Analyzing Logic Programs with Dynamic Scheduling

Kim Marriott
Dept. of Computer Science
Monash University, Clayton Vic 3168
Australia
marriott@cs.monash.edu.au

María José García de la Banda   Manuel Hermenegildo
Facultad de Informática - UPM
28660-Boadilla del Monte, Madrid
Spain
{maria,herme}@fi.upm.es

Abstract

Traditional logic programming languages, such as Prolog, use a fixed left-to-right atom scheduling rule. Recent logic programming languages, however, usually provide more flexible scheduling in which computation generally proceeds left-to-right but in which some calls are dynamically "delayed" until their arguments are sufficiently instantiated to allow the call to run efficiently. Such dynamic scheduling has a significant cost. We give a framework for the global analysis of logic programming languages with dynamic scheduling and show that program analysis based on this framework supports optimizations which remove much of the overhead of dynamic scheduling.

1 Introduction

The first logic programming languages, such as DEC-10 Prolog, used a fixed scheduling rule in which all atoms in the goal were processed left-to-right. Unfortunately, this meant that programs written in a clean, declarative style were often very inefficient, only terminated when certain inputs were fully instantiated or "ground", and (if negation was used) produced wrong results. For this reason there has been widespread interest in a class of "second-generation" logic programming languages, such as IC-Prolog, NU-Prolog, Prolog-II, Sicstus-Prolog, Prolog-III, CHIP, Prolog M, and SEPIA, etc., that provide more flexible scheduling in which computation generally proceeds left-to-right but in which some calls are dynamically "delayed" until their arguments are sufficiently instantiated to allow the call to run efficiently. Such dynamic scheduling overcomes the problems associated with traditional Prologs and their fixed scheduling. First, it allows the same program to have many different and efficient operational semantics as the operational behaviour depends on which arguments are supplied in the query. Thus, programs really behave efficiently as relations, rather than as functions. Second, the treatment of negation is sound, as negative calls are delayed until all arguments are ground. Third, it allows intelligent search in combinatorial constraint problems. Finally, dynamic scheduling allows a new style of programming in which Prolog procedures are viewed as processes which communicate asynchronously through shared variables.

Unfortunately, dynamic scheduling has a significant cost; goals, if affected by a delay declaration, must be checked to see whether they should delay or not; upon variable binding, possibly delayed calls must be woken or put in a "pending" list, so that they are woken before the next goal is executed; also, few register allocation optimizations can be performed for delayed goals; finally, space needs to be allocated for delayed goals until they are woken [1]. Furthermore, global dataflow analyses used in the compilation of traditional Prologs, such as mode analysis, are not correct with dynamic scheduling. This means that compilers for languages with dynamic scheduling are currently unable to perform optimizations which improve execution speed of traditional Prologs by an order of magnitude [19, 21, 31, 32, 33]. However, it is not simple to extend analyses for traditional Prologs to languages with dynamic scheduling, as in existing analyses the fixed scheduling is crucial to ensure correctness and termination.

Here we develop a framework for global dataflow analysis of logic languages with dynamic scheduling. This provides the basis for optimizations which remove the overhead of dynamic scheduling and promises to make the performance of logic languages with dynamic scheduling competitive with traditional Prolog.

First, we give a denotational semantics for languages with dynamic scheduling. This provides the semantic basis for our generic analysis. The main difference with denotational definitions for traditional Prolog is that sequences of delayed atoms must also be abstracted and are included in "calls" and "answers". A key feature of the semantics is to approximate sequences of delayed atoms by multisets of atoms which are annotated to indicate if they are *possibly* delayed or if they are *definitely* delayed. The use of multisets instead of sequences greatly simplifies the semantics with, we believe, little loss of precision. This is because in most "real" programs delayed atoms which wake at the same time are independent while delayed atoms which are dependent will be woken at different times.

Second, we give a generic global dataflow analysis algorithm which is based on the denotational semantics. Correctness is formalized in terms of abstract interpretation [7]. The analysis gives information about call arguments and the delayed calls, as well as implicit information about possible call schedulings at runtime. The analysis is generic in the sense that it has a parametric domain and various parametric functions. The parametric domain is the descriptions chosen to approximate sets of term equations. Different choices of descriptions and associated parametric functions provide different information and give different accuracy. The parametric functions also allow the analysis to be tailored to particular system or language dependent criteria for delaying and waking calls. Implementation of the analysis is by means of a "memoization table" in which information about the "calls" and their "answers" encountered in the derivations from a particular goal are iteratively computed.

Finally, we demonstrate the utility and practical importance of the dataflow analysis algorithm. We sketch an example instantiation of the generic analysis which gives information about groundness and freeness of variables in the delayed and actual calls. Information from the example analysis can be used to optimize target code in many different ways. In particular, it can be used to reduce the overhead of dynamic scheduling by removing unnecessary tests for delaying and awakening and by reordering goals so that atoms are not delayed. It can also be used to perform optimizations used in the compilation of traditional Prolog such as: recognizing determinate code and so allowing unnecessary backtrack points to be deleted; improving the code generated for unification; recognizing calls which are "independent" and so allow the program to be run in parallel, etc. Preliminary test results, given here, show that the analysis and associated optimizations used to reduce the overhead of dynamic scheduling give significant improvements in the performance of these languages.

Abstract interpretation of standard Prolog was suggested by Mellish [26] as a way to formalize mode analysis. Since then, it has been an active research area and many frameworks and applications have been given, for example see [9]. The approach to program analysis taken here is based on the denotational approach of Marriott *et. al.* [25]. A common implementation of abstract interpretation based analyses of Prolog is in terms of "memoization tables" [11, 13] which our analysis generalizes. To our knowledge this is the first paper to consider the global dataflow analysis of logic programming languages with delay. Related work includes Marriott *et. al.* [24] which gives a dataflow analysis for a logic programming language in which negated calls are delayed until their arguments are fully ground. However the analysis does not generalize to the case considered here as correctness relies on calls only being woken when all of their arguments are ground. Other related work is the global analysis of concurrent constraint programming languages [4, 5, 6, 14]. These languages differ from the languages considered here as they do not have a default left-to-right scheduling but instead the compiler or interpreter is free to choose any scheduling. Thus, program analysis must be correct for all schedulings. In our setting, knowledge of the default scheduling allows much more precise analysis. Related work also includes Gudeman *et. al.* [16] and Debray [12] which investigate local analyses and optimization for the compilation of Janus, a concurrent constraint programming language. The optimizations include reordering and removal of redundant suspension conditions. Debray [10] studies global analysis for compile-time fixed scheduling rules other than left-to-right. However this approach does not work for dynamic scheduling, nor for analyses to determine "freeness" information. Finally, Hanus [17] gives an analysis for improving the residuation mechanism in functional logic programming languages. This analysis handles the delay and waking of equality constraints, but does not easily extend to handle atoms as these may spawn subcomputations which in turn have delayed atoms.

In the next section we give a simple example to illustrate the usefulness of dynamic scheduling and the type of information our analysis can provide. In Section 3 we give the operational semantics of logic languages with dynamic scheduling. In Section 4 we review abstract interpretation and introduce various descriptions used in the analysis. In Section 5 we give the denotational semantics. In Section 6 we give the generic analysis, and in Section 7 we give modifications which ensure termination. In Section 8 we give an example analysis. Section 9 presents some performance results and in Section 10 we conclude.

2 Example

The following program adapted from Naish [30], illustrates the power of allowing calls to delay and the information our analysis can provide. The program `permute` is a simple definition of the relationship that the first argument is a permutation of the second argument. It makes use of the procedure $\mathbf{delete(X, Y, Z)}$ which holds if $\mathbf{Z}$ is the list obtained by removing $\mathbf{X}$ from the list $\mathbf{Y}$.

$$
\begin{array}{lll}
\texttt{permute(X,Y)} & \leftarrow & \texttt{X} = \texttt{nil}, \\
 & & \texttt{Y} = \texttt{nil}. \\
\texttt{permute(X,Y)} & \leftarrow & \texttt{X} = \texttt{U} : \texttt{X1}, \\
 & & \texttt{delete(U,Y,Z)}, \\
 & & \texttt{permute(X1,Z)}. \\
 & & \\
\texttt{delete(X,Y,Z)} & \leftarrow & \texttt{Y} = \texttt{X} : \texttt{Z}. \\
\texttt{delete(X,Y,Z)} & \leftarrow & \texttt{Y} = \texttt{U} : \texttt{Y1}, \\
 & & \texttt{Z} = \texttt{U} : \texttt{Z1}, \\
 & & \texttt{delete(X,Y1,Z1)}.
\end{array}
$$

Note that uppercase letters denote variables. Clearly the relation declaratively given by **permute** is symmetric. Unfortunately, the behavior of the program with traditional Prolog is not: Given the query, $\mathbf{Q1}$,

$$? - \texttt{permute(X, a : b : nil)}$$

Prolog will correctly backtrack through the answers $\mathbf{X = a : b : nil}$ and $\mathbf{X = b : a : nil}$. However for the query, $\mathbf{Q2}$,

$$? - \texttt{permute(a : b : nil, X)}$$

Prolog will first return the answer $\mathbf{X = a : b : nil}$ and on subsequent backtracking will go into an infinite derivation without returning any more answers.

For languages with delay the program **permute** does behave symmetrically. For instance, if the above program is given to the NU-Prolog compiler, a preprocessor will generate the following *when declarations*:

$$
\begin{array}{l}
? - \texttt{permute(X,Y)whenXorY}. \\
? - \texttt{delete(X,Y : Z,U)whenZorU}.
\end{array}
$$

These may be read as saying that the call $\mathbf{permute(X, Y)}$ should delay until $\mathbf{X}$ or $\mathbf{Y}$ is not a variable, and that the call $\mathbf{delete(X, Y : Z, U)}$ should delay until $\mathbf{Z}$ or $\mathbf{U}$ is not a variable. Of course programmers can also annotate their programs with **when** declarations. Given these declarations, both of the above queries will behave in a symmetric fashion, backtracking through the possible permutations and then failing.

What happens is that with $\mathbf{Q1}$ execution proceeds as in standard Prolog because no atoms are delayed. With $\mathbf{Q2}$, however, calls to **delete** are delayed and only woken after the recursive calls to **permute**.

The dataflow analysis developed in this paper, can be used to analyze this program with these queries. In the case of $\mathbf{Q1}$ it will determine that the overhead of delaying is not needed as no call ever delays if the second argument is ground. Furthermore, it will also determine that in all calls to **permute** the first argument will be a variable and the second argument will be ground, and in all calls to **delete** the first and third arguments will be variables, and the second will be ground. This can be used to optimize the code for unification. In the case of $\mathbf{Q2}$ it will determine that all calls to **delete** from the second clause delay. Furthermore, that in all calls to **permute** the first argument will be ground and in all calls to **delete** when unification is performed, the first and third arguments will be ground, and the second will be a variable. The reader is encouraged to check that this is indeed true! Again this information can be used to optimize the code for unification, parallelism or other purposes. The benefits obtained from the optimizations made possible with such information are illustrated by the performance results presented in Section 9.

We note that if a traditional mode analysis is performed with the query $\mathbf{Q2}$ it will ignore delaying and incorrectly generate the information that the third argument of **delete** is free (which it would be in the non-terminating execution that the analyzer would be approximating) rather than ground.

3 Operational Semantics

In this section we give some preliminary notation and an operational semantics for logic programs with dynamic scheduling.

A *logic program*, or *program*, is a finite set of clauses. A *clause* is of the form $\mathbf{H} \leftarrow \mathbf{B}$ where $\mathbf{H}$, the *head*, is an atom and $\mathbf{B}$, the *body*, is a finite sequence of literals. A *literal* is either an atom or an equation between terms. An *atom* has the form $\mathbf{p(x_1, ..., x_n)}$ where $\mathbf{p}$ is a predicate symbol and the $\mathbf{x_i}$ are distinct variables.

An *equality constraint* is essentially a conjunction of equations between terms. For technical convenience equality constraints are treated modulo logical equivalence, and are assumed to be closed under existential quantification and conjunction. Thus equality constraints are ordered by logical implication, that is $\theta \leq \theta'$ iff $\theta \Rightarrow \theta'$. The least, inconsistent equality constraint is denoted by **false**. We let $\exists_{\mathbf{W}}\, \theta$ denote the equality constraint $\exists\, \mathbf{V}_1 \exists\, \mathbf{V}_2 \cdots \mathbf{V_n}\theta$ where variable set $\mathbf{W} = \{\mathbf{V}_1, \ldots, \mathbf{V_n}\}$. We let $\overline{\exists}_{\mathbf{W}}\theta$ be constraint θ restricted to the variables $\mathbf{W}$. That is $\overline{\exists}_{\mathbf{W}}\theta$ is $\exists_{\mathbf{vars}(\theta)\setminus\mathbf{W}}\, \theta$ where function **vars** takes a syntactic object and returns the set of (free) variables occurring in it. Note that although we concentrate on equality constraints, the analysis generalizes to handle other constraints, such as arithmetic or Boolean, in the more general context of constraint logic programs [22].

Var is the set of variables, **Atom** the set of atoms,

$\mathbf{Eqns}$ the set of equality constraints, $\mathbf{Lit}$ the set of literals, and $\mathbf{Prog}$ the set of programs.

A *renaming* is a bijective mapping from $\mathbf{Var}$ to $\mathbf{Var}$. We let $\mathbf{Ren}$ be the set of renamings, and naturally extend renamings to mappings between atoms, clauses, and constraints. Syntactic objects $\mathbf{s}$ and $\mathbf{s'}$ are said to be *variants* if there is a $\rho \in \mathbf{Ren}$ such that $\rho\ \mathbf{s} = \mathbf{s'}$. The *definition of an atom* $\mathbf{A}$ *in program* $\mathbf{P}$ *with respect to variables* $\mathbf{W}$, $\mathbf{defn_P\ A\ W}$, is the set of variants of clauses in $\mathbf{P}$ such that each variant has $\mathbf{A}$ as a head and has variables disjoint from $\mathbf{W} \setminus (\mathbf{vars\ A})$.

The operational semantics of a program is in terms of its "derivations" which are sequences of reductions between "states" where a state $\langle \mathbf{G}, \theta, \mathbf{D} \rangle$ consists of the current literal sequence or "goal" $\mathbf{G}$, the current equality constraints θ, and the current sequence of delayed atoms $\mathbf{D}$. Literals in the goals are processed left-to-right. If the literal is an equality constraint, and it is consistent with the current equality constraints, it is added to these. Delayed atoms woken by the addition are processed. If the literal is an atom, it is tested to see if it is delayed. If so it is placed in the delayed atom sequence, otherwise it is replaced by the body of a clause in its definition.

More formally,

$$\mathbf{State} \quad = \quad \mathbf{Lit}{*} \times \mathbf{Eqns} \times \mathbf{Atom}{*}.$$

A state $\langle \mathbf{L} : \mathbf{G}, \theta, \mathbf{D} \rangle$ can be *reduced* as follows:

1. If $\mathbf{L} \in \mathbf{Eqns}$ and $\theta \wedge \mathbf{L}$ is satisfiable, it is reduced to $\langle \mathbf{D'} :: \mathbf{G}, \theta \wedge \mathbf{L}, \mathbf{D} \setminus \mathbf{D'} \rangle$ where $\mathbf{D'} = (\mathbf{woken\ D}\ \theta \wedge \mathbf{L})$.

2. If $\mathbf{L} \in \mathbf{Atom}$ there are two cases. If $(\mathbf{delay\ L}\ \theta)$ holds, it is reduced to $\langle \mathbf{G}, \theta, \mathbf{L} : \mathbf{D} \rangle$. Otherwise it is reduced to $\langle \mathbf{B} :: \mathbf{G}, \theta, \mathbf{D} \rangle$ for some $(\mathbf{L} \leftarrow \mathbf{B}) \in (\mathbf{defn_P\ L}\ (\mathbf{vars\ S}))$.

Note that :: denotes concatenation of sequences. A *derivation* of state $\mathbf{S}$ for program $\mathbf{P}$ is a sequence of states $\mathbf{S}_0 \rightarrow \mathbf{S}_1 \rightarrow \ldots \rightarrow \mathbf{S_n}$ where $\mathbf{S}_0$ is $\mathbf{S}$ and there is a reduction from each $\mathbf{S_i}$ to $\mathbf{S_{i+1}}$.

The above definition makes use of two parametric functions which are dependent on the systems or language being modeled. These are, $\mathbf{delay\ A}\ \theta$, which holds iff a call to atom $\mathbf{A}$ delays with the equations θ, and $\mathbf{woken\ D}\ \theta$, which is the subsequence of atoms in the sequence of delayed calls $\mathbf{D}$ that are woken by equations θ. Note that the order of the calls returned by $\mathbf{woken}$ is system dependent.

We will assume that these functions satisfy the following four conditions. The first ensures that there is a congruence between the conditions for delaying a call and waking it:

$$(1) \quad \mathbf{A} \in (\mathbf{woken\ D}\ \theta) \Leftrightarrow \mathbf{A} \in \mathbf{D} \wedge \neg\ (\mathbf{delay\ A}\ \theta).$$

The remaining conditions ensure that $\mathbf{delay}$ behaves reasonably. It should not take variable names into account:

$$(2) \quad \text{Let } \rho \in \mathbf{Ren}.\ \mathbf{delay\ A}\ \theta \Leftrightarrow \mathbf{delay}\ (\rho\ \mathbf{A})\ (\rho\ \theta).$$

It should only be concerned with the effect of θ on the variables in $\mathbf{A}$:

$$(3) \quad \mathbf{delay\ A}\ \theta \Leftrightarrow \mathbf{delay\ A}\ \bar{\exists}_{(\mathbf{vars\ A})}\theta.$$

Finally, if an atom is not delayed, adding more constraints should never cause it to delay:

$$(4) \quad \text{If } \theta \leq \theta' \text{ and } \mathbf{delay\ A}\ \theta, \text{ then } \mathbf{delay\ A}\ \theta'.$$

Although these conditions can be relaxed, they simplify the analysis presentation and are met in existing systems and languages.

The declarative semantics of a program is in terms of its "qualified answers". Consider a derivation from state $\mathbf{S}$ and program $\mathbf{P}$ with last state $\langle \mathbf{G}, \theta, \mathbf{D} \rangle$ where $\mathbf{G} = \mathbf{nil}$. It is *successful* if $\mathbf{D} = \mathbf{nil}$ and it *flounders* otherwise. We say the tuple $\langle \theta, \mathbf{D} \rangle$ is a *qualified answer to* $\mathbf{S}$. It is understood as representing the logical implication

$$\bar{\exists}_{(\mathbf{vars\ S})}(\mathbf{D} \wedge \theta) \Rightarrow \mathbf{S}.$$

For this reason we regard qualified answers $\langle \theta, \mathbf{D} \rangle$ and $\langle \theta', \mathbf{D'} \rangle$ to $\mathbf{S}$ as equivalent if

$$\bar{\exists}_{(\mathbf{vars\ S})}(\mathbf{D} \wedge \theta) \Leftrightarrow \bar{\exists}_{(\mathbf{vars\ S})}(\mathbf{D'} \wedge \theta').$$

In particular qualified answers ψ and ψ' are regarded as the same if there is a renaming ρ such that $(\rho\ \psi) = \psi'$ and $(\rho\ \mathbf{S}) = \mathbf{S}$. As there is a non-deterministic choice of the clause in an atom's definition, there may be a number of qualified answers generated from the initial state. We denote the set of qualified answers for a state $\mathbf{S}$ and program $\mathbf{P}$ by $\mathbf{qans_P\ S}$. In the case of no calls delaying, this semantics is the same as the usual operational semantics of Prolog with left-to-right scheduling.

As an example, consider the initial state $\langle \mathbf{permute}(\mathbf{X}, \mathbf{Y}), \mathbf{X} = \mathbf{a} : \mathbf{nil}, \mathbf{nil} \rangle$ and the program from Section 2. These have the (single) successful derivation shown in Figure 1.

4 Abstract Interpretation

In abstract interpretation [7] an analysis is formalized as a non-standard interpretation of the data types and functions over those types. Correctness of the analysis with respect to the standard interpretation is argued by providing an "approximation relation" which holds whenever an element in a non-standard domain describes an element in the corresponding standard domain. We define the approximation relation in terms of an "abstraction function" which maps an element in

$$
\begin{array}{cl}
\langle \mathbf{permute}(\mathbf{X},\mathbf{Y}),\ \ \theta_1,\ \ \mathbf{nil}\rangle & \textbf{where } \theta_1 \textbf{ is } \mathbf{X} = \mathbf{a} : \mathbf{nil} \\
\Downarrow & \\
\langle (\mathbf{X} = \mathbf{U} : \mathbf{X}1) : \mathbf{delete}(\mathbf{U},\mathbf{Y},\mathbf{Z}) : \mathbf{permute}(\mathbf{X}1,\mathbf{Z}),\ \ \theta_1,\ \ \mathbf{nil}\rangle & \\
\Downarrow & \\
\langle \mathbf{delete}(\mathbf{U},\mathbf{Y},\mathbf{Z}) : \mathbf{permute}(\mathbf{X}1,\mathbf{Z}),\ \ \theta_2,\ \ \mathbf{nil}\rangle & \textbf{where } \theta_2 \textbf{ is } \theta_1 \wedge \mathbf{X} = \mathbf{U} : \mathbf{X}1 \\
\Downarrow & \\
\langle \mathbf{permute}(\mathbf{X}1,\mathbf{Z}),\ \ \theta_2,\ \ \mathbf{delete}(\mathbf{U},\mathbf{Y},\mathbf{Z})\rangle & \\
\Downarrow & \\
\langle \mathbf{X}1 = \mathbf{nil} : \mathbf{Z} = \mathbf{nil},\ \ \theta_2,\ \ \mathbf{delete}(\mathbf{U},\mathbf{Y},\mathbf{Z})\rangle & \\
\Downarrow & \\
\langle \mathbf{Z} = \mathbf{nil},\ \ \theta_3,\ \ \mathbf{delete}(\mathbf{U},\mathbf{Y},\mathbf{Z})\rangle & \textbf{where } \theta_3 \textbf{ is } \theta_2 \wedge \mathbf{X}1 = \mathbf{nil} \\
\Downarrow & \\
\langle \mathbf{delete}(\mathbf{U},\mathbf{Y},\mathbf{Z}),\ \ \theta_4,\ \ \mathbf{nil}\rangle & \textbf{where } \theta_4 \textbf{ is } \theta_3 \wedge \mathbf{Z} = \mathbf{nil} \\
\Downarrow & \\
\langle \mathbf{Y} = \mathbf{U} : \mathbf{Z},\ \ \theta_4,\ \ \mathbf{nil}\rangle & \\
\Downarrow & \\
\langle \mathbf{nil},\ \ \theta_5,\ \ \mathbf{nil}\rangle & \textbf{where } \theta_5 \textbf{ is } \theta_4 \wedge \mathbf{Y} = \mathbf{U} : \mathbf{Z}.
\end{array}
$$

Figure 1: Example Derivation

the standard domain to its "best" or most precise description.

A *description* $\langle \mathcal{D}, \alpha, \mathcal{E}\rangle$ consists of a *description domain* $\mathcal{D}$ which must be a complete lattice, a *data domain* $\mathcal{E}$, and an *abstraction function* $\alpha : \mathcal{E} \to \mathcal{D}$.
We say that $\mathbf{d}$ α-approximates $\mathbf{e}$, written $\mathbf{d} \propto_\alpha \mathbf{e}$, iff $(\alpha\ \mathbf{e}) \leq \mathbf{d}$. The approximation relation is lifted to functions, Cartesian-products and sets as follows.

- Let $\langle \mathcal{D}_1, \alpha_1, \mathcal{E}_1\rangle$ and $\langle \mathcal{D}_2, \alpha_2, \mathcal{E}_2\rangle$ be descriptions, and $\mathbf{F} : \mathcal{D}_1 \to \mathcal{D}_2$ and $\mathbf{F}' : \mathcal{E}_1 \to \mathcal{E}_2$ be functions. Then $\mathbf{F} \propto \mathbf{F}'$ iff $\forall \mathbf{d} \in \mathcal{D}_1.\ \forall \mathbf{e} \in \mathcal{E}_1.\ \mathbf{d} \propto_{\alpha_1} \mathbf{e} \Rightarrow (\mathbf{F}\ \mathbf{d}) \propto_{\alpha_2} (\mathbf{F}'\ \mathbf{e})$.
- Let $\langle \mathcal{D}_1, \alpha_1, \mathcal{E}_1\rangle$ and $\langle \mathcal{D}_2, \alpha_2, \mathcal{E}_2\rangle$ be descriptions, and $\langle \mathbf{d}_1, \mathbf{d}_2\rangle : \mathcal{D}_1 \times \mathcal{D}_2$ and $\langle \mathbf{e}_1, \mathbf{e}_2\rangle : \mathcal{E}_1 \times \mathcal{E}_2$. Then $\langle \mathbf{d}_1, \mathbf{d}_2\rangle \propto \langle \mathbf{e}_1, \mathbf{e}_2\rangle$ iff $\mathbf{d}_1 \propto_{\alpha_1} \mathbf{e}_1 \wedge \mathbf{d}_2 \propto_{\alpha_2} \mathbf{e}_2$.
- Let $\langle \mathcal{D}, \alpha, \mathcal{E}\rangle$ be a description and $\mathcal{D}' \subseteq \mathcal{D}$ and $\mathcal{E}' \subseteq \mathcal{E}$. Then $\mathcal{D}' \propto \mathcal{E}'$ iff $\forall \mathbf{e} \in \mathcal{E}'.\ \exists \mathbf{d} \in \mathcal{D}'.\ \mathbf{d} \propto_\alpha \mathbf{e}$.

When clear from the context we say that $\mathbf{d}$ approximates $\mathbf{e}$ and write $\mathbf{d} \propto \mathbf{e}$ and let $\mathcal{D}$ denote both the description and the description domain.

In the analysis we will need to describe sequences of delayed atoms and sequences of woken atoms. Because of the inherent imprecision in analyzing programs with a dynamic computation rule, we cannot always be definite that a particular atom is in the sequence, but may only know that it is *possibly* in the sequence. Further, it is difficult to keep track of all possible sequence orderings. Hence, we will describe atom sequences by a multiset of annotated atoms in which each atom is annotated with **def** if it definitely appears in the sequence and **pos** if it possibly appears in the sequence. For example, the multiset $\{\langle \mathbf{p}(\mathbf{X}), \mathbf{pos}\rangle, \langle \mathbf{q}(\mathbf{Y}), \mathbf{def}\rangle\}$ describes only the sequences $\mathbf{p}(\mathbf{X}) : \mathbf{q}(\mathbf{Y}) : \mathbf{nil}$, $\mathbf{q}(\mathbf{Y}) : \mathbf{p}(\mathbf{X}) : \mathbf{nil}$ and $\mathbf{q}(\mathbf{Y}) : \mathbf{nil}$. More formally,

$$
\begin{array}{lcl}
\mathbf{Ann} & = & \{\mathbf{def}, \mathbf{pos}\} \\
\mathbf{AnnAtom} & = & \mathbf{Atom} \times \mathbf{Ann} \\
\mathbf{AnnMSet} & = & \wp\, \mathbf{AnnAtom}
\end{array}
$$

Let $\mathbf{D}^* \in \mathbf{AnnMSet}$. Define $\mathbf{def}\ \mathbf{D}^*$ to be the multiset of atoms in $\mathbf{D}^*$ that are annotated with **def** and $\mathbf{all}\ \mathbf{D}^*$ to be the multiset of all atoms in $\mathbf{D}^*$, that is the atoms annotated with either **pos** or **def**. The *atom sequence* description is

$$\langle \mathbf{AnnMSet}, \alpha_{\mathbf{AnnMSet}}, \mathbf{Atom}{*}\rangle$$

where $\alpha_{\mathbf{AnnMSet}}$ is defined by

$$\alpha_{\mathbf{AnnMSet}}\ \mathbf{D} = \{\langle \mathbf{A}, \mathbf{def}\rangle \mid \mathbf{A} \text{ in } \mathrm{D}\}$$

and **AnnMSet** is ordered by $\mathbf{D}_1^* \preceq \mathbf{D}_2^*$ iff

$$\mathbf{all}\ \mathbf{D}_1^* \subseteq \mathbf{all}\ \mathbf{D}_2{}^* \wedge \mathbf{def}\ \mathbf{D}_2^* \subseteq \mathbf{def}\ \mathbf{D}_1^*.$$

It follows that the annotated multiset $\mathbf{D}^*$ approximates the sequence $\mathbf{D}$ iff all atoms in $\mathbf{D}$ are in $\mathbf{all}\ \mathbf{D}^*$ and every atom in $\mathbf{def}\ \mathbf{D}^*$ is in $\mathbf{D}$.

We will also be interested in describing equality constraints. The analysis given in the next section is generic in the choice of description. We require that the description chosen, $\langle \mathbf{AEqns}, \alpha_{\mathbf{AEqns}}, \mathbf{Eqns}\rangle$ say, satisfies

$$\alpha_{\mathbf{AEqns}}\ \theta = \perp_{\mathbf{AEqns}} \quad \Leftrightarrow \quad \theta = \mathbf{false}$$

where $\perp_{\mathbf{AEqns}}$ is the least element in **AEqns**.

One example of an equality constraint description is the *standard equality constraint description*,

$$\langle \mathbf{Eqns}^\top, \lambda\,\theta\,.\,\theta, \mathbf{Eqns}\rangle$$

in which constraints describe themselves. More precisely, the description domain is $\mathbf{Eqns}^\top$ which is the set of constraint equalities with a new top element $\top$. $\mathbf{Eqns}^\top$ is a complete lattice ordered by

$$\theta \leq \theta' \Leftrightarrow \theta = \mathbf{false} \text{ or } \theta = \theta' \text{ or } \theta' = \top.$$

The abstraction function is the identity function, as the best description of a constraint is just itself.

5 Denotational Semantics

In this section we give a generic denotational semantics for programs with dynamic scheduling. Correctness of the denotational semantics depends on the following results about the operational semantics. The first proposition means that we can find the qualified answers of a state in terms of its constituent atoms, the second means that we can consider states modulo variable renaming, the third that we can restrict the equality constraint to the variables in the delayed atoms and the goal.

Proposition 5.1 Let $\mathbf{P} \in \mathbf{Prog}$, $\mathbf{A} \in \mathbf{Atom}$ and $\langle \mathbf{A} : \mathbf{G}, \theta, \mathbf{D} \rangle \in \mathbf{State}$. Then $\mathbf{qans_P}\ \langle \mathbf{A} : \mathbf{G}, \theta, \mathbf{D} \rangle$ is

$$\bigcup \{ \mathbf{qans_P} \langle \mathbf{G}, \theta', \mathbf{D}' \rangle \mid \langle \theta', \mathbf{D}' \rangle \in \mathbf{qans_P} \langle \mathbf{A}, \theta, \mathbf{D} \rangle \}. \quad \blacksquare$$

Proposition 5.2 Let $\mathbf{P} \in \mathbf{Prog}$, $\rho \in \mathbf{Ren}$, and $\mathbf{S} \in \mathbf{State}$.

$$\mathbf{Q} \in \mathbf{qans_P}\ \mathbf{S} \Leftrightarrow (\rho\ \mathbf{Q}) \in \mathbf{qans_P}\ (\rho\ \mathbf{S}). \quad \blacksquare$$

Proposition 5.3 Let $\mathbf{P} \in \mathbf{Prog}$ and $\langle \mathbf{G}, \theta, \mathbf{D} \rangle \in \mathbf{State}$.

$$\mathbf{qans_P}\ \langle \mathbf{G}, \theta, \mathbf{D} \rangle = \bigcup \{ \langle \theta' \wedge \theta, \mathbf{D}' \rangle | \langle \theta', \mathbf{D}' \rangle \in \mathbf{qans_P}\ \mathbf{S} \}.$$

where $\mathbf{S}$ is $\langle \mathbf{G}, \bar{\exists}_{(\mathbf{vars\ G}) \cup (\mathbf{vars\ D})} \theta, \mathbf{D} \rangle$. $\blacksquare$

Taken together these propositions mean that we can find the qualified answers to a state as long as we know the qualified answers to the "canonical" calls encountered when processing the state where a canonical call is a call that represents all of its variants and in which the constraint is restricted to the variables of the call atom and the delayed atoms. This is the basic idea behind the denotational semantics as the denotation of a program is simply a mapping from calls to answers.

The last proposition means that the meaning of a goal is independent of the order that the atoms are scheduled. Thus we can ignore the sequencing information associated with delayed atoms and treat them as multisets. It is variant of Theorem 4 in Yelick and Zachary [34].

Proposition 5.4 Let $\mathbf{P}$ be a program and $\langle \mathbf{G}, \theta, \mathbf{D} \rangle$ be a state. If $\mathbf{G}'$ is a rearrangement of $\mathbf{G}$ then,

$$\mathbf{qans_P}\ \langle \mathbf{G}, \theta, \mathbf{D} \rangle = \mathbf{qans_P}\ \langle \mathbf{G}', \theta, \mathbf{D} \rangle. \quad \blacksquare$$

In the denotational semantics atoms, bodies, clauses and programs are formalized as "environment" transformers where an environment consists of the current equality constraint description and an annotated multiset of delayed atoms. In a sense an environment is the current "answer". Thus an environment has type

$$\mathbf{Env} \quad = \quad \mathbf{AEqns} \times \mathbf{AnnMSet}$$

and the denotation of a program has type

$$\mathbf{Den} \quad = \quad \mathbf{Atom} \rightarrow \mathbf{Env} \rightarrow \wp\, \mathbf{Env}$$

as it maps a call to its set of answers.

The complete denotational definition is shown in Figure 2. The semantics is generic as it is parametric in **AEqns** the equality constraint descriptions and various parametric functions. The semantic functions associated with programs **P**, clause bodies **B**, and literals **L**, need little explanation. The only point to note is that the variable set **W** is passed around so as to ensure that there are no variable (re)naming conflicts.

The function **A** gives the meaning of an atom for the current denotation. Consider the call $\mathbf{A}\ [\![\mathbf{A}]\!]\ \mathbf{W}\ \mathbf{d}\ \langle \pi, \mathbf{D}^* \rangle$. There are three cases to consider: the first is when **A** is delayed for all equality constraints approximated by π, the second is when **A** is not delayed for any equality constraints approximated by π, and the third is when **A** is delayed for some equality constraints approximated by π, but not all. **A** is defined in terms of the parametric functions **Awake** and **Adelay**. The call $\mathbf{Awake\ A}\ \pi$ returns a description of those equality constraints which are described by π and for which **A** will not delay. Conversely, $\mathbf{Adelay\ A}\ \pi$, returns a description of those equality constraints which are described by π and for which **A** will delay. More exactly, **Awake** and **Adelay** should satisfy:

$$\begin{array}{l} \{(\mathbf{Awake\ A}\ \pi)\} \propto \{\theta \mid \pi \propto \theta \wedge \neg\, (\mathbf{delay\ A}\ \theta)\} \\ \{(\mathbf{Adelay\ A}\ \pi)\} \propto \{\theta \mid \pi \propto \theta \wedge (\mathbf{delay\ A}\ \theta)\}. \end{array}$$

Note that $\mathbf{Awake\ A}\ \pi = \bot_{\mathbf{AEqns}}$ implies

$$\pi \propto \theta \Rightarrow (\mathbf{delay\ A}\ \theta),$$

and $\mathbf{Adelay\ A}\ \pi = \bot_{\mathbf{AEqns}}$ implies

$$\pi \propto \theta \Rightarrow \neg\, (\mathbf{delay\ A}\ \theta).$$

The auxiliary function **lookup** is used to find the denotation of an atom which possibly does not delay. The call, $\mathbf{lookup\ A\ W\ d}\ \langle \pi, \mathbf{D}^* \rangle$, returns the denotation according to **d** of **A** with environment $\langle \pi, \mathbf{D}^* \rangle$. However there are complications because **d** only handles "canonical calls". Hence **lookup** must (1) restrict π to the variables in the call; (2) rename the variables introduced in the delayed atoms in the answers so that they do not interfere with the variables in **W**; and (3), combine the equality constraint description with that of the original call so as to undo the result of the restriction. **Lookup** is defined in terms of the parametric functions **Acomb** and **Arestrict**. **Acomb** combines two equality constraint descriptions and should approximate the function **add**, defined by

$$\mathbf{add}\ \theta\ \theta' = \theta \wedge \theta'.$$

The denotational semantics has semantic functions:

$$
\begin{array}{lll}
\mathbf{P} & : & \mathbf{Prog} \to \mathbf{Den} \\
\mathbf{Q} & : & \mathbf{Prog} \to \mathbf{Den} \to \mathbf{Den} \\
\mathbf{B} & : & \mathbf{Lit}* \to (\wp\,\mathbf{Var}) \to \mathbf{Den} \to \mathbf{Env} \to (\wp\,\mathbf{Env}) \\
\mathbf{L} & : & \mathbf{Lit} \to (\wp\,\mathbf{Var}) \to \mathbf{Den} \to \mathbf{Env} \to (\wp\,\mathbf{Env}) \\
\mathbf{A} & : & \mathbf{Atom} \to (\wp\,\mathbf{Var}) \to \mathbf{Den} \to \mathbf{Env} \to (\wp\,\mathbf{Env}) \\
\mathbf{E} & : & \mathbf{Eqns} \to (\wp\,\mathbf{Var}) \to \mathbf{Den} \to \mathbf{Env} \to (\wp\,\mathbf{Env}).
\end{array}
$$

It has auxiliary functions:

$$
\begin{array}{lll}
\mathbf{lookup} & : & \mathbf{Atom} \to (\wp\,\mathbf{Var}) \to \mathbf{Den} \to \mathbf{Env} \to (\wp\,\mathbf{Env}) \\
\mathbf{wmset} & : & \mathbf{AnnMSet} \to (\wp\,\mathbf{Var}) \to \mathbf{Den} \to \mathbf{Env} \to (\wp\,\mathbf{Env}) \\
\mathbf{watom} & : & \mathbf{AnnAtom} \to (\wp\,\mathbf{Var}) \to \mathbf{Den} \to \mathbf{Env} \to (\wp\,\mathbf{Env})
\end{array}
$$

and parametric functions:

$$
\begin{array}{lll}
\mathbf{Awake} & : & \mathbf{Atom} \to \mathbf{AEqns} \to \mathbf{AEqns} \\
\mathbf{Adelay} & : & \mathbf{Atom} \to \mathbf{AEqns} \to \mathbf{AEqns} \\
\mathbf{Acomb} & : & \mathbf{AEqns} \to \mathbf{AEqns} \to \mathbf{AEqns} \\
\mathbf{Arestrict} & : & \wp\,\mathbf{Vars} \to \mathbf{AEqns} \to \mathbf{AEqns} \\
\mathbf{Aadd} & : & \mathbf{Eqns} \to \mathbf{AEqns} \to \mathbf{AEqns} \\
\mathbf{Awoken} & : & \mathbf{AnnMSet} \to \mathbf{Eqns} \to \mathbf{AEqns} \to \mathbf{AnnMSet} \\
\mathbf{Adelayed} & : & \mathbf{AnnMSet} \to \mathbf{Eqns} \to \mathbf{AEqns} \to \mathbf{AnnMSet}.
\end{array}
$$

The semantic and auxiliary functions are defined by:

$$
\begin{array}{lll}
\mathbf{P}\,[\![\mathbf{P}]\!]\;\mathbf{A}\;\mathbf{e} & = & \mathbf{A}\,[\![\mathbf{A}]\!]\;(\mathbf{vars}\;\mathbf{e})\;(\mathbf{lfp}\;(\mathbf{Q}\,[\![\mathbf{P}]\!]))\;\mathbf{e} \\
\mathbf{Q}\,[\![\mathbf{P}]\!]\;\mathbf{d}\;\mathbf{A}\;\mathbf{e} & = & \mathbf{let}\;\mathbf{V} = \mathbf{vars}\;\mathbf{e}\;\mathbf{in} \\
 & & \bigcup\{(\mathbf{B}\,[\![\mathbf{B}]\!]\;\mathbf{V} \cup (\mathbf{vars}\;\mathbf{A} \leftarrow \mathbf{B})\;\mathbf{d}\;\mathbf{e}) \mid (\mathbf{A} \leftarrow \mathbf{B}) \in \mathbf{defn_P}\;\mathbf{A}\;\mathbf{V}\} \\
\mathbf{B}\,[\![\mathbf{nil}]\!]\;\mathbf{W}\;\mathbf{d}\;\mathbf{e} & = & \{\mathbf{e}\} \\
\mathbf{B}\,[\![\mathbf{L} : \mathbf{B}]\!]\;\mathbf{W}\;\mathbf{d}\;\mathbf{e} & = & \bigcup\{(\mathbf{B}\,[\![\mathbf{B}]\!]\;\mathbf{W}\;\mathbf{d}\;\mathbf{e}') \mid \mathbf{e}' \in (\mathbf{L}\,[\![\mathbf{L}]\!]\;\mathbf{W}\;\mathbf{d}\;\mathbf{e})\} \\
\mathbf{L}\,[\![\mathbf{L}]\!] & = & \mathbf{if}\;\mathbf{L} \in \mathbf{Atom}\;\mathbf{then}\;(\mathbf{A}\,[\![\mathbf{L}]\!])\;\mathbf{else}\;(\mathbf{E}\,[\![\mathbf{L}]\!]) \\
\mathbf{A}\,[\![\mathbf{A}]\!]\;\mathbf{W}\;\mathbf{d}\;\langle\pi, \mathbf{D}^*\rangle & = & \mathbf{if}\;(\mathbf{Awake}\;\mathbf{A}\;\pi) = \bot_{\mathbf{AEqns}}\;\mathbf{then}\;\{\langle\pi, \langle\mathbf{A}, \mathbf{def}\rangle \cup \mathbf{D}^*\rangle\} \\
 & & \mathbf{else\;if}\;(\mathbf{Adelay}\;\mathbf{A}, \pi) = \bot_{\mathbf{AEqns}}\;\mathbf{then}\;(\mathbf{lookup}\;\mathbf{A}\;\mathbf{W}\;\mathbf{d}\;\langle\pi, \mathbf{D}^*\rangle) \\
 & & \mathbf{else}\;\{\langle\mathbf{Adelay}(\mathbf{A}, \pi), \langle\mathbf{A}, \mathbf{pos}\rangle \cup \mathbf{D}^*\rangle\} \cup \\
 & & \quad (\mathbf{lookup}\;\mathbf{A}\;\mathbf{W}\;\mathbf{d}\;\langle(\mathbf{Awake}\;\mathbf{A}\;\pi), \mathbf{D}^*\rangle) \\
\mathbf{E}\,[\![\theta]\!]\;\mathbf{W}\;\mathbf{d}\;\langle\pi, \mathbf{D}^*\rangle & = & \mathbf{let}\;\pi' = \mathbf{Aadd}\;\theta\;\pi\;\mathbf{in} \\
 & & \mathbf{if}\;\pi' = \bot_{\mathbf{AEqns}}\;\mathbf{then}\;\emptyset \\
 & & \mathbf{else}\;(\mathbf{wmset}\;(\mathbf{Awoken}\;\mathbf{D}^*\;\theta\;\pi)\;\mathbf{W}\;\mathbf{d}\;\langle\pi', (\mathbf{Adelayed}\;\mathbf{D}^*\;\theta\;\pi)\rangle) \\
\mathbf{lookup}\;\mathbf{A}\;\mathbf{W}\;\mathbf{d}\;\langle\pi, \mathbf{D}^*\rangle & = & \mathbf{let}\;\mathbf{V} = (\mathbf{vars}\;\mathbf{A}) \cup (\mathbf{vars}\;\mathbf{D}^*)\;\mathbf{in} \\
 & & \mathbf{let}\;\mathbf{E} = \mathbf{d}\;\mathbf{A}\;\langle(\mathbf{Arestrict}\;\mathbf{V}\;\pi), \mathbf{D}^*\rangle\;\mathbf{in} \\
 & & \{\langle(\mathbf{Acomb}\;\pi\;\pi'), \mathbf{D}^{*\prime}\rangle \mid \langle\pi', \mathbf{D}^{*\prime}\rangle \in (\mathbf{rename}\;\mathbf{E}\;\mathbf{V}\;\mathbf{W})\} \\
\mathbf{wmset}\;\emptyset\;\mathbf{W}\;\mathbf{d}\;\mathbf{e} & = & \{\mathbf{e}\} \\
\mathbf{wmset}\;(\mathbf{A}^* \cup \mathbf{D}^*)\;\mathbf{W}\;\mathbf{d}\;\mathbf{e} & = & \bigcup\{(\mathbf{wmset}\;\mathbf{D}^*\;\mathbf{W}\;\mathbf{d}\;\mathbf{e}') \mid \mathbf{e}' \in (\mathbf{watom}\;\mathbf{A}^*\;\mathbf{W}\;\mathbf{d}\;\mathbf{e})\} \\
\mathbf{watom}\;\langle\mathbf{A}, \mathbf{def}\rangle\;\mathbf{W}\;\mathbf{d}\;\langle\pi, \mathbf{D}^*\rangle & = & \mathbf{lookup}\;\mathbf{A}\;\mathbf{W}\;\mathbf{d}\;\langle\pi, \mathbf{D}^*\rangle \\
\mathbf{watom}\;\langle\mathbf{A}, \mathbf{pos}\rangle\;\mathbf{W}\;\mathbf{d}\;\langle\pi, \mathbf{D}^*\rangle & = & \{\langle\pi, \mathbf{D}^*\rangle\} \cup (\mathbf{lookup}\;\mathbf{A}\;\mathbf{W}\;\mathbf{d}\;\langle(\mathbf{Awake}\;\mathbf{A}\;\pi), \mathbf{D}^*\rangle)
\end{array}
$$

Figure 2: Denotational Semantics

Arestrict restricts an equality constraint description to a set of variables and should approximate the function **restrict** defined by

$$\mathbf{restrict}\ \mathbf{W}\ \theta = \bar{\exists}_{\mathbf{W}}\theta.$$

The definition also makes use of the function call **rename E V W** which returns a variant of the environments **E** which is disjoint from the variables **W** but which leaves the variables in **V** unchanged. More exactly it returns $\rho\ \mathbf{E}$ where ρ is a renaming such that for all $\mathbf{v} \in \mathbf{V}$, $\rho\ \mathbf{v} = \mathbf{v}$ and $\mathbf{vars}\ (\rho\ \mathbf{E}) \cap (\mathbf{W} \setminus \mathbf{V}) = \emptyset$.

Equations are handled by the semantic function **E**. The function call, $\mathbf{E}\ [\![\theta]\!]\ \mathbf{W}\ \mathbf{d}\ \langle \pi, \mathbf{D}^* \rangle$, first adds the equality constraint θ to π and tests for satisfiability. If this succeeds, it then wakes up the atoms in $\mathbf{D}^*$, and processes these. The definition is parametric in the functions **Aadd**, **Awoken** and **Adelayed**. The function **Aadd** adds an equality constraint to an equality constraint description and must approximate the function **add** defined previously. **Awoken** returns the multiset of atoms that will be possibly and definitely woken by adding an equality constraint to an equality constraint description and **Adelayed** returns the multiset of atoms that will possibly and definitely remain delayed. **Awoken** must approximate **diffwoken** and **Adelayed** must approximate **diffdelay** where these are defined by

$$\begin{array}{l} \mathbf{diffwoken}\ \mathbf{D}\ \theta\ \theta' = \mathbf{woken}\ (\mathbf{D} \setminus (\mathbf{woken}\ \mathbf{D}\ \theta'))\ \theta \\ \mathbf{diffdelay}\ \mathbf{D}\ \theta\ \theta' = \mathbf{D} \setminus (\mathbf{woken}\ \mathbf{D}\ (\theta \wedge \theta')). \end{array}$$

Note that **Adelayed** may change the annotation of a delayed atom from **def** to **pos** and that **Awoken** returns a multiset of woken atoms which are also annotated.

The woken atoms are handled by the auxiliary functions **wmset** and **watom** almost exactly as if they were a clause body, the only difference is to handle the **pos** annotated atoms.

The *standard* denotational semantics, $\mathbf{P}_{\mathbf{std}}$, is obtained by from the denotational semantics by instantiating **AEqns** to the standard equality constraint descriptions and instantiating the parametric functions to the function they are required to approximate, for instance **Aadd** and **Acomb** are both instantiated to **add**. Using the four propositions given at the start of this section, it is possible to show that the denotational semantics is correct:

Theorem 5.5 Let $\mathbf{D} \in \mathbf{Atom}*$, $\theta \in \mathbf{Eqns}$, $\mathbf{A} \in \mathbf{Atom}$, and $\mathbf{P} \in \mathbf{Prog}$. Then

$$\mathbf{P}_{\mathbf{std}}\ [\![\mathbf{P}]\!]\ \mathbf{A}\ \langle \theta, \mathbf{D}^* \rangle = \mathbf{qans}_{\mathbf{P}}\ \langle \mathbf{A} : \mathbf{nil}, \theta, \mathbf{D} \rangle$$

where $\mathbf{D}^* = \alpha_{\mathbf{AnnMSet}}\ \mathbf{D}$. ■

Using results from abstract interpretation theory it follows that analyses based on the semantics are correct:

Theorem 5.6 Let $\mathbf{e} \in \mathbf{Env}$, $\mathbf{A} \in \mathbf{Atom}$, $\mathbf{P} \in \mathbf{Prog}$. If $\mathbf{e} \propto \langle \theta, \mathbf{D} \rangle$,

$$(\mathbf{P}\ [\![\mathbf{P}]\!]\ \mathbf{A}\ \mathbf{e}) \propto \mathbf{qans}_{\mathbf{P}}\ \langle \mathbf{A} : \mathbf{nil}, \theta, \mathbf{D} \rangle. \quad ■$$

Actually the denotational semantics does not exactly give the information a compiler requires for the generation of efficient code. This is because we are primarily interested in removing unnecessary tests for delaying and improving the code for unification. Therefore, we must obtain information about the *call patterns*. That is, for each atom **A** appearing in the program we want to know whether the calls to the atom initially delay, and when each call to **A** is eventually reduced, perhaps after being delayed, the value of the current equation restricted to the variables in **A**. It is straightforward to modify the denotational semantics to collect this information for atoms which are not delayed. For the case of atoms which are delayed it is more difficult as although treating the delayed atoms as a multiset does not affect the qualified answers, if more than one atom is woken it may affect the calls made in the evaluation. Because of space limitations we will ignore this extra complication but note that it has been done in the analyzer used to obtain the results presented in Section 9.

6 Implementation

The denotational equations given in the previous section can be considered as a *definition* of a class of program analyses. Read naively, the equations specify a highly redundant way of computing certain mathematical objects. On the other hand, the denotational definitions can be given a "call-by-need" reading which guarantees that the same partial result is not repeatedly recomputed and only computed if it is needed for the final result. With such a call-by-need reading the definition of **P** is, modulo syntactic rewriting, a working implementation of a generic dataflow analyzer written in a functional programming language.

In programming languages which do not support a call-by-need semantics, implementation is somewhat harder. To avoid redundant computations, the result of invoking atom **A** in the context of environment **e** should be recorded. Such memoing can be implemented using function graphs. The function graph for a function **f** is the set of pairs $\{(\mathbf{e} \mapsto \mathbf{f}(\mathbf{e})) \mid \mathbf{e} \in \mathbf{dom}\ \mathbf{f}\}$ where **dom f** denotes the domain for **f**. Computation of a function graph is done in a demand-driven fashion so that we only compute as much of it as is necessary in order to answer a given query. This corresponds to the "minimal function graph" semantics used by Jones and Mycroft [23]. However, matters are complicated by the fact that we are performing a fixpoint computation and we must iteratively compute the result by means of the function's Kleene sequence.

This idea leads to a generic algorithm for the memoization based analysis of programs with dynamic scheduling. The algorithm extends memoization based analysis for traditional Prolog. The analysis starts from a "call" and incrementally builds a *memoization table*. This contains tuples of "calls" and their "answers" which are encountered in derivations from the initial call. Calls are tuples of the form $\langle \mathbf{A}, \pi, \mathbf{D}^* \rangle$ where $\mathbf{A}$ is an atom, $\mathbf{D}^*$ is a multiset of annotated atoms describing the sequence of delayed atoms and π is an equality constraint description restricted to the variables in $\mathbf{A}$ and $\mathbf{D}^*$. An answer to a call $\langle \mathbf{A}, \pi, \mathbf{D}^* \rangle$ is of the form $\langle \pi', \mathbf{D}^{*\prime} \rangle$ where $\mathbf{D}^{*\prime}$ is a multiset of annotated atoms describing the sequence of delayed atoms and π' is an equality constraint description restricted to the variables in $\mathbf{A}$ and $\mathbf{D}^{*\prime}$. Our actual implementation has two improvements which reduce the size of the memoization table.

The first improvement, is when adding an answer to the answers of call, to remove "redundant" answers and merge similar answers together. Answers $\langle \pi_1, \mathbf{D}_1^* \rangle$ and $\langle \pi_2, \mathbf{D}_2^* \rangle$ are merged into the single answer $\langle \pi_1 \sqcup \pi_2, \mathbf{D}_1^* \rangle$ whenever $\mathbf{D}_2^* \preceq \mathbf{D}_1^*$.

The second improvement is to only consider calls modulo variable renaming. Entries in the memoization table are "canonical" and really represent equivalence classes of calls and answers.

Another possible improvement which has not been implemented yet is based on the observation that delayed atoms which are "independent" of the calling atom can never be woken when the call is executed. Such atoms need not be considered in the call as they will occur in each answer. The exact definition of independence is somewhat difficult as it really means independence from any delayed atom which could be woken in the call.

7 Termination

Correctness of the denotational semantics, Theorem 5.6, is not quite enough as it only guarantees partial correctness of an analysis, and, of course, we would also like the analysis to terminate. Given that all calls to the parametric functions terminate, the analysis will terminate iff there are a finite number of calls in the memoization table and each call has a finite number of answers. This is true if the following two requirements are met. The first is that for each finite set of variables $\mathbf{W}$ there are only a finite number of descriptions which describe some equality constraints $\bar{\exists}_{\mathbf{W}}\theta$. This is the usual requirement for the termination of memoization based analysis of standard Prolog. The second requirement is that there is a bound on the size of the annotated multisets in both the calls and the answers. In this section we sketch two modifications to the analysis which ensure that only multisets of a bounded size need be considered, albeit at some loss of accuracy. In some sense, this is a form of widening [8], however correctness does not depend on the semantics of the description domain but rather on properties of the program semantics.

The first modification allows us to only consider calls with annotated multisets of a bounded size. Correctness depends on the following property of the operational semantics:

Proposition 7.1 Let $\mathbf{P} \in \mathbf{Prog}$ and $\langle \mathbf{G}, \theta, \mathbf{D} \rangle \in \mathbf{State}$. If $\mathbf{D} = \mathbf{D}' \cup \mathbf{D}''$,

$$\mathbf{qans_P}\ \langle \mathbf{G}, \theta, \mathbf{D} \rangle = \mathbf{qans_P}\ \langle \mathbf{G} :: \mathbf{D}', \theta, \mathbf{D}'' \rangle. \quad \blacksquare$$

This means in the analysis that **lookup** can be modified to (1) remove annotated atoms $\mathbf{D}^*$ from the multiset of delayed atoms, if it is to large, (2) proceed as before, and then (3) process $\mathbf{D}^*$ using a variant of $\mathbf{B}$ which handles annotated atoms.

The second modification allows us to only consider answers with annotated multisets of a bounded size. Now a delayed atom $\mathbf{A}$ can, if it is woken, only add constraints affecting variables in $\mathbf{A}$ and variables which are local to its subcomputation. Thus in the analysis, when we encounter an answer $\langle \pi, \mathbf{D}^* \rangle$ in which the multiset $\mathbf{D}^*$ is too large, we can replace it by the answer $\langle \pi', \{\langle \top, \mathbf{pos} \rangle\} \rangle$ where $\{\pi'\}$ approximates

$$\{\theta \wedge \bar{\exists}_{(\mathbf{vars}\ \mathbf{D}^*)}\theta' \mid \pi \propto \theta \wedge \theta' \in \mathbf{Eqns}\}$$

and $\langle \top, \mathbf{pos} \rangle$ is a special annotated "atom" which signifies that there are possibly delayed atoms of indeterminate name. Note that $\langle \top, \mathbf{pos} \rangle$ can never be woken.

With these two modifications the analysis will terminate whenever the usual termination requirements for memoization based analysis of standard Prolog are met.

We can also use the idea behind the second modification to analyse modules. The problem is that when analyzing a module in isolation from the context in which it will be used we have no idea of the delayed atoms associated with calls to the module. However, the delayed atoms can only affect variables in the initial call. Thus by taking the downward closure of the initial call, we are assured to obtain correct information about the calling patterns regardless of the atoms delayed in the actual call.

Another approach to ensure termination would be to approximate the delayed multiset of atoms by a "star abstraction" [4] in which variants of the same atom are collapsed on to a single "canonical" atom.

8 Example Analysis

We now present an example of the analysis algorithm's use. In our example analysis we use "simple modes" to describe the equality constraints. We will use this

Table after 1st Iteration:

$$\langle \mathbf{permute(X,Y)}, \langle\{\mathbf{Y}\},\{\mathbf{X}\},\emptyset\rangle,\emptyset\rangle \quad \mapsto \quad \emptyset$$

Table after 2nd Iteration:

$$\begin{array}{lcl} \langle \mathbf{permute(X,Y)}, \langle\{\mathbf{Y}\},\{\mathbf{X}\},\emptyset\rangle,\emptyset\rangle & \mapsto & \{\langle\langle\{\mathbf{X},\mathbf{Y}\},\emptyset,\emptyset\rangle,\emptyset\rangle\} \\ \langle \mathbf{delete(U,Y,Z)}, \langle\{\mathbf{Y}\},\{\mathbf{U},\mathbf{Z}\},\emptyset\rangle,\emptyset\rangle & \mapsto & \emptyset \end{array}$$

Table after 3rd and Final Iteration:

$$\begin{array}{lcl} \langle \mathbf{permute(X,Y)}, \langle\{\mathbf{Y}\},\{\mathbf{X}\},\emptyset\rangle,\emptyset\rangle & \mapsto & \{\langle\langle\{\mathbf{X},\mathbf{Y}\},\emptyset,\emptyset\rangle,\emptyset\rangle\} \\ \langle \mathbf{delete(U,Y,Z)}, \langle\{\mathbf{Y}\},\{\mathbf{U},\mathbf{Z}\},\emptyset\rangle,\emptyset\rangle & \mapsto & \{\langle\langle\{\mathbf{U},\mathbf{Y},\mathbf{Z}\},\emptyset,\emptyset\rangle,\emptyset\rangle\} \end{array}$$

Analysis of $\langle \mathbf{permute(X,Y)}, \langle\{\mathbf{Y}\},\{\mathbf{X}\},\emptyset\rangle, \mathbf{nil}\rangle$.

Table after 1st Iteration:

$$\langle \mathbf{permute(X,Y)}, \langle\{\mathbf{X}\},\{\mathbf{Y}\},\emptyset\rangle,\emptyset\rangle \quad \mapsto \quad \emptyset$$

Table after 2nd Iteration:

$$\begin{array}{lcl} \langle \mathbf{permute(X,Y)}, \langle\{\mathbf{X}\},\{\mathbf{Y}\},\emptyset\rangle,\emptyset\rangle & \mapsto & \{\langle\langle\{\mathbf{X},\mathbf{Y}\},\emptyset,\emptyset\rangle,\emptyset\rangle\} \\ \langle \mathbf{delete(U,Y,Z)}, \langle\{\mathbf{U},\mathbf{Z}\},\{\mathbf{Y}\},\emptyset\rangle,\emptyset\rangle & \mapsto & \emptyset \end{array}$$

Table after 3rd and Final Iteration:

$$\begin{array}{lcl} \langle \mathbf{permute(X,Y)}, \langle\{\mathbf{X}\},\{\mathbf{Y}\},\emptyset\rangle,\emptyset\rangle & \mapsto & \{\langle\langle\{\mathbf{X},\mathbf{Y}\},\emptyset,\emptyset\rangle,\emptyset\rangle\} \\ \langle \mathbf{delete(U,Y,Z)}, \langle\{\mathbf{U},\mathbf{Z}\},\{\mathbf{Y}\},\emptyset\rangle,\emptyset\rangle & \mapsto & \{\langle\langle\{\mathbf{U},\mathbf{Y},\mathbf{Z}\},\emptyset,\emptyset\rangle,\emptyset\rangle\} \end{array}$$

Analysis of $\langle \mathbf{permute(X,Y)}, \langle\{\mathbf{X}\},\{\mathbf{Y}\},\emptyset\rangle, \mathbf{nil}\rangle$.

Figure 3: Example Analyses

mode descriptions to analyze the state corresponding to query $\mathbf{Q}1$ from Section 2. The domain used is similar to that of [28] and has been used for analyzing traditional Prolog. A mode description for equality constraint θ has the form

$$\langle \mathbf{V_{gnd}}, \mathbf{V_{free}}, \mathbf{W_{dep}} \rangle$$

where $\mathbf{V_{gnd}}$ is the set of variables that θ definitely grounds, $\mathbf{V_{free}}$ is the set of variables that θ leaves definitely "free", that is not instantiated to a non-variable term, and $\mathbf{W_{dep}}$ the set of sets of variables which θ makes possibly dependent. For example, the equality constraint

$$\mathbf{X} = \mathbf{a} \wedge \mathbf{Y} = \mathbf{Z} \wedge \mathbf{W} = \mathbf{f(V)}$$

is (most precisely) described by

$$\langle\{\mathbf{X}\},\{\mathbf{Y},\mathbf{Z}\},\{\{\mathbf{Y},\mathbf{Z}\},\{\mathbf{W},\mathbf{V}\}\}\rangle.$$

A more complete description of this description domain and abstract operations over it can be found in [28]. Of course for accuracy more complex domains could be used in the analysis.

The first state description to be analyzed is

$$\langle \mathbf{permute(X,Y)}, \langle\{\mathbf{Y}\},\{\mathbf{X}\},\emptyset\rangle, \mathbf{nil}\rangle.$$

Figure 3 shows the memoization table at each iteration in the analysis for this state description. The result of the analysis is $\{\langle\langle\{\mathbf{X},\mathbf{Y}\},\emptyset,\emptyset\rangle,\emptyset\rangle\}$. That is, if calls to **permute** have their second argument ground and first argument free, the answers will ground the first argument. As no calls were delayed in this example, the analysis was virtually the same as given by a traditional left-to-right mode analysis of the program. If the analysis is extended to give information about call patterns it shows, as promised in Section 2, that for calls to **permute** in which the second argument is ground, and the first free, no atom ever delays. Further, it shows that in all calls to **permute** the first argument will be free and the second argument will be ground, and in all calls to **delete** the first and third arguments will be free, and the second will be ground.

Now consider the state description

$$\langle \mathbf{permute(X,Y)}, \langle\{\mathbf{X}\},\{\mathbf{Y}\},\emptyset\rangle, \mathbf{nil}\rangle.$$

Figure 3 shows the memoization table at each iteration in the analysis for this state description. The result of the analysis is $\{\langle\langle\{\mathbf{X},\mathbf{Y}\},\emptyset,\emptyset\rangle,\emptyset\rangle\}$. That is, if calls to **permute** have their first argument ground and second argument free, the answers will ground the second argument. Termination is achieved by restricting calls in the memoization table so that they have an empty annotated multiset. Thus when the call

$$\langle \mathbf{permute}(\mathbf{X}1, \mathbf{Z}), \langle \{\mathbf{X}1, \mathbf{U}\}, \{\mathbf{Y}, \mathbf{Z}\}, \emptyset\rangle, \{\langle \mathbf{delete}(\mathbf{U}, \mathbf{Y}, \mathbf{Z}), \mathbf{def}\rangle\}\rangle$$

is encountered when processing the second clause of **permute**, first the call

$$\langle \mathbf{permute}(\mathbf{X}1, \mathbf{Z}), \langle \{\mathbf{X}1\}, \{\mathbf{Z}\}, \emptyset\rangle, \emptyset\rangle,$$

is looked up in the table and then, as this grounds **Z**, the call

$$\langle \mathbf{delete}(\mathbf{U}, \mathbf{Y}, \mathbf{Z}), \langle \{\mathbf{U}, \mathbf{Z}\}, \{\mathbf{Y}\}, \emptyset\rangle, \emptyset\rangle,$$

is looked up. If the analysis is extended to give information about call patterns it gives the results promised in Section 2.

9 Performance Results

We conclude with an empirical evaluation of the accuracy and usefulness of an implementation in Prolog of the analyzer presented. Our first results show that information from the analysis can be used to eliminate redundant delay declarations, leading to a large performance improvement. The last test illustrates how the analysis can be used to guide optimizations which are performed for traditional Prolog. In this case we show how implicit independent and-parallelism as detected by the analyzer can be used to parallelize the benchmark.

The benchmarks used for the evaluation were: `permute`, the permute program presented in Section 2; `qsort`, the classical quick sort program using append; `app3` which concatenates three lists by performing two consecutive calls to append; `nrev` which naively reverses a list; and `neg`, an implementation of safe negation using suspension on the groundness of the negated goal (a simple test of membership in a list). All benchmarks have been implemented in a reversible way, so that they can be used forwards and backwards, through the use of suspension declarations.

In the first test, the optimizations to eliminate unnecessary delaying were performed in two steps. The first step was to eliminate and/or relax suspension declarations as indicated by the analysis. The second step was to reorder the clause bodies provided the analysis indicated that it reduced suspension. It is important to note that although the obtained orderings are already implicit in the results of the (first) analysis, in order to eliminate suspension conditions that are redundant after the reordering, a second pass of the analysis is sometimes needed. The tests where performed with Sicstus Prolog 2.1, which is an efficient implementation of Prolog with a rich set of suspension primitives.

Due to lack of space we cannot include the code for the benchmarks and their resulting specialized versions. However, in order to give an idea of the accuracy of the analyzer and to help in understanding the efficiency results, we point out that in all cases but for `permute` the information provided by the analyzer was optimal. In the case of `permute` one condition can be relaxed beyond those inferred by the analyzer. In particular, for all the examples in their "forward" execution mode the analyzer accurately infers that no goal suspends and therefore all suspension declarations can be eliminated. With respect to the backwards execution, in all cases but `neg` the suspension conditions are either relaxed or eliminated. This does not occur for `neg` since the analyzer accurately infers that the existing groundness suspension condition is still needed for correctness. Finally, with respect to the optimizations where reordering is allowed, all backward executions are reordered in such a way that no suspension conditions are needed. Thus, we can conclude that the accuracy results for the analyzer are encouraging.

Table 1 lists execution times, expressed in seconds, for the original benchmarks and the optimized versions. Each column has the following meaning: **Name** – program name, **Query** – number of elements in the list given as query, **P** – execution time for the program written in standard Prolog, i.e. with no suspension declarations, **S** – execution time for the program written with suspension declarations, **SO** – execution time for program written with suspension declarations and optimized by removing suspension declarations as dictated by the analysis information, **S/SO** – ratio between the last two columns, **R** – execution time for the program optimized by reordering the clause bodies as dictated by the analysis information, and **R/S** – ratio between **R** and **S** columns. In the **P** column **In** stands for non-termination, and **Er** stands for a wrong result or an execution error (the fact that these cases appear shows the superiority of the version of the program with suspension declarations). Two sets of data (corresponding to two lines in the table) are given for each program, the first one corresponding to "forwards" execution of the program, the second to the "backwards" execution.

Note that in some cases the number of elements given as queries for forward execution are different from those used for the backward execution of the same program. The reason is the amount of time required by each query due to the different behaviour when running forwards (one solution) and backwards (multiple solutions).

The results are rather appealing as they show that the optimizations based on relaxing and eliminating suspension declarations using the information provided by the analyzer allows use of the more general version of the program written with suspension declarations with-

Name	Query	P	S	SO	S/SO	R	S/R
permute	8	In	27.2	24.0	1.1	0.7	38.9
	8	2.0	20.6	2.0	10.3	2.0	10.3
app3	20000	0.2	4.7	0.2	23.5	0.2	1.5
	1000	In	12.2	1.6	7.6	1.4	8.7
qsort	2000	0.8	74.3	0.8	92.9	0.8	92.9
	7	Er	20.8	4.7	4.4	0.7	29.7
nrev	300	0.2	21.4	0.2	107.0	0.2	107.0
	300	In	28.4	3.1	9.2	0.5	56.8
neg	400000	2.4	3.5	2.4	1.5	2.4	1.5
	400000	Er	3.5	3.5	1.0	2.4	1.5

Table 1: Analysis and optimization with delay

"Reversible" quick-sort, 5000 elements	Time
Standard Prolog	1.23
Suspension declarations, after analysis and reordering	1.23
Above program, parallelized, 1 processor	1.30
Above program, parallelized, 2 processors	0.81
Above program, parallelized, 4 processors	0.53
Above program, parallelized, 6 processors	0.46

Table 2: Analysis and optimization of quick-sort

out a performance penalty when executing the program in the mode that runs in Prolog. Furthermore, the analysis and resultant optimization also improves execution speed even if some suspensions still need to be used during execution. The optimizations based on reordering give even more impressive results. This is mainly explained by the fact mentioned above that for all programs the reordering has achieved the elimination of all suspension declarations.

Finally, in the last test, we show how information from the analysis can be used to perform optimizations used in the compilation of traditional Prolog. As an example we consider automatic parallelization based on the independent and parallelism model. The only program in which this kind of parallelism exists for the given queries is `qsort`. In this case the parallelism can be automatically exploited using existing tools given the information obtained from the analysis. This is because the analysis determines that there is no goal suspension in the reordered program and so the tools and techniques described in [20, 27] are applicable. These techniques can also be extended to deal with cases in which goals are delayed by extending the notion of dependence, but that is beyond the scope of this paper. A significant reduction in computation time is obtained from parallelism at least for the forward query. This is illustrated in Table 2, which shows results from running the forward query with the optimized program under &-Prolog [19], a parallel version of Sicstus Prolog, running on a commercial multiprocessor. Times are in seconds.

10 Conclusion

We have given a framework for global dataflow analysis of logic languages with dynamic scheduling. The framework extends memoization based analyses for traditional logic programming languages with a fixed left-to-right scheduling. Information from analyses based on the framework can be used to perform optimizations which remove the overhead of dynamic scheduling and also to perform optimizations used in the compilation of traditional Prolog.

A potential application of the framework is for the analysis of constraint logic programming languages which handle difficult constraints by delaying them until they become simpler. Information from an analysis based on our framework could be used to avoid testing constraints for difficulty at run-time, or to move difficult constraints to points in the program in which they are simpler, thus avoiding suspensions. An analysis specifically for this purpose has also recently been suggested by Hanus [18].

Acknowledgements

We thank Lee Naish who suggested several of the benchmarks. We also thank Peter Breuer for his comments on an earlier version of the paper.

References

[1] M. Carlsson. Freeze, Indexing, and Other Implementation Issues in the Wam. In *Fourth International Con-*

ference on Logic Programming, pages 40–58. University of Melbourne, MIT Press, May 1987.

[2] M. Carlsson. *Sicstus Prolog User's Manual.* Po Box 1263, S-16313 Spanga, Sweden, February 1988.

[3] M. Codish. A Provably Correct Algorithm for Sharing and Freeness Inference. In *1992 Workshop on Static Analysis WSA'92*, September 1992.

[4] M. Codish, M. Falaschi, and K. Marriott. Suspension Analysis for Concurrent Logic Programs. In K. Furukawa, editor, *Proc. Eighth Int'l Conf. on Logic Programming*, pages 331– 345. The MIT Press, Cambridge, Mass., 1991.

[5] M. Codish, M. Falaschi, K. Marriott and W. Winsborough. Efficient analysis of concurrent constraint logic programs. *Proc. of Twentieth Int. Coll. Automata, Languages and Programming*, A. Lingus and R. Karlsson and S. Carlsson (Ed.), LNCS Springer Verlag, pages 633-644.

[6] C. Codognet, P. Codognet, and M. Corsini. Abstract Interpretation for Concurrent Logic Languages. In S. Debray and M. Hermenegildo, editors, *Proc. North American Conf. on Logic Programming'90*, pages 215–232. The MIT Press, Cambridge, Mass., 1990.

[7] P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. *Proc. of the Fourth ACM Symposium on Principles of Programming Languages*, 238–252, 1977.

[8] P. Cousot and R. Cousot. Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation. Technical report, LIX, Ecole Polytechnique, France, 1991.

[9] P. Cousot and R. Cousot. Abstract Interpretation and Application to Logic Programs. *Journal of Logic Programming*, 13(2 and 3):103–179, July 1992.

[10] S. Debray. Static Analysis of Parallel Logic Programs. In *Fifth Int'l Conference and Symposium on Logic Programming*, Seattle,Wasinghton, August 1988. MIT Press.

[11] S. K. Debray. Static Inference of Modes and Data Dependencies in Logic Programs. *ACM Transactions on Programming Languages and Systems* **11** (3), 418–450, 1989.

[12] S.K. Debray. QD-Janus: A Sequential Implementation of Janus in Prolog. Technical Report, University of Arizona, 1993.

[13] S. K. Debray and D. S. Warren. Functional Computations in Logic Programs. *ACM Transactions on Programming Languages and Systems* **11** (3), 451–481, 1989.

[14] M. Falaschi, M. Gabbrielli, K. Marriott and C. Palamidessi. Compositional analysis for concurrent constraint programming. *IEEE Symposium on Logic in Computer Science*, Montreal, June 1993.

[15] M. Garcia de la Banda and M. Hermenegildo. A Practical Application of Sharing and Freeness Inference. In *1992 Workshop on Static Analysis WSA'92*, pages 118–125, Bourdeaux, France, September 1992.

[16] D. Gudeman, K. De Bosschere and S.K. Debray. `jc`: An Efficient and Portable Sequential Implementation of Janus. In *Proc. of 1992 Joint International Conference and Symposium on Logic Programming*, 399–413. MIT Press, November 1992.

[17] M. Hanus. On the Completeness of Residuation. In *Proc. of 1992 Joint International Conference and Symposium on Logic Programming*, 192–206. MIT Press, November 1992.

[18] M. Hanus. Analysis of Nonlinear Constraints in CLP(R). In *Proc. of 1993 International Conference on Logic Programming*, 83–99. MIT Press, June 1993.

[19] M. Hermenegildo and K. Greene. &-Prolog and its Performance: Exploiting Independent And-Parallelism. In *1990 International Conference on Logic Programming*, pages 253–268. MIT Press, June 1990.

[20] M. Hermenegildo, R. Warren, and S. Debray. Global Flow Analysis as a Practical Compilation Tool. *Journal of Logic Programming*, 3(4):349–367, August 1992.

[21] T. Hickey and S. Mudambi. Global Compilation of Prolog. *Journal of Logic Programming*, **7**, 193–230, 1989.

[22] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *Proc. Fourteenth Ann. ACM Symp. Principles of Programming Languages*, pages 111–119, 1987.

[23] N. D. Jones and A. Mycroft. Dataflow analysis of applicative programs using minimal function graphs. In *Proc. Thirteenth Ann. ACM Symp. Principles of Programming Languages*, pages 296–306. St. Petersburg, Florida, 1986.

[24] K. Marriott, H. Søndergaard, and P. Dart. A characterization of non-floundering logic programs. In S. K. Debray and M. Hermenegildo, editors, *Logic Programming: Proc. North American Conf. 1990*, pages 661–680. MIT Press, 1990.

[25] K. Marriott, H. Søndergaard and N. D. Jones. Denotational abstract interpretation of logic programs. To appear in *ACM Trans. Programming Languages and Systems.*

[26] C. S. Mellish. The automatic generation of mode declarations for Prolog programs. Technical Report 163, Dept. of Artificial Intelligence, University of Edinburgh, Scotland, 1981.

[27] K. Muthukumar and M. Hermenegildo. The CDG, UDG, and MEL Methods for Automatic Compile-time Parallelization of Logic Programs for Independent And-parallelism. In *1990 International Conference on Logic Programming*, pages 221–237. MIT Press, June 1990.

[28] K. Muthukumar and M. Hermenegildo. Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In *1991 International Conference on Logic Programming*, pages 49–63. MIT Press, June 1991.

[29] K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 13(2 and 3):315–347, July 1992.

[30] L. Naish. *Negation and Control in Prolog*, LNCS 238, Springer-Verlag, 1985.

[31] A. Taylor. LIPS on a MIPS: Results from a Prolog Compiler for a RISC. *Proc. of the 7th International Conference on Logic Programming*, 174–185, 1990.

[32] P. Van Roy and A.M. Despain. The Benefits of Global Dataflow Analysis for an Optimizing Prolog Compiler. *Proc. of the 1990 North American Conference on Logic Programming*, 501–515, 1990.

[33] R. Warren, M. Hermenegildo and S.K. Debray. On the Practicality of Global Flow Analysis of Logic Programs. *Proc. of the 5th International Conference and Symposium on Logic Programming*, 684–699, 1988.

[34] K. Yelick and J. Zachary. Moded type systems for logic programming. In *Proc. Sixteenth Annual ACM Symp. on Principles of Programming Languages*, pages 116–124. ACM, 1989.