Efficient Analysis of Constraint Logic Programs with Dynamic Scheduling[*]

María García de la Banda[†] Kim Marriott[*] Peter Stuckey[‡]

Abstract

The first logic programming languages, such as Prolog, used 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 languages include constraint logic programming languages in which constraints which are "too hard" are delayed and concurrent constraint languages in which delay on shared variables is used to provide asynchronous communication between processes. We give a new framework for the global analysis of logic programming languages with dynamic scheduling which is based on approximating the delayed atoms by a closure operator. We give an example analysis for groundness based on this framework, and give the results of an implementation which demonstrates the method is practical.

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, most "second-generation" logic programming languages, 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. Most constraint logic programming languages also employ dynamic scheduling. If a constraint is "too hard" for the solver, it is delayed until it becomes simpler. For example, in CLP($\mathcal{R}$) non-linear arithmetic constraints are delayed until they become linear.

Unfortunately, dynamic scheduling has a significant cost; literals, 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 awoken or put in a "pending" list, so that they are awoken before the next literal is executed; also, few register allocation optimizations can be performed for delayed literals; finally, space needs to be allocated for delayed literals until they are awoken [3]. 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 significantly improve execution speed of traditional Prologs [14, 20, 21]. Preliminary tests in [17] suggest that global dataflow analysis information for logic languages with dynamic scheduling allows optimizations which improve performance by an order of magnitude.

However, it is not simple to extend analyses for traditional Prolog and constraint logic programming languages to languages with dynamic scheduling, as in existing analyses the fixed scheduling is crucial to ensure

[*]This work was funded in part by ESPRIT project 7195 "ACCLAIM" and by CICYT project "IPL-D"
[†]Dept. of Computer Science, Monash University, Clayton Vic 3168, Australia. {mbanda,marriott}@cs.monash.edu.au
[‡]Dept. of Computer Science, Melbourne University, Parkville Vic 3152, Australia. pjs@cs.mu.oz.au

correctness and termination. Here we develop a framework for the global dataflow analysis of (constraint) logic languages with dynamic scheduling. To our knowledge this is the first practical framework for the analysis of this important class of languages. It 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.

Our main results are threefold. First, we give an approximate denotational semantics for languages with dynamic scheduling. This provides the semantic basis for our generic analysis. The key feature of this semantics is that delayed atom sequences are approximated by a closure operator which details the effect of the delayed atoms if the current constraint store is modified. 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 constraints. 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. A feature of the analysis which is crucial for its efficiency is that the closures (representing delayed atoms) are evaluated lazily. In effect a closure is represented as a partial function. When computing a fixpoint closures are treated as equal as long as they have the same values on the (partial) domain of interest. Third, we give empirical results which show that the overhead of the method for programs without delay is minimal and the performance on programs with delay is reasonable and considerably better than the only other comparable approach [17].

Our work extends that of Marriott *et. al.* [17] which gives the first generic dataflow analysis for logic programming languages with delay. In their analysis, sequences of delayed atoms are approximated by multisets of atoms. The problem with this approach is that the multisets of delayed atoms have unbounded size, and so to guarantee termination ad hoc widening steps were required. Thus the method is unable to handle chained dependencies with any accuracy since the widening steps required lose too much accuracy. The method is also forced to consider abstractions on very large sets of variables, this tends to make the analysis very slow for even small programs when large number of atoms are delayed. This meant that the analysis was imprecise and rather inefficient in practice. The current work removes these problems. The empirical results demonstrate that our framework is an order of magnitude faster and more accurate.

Other related work is the global analysis of concurrent constraint programming languages [4, 5, 6, 10]. 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. Debray et al. [8] gives a global analysis for logic languages with dynamic scheduling which determines parts of the program in which delay and wakeup can never occur. This allows the compilation of these parts to be optimized. In order to do this, the analyser uses a very simple model of delay behaviour which does not give accurate calling pattern information. Finally, Hanus [11, 12] gives analyses for improving the residuation mechanism in functional logic programming languages, and determining when no non-linear constraints delay in a constraint logic program. Both these analyses handles the delay and waking of constraints, but do not extend to handle atoms which may spawn subcomputations which in turn have delayed atoms.

In the next section we give the operational semantics of logic languages with dynamic scheduling. In Section 4 we give the denotational semantics. In Section 5 we give the generic analysis framework. Section 6 presents some performance results and in Section 7 we conclude. In an appendix we give an example analysis.

2 Operational Semantics

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

A *constraint logic program*, or *program*, is a finite set of rules. A *rule* 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, non-empty sequence of literals. A *literal* is either an atom or a primitive constraint. An *atom* has the form $\mathbf{p}(\mathbf{x}_1, ..., \mathbf{x_n})$ where $\mathbf{p}$ is a predicate symbol and the $\mathbf{x_i}$ are distinct variables. A *primitive constraint* is essentially a predefined predicate, such as term equations or inequalities over the reals. Arguments to a primitive constraint are terms which may be constructed using predefined functions such as real addition. The syntax given here is more restrictive than is usual, as this will simplify the rest of the paper. However the restrictions are only syntactic, as we can always rewrite an atom $\mathbf{p}(\mathbf{t}_1, ..., \mathbf{t_n})$ with arbitrary terms as arguments into $\mathbf{p}(\mathbf{x}_1, ..., \mathbf{x_n}), \mathbf{x}_1 = \mathbf{t}_1, ..., \mathbf{x_n} = \mathbf{t_n}$.

A *constraint* is a conjunction of primitive constraints. Constraints are treated modulo logical equivalence, and are assumed to be closed under existential quantification and conjunction. Thus constraints can be ordered by logical implication, that is $\theta \leq \theta'$ iff $\theta' \Rightarrow \theta$. The greatest constraint is denoted by **false**. It is the unsatisfiable constraint. The least constraint is denoted by **true**. It is the always satisfiable constraint. We let $\exists_{\mathbf{W}}\, \theta$ denote the 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 $\bar{\exists}_{\mathbf{W}}\theta$ be constraint θ restricted to the variables $\mathbf{W}$. That is $\bar{\exists}_{\mathbf{W}}\theta$ 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.

A *renaming* is a bijective mapping between variables. We let **Ren** be the set of renamings, and naturally extend renamings to mappings between atoms, rules, 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* $\mathbf{P}$, $\mathbf{defn_P}(\mathbf{A})$, is the set of variants of rules in $\mathbf{P}$ such that each variant has $\mathbf{A}$ as a head and apart from the variables in $\mathbf{A}$ has distinct new variables.

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 constraint θ, and the current sequence of delayed atoms $\mathbf{D}$. Literals in the goals are processed left-to-right. There are two cases. If the literal is a constraint, and it is consistent with the current constraint, it is added to it and delayed literals that are awoken by the addition are processed. The other case is when the literal is an atom. If it is not sufficiently instantiated to be processed it is placed in the delayed atom sequence. If the literal is not delayed, it is replaced by the body of a rule in its definition. Our definition makes use of two parametric functions which are dependent on the systems or language being modeled. These are, $\mathbf{delay}(\mathbf{A}, \theta)$, which holds iff a call to atom $\mathbf{A}$ delays with the constraint θ, and $\mathbf{awoken}(\mathbf{D}, \theta)$, which is the subsequence of atoms in the sequence of delayed atoms $\mathbf{D}$ that are awoken by constraint θ. Note that the order of the calls returned by **awoken** is system dependent.

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

1. If $\mathbf{L}$ is a primitive constraint 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 $\mathbf{D}' = \mathbf{awoken}(\mathbf{D}, \theta \wedge \mathbf{L})$.
2. If $\mathbf{L}$ is an atom and $\mathbf{delay}(\mathbf{L}, \theta)$ holds, it is reduced to $\langle \mathbf{G}, \theta, \mathbf{L} : \mathbf{D} \rangle$.
3. If $\mathbf{L}$ is an atom and $\mathbf{delay}(\mathbf{L}, \theta)$ does not hold, it is reduced to $\langle \mathbf{B} :: \mathbf{G}, \theta, \mathbf{D} \rangle$ for some $(\mathbf{L} \leftarrow \mathbf{B}) \in \mathbf{defn_P}(\mathbf{L})$.

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

The observational behavior of a program is given by its "answers" to goals. A derivation from a goal $\mathbf{G}$ for program $\mathbf{P}$ is *successful* if the last state has form $\langle \mathbf{nil}, \theta, \mathbf{D} \rangle$, that is the current goal is the empty goal. The constraint $\bar{\exists}_{\mathbf{vars}(\mathbf{G})}\theta$ is an *answer to* $\mathbf{S}$. As there is a non-deterministic choice of the rule in an atom's definition, there may be a number of answers generated from the initial state or goal. We denote the set

$$
\begin{array}{cl}
\langle \mathbf{path}(\mathbf{X},\mathbf{Y}),\ \ \theta_1,\ \ \mathbf{nil}\rangle & \quad \textbf{where } \theta_1 \textbf{ is } \mathbf{Y}=\mathbf{b} \\
\Downarrow & \\
\langle \mathbf{edge}(\mathbf{X},\mathbf{Z}) : \mathbf{path}(\mathbf{Z},\mathbf{Y}),\ \ \theta_1,\ \ \mathbf{nil}\rangle & \\
\Downarrow & \\
\langle \mathbf{path}(\mathbf{Z},\mathbf{Y}),\ \ \theta_1,\ \ \mathbf{edge}(\mathbf{X},\mathbf{Z})\rangle & \\
\Downarrow & \\
\langle \mathbf{Z}=\mathbf{Y},\ \ \theta_1,\ \ \mathbf{edge}(\mathbf{X},\mathbf{Z})\rangle & \\
\Downarrow & \\
\langle \mathbf{edge}(\mathbf{X},\mathbf{Z}),\ \ \theta_2,\ \ \mathbf{nil}\rangle & \quad \textbf{where } \theta_2 \textbf{ is } \theta_1 \wedge \mathbf{Z}=\mathbf{Y} \\
\Downarrow & \\
\langle \mathbf{X}=\mathbf{a}, \mathbf{Z}=\mathbf{b},\ \ \theta_2,\ \ \mathbf{nil}\rangle & \\
\Downarrow & \\
\langle \mathbf{Z}=\mathbf{b},\ \ \theta_3,\ \ \mathbf{nil}\rangle & \quad \textbf{where } \theta_3 \textbf{ is } \theta_2 \wedge \mathbf{X}=\mathbf{a} \\
\Downarrow & \\
\langle \mathbf{nil},\ \ \theta_4,\ \ \mathbf{nil}\rangle & \quad \textbf{where } \theta_4 \textbf{ is } \theta_3 \wedge \mathbf{Z}=\mathbf{b}.
\end{array}
$$

Figure 1: Example Derivation

of answers to a goal $\mathbf{G}$ for program $\mathbf{P}$ by $\mathbf{ans_P(G)}$. In the case when no literals delay and the constraints are term equations, this semantics is the same as the usual operational semantics of Prolog. Note that the answers to a goal must always be satisfiable, from the definition it is impossible for **false** to be a valid answer.

As an example, consider the initial state $\langle \mathbf{path}(\mathbf{X},\mathbf{Y}), \mathbf{Y}=\mathbf{b}, \mathbf{nil}\rangle$ and the program below. One of the successful derivations is shown in Figure 1.

```
?- path(X,Y) when ground(Y)
path(X,Y)  <-  X = Y
path(X,Y)  <-  edge(X,Z), path(Z,Y)
?- edge(X,Y) when ground(Y)
edge(a,b)
edge(b,c)
```

Following [17] we assume that the parametric functions **delay** and **awoken** satisfy the following four conditions. The first ensures that there is a congruence between the conditions for delaying an atom and waking it. The remaining conditions ensure that **delay** behaves reasonably: it should not take variable names into account; it should only be concerned with the effect of θ on the variables in $\mathbf{A}$; and finally if an atom is not delayed, adding more constraints should never cause it to delay.

(1) $\mathbf{A} \in \mathbf{awoken}(\mathbf{D},\theta)$ iff $\mathbf{A} \in \mathbf{D}$ and $\mathbf{delay}(\mathbf{A},\theta)$ does not hold.
(2) For any renaming ρ, $\mathbf{delay}(\mathbf{A},\theta)$ iff $\mathbf{delay}(\rho(\mathbf{A}),\rho(\theta))$.
(3) $\mathbf{delay}(\mathbf{A},\theta)$ iff $\mathbf{delay}(\mathbf{A},\bar{\exists}_{\mathbf{vars(A)}}\theta)$.
(4) If $\theta \leq \theta'$ and $\mathbf{delay}(\mathbf{A},\theta)$, then $\mathbf{delay}(\mathbf{A},\theta')$.

These conditions are crucial to the analysis we will develop, as they mean that literals behave as sets of closure operators. The conditions are met in many existing systems and languages.

For simplicity we have ignored constraints which delay. These may be modeled in our setting by wrapping them with atoms which can delay. For example delay of non-linear multiplication constraints $\mathbf{X} = \mathbf{Y} * \mathbf{Z}$ in CLP($\mathcal{R}$) is captured by the rule

$$\mathtt{mult(X,Y,Z)} \leftarrow \mathtt{X = Y * Z}.$$

where $\mathbf{delay}(\mathbf{mult}(\mathbf{X},\mathbf{Y},\mathbf{Z}),\theta)$ holds whenever θ does not constrain $\mathbf{Y}$ or $\mathbf{Z}$ to be ground.

Actually the operational 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 constraint satisfaction. Therefore, we must obtain information about the *call patterns*. That is, for each atom $\mathbf{A}$ appearing in the program we want to know whether the calls to the atom initially delay, and when each call to $\mathbf{A}$ is eventually reduced, perhaps after being delayed, the value of the current constraint restricted to the variables in **A**.

3 Closure Based Semantics

In this section we give a closure based semantics for logic programs with delay. Latter we will use the ideas in this semantics to give a slightly more complex semantics which is a suitable basis for the abstract interpretation of languages with delay.

One important feature of a semantics for analysis is that it should be compositional: the meaning of a rule body is the composition of the meaning of its constituent literals. The first, intuitive, idea for a compositional semantic definition is that literals and goals take an environment consisting of the current constraint and current sequence of delayed atoms and transform this into a new environment. However, the denotation of the literals and goals needs to be a function from a *set* of environments into a set of environments. This is because, due to non-determinism, a single environment may be transformed into a set of possible environments, and because it is convenient for descriptions to describe sets of environments.

Using this idea, we can indeed develop a compositional semantics for logic programs with dynamic scheduling (see [17]). Unfortunately, however this semantics is not a good basis for the analysis of such programs. The problem is that the number of variables and number of delayed atoms in the environment is unbounded and thus hard to finitely abstract. Consider the `path` program in the previous section. Execution of the goal

$$\mathbf{Y} = \mathbf{c}, \mathbf{path}(\mathbf{X}, \mathbf{Y})$$

will build up an unbounded collection of delayed **edge** atoms, involving an unbounded number of variables.

If we look carefully at this example, however, we realize that this problem is really an artifact. Consider a finite set of variables **W**. If there are finite number of different descriptions for constraints over **W** then there are a finite number of functions between these descriptions over **W**. It follows that, for a particular atom **A** with variables **W**, the different effects the delayed atom sequences can have on the evaluation of **A** are also finite in number. Thus, the problem is really caused by having an infinite number of different names for a finite set of "functions". For example the total groundness effect of the atom $\mathbf{path}(\mathbf{X}, \mathbf{Y})$ is captured by the function: if **Y** is ground, then so is **X**.

This leads to the idea of a semantics in which instead of carrying around the sequence of delayed atoms we carry around its denotation restricted to the variables we are currently interested in. Unfortunately naive application of this idea leads to problems, as the denotation will be a mapping from sets of environments to sets of environments and an environment will be a constraint together with a denotation represemting the delayed atoms. The recursion in these types makes such a denotation a very complex object, which is expensive to compute.

A better approach is to capture the denotation of the delayed atoms as a set of closure operators over the constraints. Marriott *et al* [16] has given a simple denotational semantics for logic programming languages with delay based on sets of closure operators. Different operators correspond to different sequences of choices. We revise this semantics here.

Definition. Let $(\mathbf{X}, \preceq)$ be a poset and **F** be an operator on **X**.

- **F** is *monotonic* if for all $\mathbf{x}, \mathbf{x}' \in \mathbf{X}$, $\mathbf{x} \preceq \mathbf{x}'$ implies $\mathbf{F}(\mathbf{x}) \preceq \mathbf{F}(\mathbf{x}')$.
- **F** is *idempotent* if $\mathbf{F} = \mathbf{F} \circ \mathbf{F}$.
- **F** is *increasing* if for all $\mathbf{x} \in \mathbf{X}$, $\mathbf{x} \preceq \mathbf{F}(\mathbf{x})$.
- **F** is a *closure operator* if **F** is monotonic, idempotent, and increasing.

The *closure based semantics* for a program $\mathbf{P}$ is the least fixpoint of the following eqquations:

$$
\begin{array}{lcl}
\mathbf{BodyClos(nil)} & = & \{\mathbf{Id_{Clos}}\} \\
\mathbf{BodyClos(L:B)} & = & \mathbf{Comb(LitClos(L), BodyClos(B))} \\
\mathbf{LitClos(L)} & = & \text{if } \mathbf{L} \text{ is an atom then } \mathbf{AtomClos(L)} \text{ else } \mathbf{ConClos(L)} \\
\mathbf{ConClos(c)} & = & \{\lambda\theta.\theta \wedge \mathbf{c}\} \\
\mathbf{AtomClos(L)} & = & \mathbf{AddDelay(A, AtomClos_{Awake}(A))} \\
\mathbf{AtomClos_{Awake}(A)} & = & \mathbf{Lub}\{\mathbf{RuleClos(A \leftarrow B)} \mid (\mathbf{A} \leftarrow \mathbf{B}) \in \mathbf{defn_P(A)}\} \\
\mathbf{RuleClos(A \leftarrow B)} & = & \mathbf{Restrict(vars(A), BodyClos(B))}.
\end{array}
$$

Figure 2: Closure Based Semantics

We let $\bullet$ denote the composition operator for closure operators. It is defined by

$$(\mathbf{F} \bullet \mathbf{G})(\mathbf{x}) = \bigsqcap\{\mathbf{x}' \mid \mathbf{x}' = \mathbf{F}(\mathbf{x}') \text{ and } \mathbf{x}' = \mathbf{G}(\mathbf{x}') \text{ and } \mathbf{x} \preceq \mathbf{x}'\}.$$

In the definition we are interested in closure operators over constraints: $\mathbf{Clos} \subseteq \mathbf{Con} \rightarrow \mathbf{Con}$. It will be convenient to regard a closure operator $\kappa \in \mathbf{Clos}$ as having a fixed variable set it is defined over, denoted by $\mathbf{vars}(\kappa)$. We extend this naturally to sets of closure operators with the same associated variables. Thus we will talk of the closure operators over a variable set $\mathbf{W}$.

The definition makes use of several auxiliary functions on closure operators defined by:

$$
\begin{array}{lcl}
\mathbf{Apply}(\mathbf{K}, \Theta) & = & \{\kappa(\theta) \mid \kappa \in \mathbf{K}, \theta \in \Theta\} \\
\mathbf{AddDelay}(\mathbf{A}, \mathbf{K}) & = & \{\lambda\theta.\text{if } \mathbf{delay}(\mathbf{A}, \theta) \text{ then } \theta \text{ else } \kappa(\theta) \mid \kappa \in \mathbf{K}\} \\
\mathbf{Comb}(\mathbf{K}, \mathbf{K}') & = & \{\kappa \bullet \kappa' \mid \kappa \in \mathbf{K}, \kappa' \in \mathbf{K}'\} \\
\mathbf{Lub}(\mathbf{K}) & = & \bigcup \mathbf{K} \\
\mathbf{Restrict}(\mathbf{W}, \mathbf{K}) & = & \{\bar{\exists}_{\mathbf{W}} \circ \kappa \mid \kappa \in \mathbf{K}\}
\end{array}
$$

as well as the constant function $\mathbf{Id_{Clos}} = \lambda\theta.\theta$. $\mathbf{Apply}(\mathbf{K}, \Theta)$ applies each closure operator in $\mathbf{K}$ to each constraint in Θ; $\mathbf{AddDelay}(\mathbf{A}, \mathbf{K})$ modifies each closure operator $\mathbf{K}$ so that if $\mathbf{A}$ delays for a constraint θ then the closure returns θ but behaves as before on constraints for which $\mathbf{A}$ does not delay; $\mathbf{Comb}(\mathbf{K}, \mathbf{K}')$ composes each closure operator in $\mathbf{K}$ with each closure operator in $\mathbf{K}'$; $\mathbf{Lub}(\mathbf{K})$ flattens a set of sets of closure operators into a set of closure operators; $\mathbf{Restrict}(\mathbf{W}, \mathbf{K})$ returns a closure operator with the same effect as $\mathbf{K}$ on the variables in $\mathbf{W}$ but with its other variables hidden. The assumptions are that for $\mathbf{Comb}(\mathbf{K}, \mathbf{K}')$, $\mathbf{vars}(\mathbf{K}') = \mathbf{vars}(\mathbf{K})$, for $\mathbf{Restrict}(\mathbf{W}, \mathbf{K})$, $\mathbf{vars}(\mathbf{K}) \supseteq \mathbf{W}$.

The closure based semantics is given in Figure 2. The denotation of a body is computed by $\mathbf{BodyClos}(\mathbf{B})$. It is the composition of the closure operators of each literal in $\mathbf{B}$. The denotation of a literal is computed by $\mathbf{LitClos}$. If the literal is a primitive constraint $\mathbf{c}$, then the denotation is just the closure operator which adds $\mathbf{c}$ to the current constraint θ. If the literal is an atom, $\mathbf{AtomClos_{Awake}}$ is used to compute the denotation of the atom if it never delays, and $\mathbf{AddDelay}$ is used to modify the closures so that they model delay of $\mathbf{A}$ appropriately. $\mathbf{AtomClos_{Awake}}$ combines the closures from each rule in the definition of the literal, first restricting the closures to the variables in $\mathbf{A}$.

Theorem 3.1 ([16])
Let $\mathbf{B}$ be a body and $\mathbf{P} \in \mathbf{Prog}$. Then $\mathbf{ans_P}(\mathbf{B}) = \mathbf{Apply}(\mathbf{BodyClos}(\mathbf{B}), \{\mathbf{true}\}) \setminus \{\mathbf{false}\}$. ■

4 Hybrid Semantics for Analysis

In this section we develop a semantics for programs with dynamic scheduling which is a suitable basis for program analysis. It is based on the closure semantics given in the last section. The reason that we do not use

The *hybrid semantics* for a program $\mathbf{P}$ is the least fixpoint of the following equations:

$$
\begin{array}{ll}
\mathbf{BodyHyb}(\mathbf{nil}, \mathbf{K}) & = \mathbf{K} \\
\mathbf{BodyHyb}(\mathbf{L} : \mathbf{B}, \mathbf{K}) & = \mathbf{BodyHyb}(\mathbf{B}, (\mathbf{LitHyb}(\mathbf{L}, \mathbf{K})) \\
\mathbf{LitHyb}(\mathbf{L}, \mathbf{K}) & = \text{if } \mathbf{L} \text{ is an atom then } \mathbf{AtomHyb}(\mathbf{L}, \mathbf{K}) \text{ else } \mathbf{ConHyb}(\mathbf{L}, \mathbf{K}) \\
\mathbf{ConHyb}(\mathbf{c}, \mathbf{K}) & = \mathbf{Add}(\mathbf{c}, \mathbf{K}) \\
\mathbf{AtomHyb}(\mathbf{L}, \mathbf{K}) & = \text{Let } \mathbf{K}' = \mathbf{Restrict}(\mathbf{vars}(\mathbf{A}), \mathbf{K}) \text{ in} \\
 & \quad \text{Let } \mathbf{K}'' = \mathbf{Lub}\{\mathbf{AtomHyb}_{\mathbf{awake}}(\mathbf{A}, \mathbf{Awake}(\mathbf{A}, \mathbf{K}')), \mathbf{AtomHyb}_{\mathbf{delay}}(\mathbf{A}, \mathbf{Delay}(\mathbf{A}, \mathbf{K}'))\} \text{ in} \\
 & \quad\ \mathbf{Comb}(\mathbf{K}, \mathbf{Extend}(\mathbf{vars}(\mathbf{K}), \mathbf{K}'')) \\
\mathbf{AtomHyb}_{\mathbf{delay}}(\mathbf{A}, \mathbf{K}) & = \mathbf{Comb}(\mathbf{K}, \mathbf{Extend}(\mathbf{vars}(\mathbf{K}), \mathbf{AtomClos}(\mathbf{A}))) \\
\mathbf{AtomHyb}_{\mathbf{awake}}(\mathbf{A}, \mathbf{K}) & = \mathbf{Lub}\{\mathbf{RuleHyb}(\mathbf{R}, \mathbf{K}) \mid (\mathbf{R}) \in \mathbf{defn}_{\mathbf{P}}(\mathbf{A})\} \\
\mathbf{RuleHyb}(\mathbf{A} \leftarrow \mathbf{B}, \mathbf{K}) & = \mathbf{Restrict}(\mathbf{vars}(\mathbf{A}), \mathbf{BodyHyb}(\mathbf{B}, \mathbf{Extend}(\mathbf{vars}(\mathbf{A} \leftarrow \mathbf{B}), \mathbf{K}))).
\end{array}
$$

Figure 3: Hybrid Semantics

the semantics of the last section directly is that this semantics cannot give precise information about calling patterns. The semantics does not encode a left to right default scheduling. This does not affect information about answers, because answers are independent of scheduling, but does affect information about call patterns which are not independent from scheduling.

The idea behind the semantic definitions given in this section is to represent a sequence of delayed atoms by a set of closure operators, and to process non-delayed atoms using the standard left-to-right scheduling in a similar manner to that used for analysis of traditional logic programs without delay. This means that in the limit for non-delayed atoms, information about calling patterns is as precise as we wish. For delayed atoms, however, we lose some information about their wakeup calling patterns, but we do achieve termination. We now formalize this "hybrid" semantics.

At first it seems that we need to view a single environment as the current constraint plus a set of closure operators which represent the denotation of the delayed atoms. In fact we can combine information about the current constraint θ with the closure operator κ in the closure operator $\kappa' = \lambda \theta'.\kappa(\theta \wedge \theta')$. We have that $\kappa'(\mathbf{true})$ is θ, because θ must be a fixpoint of κ, and that κ and κ' behave identically for all contexts in which they will be used, that is all contexts in which the current constraint implies θ. This "trick" significantly simplifies the semantic definition.

The hybrid definition makes use of the functions

$$
\begin{array}{lcl}
\mathbf{Awake}(\mathbf{A}, \mathbf{K}) & = & \{\kappa \in \mathbf{K} \mid \neg\, \mathbf{delay}(\mathbf{A}, \kappa(\mathbf{true}))\} \\
\mathbf{Delay}(\mathbf{A}, \mathbf{K}) & = & \{\kappa \in \mathbf{K} \mid \mathbf{delay}(\mathbf{A}, \kappa(\mathbf{true}))\} \\
\mathbf{Add}(\mathbf{c}, \mathbf{K}) & = & \{(\lambda \theta.\kappa(\theta \wedge \mathbf{c})) \mid \kappa \in \mathbf{K}\} \\
\mathbf{Extend}(\mathbf{W}, \mathbf{K}) & = & \mathbf{K}.
\end{array}
$$

$\mathbf{Awake}(\mathbf{A}, \mathbf{K})$ returns the closure operators in $\mathbf{K}$ which represent environments in which the atom $\mathbf{A}$ is awake; $\mathbf{Delay}(\mathbf{A}, \mathbf{K})$ returns the closure operators in $\mathbf{K}$ which represent environments in which the atom $\mathbf{A}$ is delayed; $\mathbf{Add}(\mathbf{c}, \mathbf{K})$ returns the closure operators representing the environments obtained by adding primitive constraint $\mathbf{c}$ to each environment represented by a closure operator in $\mathbf{K}$; $\mathbf{Extend}(\mathbf{W}, \mathbf{K})$ returns a closure operator whose domain is extended to the variables $\mathbf{W}$. The assumptions are that for $\mathbf{Extend}(\mathbf{W}, \mathbf{K})$, $\mathbf{vars}(\mathbf{K}) \subseteq \mathbf{W}$.

The hybrid semantics is given in Figure 3.

The denotation of a rule, given by the function **RuleHyb**, is just the denotation of its body, defined by **BodyHyb**, restricted to the variables in the head of the rule. The denotation of the body is just the composition of the denotations of the component literals. The denotation of a literal $\mathbf{L}$ is given by **LitHyb** depends on its type. If $\mathbf{L}$ is a primitive constraint then **ConHyb** uses **Add** to add it to the current environments. On the other hand, if $\mathbf{L}$ is an atom then **AtomHyb** uses the auxiliary functions **Awake** and **Delay** to split the set of environments into those environments in which $\mathbf{L}$ is not delayed and those environments in which $\mathbf{L}$ is delayed. $\mathbf{AtomHyb_{delay}}$ adds the closure corresponding to $\mathbf{L}$ to the environments in which $\mathbf{L}$ is delayed. $\mathbf{AtomHyb_{awake}}$ gives the denotation of an awake atom by combining the result for each rule defining $\mathbf{L}$. First, the closure operators representing the current environments are restricted, using **Restrict**, so that they do not constrain variables which are "irrelevant" to this call. Once the denotation is found **Comb** adds back the information about the irrelevant variables. For our purposes the ordering on $\wp\,\mathbf{Clos}$ is just the subset ordering.

Correctness of the semantics depends on correctness of the closure based semantics.

Theorem 4.1 (Correctness of Semantics)
Let $\mathbf{B}$ be a body and $\mathbf{P} \in \mathbf{Prog}$. Then $\mathbf{ans_P(B)} \subseteq \mathbf{Apply(BodyHyb(B, \{Clos_{Id}\}), \{true\})} \setminus \{\mathbf{false}\}$. ■

The inclusion is not an equality because in **AtomHyb** the link between the calling continuation and the return continuation is lost, and all calling continuations are combined with all return continuations. This is the usual loss of precision occurring in abstract interpretation of standard Prolog.

5 Generic Analysis Framework

In this section, we investigate how the hybrid semantics given in the last section can be used as a basis for program analysis. We formalize a program analysis as an abstract interpretation of the semantic equations.

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*, α, which maps an element in the standard domain $\mathbf{Y}$ to its "best" or most precise description and a *concretization function*, γ, which maps an element in the description domain $\mathbf{X}$ to the largest object it describes. Both the standard and description domain should be complete lattices and **alpha** and γ should be monotonic and *adjoint* where they are adjoint if for all $\mathbf{x} \in \mathbf{X}$ and for all $\mathbf{y} \in \mathbf{Y}$,

$$\alpha(\mathbf{y}) \leq_{\mathbf{X}} \mathbf{x} \Leftrightarrow \mathbf{y} \leq_{\mathbf{Y}} \gamma(\mathbf{x}).$$

The notion of approximation is made precise as follows: $\mathbf{x} \in \mathbf{X}$ *approximates* $\mathbf{y} \in \mathbf{Y}$ iff $\mathbf{y} \leq \gamma(\mathbf{x})$. We write this $\mathbf{x} \propto \mathbf{y}$. For *syntactic objects*, such as literals and primitive constraints, $\mathbf{S} \propto \mathbf{S'}$ iff $\mathbf{S} = \mathbf{S'}$. We extend $\propto$ to function spaces as follows. Consider $\mathbf{f} : \mathbf{X}_1 \times \cdots \times \mathbf{X_n} \to \mathbf{X'}$ and $\mathbf{g} : \mathbf{Y}_1 \times \cdots \times \mathbf{Y_n} \to \mathbf{Y'}$. We define $\mathbf{f} \propto \mathbf{g}$ iff for all $\mathbf{x}_1 \in \mathbf{X}_1$, ..., $\mathbf{x_n} \in \mathbf{X_n}$ and for all $\mathbf{y}_1 \in \mathbf{Y}_1$, ..., $\mathbf{y_n} \in \mathbf{Y_n}$, if $\mathbf{y_i} \propto \mathbf{x_i}$ for $\mathbf{i} = 1, .., \mathbf{n}$ then $\mathbf{f}(\mathbf{x}_1, ..., \mathbf{x_n}) \propto \mathbf{g}(\mathbf{y}_1, ..., \mathbf{y_n})$.

In the analysis we will be interested in describing sets of closure operators and sets of constraints. We call a description of $\wp\,\mathbf{Clos}$ a *closure description* and a description of $\wp\,\mathbf{Con}$ a *constraint description*. We shall assume that both a constraint description π and a closure description δ have an associated set of variables $\mathbf{vars}(\pi)$ and $\mathbf{vars}(\delta)$, respectively, which are the variables of the descriptions and closures they describe.

One example of a constraint description we will make use of is the *definiteness constraint description*. The description domain, denoted **Def**, is the *definite Boolean functions* [1]. The key idea in this description is to use implication to capture groundness dependencies. The reading of the function $\mathbf{x} \to \mathbf{y}$ is: "if the program variable $\mathbf{x}$ is (becomes) ground, so is (does) program variable $\mathbf{y}$." For example, the best description of the constraint set $\{\mathbf{f(x, y) = f(a, g(u, v))}\}$ is $\mathbf{x} \wedge \mathbf{y} \leftrightarrow (\mathbf{u} \wedge \mathbf{v})$. The variables associated with this description are $\{\mathbf{x, y, u, v}\}$.

We can use the hybrid semantics of Figure 3 as the basis for a generic analysis. All that is required is to replace $\wp\,\mathbf{Clos}$ and $\wp\,\mathbf{Con}$ by the closure and constraint description domain **DClos** and **DCon** say, and to provide definitions for the auxiliary functions:

$$\begin{array}{lcl lcl}
\mathbf{Apply'} & : & \mathbf{DClos} \times \mathbf{DCon} \to \mathbf{DCon} & \mathbf{Comb'} & : & \mathbf{DClos} \times \mathbf{DClos} \to \mathbf{DClos} \\
\mathbf{Add'} & : & \mathbf{Prim} \times \mathbf{DClos} \to \mathbf{DClos} & \mathbf{Lub'} & : & \wp\,\mathbf{DClos} \to \mathbf{DClos} \\
\mathbf{Awake'} & : & \mathbf{Lit} \times \mathbf{DClos} \to \mathbf{DClos} & \mathbf{Restrict'} & : & \wp\,\mathbf{Var} \times \mathbf{DClos} \to \mathbf{DClos} \\
\mathbf{Delay'} & : & \mathbf{Lit} \times \mathbf{DClos} \to \mathbf{DClos} & \mathbf{Extend'} & : & \wp\,\mathbf{Var} \times \mathbf{DClos} \to \mathbf{DClos}.
\end{array}$$

As long as these approximate the corresponding functions in the hybrid semantics then the analysis will be correct. Different choices of description domain and auxiliary functions give rise to different analyses with different accuracy and applicability.

The problem with the discussion so far is that, in a sense, we have swept the issue of how to define these parametric functions under the carpet. There has been large body of research devoted to abstract interpretation based generic analysis of logic programs, and in particular to constraint descriptions. We now show how we can leverage from this work.

Generic analysis frameworks for traditional Prolog are parametric in the choice of constraint description **DCon** and usually require abstract operations

$$\begin{array}{lcl lcl}
\mathbf{Add'_T} & : & \mathbf{Prim} \times \mathbf{DCon} \to \mathbf{DCon} & \mathbf{Comb'_T} & : & \mathbf{DCon} \times \mathbf{DCon} \to \mathbf{DCon} \\
\mathbf{Restrict'_T} & : & \wp\,\mathbf{Var} \times \mathbf{DCon} \to \mathbf{DCon} & \mathbf{Lub'_T} & : & \wp\,\mathbf{DCon} \to \mathbf{DCon} \\
\mathbf{Extend'_T} & : & \wp\,\mathbf{Var} \times \mathbf{DCon} \to \mathbf{DCon}. & & &
\end{array}$$

which must approximate the concrete operations

$$\begin{array}{lcl lcl}
\mathbf{Add_T}(\mathbf{c}, \Theta) & = & \{\mathbf{c} \wedge \theta \mid \theta \in \Theta\} & \mathbf{Comb_T}(\Theta, \Theta') & = & \{\theta \wedge \theta' \mid \theta \in \Theta, \theta' \in \Theta'\} \\
\mathbf{Restrict_T}(\mathbf{W}, \Theta) & = & \{\bar{\exists}_{\mathbf{W}}\theta \mid \theta \in \Theta\} & \mathbf{Lub_T}(\boldsymbol{\Theta}) & = & \bigcup \boldsymbol{\Theta} \\
\mathbf{Extend_T}(\mathbf{W}, \Theta) & = & \Theta. & & &
\end{array}$$

Given that **DCon** is a constraint description we can *induce* a closure description from **DCon**. The induced description domain is $\mathbf{DCon} \to \mathbf{DCon}$ and the induced abstraction function α is defined by

$$\bigsqcap\{\delta \in (\mathbf{DCon} \to \mathbf{DCon}) \mid \forall \kappa \in \mathbf{K} \,.\, \delta \propto \kappa\}.$$

We can also define many of our parametric relations over the induced closure operator descriptions from the abstract operations over **DCon**. More precisely we can define

$$\begin{array}{lcl}
\mathbf{Apply'}(\delta, \pi) & = & \delta(\pi) \\
\mathbf{Add'}(\mathbf{c}, \delta) & = & \mathbf{Comb'}(\lambda\,\pi.\mathbf{Add'_T}(\mathbf{c}, \pi), \delta) \\
\mathbf{Lub'}(\mathbf{D}) & = & \lambda\,\pi.\mathbf{Lub'_T}\{\delta(\pi) \mid \delta \in \mathbf{D}\} \\
\mathbf{Restrict'}(\mathbf{W}, \delta) & = & \lambda\,\mathbf{d}.\mathbf{Restrict'_T}(\mathbf{W}, \delta(\mathbf{Extend'_T}(\mathbf{vars}(\delta), \mathbf{d}))) \\
\mathbf{Extend'}(\mathbf{W}, \delta) & = & \lambda\,\mathbf{d}.\mathbf{Extend'_T}(\mathbf{W}, \delta(\mathbf{Restrict'_T}(\mathbf{vars}(\delta), \mathbf{d}))).
\end{array}$$

The operations **Awake′** and **Delay′** can also be naturally defined in terms of simpler operations $\mathbf{Awake'_T}$ and $\mathbf{Delay'_T}$ on **DCon**:

$$\begin{array}{lcl}
\mathbf{Awake'}(\mathbf{A}, \delta) & = & \lambda\,\mathbf{d}.\mathbf{Awake'_T}(\delta(\mathbf{d})) \\
\mathbf{Delay'}(\mathbf{A}, \delta) & = & \lambda\,\mathbf{d}.\mathbf{Delay'_T}(\delta(\mathbf{d}))
\end{array}$$

where $\mathbf{Awake'_T}(\mathbf{d})$ must approximate each $\theta \in (\gamma(\mathbf{d}) \cap \{\theta \in \mathbf{Con} \mid \neg\ \mathbf{delay}(\mathbf{A}, \theta)\}$ and $\mathbf{Delay'_T}(\mathbf{d})$ must approximate each $\theta \in \gamma(\mathbf{d}) \cap \{\theta \in \mathbf{Con} \mid \mathbf{delay}(\mathbf{A}, \theta)\}$.

The most problematic operation to define in terms of operations over **DCon** is **Comb′**. The key to its definition is the following lemma [18]:

Lemma 5.1 If $\mathbf{F}$ and $\mathbf{F}'$ are closure operators, then

1. For all ordinals $\alpha < \beta$, $(\mathbf{F} \circ \mathbf{F}')^{\alpha} \leq (\mathbf{F} \circ \mathbf{F}')^{\beta} \leq (\mathbf{F} \bullet \mathbf{F}')$

2. For some ordinal α, $\mathbf{F} \bullet \mathbf{F}' = (\mathbf{F} \circ \mathbf{F}')^{\alpha}$.

This means that we can compute a closure by using "reexecution". The problem is that the closure descriptions δ need not be extensive. This means that we must consider the "cycle points" of a function. These are a straightforward generalization of the usual fix points. For this definition to work we make the requirement that the set of constraint descriptions over a fixed set of variables is finite. This implies that the number of induced closure descriptions over a fixed set of variables is also finite.

Let $\mathbf{F} : \mathbf{X} \to \mathbf{X}$ be a possibly non-monotonic function on a finite set $\mathbf{X}$. The *cycle points* of $\mathbf{F}$ for $\mathbf{x} \in \mathbf{X}$, denoted by $\mathbf{cycle\_points}(\mathbf{F}, \mathbf{x})$ is the set

$$\{\mathbf{F}^{\mathbf{i}}(\mathbf{x}) \mid \exists\, \mathbf{i}, \mathbf{j} > 0 \,.\, \mathbf{F}^{\mathbf{i}}(\mathbf{x}) = \mathbf{F}^{\mathbf{i}+\mathbf{j}}(\mathbf{x})\}.$$

Computing the cycle points of a function over a finite set is straightforward, just compute the sequence $\mathbf{x}$, $\mathbf{F}(\mathbf{x})$, $\mathbf{F}^2(\mathbf{x})$, ... until a repeated element occurs. The cycle points are exactly the elements between the first and second occurrence.

We can define $\mathbf{Comb}'$ in terms of $\mathbf{Comb}'_{\mathbf{T}}$ as follows:

$$\mathbf{Comb}'(\delta, \delta') = \lambda\, \mathbf{d}.\mathbf{Alub}'_{\mathbf{T}}\mathbf{cycle\_points}(\mathbf{C}, \mathbf{d})$$

where $\mathbf{C}(\mathbf{d}') = \mathbf{Comb}'_{\mathbf{T}}(\delta(\mathbf{d}'), \delta(\mathbf{d}'))$. The reason we had to introduce cycle points is that the function $\mathbf{C}$ is not necessarily monotonic. This arises because the ordering on constraint descriptions reflects the number of constraints they describe, not the logical ordering on the constraints.

The denotational equations given by providing definitions for the auxiliary functions 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 in which the same partial result is not repeatedly recomputed and only computed if it is needed for the final result. In the abstract interpretation of traditional Prolog programs to avoid redundant computations, the result of invoking atom $\mathbf{A}$ in the context of environment $\mathbf{e}$ is recorded. Such memoing can be implemented using function graphs. The function graph for a function $\mathbf{f}$ is the set of pairs $\{(\mathbf{e} \mapsto \mathbf{f}(\mathbf{e})) \mid \mathbf{e} \in \mathbf{dom}(\mathbf{f})\}$ where $\mathbf{dom}(\mathbf{f})$ denotes the domain for $\mathbf{f}$. 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 [15]. 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. In our context the application of this idea leads to two generic algorithms for the memoization based analysis of programs with dynamic scheduling which extend the usual memoization based analysis for traditional Prolog.

In the first, and simplest, algorithm 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 $\mathbf{A} :: \pi$ where $\mathbf{A}$ is an atom, and π a closure description restricted to the variables in $\mathbf{A}$. An answer to a call $\mathbf{A} :: \pi$ is of the form π' where π' is a closure description restricted to the variables in $\mathbf{A}$. To improve efficiency, the analysis needs only consider calls modulo variable renaming. Entries in the memoization table are "canonical" and really represent equivalence classes of calls and answers.

As an example consider the analysis of the path program and call from Section 2 using the abstract domain $\mathbf{Def}$. Closure function $\mathbf{f}$ is represented by a definite boolean formulae, $\mathbf{F}$, where $\mathbf{f}(\mathbf{d}) = \mathbf{F} \wedge \mathbf{d}$. The initial call $\mathbf{path}(\mathbf{X}, \mathbf{Y}) :: \mathbf{Y}$ computes an answer closure $\mathbf{X} \wedge \mathbf{Y}$ using the first rule. Evaluating the second rule sets up a call to $\mathbf{edge}(\mathbf{X}, \mathbf{Z}) :: \mathbf{true}$. The answer description of $\mathbf{edge}(\mathbf{X}, \mathbf{Z}) :: \mathbf{true}$ is calculated as $\mathbf{Z} \to \mathbf{X}$. The call to $\mathbf{p}(\mathbf{Z}, \mathbf{Y})$ uses the current answer $\mathbf{Z} \wedge \mathbf{Y}$ and computes new closure answer $\mathbf{X} \wedge \mathbf{Y}$ for $\mathbf{p}(\mathbf{X}, \mathbf{Y}) :: \mathbf{true}$. This is the same as the old answer so we have reached a fixpoint, the final memo table is

$$\begin{aligned} \mathbf{edge}(\mathbf{X}, \mathbf{Z}) :: \mathbf{true} &\mapsto \mathbf{Z} \to \mathbf{X} \\ \mathbf{path}(\mathbf{X}, \mathbf{Y}) :: \mathbf{Y} &\mapsto \mathbf{X} \wedge \mathbf{Y}. \end{aligned}$$

This first generic algorithm works quite well if it is not too expensive to compute the closure descriptions of delayed atoms. This is true if the closure description domain is very simple (as in the above example), or if only constraints can delay as in many CLP languages such as CLP($\mathcal{R}$). Unfortunately with this algorithm it is difficult to provide calling pattern information for delayed atoms.

In the case that the closure description domain is a complex function and we have arbitrary delay, then we can modify the algorithm to give our second generic algorithm as follows. The idea is that we also evaluate the closure descriptions π lazily. The problem is that computing if two closure descriptions are the same is an expensive test as we must compare their value over all elements of their domain. In the second generic algorithm two functions are assumed to be the same as long as their behavior is not different on the abstract constraints upon which the functions have been assessed. This means a function evaluation table is used to store function evaluations and determine if two functions are "observationally equivalent". It also requires handling the case where two functions are assumed equivalent and later found to differ. In this case computation based on the equivalence needs to be discarded. Note that if the analysis terminates with two descriptions being "observationally equivalent" then the results of the analysis do not depend on the functions they define actually being identical, but only that they are identical on the values computed, in other words they must be "observationally equivalent". An example of this algorithm's operation is given in the Appendix.

We note that as long as the underlying constraint description domain **DCon** only has a finite number of descriptions for a given finite set of variables then both of our generic algorithms will terminate. This is because there are only a finite number of functions that can be defined mapping **DCon** to **DCon** for a given finite set of variables. In the case of the second algorithm eventually two closure descriptions must define the same functions and hence be found to be "observationally equivalent".

6 Experimental Evaluation

Examination of the example analysis in the appendix, may suggest that the analysis method is too expensive to compute. The experimental evaluation presented in this section will show that our analysis method is practical for real programs.

For efficiency the implementation describes a closure description $\mathbf{f}'$ using a tuple $(\mathbf{d}, \mathbf{f})$ where $\mathbf{d}$ is a constraint description from **DCon** which gives the known constraint information at that program point and $\mathbf{f} : \mathbf{DCon} \rightarrow \mathbf{DCon}$ describes the effect of the delayed atoms. More exactly, $\mathbf{f}' = \lambda\, \mathbf{x}.\mathbf{f}(\mathbf{Comb}'_{\mathbf{T}}(\mathbf{d}, \mathbf{x}))$. This approach reduces the number of new functions needed to be defined and gives explicit information about the current constraint description at each program point which is exactly the information required to compute the calling patterns and answer patterns. The implementation ensures that the constraint description represents all the information engendered by the function as well. That is, for each description $(\mathbf{d}, \mathbf{f})$ appearing as a calling pattern or answer pattern $\mathbf{f}(\mathbf{d}) = \mathbf{d}$. The result of each function evaluation is stored in an *evaluation table*. Its purpose is two-fold, when the same evaluation is required its value is looked up in the table to avoid recomputation, and when two functions are being compared to see if they are "observationally equivalent", the calls of interest are found in the evaluation table.

Many programs either have no delay or the delay is restricted to some area of the computation. In these cases the closure description is of the form $(\mathbf{d}, \mathbf{f})$, where $\mathbf{f}$ is equivalent to the identity function on descriptions $\mathbf{id_{DCon}}$. Our framework can be easely modified to take advantage of this by treating the function $\mathbf{id_{DCon}}$ specially: it is always the initial function description for the goal, and until an atom possibly delays it can be used as the correct function description. We can also detect cases in which a description $(\mathbf{d}, \mathbf{f})$ only describes situations in which atoms which have delayed, have all subsequently been awoken. In these cases, the description can again be replaced by $(\mathbf{d}, \mathbf{id_{DCon}})$. Note that using this modification, we can never assume equivalence between $(\mathbf{d}, \mathbf{id_{DCon}})$ and any other closure description. As we will see in the experiments, this simple modification significantly improves the efficiency of the analysis.

Two different experiments have been performed. The first one evaluates the overhead introduced by

Program	**PLAI**	**MSets**	**Id**	**Hyb**			
				Ovh	Fun	Val	Eq
aiakl	998	1.14	1.02	9.23	71	202	23
ann	4403	1.09	1.01	9.20	678	2365	235
bid	213	1.32	1.17	4.05	96	127	30
boyer	2515	1.14	0.98	6.46	388	872	204
browse	98	1.28	1.11	3.94	53	58	19
deriv	66	1.32	1.14	4.91	27	24	20
fib	15	1.24	1.18	3.70	8	9	3
grammar	86	1.29	1.11	3.41	18	35	1
hanoiapp	28	1.26	1.16	5.15	16	17	6
mmatrix	36	1.34	1.10	4.05	18	21	6
occur	40	1.29	1.15	4.38	20	25	6
peephole	1900	1.20	1.02	6.25	496	924	248
progeom	109	1.24	1.18	4.37	52	62	16
qplan	988	1.37	1.02	5.55	416	514	162
qsortapp	40	1.38	1.10	4.90	24	27	9
query	124	1.40	1.31	2.18	12	18	1
rdtok	714	1.45	1.10	7.97	303	621	159
read	715	1.41	1.04	4.80	273	282	135
serialize	684	1.14	0.98	10.77	68	241	23
tak	17	1.58	1.32	6.10	12	12	5
warplan	2536	1.29	1.08	39.77	942	4202	414
zebra	235	1.24	0.99	2.79	34	63	9
		1.29	1.10	7.00			

Table 1: Efficiency Results

our method when analyzing (parts of) programs in which no dynamic scheduling occurs. To evaluate such overhead, we have compared four analysers which result from instantiating the following frameworks over the **Def** abstract domain: the PLAI framework [19] (**PLAI**), the framework presented in [17] (**MSets**), the second generic algorithm in which the closure descriptions are computed lazily (**Hyb**), and a modified version of this algorithm which treats $\mathbf{id_{DCon}}$ specially (**Id**). The reliability of the comparison is based on the fact that all analysers have been implemented by suitably modifying the PLAI framework and adding some domain dependent functions to the **Def** abstract domain, already implemented in PLAI.

We have selected a wide set of benchmarks which have been traditionally used in the evaluation of "static" analyzers[1] and therefore do not contain suspension declarations. The results of the evaluation is shown in Table 1. For each benchmark, the information shown is the following: analysis times[2] in milliseconds using **PLAI**, overhead introduced by **MSets**, **Id** and **Hyb**. These overheads are computed computed by dividing the analysis times by those obtained with PLAI. The remaining columns show, for **Hyb**, the number of functions created (Fun), the total number of function evaluations (Val), and the total number of equivalences assumed (Eq). **Id** does not create functions for these benchmarks. However, the numbers give an idea of the complexity using **Id** on similar sized programs that do contain delay. No accuracy results are presented since, as expected, the information inferred by all analysers is the same for "static" programs.

The results show that both **MSet** and **Id** involve a reasonable small overhead (30% and 10%, respectively), when analysing programs which do not delay. The performance of **Hyb** is clearly worse than the other

[1]A complete description of this benchmarks can be found, for example, in [2].
[2]SICStus 2.1, compactcode, SPARCstation 10, one processor.

Program	**MSets**		**Id**				**Hyb**			
	≤ 1	≤ 2	Time	Fun	Val	Eq	Time	Fun	Val	Eq
simple	45 †	70	72	4	11	0	148	17	35	5
path	226 †	378 †	61	6	7	1	93	16	21	4
append3	27		23	1	0	0	84	11	12	3
	181		84	4	6	1	254	24	39	10
nrev	28		26	1	0	0	89	12	13	4
	160		93	4	6	1	330	38	44	15
permute	32		32	1	0	0	109	12	14	4
	2601 †	46196 †	805	23	37	8	795	31	50	11
qsort	67		62	1	0	0	249	28	33	9
	5300 †	314117 †	1754	85	206	41	1476	119	174	58
mortgage	69		60	1	0	0	1189	66	91	19
	555 †	4024 †	482	28	34	17	770	45	64	13
fib	6884 †	439857 †	397	24	44	6	660	37	53	8
	5098 †	72945 †	832	56	105	26	1946	102	172	34

Table 2: Efficiency for programs with dynamic scheduling

methods.

The second experiment compares **MSets**, **Hyb** and **Id** when analyzing programs with dynamic scheduling. The first benchmark is that presented in the Appendix. The second is the `path` program analyzed for the example goal. The next four are those used in the evaluation of the framework presented in [17]. The final two are the well-known CLP programs `mortgage` and `fibbonacci`, modified so that arithmetic delays until it can be computed by local propagation. All these benchmarks have two obvious modes of operation forwards and backwards declarations, and thus the benchmarks have been analyzed for both kinds of queries. The information shown in Table 2 is similar to that shown in the previous table. We give two different analysis times for **MSets**, where the size of the multiset description is restricted to ≤ 1 and ≤ 2, respectively. When these analyses are identical, only the first time is given. Both **Id** and **Hyb** are completely accurate on these benchmarks. Examples where **MSets** loses accuracy are marked with a †.

Examining the results, it is clear that whenever no suspension occurs (the first call for `app3, nrev, permute, qsort` and `mortgage`) the analysis for all methods, even **Hyb**, is reasonable fast. **Id** is always faster than **MSets**, except for `simple` using multiset size ≤ 1. In this case the analysis perform by **MSets** is simpler because it loses all accuracy and reaches the fixpoint earlier. **Id** is almost always faster than **Hyb**, the exception being `qsort` in the case it delays. In this cases **Id** requires more function evaluations due to the restriction which prevents assuming equivalences with $\mathbf{Id}_{\mathbf{DCon}}$.

MSets suffers in comparison to the other methods when delay occurs, there are three reasons for this. First **MSets** must carry around larger descriptions, because they involve the variables of the delayed atoms, second **MSets** cannot perform lub operations when atoms are delayed and hence has to treat each answer separately. These problems lead to the explosion in analysis times when the size of the multiset is increased. Finally, when there are unbounded length chains of delayed atoms, as in `path`, the second goal for each of `permute`, `qsort`, `mortgage` and both goals for `fib`, it is forced to lose substantial information. In the experiment we have considered sizes of one and two, however in practice a larger bound would usually be required to capture a reasonable number of cases where the number of delayed atoms is bounded. In the benchmarks `simple` is the only case where the number of delayed atoms is bounded and greater than one. In this case greater accuracy was achieved by increasing the bound. We can conclude that **MSets** is not practical for non-trivial programs.

An interesting observation is that no invalid equivalence assumptions were made during the analysis of any benchmark. This is due to the fact that for an equivalence to be assumed we require the first component of the abstractions (the constraint descriptions) to be identical. As a result, a difference will only appear whenever re-evaluating the functions for a more concrete value than the associated constraint description produce different answers. This can only be due to the awakening of a delayed atom in one of the functions or, more rarely, due to the fact that we are using an underlying abstract domain where re-execution may sometimes improve accuracy.

7 Conclusion

We have given a generic framework for the analysis of constraint logic programming languages in which atoms and constraints can delay. An empirical evaluation of a groundness analyzer based on our framework has demonstrated the practicality of the framework. To our knowledge it is the first practical framework for analyzing logic programming languages with delay. Information given by analyses based on our framework promise to improve the execution of logic languages with delay by an order of magnitude [17].

Our framework is also useful for the analysis of concurrent constraint languages. The difference between these languages and the languages we have considered are that they use a committed choice non-determinism when choosing which rule to reduce an atom with, and they have no fixed underlying scheduling rule. However, one of the most promising methods for the implementation of concurrent languages is to compile them into logic languages with delay [9]. Our analysis method can, of course, be used to optimize the resulting code.

The main use of our analysis framework is to give information about calling patterns. For atoms which do not delay, even in the presence of delayed atoms which may be awoken, the framework gives very precise information. For atoms which have delayed, however, the information is less precise. This is because the delayed atoms are bundled into a single closure which is continuously being reevaluated. Thus for the delayed atoms, our framework has the imprecision inherent in any reexecution based analysis [18]. We are currently investigating methods to improve this.

Acknowledgements

We thank Manuel Hermenegildo and Harald Søndergaard who were involved in the initial discussions leading to this paper.

References

[1] T. Armstrong, K. Marriott, P. Schachte and H. Søndergaard. Boolean functions for dependency analysis: Algebraic properties and efficient representation. *Proc. Static Analysis Symposium, SAS'94*. B. Le Charlier (Ed.), Springer-Verlag, Vol. 864 in LNCS, pages 266–280, 1994.

[2] F. Bueno, M. García de la Banda, and M. Hermenegildo. Effectiveness of Global Analysis in Strict Independence-Based Automatic Program Parallelization. In *International Symposium on Logic Programming*. pages 320–338. MIT Press, Ithaca November 1994.

[3] M. Carlsson. Freeze, Indexing, and Other Implementation Issues in the Wam. In *Fourth International Conference on Logic Programming*, pages 40–58. University of Melbourne, MIT Press, May 1987.

[4] M. Codish, M. Falaschi, and K. Marriott. Suspension Analyses for Concurrent Logic Programs. *ACM Trans. Programming Languages and Systems* **16** (3): 649–686, 1994.

[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] S.K. Debray, D. Gudeman, and P. Bigot. Detection and Optimization of Suspension-free logic programs. In *International Logic Programming Symposium*, pages 487–504 MIT Press, Ithaca November 1994.

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

[10] 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.

[11] 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.

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

[13] M. Hermenegildo, K. Marriott, G. Puebla and P. Stuckey. Incremental Analysis of Logic Programs. To appear in *Proc. of 1995 International Conference on Logic Programming*, MIT Press, July 1995.

Technical Report CLIP 14/94.0, Computer Science Dept, Technical Univ. of Madrid (UPM), Spain, October 1994.

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

[15] 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.

[16] K. Marriott, M. Falaschi, M. Gabbrielli and C. Palamidessi. A Simple Semantics for Logic Programming Languages with Delay. *Eighteenth Australian Computer Science Conference*, Feb. 1995.

[17] K. Marriott, M. Garcia de la Banda and M. Hermenegildo. Analyzing Logic Programs with Dynamic Scheduling. *Proc. 21st ACM Symp. Principles of Programming Languages*, pages 240–253. ACM Press, 1994.

[18] K. Marriott and H. Søndergaard. Propagation and Reexecution Reexamined. Tech. Rpt. 93/8. Dept. of Comp. Science, Melbourne University, 1993.

[19] 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.

[20] 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.

[21] 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.

Appendix – A Worked Example

In this Appendix we give an example of the closure based analysis of a simple program. The description is a somewhat simplified version of what takes place in the implementation.

Consider the analysis of the following program for the goal $\mathbf{q(X, Y, Z)}$:

$$\texttt{q(X, Y, Z) :- }{}^{0}\texttt{p(X, Y, Z)}^{1}\texttt{, p(Z, X, Y)}^{2}\texttt{, X = 1}^{3}\texttt{.}$$
$$\texttt{?- p(X, Y, Z) when ground(X).}$$
$$\texttt{p(X, Y, Z) :- }{}^{4}\texttt{X = Z}^{5}\texttt{.}$$

This program has been chosen to illustrate the use of the evaluation table and how approximate closures capture delayed atoms. We consider a groundness analysis using the **Def** domain.

Since there are initially no delayed atoms the function defining the closure description is just $\mathbf{f}_0$ which is defined to be the identity function with domain the **Def** descriptions of the variables $\{\mathbf{X}, \mathbf{Y}, \mathbf{Z}\}$. The calling pattern to $\mathbf{q(X, Y, Z)}$ is therefore

$$\mathbf{q(X, Y, Z)} :: (\mathbf{true}, \mathbf{f}_0)$$

Ensuring that all **DCon** information is available requires computing $\mathbf{f}_0(\mathbf{true}) = \mathbf{true}$.

The description at point 0 is again $(\mathbf{true}, \mathbf{f}_0)$. The first call to $\mathbf{p(X, Y, Z)}$ has the same information. We need to decide if the call is possibly delayed or not. As **true** describes all constraints, we do not know if the call delays or not, so we must consider both cases and "lub" the resulting answers together.

First consider the case when the call $\mathbf{p(X, Y, Z)} :: (\mathbf{true}, \mathbf{f}_0)$ does not delay. In this case we can include the extra information that, because the call did not delay, $\mathbf{X}$ is ground in the function describing the calling pattern. Thus at point 4 we have the description $(\mathbf{X}, \mathbf{f}_0)$ which includes the information that $\mathbf{X}$ is ground. We need to evaluate $\mathbf{f}_0(\mathbf{X}) = \mathbf{X}$ to ensure the **Def** information in the new calling pattern is complete. At point 5 we derive description $(\mathbf{X} \wedge \mathbf{Z}, \mathbf{f}_0)$. Because we want answer information to be accurate, we must ensure that nothing was awoken by the additional information. Hence we evaluate $\mathbf{f}_0(\mathbf{X} \wedge \mathbf{Z}) = \mathbf{X} \wedge \mathbf{Z}$. We now have successfully computed an answer for $\mathbf{p}$ and we add an entry to the answer table

$$\mathbf{p(X, Y, Z)} :: (\mathbf{X}, \mathbf{f}_0) \mapsto (\mathbf{X} \wedge \mathbf{Z}, \mathbf{f}_0).$$

The second case is when the call $\mathbf{p(X, Y, Z)} :: (\mathbf{true}, \mathbf{f}_0)$ delays. For an arbitrary description domain the function, $\mathbf{f}_1$, describing the result of delaying $\mathbf{p(X, Y, Z)}$ is of the form:

$$\begin{aligned} \mathbf{f}_1(\mathbf{d}) \quad = \quad & \mathbf{if}\ (\mathbf{d} \rightarrow \mathbf{ground(X)})\ \mathbf{then}\ \mathbf{f_p}(\mathbf{d}) \\ & \mathbf{elseif}\ (\mathbf{d} \rightarrow \neg\mathbf{ground(X)})\ \mathbf{then}\ \mathbf{d} \\ & \mathbf{else}\ \mathbf{f_p}(\mathbf{d} \wedge \mathbf{ground(X)}) \sqcup (\mathbf{d} \wedge \mathbf{not\_ground(X)}) \end{aligned}$$

The function involves three cases: in the first the abstraction implies that $\mathbf{p(X, Y, Z)}$ is definitely not delayed. In this case the result is given by $\mathbf{f_p}(\mathbf{d})$ which is a function that is defined to be the result of calling $\mathbf{p}$ with description $(\mathbf{d}, \mathbf{Id}^{\mathbf{X,Y,Z}})$. The second case is when the atom definitely delays. For the **Def** abstract domain this information is never available so the case can be omitted, but it is useful in other domains such as definite freeness. The third case is where it cannot be determined by the abstraction $\mathbf{d}$ whether the atom wakes up or not, the result is the lub of the results for waking up (with added information from the wakeup) and delay (with added information from the delay). The above definition corresponds to that used by the implementation, which is parametric in domain. The definition of $\mathbf{f}_1$ specialized to **Def** we will use for the purposes of this explanation is:

$$\mathbf{f}_1(\mathbf{d}) \quad = \quad \mathbf{if}\ (\mathbf{d} \rightarrow \mathbf{X})\ \mathbf{then}\ \mathbf{f_p}(\mathbf{d})\ \mathbf{else}\ \mathbf{d}.$$

Function $\mathbf{f}_1$ represents $\mathbf{AtomClos(p(X, Y, Z))}$. The overall behavior of the call $\mathbf{p(X, Y, Z)} :: (\mathbf{true}, \mathbf{f}_0)$ is the lub of the above two possibilities – delay or not delay. We have to be careful since this lub must be over the functions that the descriptions represent. $\mathbf{f}_2(\mathbf{d}) = \mathbf{f}_1(\mathbf{d}) \sqcup (\mathbf{X} \wedge \mathbf{Z} \wedge \mathbf{f}_0(\mathbf{X} \wedge \mathbf{Z} \wedge \mathbf{d})$. Determining $\mathbf{f}_2(\mathbf{true}) = \mathbf{true}$ involves computing $\mathbf{f}_1(\mathbf{true})$ and hence looking up the already computed value for $\mathbf{f}_0(\mathbf{X} \wedge \mathbf{Z})$. This causes us to add another entry to the memo table:

$$\mathbf{p}(\mathbf{X}, \mathbf{Y}, \mathbf{Z}) :: (\mathbf{true}, \mathbf{f}_0) \mapsto (\mathbf{true}, \mathbf{f}_2).$$

Combining the result of the call to $\mathbf{p}(\mathbf{X}, \mathbf{Y}, \mathbf{Z}) :: \mathbf{f}_0$ with the description at point 0, we obtain a description $(\mathbf{true}, \mathbf{f}_3)$ at program point 1 where $\mathbf{f}_3$ is defined to be $\mathbf{Comb}'\ \mathbf{f}_0\ \mathbf{f}_2$ For the **Def** abstract domain $\mathbf{Comb}'_{\mathbf{T}}$ is just conjunction and the function $\mathbf{C}\ \mathbf{d} = \mathbf{Comb}'_{\mathbf{T}}\ (\mathbf{f}\ \mathbf{d})\ (\mathbf{f}'\ \mathbf{d})$ is monotonic. Thus $\mathbf{C}$ has a single cycle point which is its least fixpoint and so $\mathbf{f}_3$ is defined to be $\mathbf{lfp}(\lambda\,\mathbf{d}.\mathbf{f}_0(\mathbf{d}) \wedge \mathbf{f}_2(\mathbf{d}))$.

Determining the calling pattern for the second call to $\mathbf{p}$ we build a renaming version of $\mathbf{f}_3$, that is

$$\mathbf{f}_4(\mathbf{d}) = \rho_4 \mathbf{f}_3(\rho_4^{-1}\mathbf{d}) \qquad \rho_4 = \{\mathbf{Z} \mapsto \mathbf{X}, \mathbf{X} \mapsto \mathbf{Y}, \mathbf{Y} \mapsto \mathbf{Z}\}.$$

Note that in the actual implementation this is avoided, we include it here for ease of explanation. The calling pattern is therefore $(\mathbf{true}, \mathbf{f}_4)$. To ensure accuracy of the constraint description information we computed $\mathbf{f}_4(\mathbf{true}) = \mathbf{true}$, which requires computing $\mathbf{f}_3(\mathbf{true})$. In turn, this involves looking up $\mathbf{f}_2(\mathbf{true})$ and $\mathbf{f}_0(\mathbf{true})$. Since $\mathbf{f}_2(\mathbf{true}) = \mathbf{true}$ and $\mathbf{f}_0(\mathbf{true}) = \mathbf{true}$ the fixpoint is reached and the evaluation of $\mathbf{f}_3(\mathbf{true})$ halts.

Examining the new calling pattern $(\mathbf{true}, \mathbf{f}_4)$ we "conjecture" that $\mathbf{f}_0$ and $\mathbf{f}_4$ are equivalent functions in order to use the answer table entry. To test if the conjecture is valid, we check they are "observationally equivalent" for all evaluations so far made for $\mathbf{f}_0$ and $\mathbf{f}_4$, that is $\mathbf{true}$, $\mathbf{X}$ and $\mathbf{X} \wedge \mathbf{Z}$. They agree for $\mathbf{true}$, but we must calculate $\mathbf{f}_4(\mathbf{X})$ and $\mathbf{f}_4(\mathbf{X} \wedge \mathbf{Z})$. Computing $\mathbf{f}_4(\mathbf{X} \wedge \mathbf{Z}) = \mathbf{X} \wedge \mathbf{Z}$ involves computing $\mathbf{f}_3(\mathbf{Z} \wedge \mathbf{Y}), \mathbf{f}_0(\mathbf{Z} \wedge \mathbf{Y}), \mathbf{f}_2(\mathbf{Z} \wedge \mathbf{Y})$ and $\mathbf{f}_1(\mathbf{Z} \wedge \mathbf{Y})$. Similarly computing $\mathbf{f}_4(\mathbf{X}) = \mathbf{X}$ involves computing $\mathbf{f}_3(\mathbf{Z}), \mathbf{f}_0(\mathbf{Z}), \mathbf{f}_2(\mathbf{Z}), \mathbf{f}_1(\mathbf{Z})$. The answers agree with $\mathbf{f}_0(\mathbf{X} \wedge \mathbf{Z})$ and $\mathbf{f}_0(\mathbf{X})$. But computing them involved new function calls to $\mathbf{f}_0$. Therefore we must determine the values for $\mathbf{f}_4(\mathbf{Z} \wedge \mathbf{Y})$ and $\mathbf{f}_4(\mathbf{Z})$ to see if the "observational equivalence" continues to hold.

Computing $\mathbf{f}_4(\mathbf{Z} \wedge \mathbf{Y})$ involves computing $\mathbf{f}_3(\mathbf{X} \wedge \mathbf{Y}), \mathbf{f}_0(\mathbf{X} \wedge \mathbf{Y}), \mathbf{f}_2(\mathbf{X} \wedge \mathbf{Y})$ and $\mathbf{f}_1(\mathbf{X} \wedge \mathbf{Y})$ The last involves computing $\mathbf{f}_{\mathbf{p}}(\mathbf{X} \wedge \mathbf{Y})$. This invokes a call to $\mathbf{p}(\mathbf{X}, \mathbf{Y}, \mathbf{Z})$ with description $(\mathbf{X} \wedge \mathbf{Y}, \mathbf{f}_0)$. The resulting description at point 5, $(\mathbf{X} \wedge \mathbf{Y} \wedge \mathbf{Z}, \mathbf{f}_0)$, produces an answer table entry

$$\mathbf{p}(\mathbf{X}, \mathbf{Y}, \mathbf{Z}) :: (\mathbf{X} \wedge \mathbf{Y}, \mathbf{f}_0) \mapsto (\mathbf{X} \wedge \mathbf{Y} \wedge \mathbf{Z}, \mathbf{f}_0)$$

and involves calculating $\mathbf{f}_0(\mathbf{X} \wedge \mathbf{Y} \wedge \mathbf{Z})$ to ensure the most precise constraint description information is determined. The result is $\mathbf{f}_1(\mathbf{X} \wedge \mathbf{Y}) = \mathbf{X} \wedge \mathbf{Y} \wedge \mathbf{Z}$. In the continuing calculation of $\mathbf{f}_3(\mathbf{X} \wedge \mathbf{Y})$ we need to then evaluate $\mathbf{f}_2(\mathbf{X} \wedge \mathbf{Y} \wedge \mathbf{Z})$ and $\mathbf{f}_1(\mathbf{X} \wedge \mathbf{Y} \wedge \mathbf{Z})$ which again sets up a new call to $\mathbf{p}(\mathbf{X} \wedge \mathbf{Y} \wedge \mathbf{Z}, \mathbf{f}_0)$, resulting in in answer table entry

$$\mathbf{p}(\mathbf{X}, \mathbf{Y}, \mathbf{Z}) :: (\mathbf{X} \wedge \mathbf{Y} \wedge \mathbf{Z}, \mathbf{f}_0) \mapsto (\mathbf{X} \wedge \mathbf{Y} \wedge \mathbf{Z}, \mathbf{f}_0)$$

The final result $\mathbf{f}_4(\mathbf{Z} \wedge \mathbf{Y}) = \mathbf{X} \wedge \mathbf{Y} \wedge \mathbf{Z}$ does not agree with $\mathbf{f}_0$ hence the conjecture is wrong — they are not the same function.

Thus we must consider the call $\mathbf{p}(\mathbf{X}, \mathbf{Y}, \mathbf{Z}) :: (\mathbf{true}, \mathbf{f}_4)$ as a new call. The answer if $\mathbf{p}$ delays is just $\mathbf{f}_1$. If $\mathbf{p}$ wakes then the description at point 4 is $(\mathbf{X}, \mathbf{f}_4)$ and the answer is $(\mathbf{X} \wedge \mathbf{Z}, \mathbf{f}_4)$. Obtaining the answer involves looking up $\mathbf{f}_4(\mathbf{X})$ and $\mathbf{f}_4(\mathbf{X} \wedge \mathbf{Z})$ to ensure the calling pattern and answer pattern information about the constraint description is precise. Let $\mathbf{f}_6(\mathbf{d}) = \mathbf{f}_1(\mathbf{d}) \sqcup (\mathbf{X} \wedge \mathbf{Z} \wedge \mathbf{f}_4(\mathbf{X} \wedge \mathbf{Z} \wedge \mathbf{d}))$, computing $\mathbf{f}_6(\mathbf{true}) = \mathbf{true}$ involves looking up $\mathbf{f}_1(\mathbf{true})$ and $\mathbf{f}_4(\mathbf{X} \wedge \mathbf{Z})$. The new answer table entries are

$$\mathbf{p}(\mathbf{X}, \mathbf{Y}, \mathbf{Z}) :: (\mathbf{X}, \mathbf{f}_4) \mapsto (\mathbf{X} \wedge \mathbf{Z}, \mathbf{f}_4)$$
$$\mathbf{p}(\mathbf{X}, \mathbf{Y}, \mathbf{Z}) :: (\mathbf{true}, \mathbf{f}_4) \mapsto (\mathbf{true}, \mathbf{f}_6)$$

Renaming the result of the call to $\mathbf{p}(\mathbf{Z}, \mathbf{X}, \mathbf{Y})$ back to the original variables is achieved using the function

$$\mathbf{f}_7(\mathbf{d}) = \rho_4^{-1} \mathbf{f}_6(\rho_4 \mathbf{d}) \qquad \rho_4 = \{\mathbf{Z} \mapsto \mathbf{X}, \mathbf{X} \mapsto \mathbf{Y}, \mathbf{Y} \mapsto \mathbf{Z}\}$$

The result at point 2 is $(\mathbf{true}, \mathbf{f}_8)$ where function $\mathbf{f}_8 = \mathbf{lfp}(\lambda\, \mathbf{d}.\mathbf{f}_3(\mathbf{d}) \wedge \mathbf{f}_7(\mathbf{d}))$. Adding the constraint information we obtain description $(\mathbf{X}, \mathbf{f}_8)$ at point 3. In order to place an entry in the table for $\mathbf{q}(\mathbf{X}, \mathbf{Y}, \mathbf{Z})$ we need to determine $\mathbf{f}_8(\mathbf{X})$. This involves calculating $\mathbf{f}_3(\mathbf{X}), \mathbf{f}_2(\mathbf{X}), \mathbf{f}_1(\mathbf{X})$ and $\mathbf{f}_\mathbf{p}(\mathbf{X})$. The last looks up the answer table to find $\mathbf{f}_\mathbf{p}(\mathbf{X}) = \mathbf{X} \wedge \mathbf{Z}$. Continuing the computation of $\mathbf{f}_3(\mathbf{X})$ involves $\mathbf{f}_2(\mathbf{X} \wedge \mathbf{Z}), \mathbf{f}_1(\mathbf{X} \wedge \mathbf{Z})$ and $\mathbf{f}_\mathbf{p}(\mathbf{X} \wedge \mathbf{Z})$. This sets up a new call to $\mathbf{p}$ eventually giving rise to the answer table entry

$$\mathbf{p}(\mathbf{X}, \mathbf{Y}, \mathbf{Z}) :: (\mathbf{X} \wedge \mathbf{Z}, \mathbf{f}_0) \mapsto (\mathbf{X} \wedge \mathbf{Z}, \mathbf{f}_0)$$

Calculation of $\mathbf{f}_8(\mathbf{X})$ continues with $\mathbf{f}_7(\mathbf{X}), \mathbf{f}_6(\mathbf{Y}), \mathbf{f}_1(\mathbf{Y})$. The next iteration in the calculation of $\mathbf{f}_8(\mathbf{X})$ involves determining $\mathbf{f}_3(\mathbf{X} \wedge \mathbf{Z})$ and $\mathbf{f}_7(\mathbf{X} \wedge \mathbf{Z})$. Computing $\mathbf{f}_3(\mathbf{X} \wedge \mathbf{Z})$ involves looking up $\mathbf{f}_2(\mathbf{X} \wedge \mathbf{Z})$. $\mathbf{f}_7(\mathbf{X} \wedge \mathbf{Z})$ is calculated using $\mathbf{f}_6(\mathbf{X} \wedge \mathbf{Y}), \mathbf{f}_4(\mathbf{X} \wedge \mathbf{Y} \wedge \mathbf{Z}), \mathbf{f}_3(\mathbf{X} \wedge \mathbf{Y} \wedge \mathbf{Z})$ and looking up $\mathbf{f}_2(\mathbf{X} \wedge \mathbf{Y} \wedge \mathbf{Z})$. Calculation of the next iteration of $\mathbf{f}_8(\mathbf{X})$ continues with $\mathbf{f}_7(\mathbf{X} \wedge \mathbf{Y} \wedge \mathbf{Z})$ and $\mathbf{f}_6(\mathbf{X} \wedge \mathbf{Y} \wedge \mathbf{Z})$. The result is $(\mathbf{X} \wedge \mathbf{Y} \wedge \mathbf{Z}, \mathbf{f}_8)$ as expected. The final answer table entry is

$$\mathbf{q}(\mathbf{X}, \mathbf{Y}, \mathbf{Z}) :: (\mathbf{true}, \mathbf{f}_0) \mapsto (\mathbf{X} \wedge \mathbf{Y} \wedge \mathbf{Z}, \mathbf{f}_8)$$

The answer information is as accurate as possible with the **Def** domain, while we can determine the calling pattern information (that $\mathbf{p}(\mathbf{X}, \mathbf{Y}, \mathbf{Z})$ is called with $\mathbf{X}$ ground) from the memoization table. Using **Id** we would detect that $\mathbf{f}_8$ could be replaced by $\mathbf{Id}_{\mathbf{Def}}$.