered, in addition to the dependency and inclusion between regular predicates. We argued for a possible way to improve the precision of the upper bound of RUL predicates. It attempts to solve the inherent limitations of top-down deterministic descriptions of term grammars. This improvement provides some form of nondeterministic top-down approximation using top-down deterministic descriptions. This is achieved by preventing tuple distributive union where some information about the subject program is known. Later this technique was used to improve the results of specialisation of interpreters for imperative programs with (recursive) procedures.

#### 8.2 Writing semantics as CLP programs

Results on analysis and specialisation of CLP may become available for use in imperative programs by specialising semantics-based interpreters for imperative programs. However, specialisers and interpreters do not always mix well. On the one hand, specialisation is a general technique for program improvement and is highly dependent on the style programs are written. On the other hand, interpreters could be written in any style thus making difficult their use in specialisation. To match these two we identified the style of structural operational semantics (as useful and general enough) to write semantics. In order to reduce the complexity of analysis and transformation we use a slightly modified form of structural operational semantics (SOS). This modified form of SOS semantics is the result of specialising the composition of statements to the instructions of the imperative language modelled. The implementation of this semantics as CLP programs aids specialisation by leaving implicit the operational semantics in the meaning of CLP clauses; carrying only one state in the predicate associated with the semantics function; hence the name one-state small-step operational semantics. We have showed how to implement most features

of standard imperative languages using this style of writing semantics. In order to be able to model procedures with parameters and block declarations we made use of the well-known concept of the stack of activation records. This concept makes it possible to describe most programming language semantics in the style of one-state small-step operational semantics.

#### 8.3 Specialisation of semantics-based interpreters

By specialising such semantics-based interpreters for imperative programs with respect to an imperative program we obtain residual CLP programs thus providing the connection between imperative programs and CLP programs. However it is not clear how to relate imperative programs to the CLP programs obtained from specialisation. To this end, some tuning of the specialiser was needed to generate residual programs where some structure of the source imperative program could be recognised. Static information from the imperative program turned out to be the best choice in aiding the specialisation. In particular, program points served this purpose by guiding the unfolding rule and the generalisation operations of our specialisers. Specialisation using program points produces residual programs where each different program point in the imperative program is associated with at least one different predicate in the residual CLP program. This resulted in a small modification of the unfolding rule and the generalisation operation of existing specialisers for CLP programs. Now imperative programs may be translated to CLP programs (through specialisation) and analysis of such residual CLP programs could be systematically related to the source imperative program. Furthermore, if some input constraints were available during specialisation of the semantics-based interpreter, specialisation of the source imperative program could be achieved by specialisation of CLP programs.

However, for specialisation of imperative programs we must provide the results of specialisation in the source imperative language rather than in CLP. To reconstruct specialised imperative programs from specialised CLP programs some properties of the residual programs generated by specialisation were exploited in conjunction with

some techniques for associating imperative statements with graph shapes. Specifically, residual programs were associated with labelled directed graphs and those graphs with abstract syntax trees for imperative programs. The use of program points for specialising imperative programs produces monovariant imperative program specialisation. When program points are combined with trace terms (an abstraction used in polyvariant logic program specialisation) some polyvariance at the imperative level may be obtained. The reconstruction of imperative programs was discussed for the class of single procedure programs.

#### 8.3.1 Languages with multiple procedures

For single procedure languages analysis and transformation results of CLP can be transferred to imperative languages; unfortunately most programs are not written in this style. The natural extension of these results is to add procedures and parameters. However specialisation of programming languages with local scope poses new problems. One of them stems from the fact that the list of variables throughout execution may change in size. Specialisation of environment handling code is highly dependent on knowing the list of variables at every program point. In order to produce residual programs where the interpretation layer associated with looking up and update of the imperative variable contents is minimal, the list of imperative variables must be known during specialisation. Another problem is that the stack of activation records contains important information about the control flow of the program. If the information contained in the stack is lost during generalisation, specialisation amounts to returning the same program i.e. the original semantics-based interpreter.

#### Specialisation with Constraints

We found that specialisation of CLP programs representing imperative language semantics required some special techniques. Constraint reasoning capabilities were integrated into a generic specialisation algorithm. In particular, a general algorithm for specialisation of logic programs was extended to handle constraints. Atoms are

replaced by pairs consisting of the atom and a set of constraints on the variables of the atom. The three constraint domains presented for program analysis (namely, linear arithmetic expressions, RUL programs and CRUL programs) were used for this purpose. This extension enables some information propagation similar to that obtained by analysis, thus providing some integration of abstract interpretation and program specialisation. The operations on the constraint domains lend themselves easily to the needs of specialisation.

A key component of the specialisation algorithm is the generalisation operation. During generalisation based on the most specific generalisation, information which is not the same position-wise in both arguments to the operation is lost. By incorporating constraints into specialisation more information may be preserved upon generalisation than with the general approach using the msg operator. In the domain of linear arithmetic expressions the upper bound is modelled by the convex hull operator. RUL constraints, in turn, are specially useful for programs containing accumulators. Finally, a combination of both naturally inherits the benefits associated with them, thus increasing the amount of information preserved during generalisation. During unfolding constraints are used to prune the search tree through consistency checks, collecting other constraints. Yet, these apparently conflicting aims (preserving information, and pruning) are achieved with specialisation using constraints.

By using constraints the specialiser could preserve more information than with the previous approach, where the only source of information was the atom. Upon generalisation with RUL constraints the contents of the list of visible variables for a program point were preserved into the constraints, and so were combinations of them. In a similar way the contents of the stack of activation records were preserved. However, this was not precise enough for the needs of specialisation; more precision of the approximation was required. Note that these two problems are related to the fact that the upper bound in the domain of RUL programs is tuple distributive; hence the additional information produced after generalisation. To cope with these problems using the same specialisers (using either the domain of RUL programs or CRUL programs) we integrated some information about the imperative program into the

semantics-based interpreter. We assume a prior analysis of the imperative program to get the variables within scope at each program point. This information forces the upper bound of RUL constraints to keep separate the information about the control of the imperative program (contained in the stack) and the list of variables of the program. In addition, the list of variables is used to produce the required polyvariance thus preventing information loss due to tuple distributivity. Hence, specialisation could be performed eliminating most of the interpretation layer of semantics-based interpreters for imperative programs.

It is worth noting that constraints were integrated into the specialiser in an attempt to improve specialisation for semantics-based interpreters of imperative programs but this integration has application to CLP programs in general, not only to interpreters.

# 8.4 Limitations and Possible directions for Future Work

- Specialisation with program points stops unfolding when a predicate with a program point is met. If the aim of specialisation is to use analysers where only some points (not all) in the imperative program are relevant then program points should be placed accordingly and thus unfolding proceeds potentially producing smaller programs more specialised. For instance, for termination analysis most program points could be removed, depending on how informative the analysis results are.
- RUL constraints are very useful for specialisation (especially for programs with accumulators). Unfortunately, the descriptive power of RUL programs is that of top-down deterministic tree automata. Because the set of regular tree languages expressed by top-down deterministic tree automata is a proper subset of the languages recognised by top-down nondeterministic tree automata it would be desirable to extend the expressive power of RUL constraints to nondeterministic

regular tree automata. This would simplify the formulation of our framework since tuple distributivity wouldn't be a problem. To our knowledge the domain for RUL programs with these characteristics has not been developed in any practical form.

- Other constraint domains could be used in addition to the ones presented, provided they have a satisfiability procedure (and some widening and narrowing where required). The inherent problems are that most analysers work well on small programs and the programs generated by our specialisers could be of substantial size. This has to do with scalability of the analysers and not with the programs we generate because we believe that the complexity of analysis is substantially reduced by specialisation. Alternatively, some extensions to the current specialisation proposal could enable the decomposition of residual into small programs corresponding to meaningful components of the source imperative program where analyses could be tried instead of using the whole residual program.
- Our proposal for imperative program recovery remains to be improved, for the case of multiple procedures. For languages with procedures and parameters by reference and by value it is difficult to decide from the callgraph and the residual program which parameters have been specialised since there would be a program point in the procedure's call and the first instruction of the procedure's body, which is not expressive enough. Between these two points the parameter passing could be represented by a complex CLP program, not only a conjunction of constraints as we would expect for flowgraph CLP programs. This is the case of languages with dynamic binding or objects (e.g. Java). In the worst case the procedure's call is made with the same arguments to a possibly specialised version of that procedure where the specialisation is in the body not in the parameters of the call.
- Constraint approximations might be an expensive tool for specialisation, and we use them everywhere in specialisation. As we noted in the exposition they

are useful for some arguments of the semantics and irrelevant for others. Thus it would be good to use the approximations more selectively during generalisation to reduce the complexity of specialisation; similarly for analysis. In the current proposal we strove to make the transfer of results from CLP to imperative languages as simple as possible demanding from the user the least knowledge about the underlying complexities of the framework. By contrast, making a selective use of constraint approximations might require a deeper understanding of the implementation of the analysers and the specialisers since it appears to be more implementation dependent.

- Throughout this exposition we only discussed single threaded languages. Concurrent languages is another avenue in which the framework could be applied. Extension of the semantics to concurrent languages would be possible, in principle, since the semantics of concurrency could potentially be expressed in CLP too.
- Programming language semantics could be very complex thus posing difficult problems to on-line specialisation. In this proposal we aimed at providing a framework for specialising and analysing imperative programs using CLP tools. The analysis phase of the specialisers did most of the work to discover control and data-flow information to perform specialisation. This is the standard approach to on-line specialisation in logic programming. Because high-level languages are growing more complex, this approach would not scale properly. Hence, it would be desirable to complement the specialiser with control and data-flow information about the imperative program to be specialised at the price of making our approach language dependent. For these reasons, it is suggested to integrate our tools into an optimising compiler, since intermediate languages are less complex and the analysis required can be provided by the compiler. Note that these analyses might be performed on 'abstract semantics' also expressed as CLP programs.
- It would be worth investigating the connection between internal graph represen-

tation in compilers with our callgraphs. This connection opens the possibility of generating CLP programs from internal graph representations of intermediate code thus making available the analysis tools of CLP to imperative languages without an explicit encoding of semantics as CLP programs.

- In order to ensure termination of CLP specialisation (linear constraints, RUL, and CRUL constraints) some extra techniques are required (e.g. homeomorphic embeddings, depth-bounds, etc.). Note that program points ensure termination for imperative program specialisation. Here we are referring to specialisation of CLP programs in general.
- The widening operator for CRUL constraints remains to be thoroughly investigated. The search for true widenings is a subject of our current research.

# Appendix A

# Semantics of three imperative languages

Here we show the semantics-based CLP interpreters for three different languages together with an example program of specialisation. Their are presented in increasing order of complexity.

#### A.1 A small language

The semantics of a small language for (recursive) procedures with block declarations follows.

```
exec(N,[p(_,halt)],rgrs_edo) :-
       write(N).
exec(N,[p(\_,endblock)],frame(F,TrP)) :-
       code id cont(F.N.TrP).
exec(N,[p(\_,endproc)],frame(F,TrP)) :-
       code_id_cont(F,N,TrP).
exec(N,[p(_,skip)|P],TrP) :-
       exec(N,P,TrP).
exec(N,[p(_,ifte(B,S1,_))|P],TrP) :-
       b_expr(N,B,tt),
       append(S1,P,SP1),
       exec(N,SP1,TrP).
exec(N,[p(_,ifte(B,_,S2))|P],TrP) :-
       b_expr(N,B,ff),
       append(S2,P,SP2),
       exec(N,SP2,TrP).
\texttt{exec([N,Nv,Np,Npst],[p(\_,assign(X,AE))|P],TrP)} \;\; :- \;\;
```

```
\mathtt{replace} (\texttt{1} \, , [\texttt{N} \, , \texttt{N} \, \texttt{v}] \, , [\texttt{N} \, , \texttt{N} \, \texttt{v} \texttt{1}] \, , \texttt{X} \, , \texttt{Vae}) \, ,
        exec([N,Nv1,Np,Npst],P,TrP).
exec(N,[p(L,while(B,S1))|P],TrP) :-
       b_expr(N,B,tt),
        append(S1,[p(L,while(B,S1))|P],SP1),
        exec(N.SP1.TrP).
exec(N,[p(_,while(B,_))|P],TrP) :-
       b_expr(N,B,ff),
        exec(N.P.TrP).
\verb|exec([A,B,Prcs,Pstms]|,[p(L,call(Prc))|P],TrP):-\\
       replace(2.[Prcs.Pstms]. .Prc.S).
        code id(L.A.(A.[].P).F).
        exec([A,B,Prcs,Pstms],S,frame(F,TrP)).
exec([Va|VlPr], [p(L,block(Dv,S))|Prg],TrP):-
        vars([Va|VlPr],N1,V,Dv),
        code_id(L,Va,(Va,V,Prg),F),
        exec(N1,S,frame(F,TrP))
b_expr(N,tt,tt).
b_expr(N,ff,ff).
b_expr(N, eq(A1, A2), TV) :-
        a_expr(N,A1,V1),
        a_expr(N, A2, V2),
        eq_test(V1,V2,TV).
b_expr(N,gt(A1,A2),TV) :-
        a_expr(N,A1,V1),
        a_expr(N,A2,V2),
        gt_test(V1,V2,TV).
b_expr(N,le(A1,A2),TV) :-
        a_expr(N,A1,V1),
        a_expr(N, A2, V2),
       le_test(V1,V2,TV).
b_expr(N, and(A1,A2),TV) :-
        b_expr(N,A1,V1),
        b_expr(N,A2,V2),
        andi (V1.V2.TV).
b_expr(N, not(A),TV) :-
        b_expr(N,A,TV1),
        noti(TV1,TV).
```

 $a_expr([N,Nv,Np,Npst],AE,Vae)$ ,

```
restore_vars(Vs,Vals,Vs1,Vals1,V) :-
a_expr(N,plus(A1,A2),V) :-
                                                                                           restrict(Vs,Vals,Vs1,Vals2),
        a_expr(N,A1,V1),
                                                                                           rst_vals(V,Vs1,Vals2,Vals1).
        a_expr(N, A2, V2),
        V is V1+V2.
                                                                                   restrict([],[],_,[]).
a_expr(N,times(A1,A2),V) :-
                                                                                   restrict([X | Xs], [V | Vs], Xs1, Vs2) :-
        a_expr(N,A1,V1),
                                                                                           member1 (X.Xs1.YN).
        a_expr(N, A2, V2),
                                                                                           keepval (YN.V.Vs1.Vs2).
        V is V1*V2
                                                                                           restrict(Xs.Vs.Xs1.Vs1).
a_expr(N,minus(A1,A2),V) :-
        a_expr(N,A1,V1),
                                                                                   keepval (ves.V.Vs1.[V|Vs1]).
        a_expr(N, A2, V2),
                                                                                   k = pval (no, _, Vs1, Vs1).
        V is V1-V2
a_expr(N,I,I) :-
                                                                                   rst_vals([],_,Vs,Vs).
                                                                                   rst_vals([(X,V)|Ws],Vs1,Vals2,Vals1) :-
        integer(I).
                                                                                           replace(1, [Vs1, Vals2], [Vs1, Vals3], X, V),
a_expr([N,Nv,Np,Npst],V,D) :-
                                                                                           rst_vals(Ws,Vs1,Vals3,Vals1).
        atom(V),
        replace(2,[N,Nv],_,V,D).
                                                                                   append([],Y,Y).
eq_test(V1,V2,tt) :- V1 =:= V2.
                                                                                   \mathtt{append}(\,[\,\mathbb{W}\,|\,\mathbb{X}\,]\,\,,\,\mathbb{Y}\,\,,\,[\,\mathbb{W}\,|\,\,\mathbb{Z}\,]\,\,)\quad :-\quad \mathtt{append}(\,\mathbb{X}\,\,,\,\mathbb{Y}\,,\,\mathbb{Z})\,\,.
eq_test(V1,V2,ff) :- V1 =\= V2.
                                                                                   member(X,[X|_]).
le_test(V1,V2,tt) :- V1 =< V2.</pre>
                                                                                   member(X,[Y|Xs]) :- member(X,Xs).
le_test(V1,V2,ff) :- V1 > V2.
                                                                                   factorial1(N) :-
gt_test(V1,V2,tt) :- V1 > V2.
                                                                                           exec([[n,f],[N,_],[fac],
gt_test(V1,V2,ff) :- V1 =< V2.
                                                                                                   [[p(10,block([var(a,f)],
                                                                                                          [p(1,ifte(gt(n,0),
andi(tt,tt,tt).
                                                                                                                [p(5,assign(a,n)),
andi(tt.ff.ff).
                                                                                                                 p(3,assign(n,minus(n,1))),
andi(ff.tt.ff).
                                                                                                                 p(4,call(fac)),
andi(ff.ff.ff).
                                                                                                                 p(2,assign(f,times(f,a)))
noti(tt.ff).
                                                                                                                [p(12,skip)])),
noti(ff,tt).
                                                                                                    p(11,endblock)])),
                                                                                                    p(9,endproc)]
\texttt{replace}\left(1,\left[\left[X\mid Xs\right],\left[\_\mid Vxs\right]\right],\left[\left[X\mid Xs\right],\left[V\mid Vxs\right]\right],X,V\right).
\texttt{replace}\left(2,\left[\left[X\mid Xs\right],\left[\forall x\mid \forall xs\right]\right],\left[\left[X\mid Xs\right],\left[\forall x\mid \forall xs\right]\right],X,\forall x\right).
                                                                                               1.
\tt replace (S, [[Y \mid Ys], [Vy \mid Vys]], [[Y \mid Ys], [Vy \mid Vzs]], X, Vx) :=
                                                                                               [p(6,assign(f,1)),
                                                                                               p(7,call(fac)),
        p(8,halt)
        \tt replace(S,[Ys,Vys],[Ys,Vzs],X,Vx).
                                                                                              ٦.
vars (N, N, [], []).
                                                                                               rgrs edo).
vars([Vars,Vals,A,B],N,L,[var(X,V)|Vdcls]):-
        member1(X,Vars,F),
                                                                                   % Beginning and end of block (points 9 and 10) may
        a_expr([Vars,Vals,A,B],V,VV),
                                                                                   % have two different environments [n,f] and [a,n,f]
        new_old(F,[Vars,Vals,A,B],N,L,X,VV,Vdcls).
                                                                                   % All other points have a single environment.
new_old(yes,[Vars,Vals,A,B],N1,[(X,VV1)|Vs],X,VV,Vdcls) :-
                                                                                   code_id(1,_,X,f1(X)).
        replace(2,[Vars,Vals],_,X,VV1),
                                                                                   code_id(2,_,X,f2(X)).
                                                                                   code_id(3,_,X,f3(X)).
        replace(1,[Vars,Vals],[Vars,Vals1],X,VV),
        vars([Vars,Vals1,A,B],N1,Vs,Vdcls).
                                                                                   code_id(4,_,X,f4(X)).
new_old(no,[Vars,Vals,A,B],N1,Vs,X,VV,Vdcls):-
                                                                                   code_id(5,_,X,f5(X)).
        vars([[X|Vars],[VV|Vals],A,B],N1,Vs,Vdcls).
                                                                                   code_id(6,_,X,f6(X)).
                                                                                   code_id(7,_,X,f7(X)).
member1(_,[],no).
                                                                                   code_id(8,_,X,f8(X)).
member1(X,[X|_],yes).
                                                                                   code_id(9,[n,f],X,f9nf(X)).
member1(X,[Y|Ys],F) :-
                                                                                   code_id(9,[a,n,f],X,f9anf(X)).
        code_id(10,[n,f],X,f10nf(X)).
        member1(X,Ys,F).
                                                                                   code_id(10,[a,n,f],X,f10anf(X)).
                                                                                   code_id(11, _, X, f11(X)).
```

```
exec_5(X1,X2,X3,X4).
code_id_cont (f1((Va,V,Code)),[Va1,Vl1|Prcs],TrP) :-
                                                                         exec_4(X1,X2,X3,X4) :-
       restore_vars(Va1,V11,Va,V12,V),
                                                                               X1=<0,
       exec([Va,V12|Prcs],Code,TrP).
                                                                               exec_6(X1,X2,X3,X4).
code_id_cont(f2((Va,V,Code)),[Va1,Vl1|Prcs],TrP) :-
                                                                         exec_5(X1,X2,X3,X4) :-
       restore_vars(Va1,V11,Va,V12,V),
                                                                               exec_7(X1,X2,X3,X4).
       exec([Va,V12|Prcs],Code,TrP).
                                                                         exec_6(X1,X2,X3,X4) :-
code_id_cont (f3((Va,V,Code)),[Va1,Vl1|Prcs],TrP) :-
                                                                               exec_8(X1,X2,X3,X4).
       restore_vars(Va1,V11,Va,V12,V),
                                                                         exec 7(X1, X2, X3, X4) :-
       exec([Va,V12|Prcs],Code,TrP).
                                                                              X5 is X1-1.
code_id_cont(f4((Va,V,Code)),[Va1,Vl1|Prcs],TrP) :-
                                                                               exec 9(X1.X5.X2.X3.X4).
       restore vars (Va1.V11.Va.V12.V).
                                                                         exec 8(X1,X2,X3,X4) :-
       exec([Va.V12|Prcs].Code.TrP).
                                                                               code id cont 13(X2.1.X1.1.X3.X4).
code id cont(f5((Va.V.Code)),[Va1.Vl1|Prcs].TrP) :-
                                                                         exec 9(X1 X2 X3 X4 X5) :-
       restore_vars(Va1,Vl1,Va,Vl2,V),
                                                                               exec_11(X1,X2,X3,X4,X5).
                                                                         exec_10(X1,X2,X3) :-
       exec([Va,V12|Prcs],Code,TrP).
code_id_cont (f6((Va,V,Code)),[Va1,Vl1|Prcs],TrP) :-
                                                                               code_id_cont_15(X2,X1,1,X3).
       restore_vars(Va1,V11,Va,V12,V),
                                                                         exec_11(X1,X2,X3,X4,X5) :-
       exec([Va,V12|Prcs],Code,TrP).
                                                                               exec_4(X2,
code_id_cont (f7((Va,V,Code)),[Va1,Vl1|Prcs],TrP) :-
                                                                                      f10anf(([a,n,f],[(a,X1)],[p(9,endproc)])),
       restore_vars(Va1,V11,Va,V12,V),
                                                                                       f4(([a,n,f],[],[p(2,assign(f,times(f,a))),
       exec([Va,V12|Prcs],Code,TrP).
                                                                                          p(11,endblock)])),
code_id_cont (f8((Va,V,Code)),[Va1,Vl1|Prcs],TrP) :-
                                                                                        frame(X3,frame(X4,X5))).
       restore_vars(Va1,V11,Va,V12,V),
                                                                         exec_12(X1,rgrs_edo) :-
       exec([Va,V12|Prcs],Code,TrP).
                                                                               write([[n,f],[X1,1],[fac],
code_id_cont(f9((Va,V,Code)),[Va1,Vl1|Prcs],TrP) :-
                                                                                      [[p(10,block([var(a,f)],
       restore_vars(Va1,V11,Va,V12,V),
                                                                                          [p(1,ifte(gt(n,0),
       exec([Va,V12|Prcs],Code,TrP).
                                                                                              [p(5,assign(a,n)),
                                                                                               p(3, as sign(n, minus(n,1))),
code_id_cont (f10nf((Va,V,Code)),[Va1,Vl1|Prcs],TrP) :-
       restore_vars(Va1,V11,Va,V12,V),
                                                                                               p(4,call(fac)),
       exec([Va,V12|Prcs],Code,TrP).
                                                                                               p(2,assign(f,times(f,a)))],
code_id_cont (f10anf ((Va,V,Code)),[Va1,Vl1|Prcs],TrP) :-
                                                                                              [p(12,skip)])),
       {\tt restore\_vars}\,({\tt Vai}\,,{\tt Vli}\,,{\tt Va}\,,{\tt Vl2}\,,{\tt V})\,,
                                                                                          p(11,endblock)])),
       exec([Va,V12|Prcs],Code,TrP).
                                                                                        p(9,endproc)]]]).
code_id_cont(f11((Va,V,Code)),[Va1,Vl1|Prcs],TrP) :-
                                                                         code_id_cont_13(f10nf(([n,f],[],[p(9,endproc)])),
       restore_vars(Va1,V11,Va,V12,V),
                                                                                         X1,X2,X3,X4,X5) :-
                                                                               exec_19(X2,X3,X4,X5).
       exec([Va,V12|Prcs],Code,TrP).
                                                                                          [p(9,endproc)])),
```

#### A.1.1 A Residual Program

The result of specialising with respect to factorial1(\_) using RUL constraints is shown next.

```
code_id_cont_13(f10anf(([a,n,f],[(a,X1)],
               X2,X3,X4,X5,X6) :-
      exec 21 (X1.X3.X4.X5.X6).
exec_14(X1,X2,X3,X4) :-
      code_id_cont_16(X3,X1,X2,1,X4).
code_id_cont_15(f4(([a,n,f],[],
                [p(2, assign(f, times(f,a))),
                p(11,endblock)])),
               X1,X2,X3) :-
code_id_cont_15(f7(([n,f],[],[p(8,halt)])),
               X1.X2.X3) :-
      exec_20(X1,X2,X3).
code_id_cont_16(f4(([a,n,f],[],
                   [p(2,assign(f,times(f,a))),
                   p(11,endblock)])),
               X1,X2,X3,X4) :-
      exec_22(X1,X2,X3,X4).
code_id_cont_16(f7(([n,f],[],[p(8,halt)])),
               X1,X2,X3,X4) :-
      exec_20(X2,X3,X4).
exec_17(X1,X2,X3) :-
     X4 is 1*X1,
```

```
exec_18(X1,X2,X4,X3).
exec_18(X1,X2,X3,frame(X4,frame(X5,X6))) :-
     code_id_cont_13(X4,X1,X2,X3,X5,X6).
exec 19(X1.X2.X3.X4) :-
     code_id_cont_15(X3,X1,X2,X4).
exec_20(X1,X2,rgrs_edo) :-
     write([[n,f],[X1,X2],[fac],
             [[p(10,block([var(a,f)],
                  [p(1,ifte(gt(n,0),
                     [p(5,assign(a,n)),
                      p(3, assign(n, minus(n, 1))),
                      p(4,call(fac)),
                      p(2,assign(f,times(f,a)))],
                     [p(12,skip)])),
               p(11,endblock)])),
               p(9,endproc)]]]).
exec_21(X1,X2,X3,X4,X5):-
     code_id_cont_16(X4,X1,X2,X3,X5).
exec_22(X1,X2,X3,X4) :-
     X5 is X3*X1,
      exec_18(X1,X2,X5,X4).
```

Note that the control flow of the source imperative program has been compiled into the specialised CLP code.

# A.2 Procedures with parameters

Next we show the semantics of a bigger programming language using locations, parameters by value and by reference, and block declarations.

```
exec(N,[p(_,halt)],rgrs_edo) :-
    write(N).

exec(N,[p(L,endproc)|P],frame(F,TrP)) :-
    code_id_cont(F,N,TrP).

exec(N,[p(L,endblock)|P],frame(F,TrP)) :-
    code_id_cont(F,N,TrP).

exec(N,[p(L,assign(X,AE))|P],TrP) :-
    assign(L,N,X,AE,P,TrP).

exec(N,[skip|P],TrP) :-
    exec(N,[p(L,ifte(B,S1,S2))|P],TrP) :-
    b_expr(N,B,tt),
    append(S1,P,SP1),
    exec(N,[p(L,ifte(B,S1,S2))|P],TrP) :-
    exec(N,[p(L,ifte(B,S1,S2))|P],TrP) :-
```

```
b_expr(N,B,ff),
     append(S2,P,SP2),
     exec(N,SP2,TrP).
exec(N,[p(L,assign(X,AE))|P],TrP) :-
     assign(L,N,X,AE,P,TrP).
exec(N,[p(L,while(B,S1))|P],TrP) :-
     append(S1,[p(L,while(B,S1))|P],S2),
     cond(L,N,B,S2,P,TrP).
exec(N,[p(L,call(Prc,Actls))|P],TrP):-
     proc_c(L,N,Prc,Actls,P,TrP).
\verb|exec([Ne,Va,Vl,Pr,Pl],[p(L,block(Dv,S))|P],TrP):=
     blockvars([Ne,Va,V1,Pr,P1],N1,Dv),
     code id(L.Va.(Va.Ne.P).F).
     exec(N1,S,frame(F,TrP)).
assign(L,[Ne,N,Nv,Np,Npst],X,AE,P,TrP) :-
          a_expr([Ne,N,Nv,Np,Npst],AE,Vae),
     replace(1,[N,Nv],[N1,Nv1],X,Vae),
     \verb|exec([Ne,N,Nv1,Np,Npst],P,TrP)|.
proc_c(L,[Ne,A,B|PrPs],Prc,Actls,P,TrP):-
     replace_p(2,PrPs,_,Prc,(Frmls,S)),
     pair(Ne, Ne1, A, B, A1, B1, Frmls, Actls),
     code_id(L,A,(A,Ne,P),F),
     exec([Ne1,A1,B1|PrPs],S,frame(F,TrP)).
b_expr(N,tt,tt).
b_expr(N,ff,ff).
\label{eq:bessel-energy}  \texttt{b\_expr}\left(\texttt{N,eq}\left(\texttt{A1,A2}\right),\texttt{TV}\right) \ :- \ \texttt{a\_expr}\left(\texttt{N,A1,V1}\right),
                           a_expr(N,A2,V2),
                  eq_test(V1,V2,TV).
b_expr(N,le(A1,A2),TV) :- a_expr(N,A1,V1),
                            a_expr(N,A2,V2),
                 le_test(V1, V2, TV).
{\tt b\_expr} \, ({\tt N}, {\tt and} \, ({\tt A1}, {\tt A2}) \, , {\tt TV}) \;\; :- \;\; {\tt b\_expr} \, ({\tt N}, {\tt A1}, {\tt V1}) \, ,
                             b_expr(N,A2,V2),
                   andi (V1, V2, TV).
b_{\texttt{expr}}(\texttt{N}, \texttt{not}(\texttt{A}), \texttt{TV}) \ :- \ b_{\texttt{expr}}(\texttt{N}, \texttt{A}, \texttt{TV1}) \,,
                       noti(TV1,TV).
a_expr(N,plus(A1,A2),V) :- a_expr(N,A1,V1),
                             a expr(N.A2.V2).
                  V is V1+V2.
a_expr(N,times(A1,A2),V) :- a_expr(N,A1,V1),
                              a expr(N.A2.V2).
                         V is V1*V2.
a_expr(N,minus(A1,A2),V) :- a_expr(N,A1,V1),
                              a_expr(N,A2,V2),
                    V is V1-V2.
a_expr(N,[],[]).
a_expr(N,[A1|As],[V1|Vs]) :-
     a_expr(N,A1,V1),
     a_expr(N,As,Vs).
a_expr(N,I,I) :-
     integer(I).
```

a\_expr([Ne,N,Nv,\_,\_],V,D) :-

 $replace(2,[N,Nv],_,V,D)$ .

atom(V),

```
eq_test(V1,V2,tt) :- V1 == V2.
                                                                                       F1 \== F,
eq_test(V1,V2,ff) :- V1 = V2.
                                                                                       pair_nv(N,N1,C,C1,B1,C2,B2,F,V).
                                                                                  pair_nv(N,N1,[],C1,B1,[(F,N)|C1],[(N,V)|B1],F,V) :-
le test(V1.V2.tt) :- V1 =< V2.
                                                                                       N1 is N+1.
le\_test(V1,V2,ff) := V1 > V2.
                                                                                  pair_v([(F,_)|_],C1,B1,C2,B1,F,L) :-
                                                                                       replace_1(C1,C2,F,L).
andi(tt.tt.tt).
                                                                                  pair_v([(F1,_)|C],C1,B1,C2,B2,F,L) :-
andi(tt.ff.ff).
                                                                                      F1 \== F.
andi(ff tt ff)
                                                                                       pair_v(C,C1,B1,C2,B2,F,L).
andi(ff,ff,ff).
                                                                                  pair v([].C1.B1.[(F.L)|C1].B1.F.L).
noti(tt ff)
                                                                                  replace(1.[Xs.Vxs].[Xs.Vxs1].X.V):-
noti(ff.tt).
                                                                                       lookup loc(X.Lx.Xs).
                                                                                       insert_loc(Lx,V,Vxs,Vxs1).
blockvars(N,N,[]).
                                                                                  replace(2,[Xs,Vxs],[Xs,Vxs],X,Vx):-
blockvars([Ne,Vars,Vals,A,B],N1,[var(X,V)|Vdcls]):-
                                                                                       lookup_loc(X,Lx,Xs),
     a_expr([Ne,Vars,Vals,A,B],V,VV),
                                                                                       lookup_val(Lx,Vx,Vxs).
     memberYN(X,Vars,YN),
                                                                                  replace(3,[Xs,Vxs],[Xs1,[(N,V)|Vxs]],N,X,V):-
     blockvarsYN(YN,[Ne,Vars,Vals,A,B],N1,X,VV,Vdcls).
                                                                                       insert_loc(X,N,Xs,Xs1).
blockvarsYN(yes,[Ne,Vars,Vals,A,B],N1,X,VV,Vdcls) :-
                                                                                  insert_loc(Lx,N,[(Lx,_)|Vxs],[(Lx,N)|Vxs]).
    replace(3,[Vars,Vals],[Vars1,Vals1],Ne,X,VV),
                                                                                  insert_loc(Lx,N,[(L,V)|Vxs],[(L,V)|Vxs1]) :-
     Ne1 is Ne+1, % New location
    blockvars([Ne1, Vars1, Vals1, A, B], N1, Vdcls).
                                                                                       insert_loc(Lx,N,Vxs,Vxs1).
blockvarsYN(no,[Ne,Vars,Vals,A,B],N1,X,VV,Vdcls):-
    Ne1 is Ne+1, % New location
                                                                                  replace_1([(N,_)|Ns],[(N,L)|Ns],N,L).
    \verb|blockvars| ([Ne1,[(X,Ne)|Vars],[(Ne,VV)|Vals],A,B],
                                                                                  replace_1([(N1,L1)|Ns],[(N1,L1)|Ns1],N,L) :-
                N1,Vdcls).
                                                                                       N\==N1.
                                                                                       replace_1(Ns,Ns1,N,L).
memberYN(_,[],no).
memberYN(X,[(X,_)|_],yes).
                                                                                  location([(X,L)|_],X,L).
\texttt{memberYN} \, (\texttt{X}\,, \texttt{[(Y,\_)|Vs]}\,, \texttt{YN)} \quad :- \quad
                                                                                  location([(Y,_)|Ys],X,L) :-
    X \== Y
                                                                                       X \ = = Y
     memberYN(X.Vs.YN).
                                                                                       location(Ys.X.L).
pair(N,N1,C,B,C1,B1,F,A) :-
                                                                                  \mathtt{replace\_p} \hspace{0.1cm} (\texttt{1,[[X | Xs],[\_|Vxs]],[[X | Xs],[V | Vxs]],}
     eval_actual_params(N,C,B,F,A,Vs),
                                                                                            X,V).
     pair1(N,N1,[C,B],[C,B],[C1,B1],F,Vs).
                                                                                  {\tt replace\_p(2,[[X | Xs],[Vx | Vxs]],[[X | Xs],[Vx | Vxs]],}
                                                                                             X.Vx).
\verb|eval_actual_params(\_,\_,\_,[],[],[])|.
                                                                                  \mathtt{replace\_p}\left(\mathtt{S}, \texttt{[[Y | Ys], [Vy | Vys]], [[Y | Zs], [Vy | Vzs]]}\right),
\verb| eval_actual_params(N,C,B,[nv(\_)|Fs],[A|As],[V|Vs]) := \\
                                                                                            X, Vx) :-
    a_expr([N,C,B|_],A,V),
                                                                                       X \== Y,
     eval actual params(N.C.B.Fs.As.Vs).
                                                                                       replace_p(S,[Ys,Vys],[Zs,Vzs],X,Vx).
eval_actual_params(N,C,B,[v(_)|Fs],[A|As],[L|Vs]) :-
    location(C,A,L),
                                                                                  app end ([], Y, Y).
     eval_actual_params(N,C,B,Fs,As,Vs).
                                                                                  append([W|X],Y,[W|Z]) :-
                                                                                       append(X,Y,Z).
pair1(N,N,_,CB,CB,[],[]).
                                                                                  lookup_loc(X,Lx,[(X,Lx)|_]).
pair1 (N,N2,[C,B],[C1,B1],E3,[nv(F)|Fs],[A|As]) :-
                                                                                  lookup_loc(X,Lx,[(Y,_)|Ys]) :-
    pair_nv(N,N1,C,C1,B1,C2,B2,F,A),
                                                                                      X\==Y.
    pair1(N1,N2,[C,B],[C2,B2],E3,Fs,As).
                                                                                       lookup_loc(X,Lx,Ys).
{\tt pair1} \; ({\tt N} \, , {\tt N2} \, , [{\tt C} \, , {\tt B}] \, , [{\tt C1} \, , {\tt B1}] \, , {\tt E3} \, , [{\tt v} \, ({\tt F}) \, | \, {\tt Fs}] \, , [{\tt A} \, | \, {\tt As}]) \;\; :-
    pair_v(C,C1,B1,C2,B2,F,A),
                                                                                  {\tt lookup\_val(X\,,Lx\,,[\,(X\,,Lx)\,|\,\_]\,)\,.}
     pair1(N,N2,[C,B],[C2,B2],E3,Fs,As).
                                                                                  lookup_val(X,Lx,[(Y,_)|Ys]) :-
                                                                                       X = Y
pair_nv(N,N1,[(F,_)|_],C1,B1,C2,B2,F,V) :-
                                                                                       lookup_val(X,Lx,Ys).
    replace(3,[C1,B1],[C2,B2],N,F,V),
        N1 is N+1.
                                                                                  clean(N,[],[]).
{\tt pair\_nv}\,({\tt N}\,,{\tt N1}\,,[\,({\tt F1}\,,\_\,)\,\,|\,\,{\tt C]}\,\,,{\tt C1}\,,{\tt B1}\,,{\tt C2}\,,{\tt B2}\,,{\tt F}\,,{\tt V})\quad:-\quad
                                                                                  \texttt{clean(N,[(N1,V)|Vs],Vs1)} : -
```

```
N =< N1,
    clean(N, Vs, Vs1).
\texttt{clean}(\texttt{N}, \texttt{[(N1,V)|Vs]}, \texttt{[(N1,V)|Vs1])} :=
   N > N1.
    clean(N, Vs, Vs1).
code_id(1,_,X,f1(X)).
code_id(2,_,X,f2(X)).
code_id(3,_,X,f3(X)).
code_id(4,_,X,f4(X)).
code_id(5,_,X,f5(X)).
code_id(6,_,X,f6(X)).
code_id(7,_,X,f7(X)).
code_id(8,[(f1,_),(a,_),(n,_),(f,_)],X,f81(X)).
code_id(8,[(t,_),(f1,_),(a,_),(n,_),(f,_)],X,f82(X)).
code_id(9,_,X,f9(X)).
code_id(10,_,X,f10(X)).
\verb|code_id_cont(f1((A,N,Code)),[N1,A1|E],TrP)|:=
    exec([N,A|E],Code,TrP).
code_id_cont(f2((A,N,Code)),[N1,A1|E],TrP) :-
    exec([N,A|E],Code,TrP).
code_id_cont(f3((A,N,Code)),[N1,A1|E],TrP) :-
    exec([N,A|E],Code,TrP).
code_id_cont(f4((A,N,Code)),[N1,A1|E],TrP) :-
    exec([N,A|E],Code,TrP).
code_id_cont(f5((A,N,Code)),[N1,A1|E],TrP) :-
    exec([N,A|E],Code,TrP).
code_id_cont(f6((A,N,Code)),[N1,A1|E],TrP) :-
    exec([N,A|E],Code,TrP).
code_id_cont(f7((A,N,Code)),[N1,A1|E],TrP) :-
    exec([N,A|E],Code,TrP).
code_id_cont(f81((A,N,Code)),[N1,A1|E],TrP) :-
    exec([N,A|E],Code,TrP).
code_id_cont(f82((A,N,Code)),[N1,A1|E],TrP) :-
    exec([N,A|E],Code,TrP).
\verb|code_id_cont(f9((A,N,Code)),[N1,A1|E],TrP)|:=
    exec([N,A|E],Code,TrP).
code_id_cont(f10((A,N,Code)),[N1,A1|E],TrP) :-
    exec([N,A|E],Code,TrP).
factorial1(N,Loc) :-
   Loc1 is Loc+1.
   Loc2 is Loc+2,
   Loc3 is Loc+3,
    exec([Loc3,[(n,Loc1),(f,Loc2)],
          [(Loc1,N),(Loc2,0)],[fac],[([nv(a),v(f1)],
             [p(1,ifte(eq(a,1),
                  [p(2,assign(f1,1))],
                   [p(8,block(
                   [var(t,1)],
                   [p(3,call(fac,[minus(a,1),t])),
                    p(6,assign(f1,times(a,t))),
                    p(9, endblock)]
                  ))])),
             p(7,endproc)]
         )]
            ],
         [p(5,call(fac,[n,f])),
          p(10,halt)],
         rgrs_edo).
```

#### A.2.1 A Specialised Program

Specialising with respect to factorial1(\_,\_) the above CLP interpreter we obtain the following residual program.

```
factorial1(X1,X2) :-
     X3 is X2+1.
     X4 is X2+2.
     X5 is X2+3
     exec 1(X5,X3,X4,X1).
exec_1(X1,X2,X3,X4) :-
     X5 is X1+1.
     exec_2(X5,X3,X1,X2,X4).
exec_2(X1,X2,X3,X4,X5) :-
     X5==1,
     exec_3(X1,X2,X3,X4,X5).
exec_2(X1,X2,X3,X4,X5) :-
     X5=\=1,
      exec_4(X1,X2,X3,X4,X5).
exec_3(X1,X2,X3,X4,X5) :-
     insert_loc_5(X2,1,[(X3,X5),(X4,X5),(X2,0)],X6),
     exec 6(X1.X2.X3.X4.X6).
exec 3(X1.X2.X3.X4.X5) :-
     insert_loc_5(X2,1,[(X3,X5),(X4,X5),(X2,0)],X6),
      exec_6(X1,X2,X3,X4,X6).
exec_4(X1,X2,X3,X4,X5) :-
     X6 is X1+1,
      exec_7(X6,X1,X2,X3,X4,X5).
insert_loc_5(X1,X2,[(X1,X3)|X4],[(X1,X2)|X4]) :-
{\tt insert\_loc\_5(X1,X2,[(X3,X4)|X5],[(X3,X4)|X6])} \ :- \ \\
     X1\==X3,
     insert_loc_5(X1,X2,X5,X6).
exec_6(X1,X2,X3,X4,X5) :-
      exec_9(X3,X4,X2,X5,rgrs_edo).
exec_7(X1,X2,X3,X4,X5,X6) :-
     lookup_val_10(X4,X7,X2,X6,X5,X6,X3,0,[]),
     X8 is X7-1,
     X9 is X1+1.
      exec_11(X9,X2,X1,X5,X3,X8,X4,X6,X5,X6,X3,
             0. N . X3.
             f81(([(f1.X3),(a,X4),(n,X5),(f,X3)],X2,
                [p(7.endproc)])).
             f5(([(n,X5),(f,X3)],X4,[p(10,halt)])),rgrs_edo).
insert_loc_8(X1,X1,X2,X3,X4,[(X1,1),(X1,X3)|X4]) :-
insert_loc_8(X1,X2,X3,X4,X5,[(X2,X3),(X1,1)|X5]) :-
exec_9(X1,X2,X3,X4,rgrs_edo) :-
      [p(1,ifte(eq(a,1),[p(2,assign(f1,1))],
             [p(8,block([var(t,1)],
```

```
[p(3,call(fac,[minus(a,1),t])),
               p(6,assign(f1,times(a,t))),
              p(9,endblock)])))),
             p(7,endproc)])]]).
lookup_val_10(X1,1,X1,X2,X3,X4,X5,X6,X7) :-
lookup_val_10(X1,X2,X3,X2,X4,X5,X6,X7,X8) :-
     X1\==X3,
     true.
exec_11(X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12,X13,
       X14.X15.X16.X17) :-
     X 6==1
      exec 12(X1.X2.X3.X4.X5.X6.X7.X8.X9.X10.X11.
             X12 X13 X14 X15 X16 X17)
exec 11 (X1.X2.X3.X4.X5.X6.X7.X8.X9.X10.X11.X12.X13.
       X14,X15,X16,X17) :-
     exec_13(X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,
             X12,X13,X14,X15,X16,X17).
exec_12(X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12,
       X13,X14,X15,X16,X17) :-
      insert_loc_8(X2,X3,X6,1,[(X7,X8),(X9,X10),
                   (X11,X12) |X13],X18),
      exec_14(X1,X2,X3,X4,X5,X18,X14,X7,X15,X16,X17).
exec_12(X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12,
       X13, X14, X15, X16, X17) :-
      insert_loc_8(X2,X3,X6,1,[(X7,X8),(X9,X10),
                  (X11,X12) |X13],X18),
      exec_14(X1,X2,X3,X4,X5,X18,X14,X7,X15,X16,X17).
exec_13(X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12,X13,
       X14,X15,X16,X17) :-
     X18 is X1+1.
      exec_15(X18,X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,
             X11.X12.X13.X14.X15.X16.X17).
exec 14(X1.X2.X3.X4.X5.X6.X7.X8.X9.X10.X11) :-
      exec_19(X3,X2,X7,X8,X4,X5,X6,frame(X9,
                                  frame(X10,X11))).
exec_15(X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12,
       X13, X14, X15, X16, X17, X18) :-
     lookup_val_10(X4,X19,X2,X7,X3,1,X8,X9,
                   [(X10,X11),(X12,X13)|X14]),
     X20 is X19-1.
     X21 is X1+1,
      exec_11(X21,X2,X1,X5,X6,X20,X4,X7,X3,1,X8,X9,
              [(X10,X11),(X12,X13)|X14],X3,
              f82(([(t,X3),(f1,X3),(a,X4),(n,X5),(f,X6)],
                  X2,[p(7,endproc)])),
              f3(([(t,X3),(f1,X15),(a,X8),(n,X5),(f,X6)],
                   X4,[p(6,assign(f1,times(a,t))),
                      p(9,endblock)])),
             frame(X16,frame(X17,X18))).
exec_16(X1,X2,X3,X4,X5,X6) :-
     lookup_val_17(X4,X7,X6),
     lookup_val_17(X2,X8,X6),
     X9 is X7*X8,
      insert_loc_5(X3,X9,X6,X10),
      exec_18(X1,X2,X3,X4,X5,X10).
exec_16(X1,X2,X3,X4,X5,X6) :-
     lookup_val_17(X4,X7,X6),
     lookup_val_17(X2,X8,X6),
     X9 is X7*X8,
```

```
insert_loc_5(X3,X9,X6,X10),
      exec_18(X1,X2,X3,X4,X5,X10).
lookup_val_17(X1,X2,[(X1,X2)|X3]) :-
lookup_val_17(X1,X2,[(X3,X4)|X5]) :-
      X1\==X3,
      lookup_val_17(X1,X2,X5).
exec_18(X1,X2,X3,X4,X5,X6) :-
      exec_6(X2,X3,X4,X5,X6).
exec_19(X1,X2,X3,X4,X5,X6,X7,X8) :-
      lookup_val_17(X4,X9,X7),
      lookup_val_17(X2,X10,X7),
      X11 is X9*X10.
      insert loc 5(X3,X11,X7,X12).
      exec_20(X1,X2,X3,X4,X5,X6,X12,X8).
exec_19(X1,X2,X3,X4,X5,X6,X7,X8) :-
      lookup_val_17(X4,X9,X7),
      lookup_val_17(X2,X10,X7),
      X11 is X9*X10,
      insert_loc_5(X3,X11,X7,X12),
      exec_20(X1,X2,X3,X4,X5,X6,X12,X8).
exec_20(X1,X2,X3,X4,X5,X6,X7,frame(X8,frame(X9,X10))) :-
      code_id_cont_21(X8,X1,X2,X3,X4,X5,X6,X7,X9,X10).
code_id_cont_21 (f 81 (([(f1,X1),(a,X2),(n,X3),(f,X4)],X5,
                 [p(7,endproc)])),X6,X7,X8,X9,X10,X11,
                                  X12,X13,X14) :-
      exec_22(X5,X1,X2,X3,X4,X12,X13,X14).
code_id_cont_21 (f82(([(t,X1),(f1,X2),(a,X3),(n,X4),
                              (f,X5)],
                X6, [p(7,endproc)])), X7, X8, X9, X10, X11,
                X12,X13,X14,X15) :-
      exec_23(X6,X1,X2,X3,X4,X5,X13,X14,X15).
exec_22(X1,X2,X3,X4,X5,X6,X7,X8) :-
      code_id_cont_24(X7,X1,f1,X2,a,X3,n,X4,f,X5,[],
                      X6.X8).
exec_23(X1,X2,X3,X4,X5,X6,X7,X8,X9) :-
      {\tt code\_id\_cont\_24\,(X8\,,X1\,,t\,,X2\,,f1\,,X3\,,a\,,X4\,,n\,,X5\,,}
                     [(f,X6)],X7,X9).
{\tt code\_id\_cont\_24} \, ({\tt f3(([(t,X1),(f1,X2),(a,X3),(n,X4),(f,X5)]} \,,
                X6, [p(6, assign(f1, times(a,t))),
                    p(9,endblock)])),
                X7,X8,X9,X10,X11,X12,X13,X14,X15,X16,X17,
                X18) :-
      exec_19(X6,X1,X2,X3,X4,X5,X17,X18).
code_id_cont_24 (f5(([(n,X1),(f,X2)],X3,[p(10,halt)])),
                X4,X5,X6,X7,X8,X9,X10,X11,X12,X13,X14,
                X15) :-
      exec_9(X3,X1,X2,X14,X15).
```

#### A.3 A subset of JVM

Here we show the code for a subset of Java Virtual Machine. It handles numeric basic types and objects using locations. 

```
t2 :-
                empty_env(test,I),
                class vars(test.V).
                vars1(I,Cp1,V),
                cont(test,(init,[]),P),
byte([],[triple(test,main_obj,((init,[]),test))],
                         \mathtt{Cp1} , \mathtt{P} , \mathtt{emptystack}) .
\label{eq:byte} \texttt{byte}\left(\_,\_,\texttt{Cp},[\texttt{p}(\_,\texttt{halt})],\texttt{emptystack}\right) \; :- \;
write(Cp).
\label{eq:byte} \texttt{byte}\left(\left[\_\left|\texttt{Ops}\right],\texttt{S},\texttt{Cp},\left[\texttt{p}\left(\_,\texttt{pop}\right)\left|\texttt{P}\right],\texttt{M}\right)\right. : -
byte(Ops,S,Cp,P,M).
\label{eq:byte} \mbox{byte} \, (\mbox{[\_,\_|0ps],S,Cp,[p(\_,pop2)|P],M)} \;\; :- \;\;
byte (Ops,S,Cp,P,M).
byte(Ops,S,Cp,[p(\_,iconst_m1)|P],M) :-
byte([-1|Ops],S,Cp,P,M).
\label{eq:byte} \mbox{byte} \, (\mbox{Ops} \, , \mbox{S} \, , \mbox{Cp} \, , \mbox{[p(\_,iconst\_0) | P]} \, , \mbox{M}) \; : -
byte([0|Ops],S,Cp,P,M).
byte(Ops,S,Cp,[p(_,iconst_1)|P],M) :-
byte([1 | Ops],S,Cp,P,M).
byte(Ops,S,Cp,[p(_,iconst_2)|P],M) :-
byte([2|Ops],S,Cp,P,M).
byte(Ops,S,Cp,[p(_,iconst_3)|P],M) :-
byte([3|Ops],S,Cp,P,M).
\label{eq:byte_state} \mbox{byte} \, (\mbox{Ops} \, , \mbox{S} \, , \mbox{Cp} \, , \mbox{[p(\_,iconst\_4) | P]} \, , \mbox{M}) \; :- \;
byte([4|Ops],S,Cp,P,M).
\label{eq:byte} \mbox{byte} \left(\mbox{Ops}\,,\mbox{S}\,,\mbox{Cp}\,,\left[\mbox{p}\left(\_\,,\mbox{dconst}\_0\right)\,|\mbox{P}\right]\,,\mbox{M}\right) \ :-
byte([d,0.0|Ops],S,Cp,P,M).
\label{eq:byte} \mbox{byte} \left(\mbox{Ops}\,,\mbox{S}\,,\mbox{Cp}\,,\left[\mbox{p}\left(\_\,,\mbox{dconst}\_1\right)\,|\mbox{P}\right]\,,\mbox{M}\right) \ :-
byte([d,1.0|Ops],S,Cp,P,M).
\label{eq:byte} \mbox{byte} \, (\mbox{Ops} \, , \mbox{S} \, , \mbox{Cp} \, , \mbox{[p(\_, dconst\_2) |P]} \, , \mbox{M}) \ :- \  \,
\verb|byte([d,2.0|0ps],S,Cp,P,M)|.
\label{eq:byte} \mbox{byte} \, (\mbox{Ops} \, , \mbox{S} \, , \mbox{Cp} \, , \mbox{[p(\_, dconst\_3) |P]} \, , \mbox{M}) \; : -
byte([d,3.0|0ps],S,Cp,P,M).
\label{eq:byte} \mbox{byte} \, (\mbox{Ops}\,, \mbox{S}\,, \mbox{Cp}\,, \mbox{[p(\_,dconst\_4) |P]}\,, \mbox{M}) \ :- \  \,
byte([d,4.0|0ps],S,Cp,P,M).
\label{eq:byte} & \texttt{byte}\left( [\texttt{d}, \texttt{D} \,|\, \texttt{Ops}] \,, \texttt{S}, \texttt{Cp} \,, [\texttt{p}(\_\,, \texttt{d2i}) \,|\, \texttt{P}] \,, \texttt{M} \right) \;: -
double2int(D,I),
byte([I | Ops], S, Cp, P, M).
byte([D|Ops],S,Cp,[p(_,istore_1)|P],M) :-
update(1.S.S1.var(1).D).
byte(Ops,S1,Cp,P,M).
byte([D|Ops].S.Cp.[p( .istore 2)|P].M) :-
update(1,S,S1,var(2),D),
byte(Ops,S1,Cp,P,M).
byte([D|Ops],S,Cp,[p(_,istore_3)|P],M) :-
update(1,S,S1,var(3),D),
byte(Ops,S1,Cp,P,M).
\label{eq:byte} \mbox{byte} \left( \left[ \mbox{D} \, | \, \mbox{Ops} \right], \mbox{S}, \mbox{Cp} \, , \left[ \mbox{p} \left( \mbox{\_,istore} \left( \mbox{N} \right) \right) | \, \mbox{P} \right] \, , \mbox{M} \right) \; : -
update(1,S,S1,var(N),D),
\verb|byte(Ops,S1,Cp,P,M|)|.
\label{eq:byte} \mbox{byte} \, (\mbox{Ops}\,, \mbox{S}\,, \mbox{Cp}\,, \mbox{[p(\_,iload\,(\mbox{N}\,))\,|P]}\,, \mbox{M}) \;\; :-
update(2,S,S,var(N),D),
byte([D|Ops],S,Cp,P,M).
```

```
update(2,S,S,var(N),D),
byte([d,D|Ops],S,Cp,P,M).
\label{eq:byte_system} \texttt{byte}\,(\texttt{Ops}\,, \texttt{S}\,, \texttt{Cp}\,, \texttt{[p(\_,iinc(V,C))|P]}\,, \texttt{M}) \;\; :-
update(2,S,S,var(V),Dv),
add_s(Dv,C,Dvpo),
update(1,S,S1,var(V),Dvpo),
byte(Ops,S1,Cp,P,M).
\label{eq:byte} \mbox{byte}\left(\mbox{[A,B|Ops],S,Cp,[p(\_,iadd)|P],M}\right) : -
add s(A.B.AB).
byte([AB|Ops],S,Cp,P,M).
\label{eq:byte} \mbox{byte}\left( \mbox{[A,B|Ops],S,Cp,[p(\_,isub)|P],M} \right) \ :- \ .
sub s(B.A.AB).
\verb|byte([AB|Ops],S,Cp,P,M)|.
\label{eq:byte} \mbox{byte}\left( \mbox{[A,B|Ops],S,Cp,[p(\_,idiv)|P],M} \right) \ :- \ .
div_s(B,A,AB),
byte([AB|Ops],S,Cp,P,M).
\label{eq:byte} \mbox{byte}\left(\mbox{[A,B|Ops],S,Cp,[p(\_,imul)|P],M}\right) :=
mul_s(A,B,AB),
byte([AB|Ops],S,Cp,P,M).
\label{eq:byte} \mbox{byte} \left( \left[ \mbox{d} \, , \mbox{d} \, , \mbox{d} \, , \mbox{B} \, | \, \mbox{Ops} \right] \, , \mbox{S} \, , \mbox{Cp} \, , \, \left[ \mbox{p} \left( \, \_ \, , \, \mbox{dadd} \right) \, | \, \mbox{P} \right] \, , \mbox{M} \right) \; : - \;
add_d(A,B,AB),
byte([AB|Ops],S,Cp,P,M).
\label{eq:byte} \mbox{byte} \left( \left[ \mbox{d} \, , \mbox{d} \, , \mbox{d} \, , \mbox{B} \, | \, \mbox{Ops} \right] \, , \mbox{S} \, , \mbox{Cp} \, , \left[ \mbox{p} \left( \mbox{\_} \, , \, \mbox{ds ub} \right) \, | \, \mbox{P} \right] \, , \mbox{M} \right) \; : -
sub d(B.A.AB).
byte([AB|Ops],S,Cp,P,M).
\label{eq:byte} \mbox{byte} ( \mbox{[d,A,d,B|Ops]} \mbox{,S,Cp,[p(\_,ddiv)|P],M} ) : -
div_d(B,A,AB),
byte([AB|Ops],S,Cp,P,M).
\label{eq:byte} \mbox{byte} \, (\mbox{[d,A,d,B|Ops]} \, , \mbox{S,Cp,[p(\_,dmul)|P]} \, , \mbox{M}) \; : -
mul_d(A,B,AB),
byte([d,AB|Ops],S,Cp,P,M).
\label{eq:byte} \mbox{byte}\left(\mbox{Ops}\,,\mbox{S}\,,\mbox{Cp}\,,\,\left[\mbox{p}\,(\mbox{\_}\,,\mbox{bipush}(\mbox{D})\,)\,|\mbox{P}\right]\,,\mbox{M}\right) \ :-
byte([D|Ops],S,Cp,P,M).
\label{eq:byte} \mbox{byte}\left(\mbox{Ops}\,,\mbox{S}\,,\mbox{Cp}\,,\,\left[\mbox{p}\left(\mbox{\_}\,,\mbox{sipush}(\mbox{D})\right)\,|\,\mbox{P}\right]\,,\mbox{M}\right) \ :-
\texttt{byte}\left( \texttt{[D|Ops],S,Cp,P,M} \right) \,.
\label{eq:byte} \mbox{byte} \left(\mbox{Ops}\,,\mbox{S}\,,\mbox{Cp}\,,\,\mbox{[p(\_\,,ldc\,(D))\,|P]}\,,\mbox{M}\right) \ :- \ .
byte([D|Ops],S,Cp,P,M).
byte(Ops,S,Cp,[p(_,ldc_w(D))|P],M) :-
byte([D|Ops],S,Cp,P,M).
\label{eq:byte_problem} \mbox{byte} \left(\mbox{Ops}\,,\mbox{S}\,,\mbox{Cp}\,,\, \left[\mbox{p}\,(\mbox{$\_$},\mbox{1dc}\,2\mbox{$\_$w}\,(\mbox{D})\,)\,|\,\mbox{P}\,\right]\,,\,\mbox{M}\right) \ :- \ \ \mbox{$\%$ Load a 2 word item!}
byte([d,D|Ops],S,Cp,P,M).
\label{eq:byte} \mbox{byte} \, (\mbox{Ops}\,, \mbox{S}\,, \mbox{Cp}\,, \, \mbox{[p(\_\,,goto(Label))\,|\_]}\,, \mbox{M}) \;\; :-
label(Label.P1).
byte(Ops.S.Cp.P1.M).
byte([B,A|Ops],S,Cp,[p(_,if_icmplt(Label))|_],M) :-
lt(A,B,tt),
label(Label,P1),
byte(Ops,S,Cp,P1,M).
byte([B,A|Ops],S,Cp,[p(_,if_icmplt(_))|P],M) :-
lt(A,B,ff),
byte(Ops,S,Cp,P,M).
byte([B,A|Ops],S,Cp,[p(_,if_icmple(Label))|_],M) :-
le(A,B,tt),
label(Label,P1),
byte(Ops,S,Cp,P1,M).
\label{eq:byte} \mbox{byte} \left( \mbox{[B,A|Ops],S,Cp,[p(\_,if\_icmple(\_))|P],M} \right) \ :- \ .
le(A,B,ff),
byte(Ops,S,Cp,P,M).
\label{eq:byte} \mbox{byte} \left( \mbox{[B,A|Ops],S,Cp,[p(\_,if\_icmpeq(Label))|\_],M} \right) \ :- \ .
eq(A,B,tt),
```

```
label(Label, P1),
                                                                                 \label{eq:byte} \mbox{byte}(\mbox{\tt [A|Ops]}\mbox{\tt ,S,Cp,[p(\_,ifge(Label))|\_],M)} \mbox{\tt :-}
\verb|byte(Ops,S,Cp,P1,M)|.
                                                                                 ge(A,0,tt),
\label{eq:byte} \mbox{byte([B,A|Ops],S,Cp,[p(\_,if\_icmpeq(\_))|P],M)} := \mbox{}
                                                                                 label(Label,P1),
eq(A,B,ff),
                                                                                 byte(Ops,S,Cp,P1,M).
byte(Ops,S,Cp,P,M).
                                                                                 \label{eq:byte} \mbox{byte([A|Ops],S,Cp,[p(\_,ifge(\_))|P],M)} : \mbox{-}
byte([B,A|Ops],S,Cp,[p(_,if_icmpne(Label))|_],M) :-
                                                                                 ge(A,0,ff),
ne(A,B,tt),
                                                                                 byte(Ops,S,Cp,P,M).
label(Label.P1).
                                                                                 byte([d,B,d,A|Ops],S,Cp,[p(\_,dcmp1)|P],M) :-\\
\verb|byte(Ops,S,Cp,P1,M)|.
                                                                                 cmpd(A,B,Tv),
\verb"byte([Tv|Ops],S,Cp,P,M")".
ne(A,B,ff),
                                                                                 \label{eq:byte} \mbox{byte}(\mbox{\tt [d,B,d,A|Ops]}\,,\mbox{\tt S,Cp},\mbox{\tt [p(\_,dcmpg)|P]}\,,\mbox{\tt M})\ :-
                                                                                 cmpd(B,A,Tv),
byte(Ops,S,Cp,P,M).
\label{eq:byte} \mbox{byte([B,A|Ops],S,Cp,[p(\_,if\_icmpge(Label))|\_],M)} := \mbox{}
                                                                                 byte([Tv|Ops],S,Cp,P,M).
                                                                                 byte(Ops,_,Cp,[p(_,return)|_],M) :-
ge(A.B.tt).
label (Label, P1),
                                                                                 code_id_cont(void,Ops,Cp,M).
byte(Ops,S,Cp,P1,M).
                                                                                 %restore(void,Ops,Ops1,S1,M,M1,P),
byte([B,A|Ops],S,Cp,[p(_,if_icmpge(_))|P],M) :-
                                                                                 %byte(Ops1,S1,Cp,P,M1).
ge(A,B,ff),
                                                                                 byte(Ops,_,Cp,[p(_,ireturn)|_],M) :-
byte(Ops,S,Cp,P,M).
                                                                                 code_id_cont(int,Ops,Cp,M).
\label{eq:byte} \mbox{byte([B,A|Ops],S,Cp,[p(\_,if\_icmpgt(Label))|\_],M)} := \mbox{}
                                                                                 %restore(int,Ops,Ops1,S1,M,M1,P),
                                                                                 %byte(Ops1,S1,Cp,P,M1).
gt(A,B,tt),
label(Label,P1),
                                                                                 byte(Ops,_,Cp,[p(_,dreturn)|_],M) :-
byte(Ops,S,Cp,P1,M).
                                                                                 code_id_cont (double,Ops,Cp,M).
%restore(double,Ops,Ops1,S1,M,M1,P),
gt(A.B.ff).
                                                                                 %byte(Ops1,S1,Cp,P,M1).
byte(Ops,S,Cp,P,M).
                                                                                 \label{eq:byte} \texttt{byte}(\texttt{[A,B|Ops],S,Cp,[p(\_,swap)|P],M}) \;\; :-
\label{eq:byte} \mbox{byte([A|Ops],S,Cp,[p(\_,iflt(Label))|\_],M)} :-
                                                                                 byte([B,A|Ops],S,Cp,P,M).
lt(A,0,tt),
                                                                                 label(Label,P1),
                                                                                 \verb|byte([A,A|Ops],S,Cp,P,M)|.
byte(Ops,S,Cp,P1,M).
                                                                                 \label{eq:byte} \mbox{byte}( \mbox{$[A\,,B\,|\,0ps]$} \mbox{,S,Cp,$$[p(\_\,,dup\_x1)\,|\,P]$} \mbox{,M}) :=
\label{eq:byte} \texttt{byte}(\texttt{[A|Ops],S,Cp,[p(\_,iflt(\_))|P],M}) \; :- \;
                                                                                 byte([A,B,A|Ops],S,Cp,P,M).
lt(A,0,ff),
                                                                                 byte(Ops,S,Cp,[p(_,new(C1))|P],M) :-
byte(Ops,S,Cp,P,M).
                                                                                 store([C1,[],[]],Cp,Cp1,Rf),
\label{eq:byte} \mbox{byte([A|Ops],S,Cp,[p(\_,ifle(Label))|\_],M)} :=
                                                                                 byte([Rf | Ops],S,Cp1,P,M).
le(A.O.tt).
                                                                                 \label{eq:byte} \mbox{byte}(\mbox{Ops}\,,\mbox{S}\,,\mbox{Cp}\,,\mbox{[p(\_,aconst\_null)|P]}\,,\mbox{M}) \ :- \ .
label (Label, P1),
                                                                                 \verb|byte([null|0ps],S,Cp,P,M)|.
byte(Ops,S,Cp,P1,M).
                                                                                 \label{eq:byte} \mbox{byte([Rf|Ops],S,Cp,[p(\_,astore\_0)|P],M)} :-
\label{eq:byte} \mbox{byte([A|Ops],S,Cp,[p(\_,ifle(\_))|P],M)} :-
                                                                                 update(1,S,S1,var(0),Rf),
le(A,0,ff),
                                                                                 byte(Ops,S1,Cp,P,M).
byte(Ops,S,Cp,P,M).
                                                                                 \label{eq:byte} \mbox{byte([A|Ops],S,Cp,[p(\_,ifeq(Label))|\_],M)} : -
                                                                                 update(1,S,S1,var(I),Rf),
eq(A,0,tt),
                                                                                 byte(Ops,S1,Cp,P,M).
label (Label, P1),
                                                                                 byte(Ops,S,Cp,[p(_,aload_0)|P],M) :-
byte(Ops.S.Cp.P1.M).
                                                                                 update(2,S,S,var(0),Rf),
byte([A|Ops],S,Cp,[p(_,ifeq(_))|P],M) :-
                                                                                 byte([Rf|Ops].S.Cp.P.M).
eq(A,0,ff),
                                                                                 byte(Ops,S,Cp,[p(\_,aload(I))|P],M) := %I<4, 0<I
byte(Ops,S,Cp,P,M).
                                                                                 update(2,S,S,var(I),Rf),
\label{eq:byte} \mbox{byte([A|Ops],S,Cp,[p(\_,ifne(Label))|\_],M)} :-
                                                                                 byte([Rf | Ops],S,Cp,P,M).
ne(A,0,tt),
                                                                                 byte(Ops,_,Cp,[p(_,areturn)|_],M) :-
label (Label.P1).
                                                                                 code_id_cont (ref,Ops,Cp,M) .
byte(Ops,S,Cp,P1,M).
                                                                                 %restore(ref,Ops,Ops1,S1,M,M1,P1),
byte([A|Ops],S,Cp,[p(_,ifne(_))|P],M) :-
                                                                                 %byte(Ops1,S1,Cp,P1,M1).
ne(A,0,ff),
                                                                                 byte([Rf | Ops],S,Cp,[p(_,ifnull(Lbl))|_],M) :-
byte(Ops,S,Cp,P,M).
                                                                                 null_obj(Rf,Cp,tt),
\label{eq:byte} \mbox{byte([A|Ops],S,Cp,[p(\_,ifgt(Label))|\_],M)} : -
                                                                                          label(Lbl,P1),
gt (A, 0, tt),
                                                                                 byte(Ops,S,Cp,P1,M).
label(Label, P1),
                                                                                 byte(Ops,S,Cp,P1,M).
                                                                                 null_obj(Rf,Cp,ff),
\label{eq:byte} \mbox{byte([A|Ops],S,Cp,[p(\_,ifgt(\_))|P],M)} := \mbox{}
                                                                                 byte(Ops,S,Cp,P,M).
gt(A,0,ff),
                                                                                 byte(Ops,S,Cp,P,M).
                                                                                 null_obj(Rf,Cp,tt),
```

```
byte(Ops,S,Cp,P,M).
                                                                                   4 is Boolean
byte([Rf|Ops],S,Cp,[p(_,ifnonnull(Lb1))|_],M) :-
                                                                                   5 is Char
null_obj(Rf,Cp,ff),
                                                                                   6 is Float
label(Lbl,P1),
                                                                                   7 is Double
byte(Ops,S,Cp,P1,M).
                                                                                   8 is Byte
byte(Ops,S,Cp,[p(_,putfield(Fld,Fldcr))|P],M) :-
                                                                                   9 is Short
pop_ops(Ops,[Rf|Ops1],Fldcr,D),
                                                                                   10 is Int
update_fld(1,Cp,Cp1,Rf,Fld,Fldcr,D),
                                                                                   11 is Long
\verb|byte(Ops1,S,Cp1,P,M|)|.
% Here the environment handling predicates in order of appearance
update_fld(2,Cp,Cp,Rf,Fld,Fldcr,D),
push_ops(Ops,Ops1,Fldcr,D),
                                                                                   update(Sel.S.S1.var(I).D) :-
byte(Ops1,S,Cp,P,M).
                                                                                   update_idx(Sel,S,S1,0,I,D).
byte(Ops,S,Cp,[p(\_,random)|P],M) :-
                                                                                   update_idx(1,[_|S],[D|S],I,I,D).
                                                                                   update_idx(2,[D|S],[D|S],I,I,D).
read(D),
                                                                                   \label{local_state} \verb"update_idx(Sel,[X|S],[X|S1],I,V,D) := \\
byte([d,D|Ops],S,Cp,P,M).
\label{eq:byte} \mbox{byte} \, (\mbox{Ops}\,, \mbox{S}\,, \mbox{Cp}\,, \mbox{[p(\_,nl)}\,|\mbox{P]}\,, \mbox{M}) \;\; :-
                                                                                   I<V,
                                                                                   I1 is I+1,
                                                                                   allow_more(S,Si,I,V),
byte (Ops,S,Cp,P,M).
\label{eq:byte} \mbox{byte} \left( \left[ \mbox{Strg} \left| \mbox{Ops} \right] \mbox{,S} \mbox{,Cp}, \left[ \mbox{p} \left( \mbox{\_,write} \right) \left| \mbox{P} \right] \mbox{,M} \right) \right. : -
                                                                                   update_idx(Sel,Si,S1,I1,V,D).
write(Strg),
byte(Ops,S,Cp,P,M).
                                                                                   allow_more([],L,I,V) :-
byte([Strg1,Strg2|Ops],S,Cp,[p(_,append)|P],M) :-
                                                                                   Lngth is V-I,
append_strg(Strg1,Strg2,Strng3),
                                                                                   length(L,Lngth).
byte([Strng3|Ops],S,Cp,P,M).
                                                                                   allow_more([A|B],[A|B],_,_).
\label{eq:byte} \mbox{byte}\left(\mbox{Ops}\,,\mbox{S}\,,\mbox{Cp}\,,\left[\,\mbox{p}\left(\mbox{L}\,,\mbox{invok\,evirtual}\left(\mbox{Mth}\right)\right)\,|\,\mbox{P}\right]\,,\mbox{M}\right)\ :-
                                                                                   append_strg(A,B,C) :-
code_id(L,locals(S),Locals),
                                                                                   name (A, La),
code_id(L,code(P),Code),
                                                                                   name (B.Lb).
args_match(Mth,Ops,Ops1,Locals,S1,M,M1,Code,Rf1),
                                                                                   append(La,Lb,Lc),
cclass(Rf1,C1),
                                                                                   name (C, Lc).
cont v(Cl.Mth.P1).
byte(Ops1,[Rf1|S1],Cp,P1,M1).
                                                                                   name atomic(X.Y) :-
\label{eq:byte} \mbox{byte} \left(\mbox{Ops}\,,\mbox{S}\,,\mbox{Cp}\,,\mbox{[p(L,invokespecial(Mth,init))|P]}\,,\mbox{M}\right) \ :- \ .
                                                                                   atom(X).
                                                                                   name (X,Y).
code_id(L,locals(S),Locals),
                                                                                   name_atomic(X,Y) :-
code_id(L,code(P),Code),
args_match(Mth,Ops,Ops1,Locals,S1,M,M1,Code,Rf),
                                                                                   atom(X).
cclass(Rf,C1),
                                                                                   name(X.Y).
new_cclass(Rf,Rf1,Super),
                                                                                   double2int(D.I) :-
cont_e(Cl,Mth,P1,Super),
                                                                                   I is D//1.
new vars(Cp.Cp1.Rf1).
                                                                                                     % Integer division
byte(Ops1,[Rf1|S1],Cp1,P1,M1).
byte(Ops,S,Cp,[p(L,invokespecial(Mth,other))|P],M) :-
                                                                                   add_s(A,B,AB) :-
code_id(L,locals(S),Locals),
                                                                                   %\{AB = A+B\}.
code_id(L,code(P),Code),
                                                                                   AB is A+B.
args_match(Mth,Ops,Ops1,Locals,S1,M,M1,Code,Rf),
                                                                                   sub_s(A,B,AB) :-
cclass(Rf,Cl),
                                                                                   %\{AB = A-B\}.
new_cclass(Rf,Rf1,Super),
                                                                                   AB is A-B.
cont_e(Cl,Mth,P1,Super),
                                                                                   div_s(A,B,AB) :-
byte(Ops1,[Rf1|S1],Cp,P1,M1).
                                                                                   AB is A//B. % Integer division!
                                                                                   mul_s(A,B,AB) :-
   Comment for invokevirtual and invokespecial:
                                                                                   AB is A*B.
This last one should seek into private methods and
                                                                                   mul_d(A,B,AB) :-
In the case of Mth= init() it should take an
                                                                                   AB is A*B.
uninstantiated object as reference
(i.e. only initialised on its static fields).
                                                                                   cont_v(Cl,Mth,P) :-
   Comment for newarray:
                                                                                   cont(Cl,Mth,P).
Its argument take values between 6 and 11.
```

 $code_id(f8,X,ff8(X))$ .

```
cont_e(Cl,Mth,P1,Cl1) :-
                                                                                 code_id(16,X,f16(X)).
cont(Cl,Mth,P),
                                                                                 code_id(64,X,f64(X)).
cont_e_check (P,P1,C1,Mth,Cl1).
                                                                                 code_id(67,X,f67(X)).
                                                                                 code_id(86,X,f86(X)).
cont_e_check(null,P,C1,Mth,C11) :-
                                                                                code_id(98,X,f98(X)).
superc(C1,S),
                                                                                 code_id(106, X, f106(X)).
cont_e(S,Mth,P,Cl1).
                                                                                 code_id(109, X, f109(X)).
cont_e_check(P,P,C1,_,C1) :-
                                                                                code_id(114,X,f114(X)).
P \== null.
                                                                                 code_id(116, X, f116(X)).
                                                                                code_id(35,X,f35(X)).
cont(X Y) :-
                                                                                 code id(39.X.f39(X)).
cont(X.Y.).
                                                                                 code id(100.X.f100(X)).
                                                                                 code id(40.X.f40(X)).
noti(ff tt)
noti(tt,ff).
                                                                                code_id_cont (void,[f6(locals(S))|Ops],Cp,
                                                                                                      frame(f6(code(P)),M)) :-
lt(A,B,tt) :-
                                                                                 byte(Ops,S,Cp,P,M).
                                                                                 code_id_cont (void,[ff8(locals(S))|Ops],Cp,
A < B.
lt(A,B,ff) :-
                                                                                                      ff8(code(P)),M)) :-
A >= B.
                                                                                 byte(Ops,S,Cp,P,M).
le(A,B,tt) :-
                                                                                 code_id_cont (void,[f16(locals(S))|Ops],Cp,
A =< B.
                                                                                                      f16(code(P)),M)) :-
le(A,B,ff) :-
                                                                                 byte(Ops,S,Cp,P,M).
                                                                                 code_id_cont (void,[f64(locals(S))|Ops],Cp,
eq(A,B,tt) :-
                                                                                                      f64(code(P)),M)) :-
A =:= B.
                                                                                 byte(Ops,S,Cp,P,M).
eq(A,B,ff) :-
                                                                                 code_id_cont (void,[f67(locals(S))|Ops],Cp,
A =\= B.
                                                                                                      f67(code(P)),M)) :-
ne(A,B,tt) :-
                                                                                 \verb|byte(Ops,S,Cp,P,M)|.
A =\= B.
                                                                                 {\tt code\_id\_cont} \ ({\tt void}, [{\tt f86}({\tt locals}\,({\tt S})\,)\,|\,{\tt Ops}]\,, {\tt Cp}\,,
ne(A,B,ff) :-
                                                                                                      f86(code(P)),M)) :-
A =:= B.
                                                                                 \verb|byte(Ops,S,Cp,P,M)|.
gt(A,B,tt) :-
                                                                                 code_id_cont(void,[f98(locals(S))|Ops],Cp,
A > B
                                                                                                      f98(code(P)),M)) :-
gt(A,B,ff) :-
                                                                                 byte(Ops,S,Cp,P,M).
A =< B.
                                                                                 {\tt code\_id\_cont} \ ({\tt void}, [{\tt f106} ({\tt locals} \ ({\tt S})) \ | {\tt Ops}] \ , {\tt Cp},
ge(A,B,tt) :-
                                                                                                      f106(code(P)),M)) :-
A >= B.
                                                                                 byte(Ops,S,Cp,P,M).
ge(A,B,ff) :-
                                                                                 code_id_cont (void,[f109(locals(S))|Ops],Cp,
                                                                                                      f109(code(P)),M)) :-
A < B.
                                                                                 byte(Ops,S,Cp,P,M).
                                                                                 {\tt code\_id\_cont} \; ({\tt void}, [{\tt f114}({\tt locals}\,({\tt S})) \; | {\tt Ops}] \;, {\tt Cp}, \\
cmpd(A,B,-1) :-
A < B.
                                                                                                      f114(code(P)).M)) :-
cmpd(A.B.0) :-
                                                                                 byte(Ops.S.Cp.P.M).
A = B.
                                                                                 code_id_cont (void, [f116(locals(S)) | Ops], Cp,
cmpd(A,B,1) :-
                                                                                                     f116(code(P)),M)) :-
A > B.
                                                                                 byte(Ops,S,Cp,P,M).
                                                                                 code_id_cont (void,[f35(locals(S))|Ops],Cp,
                                                                                                      f35(code(P)),M)) :-
% The first argument corresponds to the possible load
                                                                                byte(Ops,S,Cp,P,M).
% opcodes (e.g. iload, etc)
                                                                                 code_id_cont (void,[f39(locals(S))|Ops],Cp,
                                                                                                      f39(code(P)),M)) :-
\tt restore(void,[locals(S) | Ops], Ops, S, [code(P) | M], M, P) \;.
                                                                                 byte(Ops,S,Cp,P,M).
\verb"restore(int,[I,locals(S)|0ps],[I|0ps],S,[code(P)|M],M,P)".
                                                                                 code_id_cont (void,[f40(locals(S))|Ops],Cp,
\verb"restore" (\verb"ref", [R, locals" (S) | Ops]", [R | Ops]", S, [code" (P) | M]", M, P)".
                                                                                                      f40(code(P)),M)) :-
\verb"restore(double,[d,D,locals(S)|Ops],[d,D|Ops],S,[code(P)|M],M,P)".
                                                                                 byte(Ops,S,Cp,P,M).
\tt restore(float,[f,F,locals(S)|0ps],[f,F|0ps],S,[code(P)|M],M,P)\;.
                                                                                 code_id_cont (void,[f100(locals(S))|Ops],Cp,
\texttt{restore}(\texttt{long}\,, \texttt{[L\,,locals\,(S)\,|Ops]}\,, \texttt{[L\,|Ops]}\,, \texttt{S\,,[code\,(P)\,|M]}\,, \texttt{M\,,P)}\;.
                                                                                                      f100(code(P)),M)) :-
                                                                                 byte(Ops,S,Cp,P,M).
code_id(6,X,f6(X)).
```

```
code_id_cont(int,[I,f6(locals(S))|Ops],[I|Ops],Cp,
                                                                           code_id_cont(double, [d,D,f86(locals(S))|Ops], [d,D|Ops], Cp,
                   frame(f6(code(P)),M)) :-
                                                                                               frame(f86(code(P)),M)) :-
byte (Ops,S,Cp,P,M).
                                                                           byte(Ops,S,Cp,P,M).
code_id_cont(int,[I,ff8(locals(S))|Ops],[I|Ops],Cp,
                                                                           \verb|code_id_cont| (\verb|double|, [d,D,f| 98(locals(S)) | Ops], [d,D | Ops], Cp, \\
                   frame(ff8(code(P)),M)) :-
                                                                                               frame(f98(code(P)),M)) :-
byte(Ops,S,Cp,P,M).
                                                                           byte(Ops,S,Cp,P,M).
code_id_cont(int,[I,f16(locals(S))|0ps],[I|0ps],Cp,
                                                                           code_id_cont(double,[d,D,f106(locals(S))|Ops],[d,D|Ops],Cp,
                  frame(f16(code(P)),M)) :-
                                                                                               frame(f106(code(P)),M)) :-
\verb|byte(Ops,S,Cp,P,M)|.
                                                                           \verb|byte(Ops,S,Cp,P,M)|.
\verb|code_id_cont(int,[I,f64(locals(S))|0ps],[I|0ps],Cp|,\\
                                                                           code_id_cont(double,[d,D,f109(locals(S))|Ops],[d,D|Ops],Cp,
                   frame(f64(code(P)),M)) :-
                                                                                               frame(f109(code(P)).M)) :-
byte(Ops,S,Cp,P,M).
                                                                           byte(Ops,S,Cp,P,M).
code_id_cont(int,[I,f67(locals(S))|Ops],[I|Ops],Cp,
                                                                           code id cont(double, [d.D.f114(locals(S))|Ops], [d.D|Ops], Cp.
                  frame(f67(code(P)).M)) :-
                                                                                               frame(f114(code(P)),M)) :-
byte (Ops,S,Cp,P,M).
                                                                           byte (Ops,S,Cp,P,M).
                                                                           code_id_cont(double, [d,D,f116(locals(S))|Ops], [d,D|Ops], Cp,
code_id_cont(int, [I,f86(locals(S))|Ops], [I|Ops],Cp,
                   frame(f86(code(P)),M)) :-
                                                                                               frame(f116(code(P)),M)) :-
byte (Ops,S,Cp,P,M).
                                                                           byte (Ops, S, Cp, P, M).
code_id_cont(int,[I,f98(locals(S))|Ops],[I|Ops],Cp,
                                                                           code_id_cont(double,[d,D,f35(locals(S))|Ops],[d,D|Ops],Cp,
                   frame(f98(code(P)),M)) :-
                                                                                               frame(f35(code(P)),M)) :-
byte (Ops,S,Cp,P,M).
                                                                           byte (Ops,S,Cp,P,M).
code_id_cont(int,[I,f106(locals(S))|Ops],[I|Ops],Cp,
                                                                           code_id_cont(double, [d,D,f39(locals(S))|Ops], [d,D|Ops], Cp,
                   frame(f106(code(P)),M)) :-
                                                                                               frame(f39(code(P)),M)) :-
byte(Ops,S,Cp,P,M).
                                                                           byte(Ops,S,Cp,P,M).
code_id_cont(int,[I,f109(locals(S))|Ops],[I|Ops],Cp,
                                                                           code id cont(double, [d.D.f40(locals(S))|Ops], [d.D|Ops], Cp.
                   frame(f109(code(P)),M)) :-
                                                                                               frame(f40(code(P)),M)) :-
byte(Ops,S,Cp,P,M).
                                                                           byte(Ops,S,Cp,P,M).
code_id_cont(int,[I,f114(locals(S))|Ops],[I|Ops],Cp,
                                                                            \verb|code_id_cont(double,[d,D,f100(locals(S))|Ops],[d,D|Ops],Cp,|\\
                   frame(f114(code(P)),M)) :-
                                                                                               frame(f100(code(P)),M)) :-
byte(Ops,S,Cp,P,M).
                                                                           byte (Ops,S,Cp,P,M).
code_id_cont(int,[I,f116(locals(S))|Ops],[I|Ops],Cp,
                   frame(f116(code(P)),M)) :-
\verb|byte(Ops,S,Cp,P,M)|.
                                                                            \verb|code_id_cont(ref,[R,f6(locals(S))|0ps],[R|0ps],Cp|,\\
code_id_cont(int,[I,f35(locals(S))|Ops],[I|Ops],Cp,
                                                                                               frame(f6(code(P)),M)) :-
                   frame(f35(code(P)),M)) :-
                                                                           byte(Ops,S,Cp,P,M).
byte (Ops,S,Cp,P,M).
                                                                           \verb|code_id_cont(ref,[R,ff8(locals(S))|Ops],[R|Ops],Cp|,\\
code_id_cont(int,[I,f39(locals(S))|Ops],[I|Ops],Cp,
                                                                                               frame(ff8(code(P)),M)) :-
                   frame(f39(code(P)),M)) :-
                                                                           byte(Ops,S,Cp,P,M).
byte(Ops,S,Cp,P,M).
                                                                           \verb|code_id_cont(ref,[R,f16(locals(S))|0ps],[R|0ps],Cp|,\\
code_id_cont(int,[I,f40(locals(S))|0ps],[I|0ps],Cp,
                                                                                               frame(f16(code(P)),M)) :-
                   frame(f40(code(P)),M)) :-
                                                                           \verb|byte(Ops,S,Cp,P,M)|.
                                                                           code_id_cont(ref,[R,f64(locals(S))|Ops],[R|Ops],Cp,
byte (Ops,S,Cp,P,M).
code_id_cont(int,[I,f100(locals(S))|Ops],[I|Ops],Cp,
                                                                                               frame(f64(code(P)).M)) :-
                  frame(f100(code(P)),M)) :-
                                                                           byte(Ops.S.Cp.P.M).
byte (Ops.S.Cp.P.M).
                                                                           code_id_cont(ref,[R,f67(locals(S))|Ops],[R|Ops],Cp,
                                                                                               frame(f67(code(P)),M)) :-
code_id_cont(double,[d,D,f6(locals(S))|Ops],[d,D|Ops],Cp,
                                                                           byte (Ops,S,Cp,P,M).
                  frame(f6(code(P)),M)) :-
                                                                           code_id_cont(ref,[R,f86(locals(S))|Ops],[R|Ops],Cp,
                                                                                               frame(f86(code(P)),M)) :-
byte(Ops,S,Cp,P,M).
\verb|code_id_cont(double,[d,D,ff8(locals(S))|0ps],[d,D|0ps],Cp,|\\
                                                                           byte(Ops,S,Cp,P,M).
                   frame(ff8(code(P)),M)) :-
                                                                           code_id_cont(ref,[R,f98(locals(S))|Ops],[R|Ops],Cp,
byte(Ops,S,Cp,P,M).
                                                                                               frame(f98(code(P)),M)) :-
\verb|code_id_cont(double,[d,D,f16(locals(S))|0ps],[d,D|0ps],Cp,|\\
                                                                           byte(Ops,S,Cp,P,M).
                                                                           {\tt code\_id\_cont} \ ({\tt ref}\ , [{\tt R}\ , {\tt f106}\ ({\tt locals}\ ({\tt S}))\ |{\tt Ops}]\ , [{\tt R}\ |{\tt Ops}]\ , {\tt Cp}\ ,
                   frame(f16(code(P)),M)) :-
byte (Ops,S,Cp,P,M).
                                                                                               frame(f106(code(P)),M)) :-
code_id_cont(double,[d,D,f64(locals(S))|Ops],[d,D|Ops],Cp,
                                                                           byte (Ops,S,Cp,P,M).
                   frame(f64(code(P)),M)) :-
                                                                            code_id_cont(ref,[R,f109(locals(S))|Ops],[R|Ops],Cp,
byte(Ops,S,Cp,P,M).
                                                                                               frame(f109(code(P)),M)) :-
{\tt code\_id\_cont}\,({\tt double}\,, {\tt [d,D,f67}\,({\tt locals}\,({\tt S})\,)\,|{\tt Ops}]\,, {\tt [d,D}\,|{\tt Ops}]\,, {\tt Cp}\,,
                                                                           byte(Ops,S,Cp,P,M).
                   frame(f67(code(P)),M)) :-
                                                                            \verb|code_id_cont(ref,[R,f114(locals(S))|0ps],[R|0ps],Cp|,\\
\verb|byte(Ops,S,Cp,P,M)|.
                                                                                               frame(f114(code(P)),M)) :-
```

```
byte(Ops,S,Cp,P,M).
\verb|code_id_cont(ref,[R,f116(locals(S))|0ps],[R|0ps],Cp|,\\
                                                                       new_obj(Cp,Cp,[],[]).
                                                                       new_obj(Cp,Cp2,[[N,pair(Type,N1)]|01],[var(N,Type)|Vars]) :-
                  frame(f116(code(P)),M)) :-
byte(Ops,S,Cp,P,M).
                                                                       Cp=[[N1|Mo],Hp],
code_id_cont (ref, [R,f35(locals(S))|0ps], [R|0ps],Cp,
                                                                       Nl1 is Nl + 1,
                  frame(f35(code(P)),M)) :-
                                                                       Cp1=[[Nl1|Mo],[(Nl,null)|Hp]],
byte(Ops,S,Cp,P,M).
                                                                       new_obj(Cp1,Cp2,O1,Vars).
code_id_cont(ref,[R,f39(locals(S))|0ps],[R|0ps],Cp,
                  frame(f39(code(P)),M)) :-
                                                                       initial v(int.0).
byte(Ops,S,Cp,P,M).
                                                                       initial v(short.0).
code_id_cont (ref, [R,f40(locals(S))|Ops], [R|Ops],Cp,
                                                                       initial_v(byte,0).
                  frame(f40(code(P)),M)) :-
                                                                       initial v(char.0).
                                                                       initial_v(long,0).
byte(Ops,S,Cp,P,M).
code_id_cont (ref, [R,f100(locals(S))|Ops], [R|Ops], Cp,
                                                                       initial_v(float,0.0).
                  frame(f100(code(P)),M)) :-
                                                                       initial_v(double,0.0).
byte(Ops,S,Cp,P,M).
                                                                       vm_basic_type(byte,yes).
empty_env(C,[C,[[C,[[],[]],[[],[]],[[],[]]],[]]).
                                                                       vm_basic_type(char, yes).
                                                                       vm_basic_type(short,yes).
classname([C1|_],C1).
                                                                       vm_basic_type(long,yes).
                                                                       vm_basic_type(double,yes).
classlist([C,A,B],[C,B],A). % Instance variables list A!
                                                                       vm_basic_type(float,yes).
                                                                       vm_basic_type(X,no) :-
objects([C,A,B],[C,A],B).
                                                                       \t X = byte,
vars1(S,Cp2,Vars) :-
                                                                       \t X = char,
classname(S,C1),
                                                                       \+ X = short,
classlist(S,_,Lc),
                                                                       append(Lcb,[[Cl|Purv]|Lca],Lc),
                                                                       \+ X = double,
add_vars1 (Purv, Purv1, Vars, Objs),
                                                                       append(Lcb, [[C1|Purv1]|Lca], Lc1),
Cp = [[1,[C1,Lc1,[]]],[]],
                                                                       \verb|store(InsCl,[Cp,Tmp],[Cp1,[(N,InsCl)|Tmp]],pair(Cl,N)) :=
new_obj(Cp,Cp1,Objects1,Objs),
                                                                       classname (InsCl,Cl),
Cp1=[[N1,Mo],Hp],
                                                                       new_location(Cp,Cp1,N).
objects(Mo,Rst,_),
                                                                       new_location([N | B], [N1 | B], N) :-
objects(Mo1,Rst,Objects1),
Cp2=[[N1,Mo1],Hp].
                                                                       N1 is N+1.
                                                                       {\tt args\_match(Fldcr, Ops, [Locals \,|\, Ops1]\,, Locals\,, S1\,, M\,,}
add_vars1 (Purv,Purv,[],[]).
                                                                                      frame(Code,M),Code,Rf1) :-
add_vars1(Purv,Purv2,[V|Vars],Objs) :-
                                                                       Fldcr=(_,LArgs),
V=var(_,Type,_) ,
                                                                       pop2locals(LArgs, Ops, [Rf | Ops1], [], S1),
vm basic type(Type,YN).
                                                                       new_cmethod(Rf,Rf1,Fldcr).
add_vars1_check(YN,Purv,Purv2,V,Vars,Objs).
                                                                       cclass(pair(C1,_),C1).
add_vars1_check(yes,Purv,Purv2,V,Vars,Objs) :-
                                                                       cclass(triple(_,_,(_,Cl1)),Cl1).
V=var(Label, Type, N),
add_var1(Purv,Purv1,N,Type,Label),
                                                                       cmethod(triple(\_,\_,(M1,\_)),M1).
add_vars1 (Purv1,Purv2,Vars,Objs).
add_vars1_check(no,Purv,Purv2,V,Vars,Objs) :-
                                                                       new_cclass(triple(C1,Obj,(Mth,_)),triple(C1,Obj,(Mth,C11)),C11).
V = var(_,Type,N),
Objs = [var(N,Type)|Objs1],
                                                                       new\_cmethod(triple(Cl,Obj,(\_,Cl3)),triple(Cl,Obj,(Mth,Cl3)),Mth)\;.
add_vars1 (Purv, Purv2, Vars, Objs1).
                                                                       {\tt new\_cmethod(pair(Cl,Obj),triple(Cl,Obj,(Mth,Cl)),Mth)}\,.
\verb"add_var1([[Ns,Vs]|Prv],[[[N|Ns],[Val|Vs]]|Prv],N,Type,public) :-
                                                                       pop2locals([],Ops,Ops,S,S).
                                                                       pop2locals([int |LArgs],[I|Ops],Ops1,S,S1) :-
initial_v(Type, Val).
\verb| add_vari([Pu,[Ns,Vs]|Pv],[Pu,[[N|Ns],[Val|Vs]]|Pv],N,Type,protct) :- pop2locals(LArgs,Ops,Ops1,[I|S],S1). \\
                                                                       \verb"pop2locals"([short | LArgs], [I | Ops], Ops1, S, S1) :-
initial_v(Type, Val).
\verb"add_vari([Pu,Pr,[Ns,Vs]],[Pu,Pr,[[N|Ns],[Val|Vs]]],N,Type,privat) :- pop2locals(LArgs,Ops,Ops1,[I|S],S1).
initial_v(Type,Val).
                                                                       pop2locals([char|LArgs],[I|Ops],Ops1,S,S1) :-
```

```
pop2locals(LArgs,Ops,Ops1,[I|S],S1).
                                                                            update_fld(S,Cp,Cp1,Rf,N,int,I) :-
pop2locals([ref |LArgs],[I|Ops],Ops1,S,S1) :-
                                                                           update_int(Rf,S,Cp,Cp1,N,I).
pop2locals(LArgs,Ops,Ops1,[I|S],S1).
                                                                           update_fld(S,Cp,Cp1,Rf,N,short,Sh) :-
pop2locals([byte|LArgs],[I|Ops],Ops1,S,S1) :-
                                                                           update_int(Rf,S,Cp,Cp1,N,Sh).
pop2locals(LArgs,Ops,Ops1,[I|S],S1).
                                                                           update_fld(S,Cp,Cp1,Rf,N,byte,B) :-
pop2locals([float|LArgs],[I|Ops],Ops1,S,S1) :-
                                                                           update_int(Rf,S,Cp,Cp1,N,B).
pop2locals(LArgs,Ops,Ops1,[I|S],S1).
                                                                           update_fld(S,Cp,Cp1,Rf,N,float,F) :-
pop2locals([double|LArgs],[d,D|Ops],Ops1,S,S1) :-
                                                                           update_int(Rf,S,Cp,Cp1,N,F).
\verb"pop2locals" (LArgs, Ops, Ops1, [D,\_|S], S1).
                                                                           update_fld(S,Cp,Cp1,Rf,N,char,Ch) :-
\verb"pop2locals" ([long|LArgs],[l,L|Ops],Ops1,S,S1) :-
                                                                           update_int(Rf,S,Cp,Cp1,N,Ch).
\verb"pop2locals" (LArgs, Ops, Ops1, [L,\_|S], S1).
                                                                           update_fld(S,Cp,Cp1,Rf,N,ref,R) :-
                                                                           update_ptr(Rf,S,Cp,Cp1,N,R).
null ref(null)
                                                                           update_fld(S,Cp,Cp1,Rf,N,double,D) :-
                                                                           update_int(Rf,S,Cp,Cp1,N,D).
null_obj(null,_,tt).
                                                                           update_fld(S,Cp,Cp1,Rf,N,long,L) :-
                                                                           update_int(Rf,S,Cp,Cp1,N,L).
null_obj(Rf,Cp,Tv) :-
functor (Rf,F,_),
update\_int(triple(Cl,main\_obj,\_),1,[Cp,Hp],[Cp1,Hp],N,V) := \\
lookup_obj(Cp,Rf,Obj_c),
                                                                            Cp=[N,[C1,Var,Objs]],
obj_cnts(Obj_c,Cnts),
                                                                            Cp1=[N,[C1,Var1,Objs]],
null (Cnts, Tv).
                                                                           update_m(1,Cl,Var,Var1,N,V).
                                                                            update_int(triple(C1,main_obj,_),2,[Cp,Hp],[Cp,Hp],N,V) :-
null(null,tt).
                                                                            Cp=[N,[C1,Var,_]],
null(N.ff) :-
                                                                           update_m(2,C1,Var,Var,N,V).
functor (N, F, _),
                                                                            update_int(triple(_,0,_),1,[Cp,Hp],[Cp,Hp1],N,V) :-
integer (0),
                                                                            update_rf12(Hp,Hp1,(0,0c,0c1)),
obj_cnts((_,A),A).
                                                                            classlist(Oc,R,Var),
                                                                            classname(Oc.C1).
% The case of pair (C1,0) is not included because its contents
                                                                            update_m(1,Cl,Var,Var1,N,V),
% aren't null, by definition above!!
                                                                            classlist(Oc1,R,Var1).
lookup_obj([_,Hp],pair(_,0),(0,0c)) :-
                                                                            update_int(triple(_,0,_),2,[Cp,Hp],[Cp,Hp],N,V) :-
integer (0),
                                                                           integer (0),
update_rf12(Hp,Hp,(0,0c,0c)).
                                                                           update_rf12(Hp,Hp,(0,0c,0c)),
lookup\_obj([[\_,Cp]|\_],triple(Cl,main\_obj,\_),(main\_obj,Cp)) \; :- \;
                                                                           classlist(Oc,_,Var),
classname(Cp,Cl).
                                                                            classname(Oc.Cl).
lookup_obj([_,Hp],triple(_,Obj,_),(Obj,Obj_c)) :-
                                                                           update_m(2,C1,Var,Var,N,V).
integer (Obj),
                                                                           update\_int\left(pair\left(\texttt{Cl}\,,\texttt{main\_obj}\right),\texttt{1},\left[\texttt{Cp}\,,\texttt{Hp}\right],\left[\texttt{Cp1}\,,\texttt{Hp}\right],\texttt{N}\,,\texttt{V}\right) :=
                                                                           Cp=[N,[C1,Var,Objs]],
update_rf12(Hp,Hp,(0bj,0bj_c,0bj_c)).
                                                                           Cp1=[N,[Cl,Var1,Objs]],
unique_ref (triple(_ ,N ,_) ,N) .
                                                                           update_m(1,Cl,Var,Var1,N,V).
unique\_ref(pair(\_,N),N).
                                                                           {\tt update\_int(pair(Cl,main\_obj),2,[Cp,Hp],[Cp,Hp],N,V)} \ :- \\
                                                                           Cp=[N.[C1.Var.]].
push_ops(Ops,[I |Ops],int,I).
                                                                           update m(2.Cl.Var.Var.N.V).
                                                                           {\tt update\_int(pair(\_,0),1,[Cp,Hp],[Cp,Hp1],N,V)} \; :- \;
push_ops(Ops,[S|Ops],short,S).
push_ops(Ops,[B|Ops],byte,B).
                                                                           integer (0).
push_ops(Ops,[F|Ops],float,F).
                                                                           update_rf12(Hp,Hp1,(0,0c,0c1)),
push_ops(Ops,[C|Ops],char,C).
                                                                           classlist(Oc,R,Var),
push_ops(Ops,[R|Ops],ref,R).
                                                                           classname(Oc.Cl).
push_ops(Ops,[d,D|Ops],double,D).
                                                                           update_m(1,Cl,Var,Var1,N,V),
push_ops(Ops,[1,L|Ops],long,L).
                                                                           classlist(Oc1.R.Var1).
                                                                           update_int(pair(_,0),2,[Cp,Hp],[Cp,Hp],N,V) :-
pop_ops([I | Ops], Ops, int, I).
                                                                           integer (0).
                                                                           update_rf12(Hp,Hp,(0,0c,0c)),
pop\_ops([S|Ops],Ops,short,S).
pop\_ops([B|Ops],Ops,byte,B).
                                                                           classlist(Oc,_,Var),
pop\_ops([F|Ops],Ops,float,F).
                                                                           classname(Oc,C1),
pop\_ops([C|Ops],Ops,char,C).
                                                                           update_m(2,C1,Var,Var,N,V).
pop_ops([R |Ops],Ops,ref,R).
pop\_ops([d,D|Ops],Ops,double,D).
                                                                           update_rf12(Hp,Hp2,(0,0c,0c1)):-
pop_ops([1,L|0ps],0ps,long,L).
                                                                           update_rf2(F,Nwo,Hp,Hp1,(0,0c,0c1)),
                                                                           \tt decide\_srch(F,Nwo,Hp,Hp1,Hp2,(0,0c,0c1)).
```

```
update_q(M,Ps,Ps1,N,V,F).
decide_srch(retry,Nwo,Hp,_,Hp2,(_,Oc,Oc1)) :-
update_rf12(Hp,Hp2,(Nwo,Oc,Oc1)).
                                                                                          update_s(_,[[],[]],[[],[]],_,_,no).
decide_srch(done,_,_,Hp2,Hp2,_).
                                                                                          update_s(1,[[N|Ns],[_|Vs]],[[N|Ns],[V|Vs]],N,V,yes).
                                                                                          update\_s\left(2,\left[\left[N\left|Ns\right],\left[V\left|Vs\right]\right],\left[\left[N\left|Ns\right],\left[V\left|Vs\right]\right],N,V,yes\right).\right.
update_rf2(done,_,[(0,0c)|0bjs],[(0,0c1)|0bjs],(0,0c,0c1)) :-
                                                                                          functor (Oc,F,_),
                                                                                          update_s(S,[Ns,Vs],[Ns,Vs1],N1,V1,F).
\+ F = triple.
\label{local_problem} $$ up\,dat\,e\_rf\,2\,(r\,etry\,,\,Nw\,o\,,\,[\,(\,0\,,\,0c\,2)\,\,|\,\,0b\,js\,]\,\,,\,(\,0\,,\,\,\,\,\,\,\,,\,\,\,\,\,\,)\,) \ := \ (\,0\,,\,0c\,2\,)\,\,|\,\,0b\,js\,]\,\,,\,(\,0\,,\,\,\,\,\,\,\,\,\,\,\,\,)\,
                                                                                          {\tt update\_ptr(triple(Cl,main\_obj,\_),1,[Cp,Hp],[Cp,Hp1],N,R)} \ :- \\
0c2 = pair(\_,Nwo).
                                                                                          Cp=[_,[C1,_,Objs]],
\label{local_problem} $$ up\,dat\,e\_rf\,2\,(r\,etry\,,\,Nw\,o\,,\,[\,(\,0\,,\,0c\,2)\,\,|\,\,0b\,js\,]\,\,,\,(\,0\,,\,\,\,\,\,\,\,,\,\,\,\,\,\,)\,) \ := \ (\,0\,,\,0c\,2\,)\,\,|\,\,0b\,js\,]\,\,,\,(\,0\,,\,\,\,\,\,\,\,\,\,\,\,\,)\,
                                                                                          update_rf(Objs,Objs,[N,Rf]),
                                                                                          unique_ref(Rf,Uid),
Oc2 = triple(_,Nwo,_).
 update\_rf2(F,Nwo,[(02,0c2)|0bjs],[(02,0c2)|0bjs1],(0,0c,0c1)) := \\
                                                                                          update_rf1(Hp,Hp1,(Uid,R)).
                                                                                          \label{local_ptr} \verb"update_ptr"(triple(C1,main_obj,\_),2,[Cp,Hp],[Cp,Hp],N,Rf) :-
update_rf2(F,Nwo,Objs,Objs1,(0,Oc,Oc1)).
                                                                                          Cp=[_,[C1,_,Objs]],
                                                                                          update_rf(Objs,Objs,[N,R]),
                                                                                          cnts2rf(R,Rf).
up\,dat\,e\_rf\,1\,([\,({\tt N\,,\,\_})\,\,|\,{\tt Cls}]\,,[\,({\tt N\,,Cnt}\,{\tt s}\,{\tt 1}\,)\,\,|\,{\tt Cls}]\,\,,({\tt N\,,Cnt}\,{\tt s}\,{\tt 1}\,)\,\,)\,\,.
update_rf1([(N,Cnts)|Cls],[(N,Cnts)|Cls1],(N1,Cnts1)) :-
                                                                                          \label{eq:update_ptr(pair(_,0),1,[Cp,Hp],[Cp,Hp1],N,R)} update_ptr(pair(_,0),1,[Cp,Hp],[Cp,Hp1],N,R) :-
                                                                                          integer(0),
                                                                                          update_rf12(Hp,Hp,(0,0c,0c)),
update_rf1(Cls,Cls1,(N1,Cnts1)).
                                                                                          objects(Oc,_,Obj),
update_rf([[N,Cnts]|Cls],[[N,Cnts]|Cls],[N,Cnts]).
                                                                                          update_rf(Obj,Obj,[N,Rf]),
update_rf([[N,Cnts]|Cls],[[N,Cnts]|Cls1],[N1,Cnts1]) :-
                                                                                          unique_ref(Rf,Uid),
                                                                                          update_rf1(Hp,Hp1,(Uid,R)). % It cannot refer to the same object!
update_rf(Cls,Cls1,[N1,Cnts1]).
                                                                                          update_ptr(pair(_,0),2,[Cp,Hp],[Cp,Hp],N,Rf) :-
append([],Y,Y).
                                                                                          update_rf12(Hp,Hp,(0,0c,0c)),
append([W|X],Y,[W|Z]) :-
                                                                                          objects(Oc,_,Objs),
append(X,Y,Z).
                                                                                          update_rf(Objs,Objs,[N,R]),
                                                                                          cnts2rf(R,Rf).
append1([],Y,Y).
                                                                                          \label{eq:condition} \verb"update_ptr(triple(\_,0,\_),1,[Cp,Hp],[Cp,Hp1],N,R) :=
append1([[C1|Cs1]|X],[[C|Cs]|Y],[[C1|Cs1]|Z]) :-
                                                                                          integer(0),
\+ C=C1.
                                                                                          update_rf12(Hp,Hp,(0,0c,0c)),
append1(X,[[C|Cs]|Y],Z).
                                                                                          objects(Oc,_,Obj),
                                                                                          update_rf(Obj,Obj,[N,Rf]),
element((D,Dc),[(D,Dc)|C],[],C).
                                                                                          unique_ref(Rf,Uid),
element((D,Dc),[(A,Ac)|C],[(A,Ac)|B],E) :-
                                                                                          update_rf1(Hp,Hp1,(Uid,R)).
D=\=A.
                                                                                          \label{eq:condition} update\_ptr(triple(\_,O,\_),2,[Cp,Hp],[Cp,Hp],N,Rf) :-
\mathtt{element}\,(\,(\mathtt{D}\,,\mathtt{Dc})\,,\mathtt{C}\,,\mathtt{B}\,,\mathtt{E})\;.
                                                                                         integer (0),
                                                                                          {\tt update\_rf12(Hp,Hp,(0,0c,0c))},
\mathtt{up\,dat\,e\_m\,(M\,,C\,,Lc\,,Lc1\,,N\,,V)} \quad : \quad \cdot
                                                                                          objects(Oc,_,Objs),
                                                                                          update_rf(Objs,Objs,[N,R]),
append1(Lcb,[[C|Purv]|Lca],Lc),
update g(M.Purv.Purv1.N.V.F).
                                                                                          cnts2rf(R,Rf).
decide_m(F,M,Lc,C,Purv1,Lcb,Lca,Lc1,N,V).
                                                                                          cnts2rf(null,null).
decide_m(yes,_,_,C,P,L1,L2,Lc1,_,_) :-
                                                                                          cnts2rf([_,null],null).
append(L1,[[C|P]|L2],Lc1).
                                                                                          cnts2rf([_,loc(R)],R).
decide_m(no,M,Lc,C,_,_,Lc1,N,V) :-
                                                                                         cnts2rf(pair(A,B),pair(A,B)).
superc(C,Super),
                                                                                          cnts2rf(triple(A,B,C),triple(A,B,C)).
append1(Lcb, [[Super, Pu, Pr | Pv] | Lca], Lc),
update_q(M,[Pu,Pr],[Pu1,Pr1],N,V,F),
                                                                                          new_vars(Cp,Cp2,Rf) :-
\mathtt{decide\_m(F,M,Lc,Super,[Pu1,Pr1|Pv],Lcb,Lca,Lc1,N,V)}\;.
                                                                                          cclass(Rf.Cl).
                                                                                          empty_env(Cl,Inst0),
update_q(_,[],_,_,_,no).
                                                                                          classlist(Inst0,_,[[C1|Purv]]),
\mathtt{update\_q}(\texttt{M}, \texttt{[P|Ps]}, \texttt{Ps1}, \texttt{N}, \texttt{V}, \texttt{F1} \quad) \; : \neg
                                                                                          class_vars(C1,Vars),
update_s(M,P,P1,N,V,F),
                                                                                          add_vars1 (Purv, Purv1, Vars, Objs),
decide_s(F,F1,M,[P|Ps],P1,Ps1,N,V).
                                                                                          new_obj(Cp,Cp1,NObjs,Objs),
                                                                                          {\tt classlist\,(Inst,[Cl,NObjs],[[Cl|Purv1]])}\,,
decide_s(yes,yes,_,[_|Ps],P1,[P1|Ps],_,_).
                                                                                          update_inst(Cp1,Cp2,Rf,Inst).
decide_s(no,F,M,[P|Ps],_,[P|Ps1],N,V) :-
```

```
update_inst([Cp,Hp],[Cp1,Hp],triple(_,main_obj,_),Inst) :-
                                                                                       var(privat,circle,cir),
                                                                       var(privat,cylinder,cyl),
Cp=[N,Inst0],
append_inst(Inst0,Inst,Inst1),
                                                                       var(privat, shape, s1),
Cp1=[N,Inst1].
                                                                       var(privat, shape, s2),
update\_inst([Cp,Hp],[Cp,Hp1],triple(\_,0,\_),Inst) :-
                                                                       var(privat, shape, s3),
integer (0),
                                                                       var(privat, shape, s4)]).
element((0,Inst0),Hp,Hpb,Hpa),
                                                                      superc(test,top).
append_inst(Inst0,Inst,Inst1),
element ((0, Inst1), Hp1, Hpb, Hpa).
                                                                      {\tt Compiled\ from\ Cylinder.java}
append_inst(Inst0,Inst,Inst2) :-
                                                                      public synchronized class Cylinder extends Circle
classlist(Inst0.Rest0.Vars0).
                                                                          / ACC SUPER bit set /
classlist(Inst. .Vars).
append(Vars0, Vars, Vars1),
                                                                          protected double height;
classlist(Inst1.Rest0.Vars1).
objects (Inst1, Rest1, Objs1),
                                                                          public Cylinder(double,double,double,double);
objects(Inst,_,Objs),
                                                                      / (DDDD)V /
append(Objs1,Objs,Objs2),
                                                                          public void setHeight(double);
objects (Inst2,Rest1,Objs2).
                                                                      / (D)V /
                                                                          public double getHeight();
%fin
                                                                          public double area();
% Bytecode
                                                                          public double volume();
                                                                          public java.lang.String toString();
                                                                         ()Ljava/lang/String;
Compiled from Test.java
                                                                          public java.lang.String getName();
public synchronized class Test extends java.applet.Applet
                                                                          ()Ljava/lang/String;
    / ACC_SUPER bit set /
    public void init();
  ()V /
                                                                      cont(cylinder, (cylinder, [double,double,double,double]),[
    public void paint(java.awt.Graphics);
                                                                       p(60.aload 0).
   (Ljava/awt/Graphics;)V /
                                                                       p(61,dload(3)),
    public Test();
                                                                       p(62,dload(5)),
   ()V /
                                                                       p(63,dload(7)),
}
                                                                       p(64,invokespecial((circle,[double,double,double]),init)),
                                                                                         % <Method Circle(d.d.d)>
                                                                       p(65,aload_0),
cont(test,(init,[]),[
                                                                       p(66.dload(1)).
p(1,aload_0).
                                                                       p(67,invokevirtual((setheight,[double]))),
p(2,new(point)), % <Class Point>
                                                                                         % <Method void setHeight(d)>
p(3.dup).
                                                                       p(68.return)]).
p(4,1dc2_w(7.0)), % <Double 7.0>
p(5,ldc2_w(11.0)), % <Double 11.0>
p(6,invokespecial((point,[double,double]),init)),
                                                                      cont(cylinder,(setheight,[double]),[
           % <Method Point(double,double)>
                                                                       p(69,aload_0),
p(7,putfield(po,ref)), % <Field Point po>
                                                                       p(70,dload(1)),
p(8,aload_0),
                                                                       p(71,dconst_0),
p(a8,new(circle)), % <Class Circle>
                                                                       p(72,dcmp1),
p(b8,dup),
                                                                       p(73,iflt((cylinder,(setheight,[double]),16))),
p(c8,1dc2_w(3.5)), % <Double 3.5>
                                                                       p(74.dload(1)).
p(d8,1dc2_w(22.0)), % <Double 22.0>
                                                                       p(75,goto((cylinder,(setheight,[double]),17)))]).
p(e8,1dc2_w(8.0)), % <Double 8.0>
p(f8,invokespecial((circle,[double,double,double]),init)),
                                                                      label1((cylinder,(setheight,[double]),16),[
           % <Mth Circle(do,do,do)>
                                                                       p(76, dconst_0),
p(g8,putfield(cir,ref)), % <Field Circle circle>
                                                                       p(77,putfield(height,double)), % <Field double height>
p(a33,halt)]).
                                                                       p(78,return)]).
class_vars(test,[var(privat,point,po),
                                                                      label1((cylinder,(setheight,[double]),17),[
```

```
p(79,putfield(height,double)), % <Field double height>
                                                                        class_vars(circle,[var(protct,double,radius)]).
p(80,return)]).
                                                                        cont(circle,(circle,[]),[
                                                                        p(103,aload_0),
                                                                        p(104,dconst_0),
cont(cylinder, (getHeight,[]),[
                                                                        p(105,dconst_0),
p(81,aload_0),
                                                                        p(106,invokespecial((point,[double,double]),init)),
p(82,getfield(height,double)), % <Field double height>
                                                                                      % <Method Point(double,double)>
p(83,dreturn)]).
                                                                        p(107,aload_0),
                                                                        p(108,dconst_0),
                                                                        p(109,invokevirtual((setradius,[double]))),
cont(cylinder,(area,[]),[
                                                                                            % <Method void setRadius(double)>
p(84,1dc2_w(2.0)), % < Double 2.0>
p(85,aload_0),
                                                                        p(110.return)]).
p(86,invokespecial((area,[]),other)), % <Method double area()>
p(87,dmul).
                                                                        cont(circle,(circle,[double,double,double]),[
p(88,1dc2_w(6.28318)), % < Double 6.28318>
                                                                        p(111,aload_0),
p(89,aload_0),
                                                                        p(112,dload(3)),
                                                                        p(113,dload(5)),
p(90,getfield(radius,double)), % <Field double radius>
p(91,dmul),
                                                                        p(114,invokespecial((point,[double,double]),init)),
p(92,aload_0),
                                                                                            % <Method Point(double,double)>
p(93,getfield(height,double)), % <Field double height>
                                                                        p(115,aload_0),
p(94,dmul),
                                                                         p(a116,dload(1)),
p (95, dadd),
                                                                         p(116,invokevirtual((setradius,[double]))),
p(96,dreturn)]).
                                                                                            % <Method void setRadius(double)>
                                                                        p(117,return)]).
cont(cylinder,(volume,[]),[
                                                                        cont(circle, (setradius, [double]),[
p(97,aload_0),
p(98,invokespecial((area,[]),other)), % <Method double area()>
                                                                         p(118,aload_0),
 p(99,aload_0),
                                                                         p(119,dload(1)),
p\,(\texttt{100,getfield(height,double))}\,,\quad \text{\%} \,\, \texttt{`Field double height'}
                                                                        p(120,dconst_0),
p(101,dmul),
                                                                         p(121,dcmpl),
p(102, dreturn)]).
                                                                        \tt p(122,iflt((circle,(setradius,[double]),11))),
                                                                         p(123,dload(1)),
class_vars(cylinder,[var(protct,double,height)]).
                                                                        p(124,goto((circle,(setradius,[double]),12)))]).
superc(cylinder,circle).
                                                                        {\tt label1} \; (({\tt circle}, ({\tt setradius}, [{\tt double}]), {\tt l1}), [
                                                                        p(125,dconst_0),
                                                                        {\tt p(126,putfield(radius,double)), \quad \% < Field \ double \ radius>}
Compiled from Circle.java
                                                                        p(127,return)]).
public synchronized class Circle extends Point
    / ACC_SUPER bit set /
                                                                        label1((circle,(setradius,[double]),12),[
                                                                        p(128,putfield(radius,double)), % <Field double radius>
                                                                        p(129,return)]).
    protected double radius:
/ D /
   public Circle();
                                                                        cont(circle,(getradius,[]),[
/ () V /
                                                                        p(130.aload 0).
   public Circle(double, double, double);
                                                                        p(131,getfield(radius,double)), % <Field double radius>
/ (DDD)V /
                                                                        p(132,dreturn)]).
   public void setRadius(double);
/ (D) V /
                                                                        cont(circle,(area,[]),[
   public double getRadius();
                                                                        p(133,ldc2_w(3.14159)), % <Double 3.14159>
/ ()D /
                                                                        p(134,aload_0),
                                                                        p(135,getfield(radius,double)), % <Field double radius>
   public double area();
                                                                        p(136,dmul),
   public java.lang.String toString();
                                                                        p(137,aload_0),
/ ()Ljava/lang/String; /
                                                                        p(138,getfield(radius,double)), % <Field double radius>
   public java.lang.String getName();
                                                                        p(139,dmul),
/ ()Ljava/lang/String; /
                                                                        p(140, dreturn)]).
}
*/
                                                                        superc(circle,point).
```

```
() V /
Compiled from Point.java
public synchronized class Point extends Shape
   / ACC SUPER bit set /
                                                                   class_vars(shape,[]).
   protected double x;
/ D /
                                                                   cont(shape,(area,[]),[
   protected double y;
                                                                    p(41,dconst_0),
/ D /
                                                                    p(42, dreturn)]).
   public Point(double, double);
                                                                   cont(shape,(volume,[]),[
/ (DD) V /
   public void setPoint(double, double);
                                                                    p(43,dconst_0),
/ V(dd) /
                                                                    p(44, dreturn)]).
   public double getX();
/ ()D /
                                                                   cont(shape,(shape,[]),[
   public double getY();
                                                                    p(52,return)]).
/ ()D /
   public java.lang.String toString();
                                                                   superc(shape,top).
/ ()Ljava/lang/String; /
                                                                   label((C1, (Mth, Sig), L), P) :-
   public java.lang.String getName();
   ()Ljava/lang/String; /
                                                                   label1((Cl, (Mth, Sig), L), P).
}
                                                                   cont(C1,Mth,null) :-
class_vars(point,[var(protct,double,x),
                                                                    \+ (C1,Mth) = (test,(init,[])),
         var(protct,double,y)]).
                                                                   \+ (C1,Mth) = (cylinder,(cylinder,
                                                                                [double,double,double,double])),
cont (point, (point, [double, double]),[
                                                                   \+ (C1,Mth) = (cylinder,(setheight,[double])),
 p(34,aload_0),
                                                                   \+ (C1,Mth) = (cylinder,(getHeight,[])),
 \+ (C1,Mth) = (cylinder,(area,[])),
p(36,aload_0),
                                                                   \+ (C1,Mth) = (cylinder,(volume,[])),
p(37,dload(1)),
                                                                   \+ (C1,Mth) = (circle,(circle,[])),
 p(38,dload(3)),
                                                                   \+ (C1,Mth) = (circle,(circle,[double,double,double])),
 p(39,invokevirtual((setpoint,[double,double]))),
                                                                   \+ (C1,Mth) = (circle,(setradius,[double])),
                  % <Method void setPoint(d,d)>
                                                                   \+ (C1,Mth) = (circle, (getradius, [])),
                                                                   \+ (C1,Mth) = (circle,(area,[])),
 p(40,return)]).
                                                                   \+ (C1,Mth) = (point,(point,[double,double])),
                                                                   \+ (C1,Mth) = (point,(setpoint,[double,double])),
cont (point, (setpoint, [double, double]),[
                                                                   \+ (C1,Mth) = (shape,(area,[])),
p(45,aload_0),
                                                                   \+ (C1,Mth) = (shape,(volume,[])),
p(46,dload(1)),
                                                                   \+ (C1,Mth) = (shape,(shape,[])),
p(47,putfield(x,double)), % <Field double x>
                                                                   \+ (C1.Mth) = (treetest.(main.[])).
p(48,aload_0),
                                                                   \+ (C1,Mth) = (treetest,(treetest,[])),
p(49.dload(3)).
p(50,putfield(y,double)), % <Field double y>
                                                                   \+ (C1,Mth) = (treenode,(treenode,[int])),
p(51,return)]).
                                                                   \+ (C1,Mth) = (treenode,(insert,[int])),
                                                                   \+ (C1,Mth) = (tree,(tree,[])),
superc(point, shape).
                                                                   \+ (C1,Mth) = (tree,(insertnode,[int])),
                                                                   \+ (C1,Mth) = (tree,(preordertraversal,[])),
                                                                   \+ (C1,Mth) = (tree,(preorderhelper,[ref])),
Compiled from Shape.java
                                                                   \+ (C1,Mth) = (tree,(inordertraversal,[])),
public abstract synchronized class Shape extends
                                                                   \+ (C1,Mth) = (tree,(inorderhelper,[ref])),
java.lang.Object
                                                                   \+ (C1,Mth) = (tree,(postordertraversal,[])),
   / ACC_SUPER bit set /
                                                                   \+ (C1,Mth) = (tree,(postorderhelper,[ref])).
   public double area();
/ ()D /
   public double volume();
/ ()D /
   public abstract java.lang.String getName();
/ ()Ljava/lang/String; /
   public Shape();
```

## A.3.1 A Specialised Program

Next, the result of specialising with respect to <-byte([],[triple...), in t2 below, with no input constraints.

```
t2 :-
        empty_env(test,I),
        class_vars(test,V),
        vars1(I,Cp1,V),
        cont(test,(init,[]),P),
        \verb|byte([],[triple(test,main_obj,((init,[]),test))]|,\\
             Cp1,P,emptystack).
byte([].
     [triple(test.main obj.((init.[]).test))].
         [[8,[test,[[test,[[],[]],[[],[]],[[],[]]],
         [[po,pair(point,1)],[cir,pair(circle,2)],
          [cvl,pair(cylinder,3)],
          [s1,pair(shape,4)],
          [s2,pair(shape,5)],
          [s3,pair(shape,6)],
          [s4,pair(shape,7)]]]],
          [(7,null),(6,null),(5,null),(4,null),
            (3,null),(2,null),(1,null)]],
            [p(1.aload 0).
             p(2,new(point)),
             p(3,dup),
             p(4,1dc2_w(7.0)),
             p(5,1dc2_w(11.0)),
             p(6,invokespecial((point,[double,double]),init)),
             p(7,putfield(po,ref)),
             p(8,aload_0),
             p(a8, new(circle)),
             p(b8,dup),
             p(c8,1dc2_w(3.5)),
             p(d8,1dc2_w(22.0)),
             p(e8,1dc2_w(8.0)),
             p(f8, invokespecial((circle, [double, double, double]),
                         init)),
             p(g8,putfield(cir,ref)),p(a33,halt)],
         emptystack) :-
      byte_1.
bvte 1 :-
      byte 2.
bvte 2 :-
      byte 3.
byte_3 :-
      byte_4.
byte_4 :-
      byte_5.
      byte_6(
        f6(locals([triple(test,main_obj,((init,[]),test))])),
            triple(test,main_obj,((init,[]),test)),[],
              point,8,7.0,X1,11.0,X2,[[9,[test,
```

```
[[test,[[],[]],[[],[]],[[],[]]],
               [[po,pair(point,1)],[cir,pair(circle,2)],
                [cyl,pair(cylinder,3)],
                [s1,pair(shape,4)],
                [s2,pair(shape,5)],
                [s3,pair(shape,6)],
                [s4,pair(shape,7)]]],
                  [(8,[point,[[point,[[],[]],[[y,x],[0.0,0.0]],
                  [[],[]]]],[]]),
                  (7, null), (6, null), (5, null), (4, null), (3, null),
                  (2, null), (1, null)]],
        \tt f6(code([p(7,putfield(po,ref)),p(8,aload_0),
                  p(a8,new(circle)),p(b8,dup),p(c8,ldc2_w(3.5)),
                  p(d8,1dc2_w(22.0)),p(e8,1dc2_w(8.0)),
                  p(f8,invokespecial((circle,[double,double,
                             double]),init)),
                  p(g8,putfield(cir,ref)),p(a33,halt)])),
         emptystack).
byte_6(X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12,X13):-
      byte_7(X5, X6, X1, X2, X3, X4, X7, X8, X9, X10, X11, X12, X13).
byte_7 (X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12,X13) :-
      byte_8(X1,X2,X7,X8,X9,X10,X3,X4,X5,X6,X11,X12,X13).
byte_8(X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12,X13):-
      byte_9(X7,X8,X9,X10,X1,X2,X3,X4,X5,X6,X11,X12,X13).
byte 9(X1.X2.X3.X4.X5.X6.X7.X8.X9.X10.X11.X12.X13) :-
      byte_10(X5,X6,X1,X2,X3,X4,X7,X8,X9,X10,X11,X12,X13).
byte_10(X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12,X13) :-
      byte_11 (X7,X1,X2,X3,X4,X5,X6,X8,X9,X10,X11,X12,X13).
byte_11(X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12,X13) :-
      {\tt byte\_12\,(X9\,,X1\,,X2\,,X3\,,X4\,,X5\,,X6\,,X7\,,X8\,,X10\,,X11\,,X12\,,X13)}\,.
byte_12(X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12,X13) :-
      \label{eq:byte_14} \ \text{byte_14} \ (\texttt{X1}\,,\texttt{X2}\,,\texttt{X3}\,,\texttt{X4}\,,\texttt{X5}\,,\texttt{X6}\,,\texttt{X7}\,,\texttt{X8}\,,\texttt{X9}\,,\texttt{X10}\,,\texttt{X11}\,,\texttt{X12}\,,\texttt{X13}\,,\texttt{X14}\,,\texttt{X15}) \; .
byte_14(X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12,X13,X14,X15) :-
      byte_15(X3,X1,X2,X4,X5,X6,X7,X8,X9,X10,X11,X12,X13,X14,X15).
byte_15(X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12,X13,X14,X15) :-
      {\tt update\_int\_70\,(X2,X3,setpoint\,,[double]\,,point\,,X13,X16,x\,,X1)\,,}
      byte_16(X2,X3,X1,X4,X5,X6,X7,X8,X9,X10,X11,X12,X16,X14,X15).
byte_16(X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12,X13,X14,X15) :-
      byte_17(X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12,X13,X14,X15).
byte_17(X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12,X13,X14,X15) :-
      byte 18(X5.X1.X2.X3.X4.X6.X7.X8.X9.X10.X11.X12.X13.X14.X15).
byte 18(X1.X2.X3.X4.X5.X6.X7.X8.X9.X10.X11.X12.X13.X14.X15) :-
      update int 70(X2.X3.setpoint.[double].point.X13.X16.v.X1).
      byte_19(X2,X3,X4,X5,X1,X6,X7,X8,X9,X10,X11,X12,X16,X14,X15).
byte_19(X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12,X13,X14,X15) :-
      {\tt byte\_20\,(X7\,,X8\,,X9\,,X10\,,X1\,,X2\,,X3\,,X4\,,X5\,,X6\,,X13\,,X14\,,X15)}\;.
byte_20(X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12,X13) :-
      code_id_cont_48(X1,X2,X3,X4,X11,X12,X13).
byte_21(X1,X2,X3,X4,X5) :-
      update_ptr_71(X2,X4,X6,X1),
      byt e_22(X3,X6,X5).
byte_22(X1,X2,X3) :-
      byte_23(X1,X2,X3).
byte_23(X1,[[X2|X3],X4],X5) :-
      X6 is X2+1,
      byte_24(X2,X1,X6,X3,X4,X5).
byte_24(X1,X2,X3,X4,X5,X6) :-
      byte_25(X1,X2,X3,X4,X5,X6).
byte_25(X1,X2,X3,X4,X5,X6) :-
```

```
byte_26(X1,X2,X3,X4,X5,X6).
                                                                          decide_m_37 (no, X1, X2, X3, X4, X5, X6, X7, X8) :-
byte_26(X1,X2,X3,X4,X5,X6):-
                                                                                superc_42(X2,X9),
      byte_27(X1,X2,X3,X4,X5,X6).
                                                                                 append1_35(X10,X9,[X11,X12|X13],X14,X1),
byte_27(X1,X2,X3,X4,X5,X6):-
                                                                                 update_s_39(X11,X15,X7,X8,X16),
      byte_28(X1,X2,X3,X4,X5,X6).
                                                                                 decide_s_43(X16,X17,X11,X12,X15,X18,X19,X7,X8),
byte_28(X1,X2,X3,X4,X5,X6):-
                                                                                 {\tt decide\_m\_37\,(X17,X1,X9,[X18,X19|X13],X10,X14,X6,X7,X8)}\,.
      update_inst_61([[X3|X4],[(X1,[circle,[],[]])|X5]],
                                                                          update_rf2_38(done,X1,X2,
                      X7, X1, circle, [double], radius, [], []),
                                                                                         [point,[[point,[[],[]],[[y,x],[11.0,7.0]],
      byte_29(X1,X2,X8,X9,X10,X7,X6).
                                                                                             [[],[]]],[]],X3,X4,X5,X6,X7,X8,X9,
byte_29(X1,X2,X3,X4,X5,X6,X7) :-
                                                                                              [(X2,[point,X10,[]]),(X3,null),
      byte_30(X1,X2,X3,X4,X5,X6,X7).
                                                                                              (X4.null), (X5.null), (X6.null),
                                                                                              (X7.null).(X8.null)|X9].X2.point.
byte 30(X1, X2, X3, X4, X5, X6, X7) :-
                                                                                              [[point,[[],[]],[[y,x],[11.0,7.0]],
      byte 31 (X1.X2.X3.X4.X5.X6.X7).
byte 31 (X1 X2 X3 X4 X5 X6 X7) :-
                                                                                             [[],[]]]],[],X10) :-
      byte_32(X1,X2,X3,X4,X5,X6,X7).
                                                                                true.
byte_32(X1,X2,X3,X4,X5,X6,X7) :-
                                                                          update_s_39([[],[]],[[],[]],X1,X2,no) :-
      update_inst_61(X6,X8,X1,point,[],y,[x],[0.0]),
                                                                          update_s_39([[X1 | X2],[X3 | X4]],[[X1 | X2],[X5 | X4]],X1,X5,yes) :-
      byte_6(
        f114(locals([
                                                                          update_s_39([[X1|X2],[X3|X4]],[[X1|X2],[X3|X5]],X6,X7,X8) :-
              triple(circle, X1, ((circle, [double, double, double]),
                      circle)),
                      3.5, X3, 22.0, X4, 8.0, X5])),
                                                                                 update_s_39([X2,X4],[X2,X5],X6,X7,X8).
        ff8(locals([triple(test,main_obj,((init,[]),test))])),
                                                                          decide_s_40 (yes, yes, X1, X2, X3, [X3|X2], X4, X5) :-
               pair(circle,X1),
                [\texttt{triple(test,main\_obj,((init,[]),test))} \,|\, \texttt{X2}]\,,
                                                                          decide_s_40 (no, X1, X2, X3, X4, [X2|X5], X6, X7) :-
               circle, X1, 22.0, X9, 8.0, X10, X8,
                                                                                 update_q_36(X3,X5,X6,X7,X1).
        f114(code([p(115,aload_0),p(a116,dload(1)),
                                                                          append_41([],X1,X1) :-
               p(116,invokevirtual((setradius,[double]))),
               p(117,return)])),
                                                                          append_41([X1|X2],X3,[X1|X4]) :-
        frame(ff8(code([p(g8,putfield(cir,ref)),p(a33,halt)])),X7)).
                                                                                append_41(X2,X3,X4).
update_rf2_33(done, X1, X2, X3, X4, X5, X6, X7, X8, X9, X10, X11, X12, X13, X14, superc_42(test, top) :-
              X15,X16,[(X2,[X3,X17,[]]),(X8,X9),
               (X10,null), (X11,null), (X12,null),
                                                                          superc_42(cylinder,circle) :-
               (X13,null), (X14,null), (X15,null) | X16], X2, X3,
              [[X3,[[],[]],[[X4|X5],[0.0|X6]],
                                                                          superc_42(circle,point) :-
              [[],[]]|X7],[],X17) :-
                                                                                true.
                                                                          superc_42(point, shape) :-
update_rf2_33(X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12,X13,
                                                                                true.
              X14, X15, X16, X17, [(X3, [X4, [[X4, [[], []],
                                                                          superc_42(shape,top) :-
              [[X5|X6],[0.0|X7]],[[],[]]]|X8],[]])|X18],
                                                                                true.
              X19,X20,X21,X22,X23) :-
                                                                          superc_42(treetest,top) :-
      X19=\=X3.
                                                                                true.
      update_rf2_44(X1,X2,X9,X10,X11,X12,X13,X14,X15,X16,
                                                                          superc 42(treenode.top) :-
                    X17,X18,X19,X20,X21,X22,X23).
                                                                                true.
decide_srch_34(retry, X1, X2, X3, X4, X5, X6, X7, X8, X9) :-
                                                                          superc 42(tree.top) :-
      update_rf2_47(X10,X11,X2,X12,X1,X6,X7,X8,X9),
                                                                                true.
      decide_srch_34(X10,X11,X2,X12,X4,X1,X6,X7,X8,X9).
                                                                          decide_s_43 (yes, yes, X1, X2, X3, X3, X2, X4, X5) :-
decide_srch_34(done,X1,X2,X3,X3,X4,X5,X6,X7,X8) :-
                                                                                true.
                                                                          decide_s_43 (no,X1,X2,X3,X4,X2,X5,X6,X7) :-
      true.
append1_35([],X1,X2,X3,[[X1|X2]|X3]):-
                                                                                update_s_39(X3,X8,X6,X7,X9),
                                                                                decide_s_45(X9,X1,X3,X8,X5,X6,X7).
      true.
append1_35([[X1 | X2] | X3], X4, X5, X6, [[X1 | X2] | X7]) :-
                                                                          update_rf2_44(done, X1, X2, [point, [[point, [[], []], [[y, x],
                                                                                           [11.0.7.0]].
      append1_35(X3,X4,X5,X6,X7).
                                                                                          [[],[]]],[]],X3,X4,X5,X6,X7,X8,X9,[(X2,
update_q_36([],X1,X2,X3,no) :-
                                                                                           [point, X10, []]),
                                                                                          (X3, null), (X4, null), (X5, null), (X6, null),
update_q_36([X1|X2],X3,X4,X5,X6) :-
                                                                                          (X7, null), (X8, null) | X9], X2, point,
      update_s_39(X1,X7,X4,X5,X8),
                                                                                          [[point,[[],[]],[[y,x],[11.0,7.0]],
      {\tt decide\_s\_40(X8,X6,X1,X2,X7,X3,X4,X5)}\;.
                                                                                           [[],[]]],[],X10) :-
decide_m_37(yes,X1,X2,X3,X4,X5,X6,X7,X8) :-
      append_41(X4,[[X2|X3]|X5],X6).
                                                                          {\tt update\_rf2\_44(X1\,,X2\,,X3\,,X4\,,X5\,,X6\,,X7\,,X8\,,X9\,,X10\,,X11\,,[\,(X3\,,X4)\,,}
```

```
(X5,null),(X6,null),(X7,null),(X8,null),
                                                                                       [s4,pair(shape,7)]]]],X3],X4) :-
             (X9,null),(X10,null)|X12],
                                                                              update_rf2_47(X5,X6,X2,X7,9,X8,X9,X10,X9),
              X13,X14,X15,X16,X17) :-
                                                                              decide_srch_34(X5, X6, X2, X7, X2, 9, X8, X9, X10, X9),
     X13=\=X3.
                                                                              update_rf_53(X10,po,X11),
      X13=\=X5,
                                                                              unique_ref_54(X11,X12),
     X13=\=X6.
                                                                              update_rf1_52(X2,X3,X12,X4).
     X13=\=X7.
                                                                        byte_51(X1,X2,X3,X4,X5,X6,X7,X8,X9) :-
     X13=\=X8.
                                                                              byte_55(X1,X2,X3,X4,X5,X6,X7,X8,X9).
     X13=\=X9.
                                                                        update_rf1_52([(X1,X2)|X3],[(X1,X4)|X3],X1,X4) :-
     X13=\=X10.
                                                                              true.
      update_rf2_46(X1, X2, X11, X12, X13, X14, X15, X16, X17).
                                                                        update_rf1_52([(X1,X2)|X3],[(X1,X2)|X4],X5,X6) :-
decide_s_45(yes,yes,X1,X2,X2,X3,X4) :-
                                                                              X1=\=X5
      true
                                                                              update_rf1_52(X3,X4,X5,X6).
                                                                        update rf 53([[X1.X2]|X3],X1.X2) :-
decide s 45(no.no.X1.X2.X1.X3.X4) :-
      true.
                                                                              true.
update_rf2_46(retry,8,[(1,pair(point,8))],[(1,pair(point,8))],
                                                                        update_rf_53([[X1,X2]|X3],X4,X5) :-
              1,X1,X2,X3,X4) :-
                                                                              \t x1=x4
                                                                              update_rf_53(X3,X4,X5).
      true.
update_rf2_47(done, X1, [(X2, [X3, X4, X5]) | X6],
                                                                        unique_ref_54(triple(X1,X2,X3),X2) :-
               [(X2,[X3,X7,X5])|X6],X2,X3,X4,X5,X7) :-
                                                                              true.
                                                                        unique_ref_54(pair(X1,X2),X2) :-
      true.
update_rf2_47(retry,X1,[(X2,pair(X3,X1))|X4],
                                                                              true.
              [(X2,pair(X3,X1))|X4],X2,X5,X6,X7,X8) :-
                                                                        byte_55(X1,X2,X3,X4,X5,X6,X7,X8,X9) :-
                                                                              byte_56(X1,X5,X6,X7,X2,X3,X4,X10,X8,X9).
      true.
update_rf2_47(retry, X1, [(X2, triple(X3, X1, X4)) | X5],
                                                                        byte 56(X1.X2.X3.X4.X5.X6.X7.X8.X9.X10) :
               [(X2,triple(X3,X1,X4))|X5],X2,X6,X7,X8,X9) :-
                                                                              byte_57(X1,X2,X3,X4,X5,X6,X7,X8,X9,X10).
                                                                        byte_57(X1,X2,X3,X4,X5,X6,X7,X8,X9,X10) :-
update_rf2_47(X1,X2,[(X3,X4)|X5],[(X3,X4)|X6],X7,X8,X9,X10,X11) :-
                                                                              byte_58(X1,X2,X3,X4,X5,X6,X7,X8,X9,X10).
      X7=\=X3.
                                                                        byte_58(X1,X2,X3,X4,X5,X6,X7,X8,X9,X10) :-
      update_rf2_47(X1, X2, X5, X6, X7, X8, X9, X10, X11).
                                                                              byte_59(X1,X2,X3,X4,X5,X6,X7,X8,X9,X10).
code_id_cont_48(f6(locals([triple(test,main_obj,
                                                                        byte_59(X1,X2,X3,X4,X5,X6,X7,X8,X9,X10) :-
                   ((init,[]),test))])),
                                                                              byte_60(X1,X2,X3,X4,X5,X6,X7,X8,X9,X10).
                 X1,X2,X3,X4,f6(code([p(7,putfield(po,ref)),
                                                                        byte_60(X1,X2,X3,X4,X5,X6,X7,X8,X9,X10) :-
                  p(8,aload_0),
                                                                              byte_62(X1,X2,X3,X4,X5,X6,X7,X8,X9,X10).
                  p(a8,new(circle)), p(b8,dup), p(c8,1dc2_w(3.5)),
                                                                        {\tt update\_inst\_61([[X1,[X2,X3,X4]],X5],[[X1,[X2,X6,X7]],X5],}
                  p(d8,1dc2_w(22.0)), p(e8,1dc2_w(8.0)),
                                                                                        main_obj, X8, X9, X10, X11, X12) :-
                  p(f8,invokespecial((circle,
                                                                              {\tt app\,end\_41\,(X3,[[X8,[[],[]],[[X10\,|X11],[0.0\,|X12]],[[],[]]]],}
                       [double,double,double]),init)),
                                                                                        X6).
                  p(g8,putfield(cir,ref)),p(a33,halt)])),X5):-
                                                                              append_41(X4,[],X7),
      byte_21(X1,X2,X3,X4,X5).
                                                                              true.
code_id_cont_48(f114(locals([triple(circle,X1,
                                                                        update_inst_61([X1,X2],[X1,X3],X4,X5,X6,X7,X8,X9) :-
                 ((circle, [double, double, double]), circle)),
                                                                              integer (X4),
                                                                              element_63(X4,X10,X11,X12,X2,X13,X14),
                  3.5.X2.22.0.X3.8.0.X41)).X5.X6.X7.X8.
                f114(code([p(115.aload 0).p(a116.dload(1)).
                                                                              append_41(X11,[[X5,[[],[]],[[X7|X8],[0.0|X9]],[[],[]]]],
                      p(116,invokevirtual((setradius,[double]))),
                                                                                        X15).
                      p(117,return)])),X9) :-
                                                                              append_41(X12,[],X16),
      byte_49(X5,X6,X7,X1,X2,X3,X4,X8,X9).
                                                                              element_63(X4,X10,X15,X16,X3,X13,X14).
byte_49(X1,X2,X3,X4,X5,X6,X7,X8,X9) :-
                                                                        byte_62(X1,X2,X3,X4,X5,X6,X7,X8,X9,X10) :-
      byte_51 (X4,X1,X2,X3,X5,X6,X7,X8,X9).
                                                                              byte_64(X1,X2,X3,X4,X5,X6,X7,X8,X9,X10).
update_ptr_50(triple(test,main_obj,((init,[]),test)),X1,X2,
                                                                        element_63(X1,X2,X3,X4,[(X1,[X2,X3,X4])|X5],[],X5) :-
              [[X1,[test,[[test,[[],[]],[[],[]],[[],[]]],
                                                                              true.
              [[po,pair(point,1)],[cir,pair(circle,2)],
                                                                        element_63(X1,X2,X3,X4,[(X5,X6)|X7],[(X5,X6)|X8],X9) :-
              [cyl,pair(cylinder,3)],[s1,pair(shape,4)],
                                                                              element_63(X1,X2,X3,X4,X7,X8,X9).
              [s2,pair(shape,5)],[s3,pair(shape,6)],
              [s4,pair(shape,7)]]]],X3],X4) :-
                                                                        byte_64(X1,X2,X3,X4,X5,X6,X7,X8,X9,X10) :-
      update_rf1_52(X2,X3,1,X4).
                                                                              byt e_65 (X1,X2,X3,X4,X5,X6,X7,X8,X9,X10).
update_ptr_50(pair(circle,9),X1,X2,[[X1,[test,
                                                                        byte_65(X1,X2,X3,X4,X5,X6,X7,X8,X9,X10) :-
               [[test,[[],[]],[[],[]],[[],[]]],
                                                                              update_int_70(circle,X1,setradius,[],circle,X9,X11,
               [[po,pair(point,1)],[cir,pair(circle,2)],
                                                                                            radius, 3.5),
               [cyl,pair(cylinder,3)],[s1,pair(shape,4)],
                                                                              byt e_66 (X1 ,X2 ,X3 ,X4 ,X5 ,X6 ,X7 ,X8 ,X11 ,X10) .
               [s2,pair(shape,5)],[s3,pair(shape,6)],
                                                                        byte_66(X1,X2,X3,X4,X5,X6,X7,X8,X9,X10) :-
```

```
byte_67(X5,X6,X7,X1,X2,X3,X4,X9,X10).
byte_67(ff8(locals([triple(test,main_obj,((init,[]),test))])),
             X1,X2,X3,X4,X5,X6,X7,
        {\tt frame(ff8(code([p(g8,putfield(cir,ref)),p(a33,halt)])),}
        X8)) :-
      byte_68(X1,X2,X7,X8).
{\tt byte\_68\,(X1\,,[triple(test,main\_obj,((init,[])\,,test))\,|X2]}\,,
               [[X3,[test,X4,X5]],X6],X7) :-
      {\tt update\_rf\_53(X5,cir,X8)}\,,
      unique_ref_54(X8,X9),
      {\tt update\_rf1\_52}\,({\tt X6},{\tt X10},{\tt X9},{\tt X1})\,,
      byte_69(X2,X3,X4,X5,X10,X7).
byte_69(X1,X2,X3,X4,X5,emptystack):-
      write([[X2,[test,X3,X4]],X5]).
update_int_70(X1,main_obj,X2,X3,X4,[[X5,[X1,X6,X7]],X8],
               [[X5,[X1,X9,X7]],X8],X5,X10) :-
      append1_35(X11,X1,X12,X13,X6),
      update_q_36(X12,X14,X5,X10,X15),
      decide_m_37(X15,X6,X1,X14,X11,X13,X9,X5,X10).
update_int_70(X1,X2,X3,X4,X5,[X6,X7],[X6,X8],X9,X10):-
      integer (X2),
      update_rf2_47(X11,X12,X7,X13,X2,X14,X15,X16,X17),
      decide_srch_34(X11,X12,X7,X13,X8,X2,X14,X15,X16,X17),
      append1_35(X18,X14,X19,X20,X15),
      update_q_36(X19,X21,X9,X10,X22),
      decide_m_37(X22,X15,X14,X21,X18,X20,X17,X9,X10),
update_ptr_71(triple(test,main_obj,((init,[]),test)),
               [[X1,[test,X2,X3]],X4],[[X1,[test,X2,X3]],X5],
      update_rf_53(X3,po,X7),
      unique_ref_54(X7,X8),
      update_rf1_52(X4,X5,X8,X6).
update_ptr_71(pair(circle,X1),[X2,X3],[X2,X4],X5) :-
      integer(X1),
      update_rf2_47(X6,X7,X3,X8,X1,X9,X10,X11,X10),
      decide_srch_34(X6,X7,X3,X8,X3,X1,X9,X10,X11,X10),
      update_rf_53(X11,po,X12),
      unique_ref_54(X12,X13),
      {\tt update\_rf1\_52}\,({\tt X3,X4,X13,X5})\,.
```

## **Bibliography**

- [And94] Lars Ole Andersen. Program Analysis and Specialization for the C programming Language. PhD thesis, DIKU, University of Copenhagen, Denmark, May 1994.
- [App98] Andrew W. Appel. SSA is functional programming. *ACM SIGPLAN Notices*, 33(4):17–20, 1998.
- [Asa99] Kenichi Asai. Binding-time analysis for both static and dynamic expressions. In A. Cortesi and G. Filé, editors, *Static Analysis Symposium*, pages 117–133. Springer-Verlag, LNCS 1694, 1999.
- [ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1986.
- [BCD90] A. Bossi, N. Cocco, and S. Dulli. A method for specializing logic programs. ACM Transactions on Programming Languages and Systems, 12(2):253–302, 1990.
- [BF96] Sandrine Blazy and Philippe Facon. An automatic interprocedural analysis for the understanding of scientific application programs. In O. Danvy, Robert Glück, and Peter Thieman, editors, *International Seminar on Partial Evaluation*, pages 1–16. Springer-Verlag, LNCS 1110, 1996.

[BGS94] David F. Bacon, Susan Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, December 1994.

- [BK96] F. Benoy and A. King. Inferring argument size relations in CLP( $\mathcal{R}$ ). In Proceedings of the 6th International Workshop on Logic Program Synthesis and Transformation, pages 204–223, Sweden, 1996. Springer-Verlag, LNCS 1207.
- [BMSU86] Francois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D. Ullman. Magic sets and other strange ways to implement logic programs. In *Proceedings of the Fifth ACM Symposium on Principles of Database Systems*, pages 1–15, 1986.
- [Boo94] Grady Booch. Object-Oriented Analysis and Design. Benjamin Cummings, second edition, 1994.
- [CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximations of fixpoints. In Conference Record of the 4th Annual ACM Symposium on Principles of Programming Languages, pages 238–252, Los Angeles, 1977.
- [CC92a] Patrick Cousot and Radhia Cousot. Abstract interpretation and application to logic programs. The Journal of Logic Programming, pages 103–179, 1992.
- [CC92b] Patrick Cousot and Radhia Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In Programming Language Implementation and Logic Programming, pages 269–295, Leuven, Belgium, 1992. Springer-Verlag, LNCS 631.
- [CD91] Charles Consel and Olivier Danvy. Static and dynamic semantic processing. In ACM Press, editor, Conference Record of the Eighteenth An-

nual ACM Symposium on Principles of Programming Languages, pages 14–24, Orlando, Florida, 1991.

- [CDG<sup>+</sup>99] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D Lugiez, S. Tison, and M. Tommasi. Tree Automata Techniques and Applications. 1999. http://www.grappa.univ-lille.fr/tata.
- [CE81] Keith L. Clark and M. H. Van Emden. Consequence verification flowcharts. *IEEE Transactions on Software Engineering*, 7(1):52–60, 1981.
- [CFR<sup>+</sup>91] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.
- [CH78] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the Conference Record of the 5th ACM Symposium on Principles of Programming Languages*, pages 84–97, Albuquerque, New Mexico, 1978.
- [CK91] Charles Consel and Siau Cheng Khoo. Parameterized partial evaluation. SIGPLAN Notices, 26(6):92–106, 1991.
- [Con93] Charles Consel. A tour of schism. In ACM Press, editor, Partial Evaluation and Semantics-based Program Manipulation, pages 145–154, New York, NY, 1993.
- [Cou97] Patrick Cousot. Abstract interpretation based static analysis parametrized by semantics. In *Proceedings of the Fourth International Symposium on Static Analysis*, SAS'97, pages 388–394, Paris, France, 1997. LNCS 1302, Springer-Verlag.

[CPPGM91] Alberto Coen-Porisini, Flavio De Paoli, Carlo Ghezzi, and Dino Mandrioli. Software specialization via symbolic execution. *IEEE Transactions* on Software Engineering, 17(9):884–899, 1991.

- [Das98] Manuvir Das. Partial Evaluation using Dependence Graphs. PhD thesis, University of Wisconsin-Madison, USA, 1998.
- [Die96] Stephan Diehl. Semantics-Directed Generation of Compilers and Abstract Machines. PhD thesis, University of Saarbruecken, Germany, 1996.
- [dWG93] D.A. de Waal and J. Gallagher. Logic program specialisation with deletion of useless clauses (poster asbtract). In D. Miller, editor, Proceedings of the International Logic Programming Symposium. MIT Press, 1993.
  Full version in CSTR-92-33, Dept. Computer Science, U. of Bristol.
- [EK76] M. H. Van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. Journal of the ACM, 23(4):733-742, October 1976.
- [Gal91] John P. Gallagher. A system for specialising logic programs. Technical Report TR-91-32, University of Bristol, November 1991.
- [Gal93] John P. Gallagher. Tutorial on specialisation of logic programs. In Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, pages 88–98, Copenhagen, Denmark, 1993. ACM Press.
- [Gau97] Ronan Gaugne. Static debugging of C programs: Detection of pointer errors in recursive data structures. Technical Report RR-3232, INRIA, Institut National de Recherche en Informatique et en Automatique, August 1997.

[GB90] J. Gallagher and M. Bruynooghe. Some low-level source transformations for logic programs. In *Proceedings of Meta90 Workshop on Meta Programming in Logic*. Katholieke Universiteit Leuven, Belgium, 1990.

- [GB91] J. Gallagher and M. Bruynooghe. The derivation of an algorithm for program specialisation. New Generation Computing, 9(3&4):305–333, 1991.
- [GDL95] R. Giacobazzi, S. Debray, and G. Levi. Generalized Semantics and Abstract Interpretation for Constraint Logic Programs. *The Journal of Logic Programming*, 25(3):191–248, 1995.
- [GdW92] John P. Gallagher and D. A. de Waal. Regular approximations of logic programs and their uses. Technical Report TR-92-06, University of Bristol, March 1992.
- [GdW94] J. Gallagher and D.A. de Waal. Fast and precise regular approximation of logic programs. In P. Van Hentenryck, editor, Proceedings of the International Conference on Logic Programming (ICLP'94), Santa Margherita Ligure, Italy, pages 599–613. MIT Press, 1994.
- [GL96] John P. Gallagher and L. Lafave. Regular approximation of computation paths in logic and functional languages. In O. Danvy, R. Glück, and P. Thiemman, editors, *Partial Evaluation*, pages 115–136. Springer-Verlag, LNCS 1110, 1996.
- [GM97] Valérie Gouranton and Daniel Le Métayer. Formal development of static program analysers. In *Proceedings of the 8th Israeli Conference on Computer Systems and Software Engineering*, pages 101–110, Israel, 1997.
- [GMS98] Arne John Glenstrup, Henning Makholm, and Jens Peter Secher. C-mix: Specialization of C programs. Notes from the Partial Evaluation Summer School held at DIKU Copenhagen, Denmark, 1998.

[GP00] John P. Gallagher and Julio C. Peralta. Using regular approximations for generalisation during partial evaluation. In Workshop on Partial Evaluation and Program Manipulation, pages 44–51. ACM Press, 2000. Also in ACM SIGPLAN Notices 34(11), Nov-99.

- [Gup99] Gopal Gupta. Horn logic denotations and their uses. In The Logic Programming Paradigm: A 25-Year Perspective. Springer Series in Artificial Intelligence, 1999.
- [Har99] Pieter H. Hartel. LETOS-a lightweight execution tool for operational semantics. Software—Practice & Experience, 29(15):1379-1416, September 1999.
- [HCC94] Pascal Van Hentenryck, Agostino Cortesi, and Baudouin Le Charlier.

  Type analysis of Prolog using type graphs. The Journal of Logic Programming, 22(3):179–210, 1994.
- [HDL<sup>+</sup>98] John Hatcliff, Matthew B. Dwyer, Shawn Laubach, Jason Mayans, and Nanda Muhammad. Automatically specializing software for finite state verification. Technical Report TR-98-4, Kansas State University, Department of Computing and Information Sciences, 1998.
- [Hen90] Matthew Hennessy. The Semantics of Programming Languages: An Elementary Introduction using Structural Operational Semantics. John Wiley and Sons, 1990.
- [HK99] Jacob M. Howe and Andy King. Specialising finite domain programs using polyhedra. Technical Report CS-99-16, Universita' Ca'Foscari Di Venezia, Dipartamento di Informatica, 1999. Full paper to appear in vol. 1817 of LNCS.
- [HMT98] John Hatcliff, Torben Æ. Morgensen, and Peter Thiemann, editors. Partial Evaluation: Practice and Theory, Copenhagen, Denmark, July 1998.
   DIKU, Springer-Verlag, LNCS 1706. International Summer School.

[HNC97] Luke Hornof, Jaques Noyé, and Charles Consel. Effective specialization of realistic programs via use sensitivity. In Pascal Van Hentenryck, editor, *Static Analysis Symposium*, pages 293–318. Springer-Verlag, LNCS 1302, 1997.

- [Hol95] C. Holzbaur. Ofai clp(q,r) manual, edition 1.3.3. Technical Report TR-95-09, Austrian Research Institute for Artificial Intelligence, Vienna, 1995.
- [HS91] Timothy J. Hickey and Donald A. Smith. Toward the partial evaluation of CLP languages. In *PEPM*, *SIGPLAN Notices Vol. 26 No. 9*, pages 43–51. ACM Press, 1991.
- [HU79] J. E. Hopcroft and J. D. Ullman. Introduction to Automata Theory, Languages and Computation. Addison Wesley, Reading, Mass., 1979.
- [HWD92] Manuel Hermanegildo, R. Warren, and S. K. Debray. Global flow analysis as a practical compilation tool. *The Journal of Logic Programming*, 13(2 and 3):349–366, July 1992.
- [JB92] G. Janssens and M. Bruynooghe. Deriving descriptions of possible values of programs by means of abstract interpretation. *The Journal of Logic Programming*, 13(2–3):295–258, 1992.
- [JGS93] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. Partial Evaluation and Automatic Program Generation. Prentice Hall International Series, 1993.
- [JM82] Neil D. Jones and Steven S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures.

  In ACM Press, editor, Conference Record of the Ninth Symposium on Principles of Programming Languages, pages 66–74, 1982.
- [JM94] Joxan Jaffar and Michael J. Maher. Constraint logic programming: A survey. The Journal of Logic Programming, 19(20):503–581, 1994.

[JN95] Neil D. Jones and Flemming Nielson. Abstract interpretation: a semantics-based tool for program analysis. In *Handbook of Logic in Computer Science*, 122 pages. Oxford University Press, 1995.

- [Jon88] Neil D. Jones. Automatic program specialisation: A re-examination from basic principles. In D. Bjørner, A. P. Erschov, and N. D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 225–282. Amsterdam: North-Holland, 1988.
- [Jon96] Neil D. Jones. What not to do when writing an interpreter for specialisation. In Olivier Danvy, Robert Glück, and Peter Thiemann, editors, Partial Evaluation, International Seminar, pages 216–237, Dagstuhl Castle, Germany, 1996. Springer-Verlag, LNCS 1110.
- [Jon97] Neil D. Jones. Combining abstract interpretation and partial evaluation. In Pascal V. Hentenryck, editor, *Symposium on Static Analysis*, (SAS'97), pages 396–405. Springer-Verlag, LNCS 1302, 1997.
- [KKZG95] Paul Kleinrubatscher, Albert Kriegshaber, Robert Zöling, and Robert Glück. Fortran program specialization. SIGPLAN Notices, 30(4):61–70, 1995.
- [Knu73] Donald E. Knuth. The Art of Computer Programming, volume 3 of Sorting and Searching. Addison-Wesley Publishing Company, 1973.
- [Kow79] Robert Kowalski. Logic for Problem Solving. North Holland, 1979.
- [KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language: ANSI C.* Prentice-Hall, 2nd edition, 1988.
- [Laf99] Laura Lafave. A Constraint-based Partial Evaluator for Functional Logic Programs and its Application. PhD thesis, University of Bristol, UK, 1999.

[Leu97] Michael Leuschel. Advanced Techniques for Logic Program Specialisation. PhD thesis, Katholieke Universiteit Leuven, Department of Computer Science, Leuven, Belgium, May 1997.

- [Leu98] Michael Leuschel. Program specialisation and abstract interpretation reconciled. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 220–234, Manchester, UK, 1998. MIT Press.
- [LG97] L. Lafave and John P. Gallagher. Constraint-based partial evaluation of rewritting-based functional logic programs. In N. Fuchs, editor, LOP-STR'97, pages 168–188. Springer-Verlag, LNCS 1463, 1997.
- [LJ99] Michael Leuschel and Jesper Jørgensen. Efficient specialisation in Prolog using a hand-written compiler generator. Technical Report DSSE-TR-99-6, Department of Electronics and Computer Science, University of Southampton, September 1999.
- [Llo87] J. W. Lloyd. Foundations of Logic Programming. Springer Verlag, 2nd edition, 1987.
- [LM96] M. Leuschel and B. Martens. Global control for partial deduction through characteristic atoms and global trees. In *Partial Evaluation*, pages 263–283. Springer-Verlag, LNCS 1110, 1996.
- [LM99] Michael Leuschel and Thierry Massart. Infinite state model checking by abstract interpretation and program specialisation. Technical Report CS-99-16, Universita' Ca'Foscari Di Venezia, Dipartamento di Informatica, 1999. Full paper to appear in vol. 1817 of LNCS.
- [LS91] J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. The Journal of Logic Programming, 11(3&4):217–242, 1991.

[MB92] T. Mogensen and A. Bondorf. Logimix: A self-applicable partial evaluator for Prolog. In K.-K. Lau and T. Clement, editors, LOPSTR 92.
Workshops in Computing, pages 214–227. Springer-Verlag, 1992.

- [Mey91] Uwe Meyer. Techniques for partial evaluation of imperative programs. In Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation, pages 94–115, New Haven, Conneticut, 1991. ACM Press.
- [MG95] Bern Martens and John P. Gallagher. Ensuring global termination of partial deduction while allowing flexibel polyvariance. In L. Sterling, editor, *Proceedings of ICLP'95*, pages 597–613, Kanagawa, Japan, 1995.
- [Mil99] Per Mildner. Type Domains for Abstract Interpretation: A Critical Study. PhD thesis, Uppsala University Information Technology, Sweden, 1999.
- [Mis84] P. Mishra. Towards a theory of types in Prolog. In *International Symposium on Logic Programming*, pages 289–298. IEEE, 1984.
- [MNL88] K. Marriot, L. Naish, and J.-L. Lassez. Most specific logic programs. In Proceedings of the Fifth International Conference and Symposium on Logic Programming, volume 2, pages 909–923, Washington, Seattle, 1988. MIT Press.
- [Mos91] Peter D. Mosses. An introduction to action semantics. Technical Report PB-370, Dept. of Computer Science, Univ. of Aarhus, 1991.
- [Mou97] Bárbara Moura. Bridging the Gap Between Functional and Imperative Languages. PhD thesis, Institut de Formation Supériore en Informatique et Communication, Université de Rennes, France, April 1997.
- [MS96] Daniel Le Métayer and David Schmidt. Structural operational semantics as a basis for static program analysis. *ACM Computing Surveys*, 28(2):340–343, June 1996.

[MS98] Kim Marriot and Peter J. Stuckey. *Programming with Constraints: An Introduction*. MIT Press, 1998.

- [Muc97] Steven S. Muchnick. Advanced Compiler Design and Implementation.

  Morgan Kaufman, 1997.
- [Nie89] Flemming Nielson. Two-level semantics and abstract interpretation.

  Theoretical Computer Science, (69):117–242, 1989.
- [Nie96] Flemming Nielson. Semantics-directed program analysis: A toolmaker's perspective. In *Third International Symposium*, SAS'96, pages 2–21. Springer-Verlag, LNCS 1145, 1996.
- [Nie97] Flemming Nielson. Perspectives on program analysis. *ACM SIGPLAN Notices*, 32(1):89–91, January 1997.
- [NN88] Flemming Nielson and Hanne Riis Nielson. Two-level semantics and code generation. *Theoretical Computer Science*, (56):59–133, 1988.
- [NN92] Hanne Riis Nielson and Flemming Nielson. Semantics with Applications. John Wiley and Sons, 1992.
- [NN97] Hanne Riis Nielson and Flemming Nielson. Analysis of programs. In Encyclopedia of Mathematics, Supplement, volume 1. Kluwer Academic Press, 1997.
- [NP92] Vivek Nirke and William Pugh. Partial evaluation of high-level imperative programming languages, with applications in hard real-time systems. In ACM Press, editor, Conference Record of the Nighteenth Symposium on Principles of Programming Languages, pages 269–280, Albuquerque, New Mexico, 1992.
- [Pet98] Mikael Pettersson. Compiling natural semantics. In *Lecture Notes in Computer Science*, volume 1549. Springer Verlag, 1998.

[PG99] Julio C. Peralta and John P. Gallagher. Imperative program specialisation: An approach using CLP. Technical Report CS-99-16, Universita' Ca'Foscari Di Venezia, Dipartamento di Informatica, 1999. Full paper to appear in vol. 1817 of LNCS.

- [PGS98] Julio C. Peralta, John P. Gallagher, and Hüseyin Sağlam. Analysis of imperative programs through analysis of constraint logic programs. In G. Levi, editor, Static Analysis Symposium, pages 246–261. Springer-Verlag, LNCS 1503, 1998.
- [Plo70] Gordon Plotkin. Building in equational theories. *Machine Intelligence*, 5:153–163, 1970.
- [PP94] Alberto Pettorossi and Maurizio Proietti. Transformations of logic programs: Foundations and techniques. *The Journal of Logic Programming*, 19(20):261–320, 1994.
- [PPR97] Alberto Pettorossi, Maurizio Proietti, and Sophie Renault. Reducing nondeterminism while specializing logic programs. In Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 414–427, Paris, France, 15–17 January 1997.
- [Rep98] Thomas Reps. Program analysis via graph reachability. Technical Report TR-1386, Computer Sciences Department, University of Wisconsin, August 1998. Extended version of invited paper in Proceedings of ILPS97.
- [Rey70] J. C. Reynolds. Transformational systems and the algebraic structure of atomic formulas. *Machine Intelligence*, pages 135–151, 1970.
- [RLK86] J. Rohmer, R. Lescoeur, and J. M. Kersit. The Alexander method—a technique for the processing of recursive axioms in deductive databases.

  New Generation Computing, (4):273–285, 1986.

[Ros89] Brian J. Ross. The partial evaluation of imperative programs using Prolog. In H. Abramson and M. H. Rogers, editors, Meta-Programming in Logic Programming, pages 341–363. MIT Press, 1989.

- [RRS95] C. R. Ramakrishnan, I. V. Ramakrishnan, and R. C. Sekar. A symbolic constraint solving framework for analysis of constraint logic programs. In Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation, pages 12–23, June 1995.
- [RS98] John L. Ross and Mooly Sagiv. Building a bridge between pointer aliases and program dependences. In European Symposium On Programming, pages 221–235, Lisbon, Portugal, 1998. Springer-Verlag, LNCS 1381.
- [Sağ98] H. Sağlam. A Toolkit for Static Analysis of Constraint Logic Programs. PhD thesis, Bristol University, Department of Computer Science, Bristol, U.K., March 1998.
- [Sah91] Dan Sahlin. An Automatic Partial Evaluator for Full Prolog. PhD thesis,The Royal Institute of Technology, Stockholm, Sweden, May 1991.
- [Sch86] David A. Schmidt. *Denotational Semantics*. Boston Massachusetts: Allyn and Bacon, 1986.
- [SFRW98] Mooly Sagiv, Nissim Francez, Michael Rodeh, and Reinhard Wilhelm. A logic-based approach to program flow analysis. Acta Informatica, 35(6):457-504, June 1998.
- [SG98a] H. Sağlam and J. Gallagher. Constrained regular approximation of logic programs. In N. Fuchs, editor, Logic Program Synthesis and Transformation (LOPSTR'97), pages 282–299. Springer-Verlag, LNCS 1463, 1998.
- [SG98b] Morten Heine B. Sørensen and Robert Glück. Introduction to super-compilation. In John Hatcliff, Torben Æ. Morgensen, and Peter Thiemann, editors, *Partial Evaluation: Practice and Theory*, pages 246–270. Springer-Verlag, LNCS 1706, 1998.

[SK95] Kenneth Slonneger and Barry L. Kurtz. Formal Syntax and Semantics of Programming Languages. Addison-Wesley Publishing Company, 1995.

- [Smi91] Donald A. Smith. Partial evaluation of pattern matching in constraint logic programming languages. In *PEPM*, *SIGPLAN Notices Vol. 26 No.* 9, pages 62–71. ACM Press, 1991.
- [VCH96] Clark Verbrugge, Phong Co, and Laurie Hendren. Generalized constant propagation a study of C. In *Compiler Construction'96*, pages 74–90. Springer-Verlag, LNCS 1060, 1996.
- [VM97] Wim Vanhoof and Bern Martens. To parse or not to parse. In N. Fuchs, editor, Logic Program Synthesis and Transformation (LOPSTR'97), pages 322–342. Springer-Verlag, LNCS 1463, 1997.
- [VMSV98] Wim Vanhoof, Bern Martens, Danny De Schreye, and Karel De Vlaminck. Specialising the other way around. In Joxan Jaffar, editor, Proceedings of the Joint International Conference and Symposium on Logic Programming, pages 279–293. MIT Press, 1998.
- [WZ85] Mark N. Wegman and Frank Kenneth Zadeck. Constant propagation with conditional branches. In Conference Record of the Twelfth Symposium on POPL, pages 291–299, New Orleans, Luisiana, 1985.
- [YS91] Eyal Yardeni and Ehud Shapiro. A type system for logic programs. The Journal of Logic Programming, pages 125–153, 1991.
- [Zob87] J. Zobel. Derivation of polymorphic types for Prolog programs. In Proceedings of the Fourth International Conference on Logic Programming, pages 817–838. MIT Press, 1987.