Analysis of Constraint Logic Programs

Kim Marriott
IBM T. J. Watson Research Center,
P. O. Box 704, Yorktown Heights NY 10598, USA

Harald Søndergaard
Dept. of Computer Science,
University of Melbourne,
Parkville 3052 Vic., Australia

Abstract

Increasingly, compilers for logic programs are making use of information from dataflow analyses to aid in the generation of efficient target code. However, generating efficient target code from constraint logic programs is even more difficult than from logic programs. Thus we feel that good compilation of constraint logic programs will require sophisticated analysis of the source code. Here we present a simple framework for the abstract interpretation of constraint logic programs which is intended to serve as the basis for developing such analyses. We show the framework’s practical usefulness by sketching two dataflow analyses that are based on it and indicating how they can be used to improve the code generated by a compiler.

1 Introduction

Dataflow analysis is an essential component of many programming tools. Program such as “debuggers” and type checkers use dataflow analyses to identify errors in a program, while compilers and other program transformers use dataflow analyses to guide various optimizations. Accurate dataflow analyses are usually complex and so they are difficult to design and to prove correct. Fortunately, the theory of abstract interpretation provides some help.

Abstract interpretation formalizes dataflow analysis by viewing it as “approximate” computation, in which computation is performed with descriptions of data rather than the data themselves. P. and R. Cousot [3, 4] developed abstract interpretation as a precise framework for discussing approximate computation of imperative programs. The advantage of such a framework is that it serves as a basis for the better understanding of dataflow analyses, and for discussing their correctness. The work of P. and R. Cousot has been extended to functional and logic programming languages. Such extensions are not straightforward. For example, dataflow analysis of logic
programs differs markedly from the analysis of programs written in conventional languages because the dataflow is bi-directional, owing to unification, and the control flow is more complex, owing to backtracking.

Abstract interpretation of logic programs was first mentioned by Mellish [16] where it was suggested as a way to formalize mode analysis. Since then, abstract interpretation of logic programs has been an active research area and many general frameworks and applications have been given. Following Marriott and Søndergaard [13], the approaches can be broadly divided into two classes. The first, and most important class for logic program compilers, are top-down analyses (for example [19, 17, 5, 9, 2, 10, 15]). A top-down analysis approximates the SLD-refutation semantics and is intended to give information about “call-patterns”. The second class are the bottom-up analyses (for example [10, 11, 12, 1]). These are based on simpler $T_P$-like semantics and give information about “success-patterns”.

In this paper we generalize top-down abstract interpretation of logic programs to the wider class of constraint logic programs (CLP) [8]. A CLP program is a rule-based program in which constraints over some predefined domain are allowed in the rule bodies. Such a program is evaluated similarly to a logic program except that a test for constraint satisfaction replaces unification. Testing for constraint satisfaction may be very expensive, much more so than unification. Thus, it is important that tests for satisfaction be optimized wherever possible in the code generated by a CLP compiler. Thus we feel that any “good” CLP compiler will require non-trivial dataflow analysis of a program to determine when such optimizations are possible. This paper is intended to provide a framework for such analyses.

We now indicate by means of an example program how dataflow analysis of CLP programs can help their efficient implementation.

**Example 1.1** Consider the following CLP(R) [7] program to compute mortgage repayments with parameters: $x_p$, the principal; $x_t$, the life of the mortgage (in months); $x_i$, the annual percentage interest rate; $x_r$, the monthly repayment; and $x_b$, the outstanding balance.

\[
\text{mortgage}(x_p, x_t, x_i, x_r, x_b) \leftarrow \\
\quad x_t = 1 \land \tag{1} \\
\quad x_b + x_r = x_p \ast (1 + x_i/1200). \tag{2}
\]

\[
\text{mortgage}(x_p, x_t, x_i, x_r, x_b) \leftarrow \\
\quad x_t > 1 \land \\
\quad \text{mortgage}(x_p \ast (1 + x_i/1200) - x_r, x_t - 1, x_i, x_r, x_b). \tag{3}
\]

This program can be queried in many different ways. The query $Q_1 = \text{mortgage}(100000, 360, 12, x_r, 0)$ has answer constraint $x_r = 1028.61$, the query $Q_2 = \text{mortgage}(x_p, 360, 12, 1028.61, 0)$ has answer constraint $x_p = 100000$, and the query $Q_3 = \text{mortgage}(x_p, 120, 12, x_r, x_b)$ has answer constraint $x_p = 0.303 \ast x_b + 69.70 \ast x_r$. In all of these queries the efficiency of the
program execution may be improved by taking the form of the query into account.

For example, a “determinacy” analysis akin to that given by Debray and Warren [6] shows that whenever the second argument, \( x_t \), is “ground”, no backtracking need take place after 1. This information allows a compiler to generate code in which a redundant choicepoint is not constructed for the first clause. This is true for each of the queries given above. A “freeness” analysis akin to that given by Mellish [16] shows that if the last argument, \( x_b \), is not constrained in the query then it is not constrained in subsequent calls to mortgage. In such a case adding the constraint 2 can never lead to unsatisfiability. This information allows a compiler to generate code in which an unnecessary test for satisfiability is not made when executing the first clause. This is true for query \( Q_1 \). A “groundness” or “definiteness” analysis akin to that given by Mellish [16] shows that if the first, third and fourth arguments, \( x_p \), \( x_i \) and \( x_r \), are uniquely constrained in the query then this holds true in subsequent calls to mortgage. This means that the constraints in 3 can be optimized to a sequence of assignments. This is true for query \( Q_2 \).

In the remainder of this paper we present a framework for the development of such dataflow analyses. In Section 2 abstract interpretation is introduced. In Section 3 we refine the operational semantics of constraint logic programs to give a semantics which is suitable for dataflow analysis. In Section 4 we sketch two example dataflow analyses; the first is for “freeness”, the second for “definiteness”. Finally we summarize the main results of the paper in Section 5. Notation used in this paper is explained in Section 2 of an accompanying paper [14].

2 Abstract Interpretation

Abstract interpretation formalizes the idea of “approximate computation” in which computation is performed with descriptions of data rather than the data itself. Approximate computation is well-known from everyday use, for example the rules of signs, such as “plus times minus yields minus.” Approximate computation is formalized as evaluating a formula or a program, not over its standard domain, but by evaluating it over a non-standard domain of descriptions. Of course, when performing approximate computations, one must reinterpret all operators so as to apply to descriptions rather than to proper values.

Abstract interpretation is powerful because it is semantics-based and thus concerned with a programming language as a whole rather than the analysis of particular programs. But, as Nielson [18] observed, the generality of abstract interpretation can be taken further. Assume we are given a language in which one can express the semantics of a wide variety of programming languages. The theory of abstract interpretation may then be developed
in the framework of this *meta-language* once and for all. In this way, we
need not “reinvent” abstract interpretation for all different programming
languages—each is just a special instance of the general theory.

This is the approach we take, and the semantic definitions given here are
expressed in a meta-language given by Marriott, Søndergaard and Jones [15].
This meta-language is specifically intended for the definition of the seman-
tics of “logic programming-like” languages and is based on the formalism of
denotational semantics. The usefulness and suitability of the meta-language
should be apparent from the following sections: both standard and non-
standard semantics are easily presented in the meta-language. By choosing
the right level of abstraction, they can be made highly congruent: if the
standard semantics employs a certain operator on the standard domain, the
non-standard semantics should use a very similar operator on the corre-
sponding non-standard domain. We now summarize the main features of
the meta-language and the associated abstract interpretation theory.

The meta-language is one of typed lambda expressions, and the types
are given by

\[
T \in \text{Type} ::= S \mid L
\]
\[
L \in \text{Lat} ::= D \mid T \to L
\]

where \( S \in \text{Stat} \), \( D \in \text{Dyn} \), \( \text{Stat} \) is a collection of static types, and \( \text{Dyn} \) is a
collection of dynamic types. The difference between the two kinds is that (the
interpretation of) a static type remains the same throughout all (standard
and non-standard) semantics, whereas a dynamic type may change. We call
\( \text{Stat} \cup \text{Dyn} \) the collection of base types, and \( \text{Lat} \) is the collection of lattice
types. The syntax of the meta-language is given by

\[
e ::= c_i \quad \text{(base functions)}
\]
\[
| x_i \quad \text{(variables)}
\]
\[
| \lambda x. e \quad \text{(lambda abstraction)}
\]
\[
| e \ e' \quad \text{(function application)}
\]
\[
| \text{lfp} \ e \quad \text{(least fixed point operation)}
\]
\[
| \bigcup_{x \in e} e' \quad \text{(least upper bound operation)}
\]

Static types are interpreted as posets (ordered by identity) and dynamic
types are interpreted as complete lattices. A *type interpretation* \( I \) thus
assigns a structure \( (I E) \) to each base type \( E \). Type interpretations can be
naturally extended from the base types to all types. An *interpretation* \( I \)
denotes a type interpretation (by an abuse of notation also called \( I \)) together
with an assignment of an element of \( (I T) \) to each base function \( c \) of type \( T \).
By natural extension this gives the semantics of the meta-language.

By varying the interpretation, one may obtain different semantics from
the same set of semantic equations. The *standard interpretation* gives the
usual input/output behaviour of the program while dataflow analyses may
be expressed as *non-standard* interpretations. The role of abstract inter-
pretation is to give relationships between the standard and non-standard
interpretations which guarantee that the analysis safely approximates the standard semantics.

To do this, one must first formalize what the descriptions in the non-standard semantics mean in terms of the elements in the standard semantics. Following P. and R. Cousot [3, 4] this is done by means of concretization functions. Intuitively, a concretization function maps a description to the largest object which it describes.

**Definition.** A concretization function $\gamma$ from $A$ to $E$ is a monotonic function from $A$ to $E$ such that there is a (monotonic) adjoint function $\hat{\gamma}$ from $E$ to $A$. A concretization family, $\Gamma = \{\Gamma_D\}$, from interpretation $I$ to interpretation $I'$ is an indexed family of functions such that for each $D \in \text{Dyn}$, $\Gamma_D$ is a concretization function from $(I_D)$ to $(I'_D)$. ■

Descriptions are ordered according to how large a set of objects they apply to; intuitively, the more imprecise, the “higher” they sit in the ordering. In accordance with this, we now define what it means for a description (or function) to “safely approximate” an object (or function).

**Definition.** Let $\Gamma$ be a concretization family from $I$ to $I'$. Define the relation (safely) approximates, written $\text{appr}[\Gamma]$, by

\[
\begin{align*}
    u \text{ appr}[\Gamma] S v & \iff v = u \text{ where } S \in \text{Stat} \\
    u \text{ appr}[\Gamma] D v & \iff v \leq \gamma_D u \text{ where } D \in \text{Dyn} \\
    u \text{ appr}[\Gamma] T \rightarrow T' v & \iff \forall u', v'. u' \text{ appr}[\Gamma] T v' \Rightarrow (u u') \text{ appr}[\Gamma] T' (v v'). \phantom{\text{appr}[\Gamma]} ■
\end{align*}
\]

When the concretization family $\Gamma$ and type $T$ is clear from the context we write $\text{appr}[\Gamma]_T$ as $\text{appr}$.

**Example 2.1** Assume that the standard interpretation of a dynamic type $T$ is $E = \mathcal{P} \text{Sub}$ where $\text{Sub}$ is the set of substitutions. In a non-standard interpretation we can interpret $T$ as the set $A = \mathcal{P} \text{Var}$ ordered with the superset relation. Intuitively $U \in A$ describes $\Theta \in E$ iff each $\theta \in \Theta$ grounds all variables in $U$. This intuitive meaning is captured by the concretization function $\gamma$ from $A$ to $E$ defined by

\[
\gamma U = \{\theta \in \text{Sub} \mid \forall V \in U. \theta \text{ grounds } V\}.
\]

Let $\theta_1 = \{x \mapsto a, y \mapsto z\}$ and $\theta_2 = \{x \mapsto f(a), y \mapsto a\}$. The set $\{\theta_1, \theta_2\}$ is approximated by the description $\{x\}$. Let $\text{restrict} : \mathcal{P} \text{Var} \rightarrow E \rightarrow E$ be the function such that $\text{restrict} U \Theta$ is the set of substitutions in $\Theta$ restricted to the variables in $U$. Then $\text{restrict}$ is approximated by the function $\text{restrict}' : \mathcal{P} \text{Var} \rightarrow A \rightarrow A$ defined by $\text{restrict}' U U' = U \cap U'$. ■

The following fundamental result allows us to argue inductively the correctness of a whole dataflow analysis (or non-standard semantics) once certain
base functions are shown to be in the relation safely approximates. This applies not only to constraint logic programming languages as discussed in this paper, but to any language whose semantics can be expressed in the meta-language.

**Proposition 2.1** Let $I$ and $I'$ be interpretations and let $e$ be a closed expression. If $(I c) \text{ appr } (I' c)$ holds for every base function $c$, then, $(I e) \text{ appr } (I' e)$.

The following result is also important as it allows the stepwise development of approximations and proof of approximation.

**Proposition 2.2** Let $I$, $I'$ and $I''$ be interpretations and let $e$ be a closed expression. Then

$$(I e) \text{ appr } (I' e) \land (I' e) \text{ appr } (I'' e) \Rightarrow (I e) \text{ appr } (I'' e).$$

3 Operational Semantics of CLP

In this section we first give an operational semantics which captures the essence of a CLP interpreter’s evaluation of a constraint logic program. We then refine this semantics until we obtain a semantics which is a suitable basis for the abstract interpretation of CLP programs.

First some notation. A constraint logic program, or program, is a finite set of clauses. A clause is of the form $H \leftarrow \pi \land B$ where $H$, the head, is an atom, $\pi$ is a constraint and $B$, the body, is a finite sequence of atoms. Note that $\pi$ may be a conjunction of primitive constraints or the vacuous constraint $true$. We let $\text{Var}$ denote the set of variables, $\text{Atom}$ the set of atoms, $\text{Con}$ the set of constraints, $\text{Clause}$ the set of clauses, and $\text{Prog}$ the set of CLP programs. $\text{Con}$ is assumed to contain $true$ and to be closed under conjunction and existential quantification.

In the semantic definitions, finite failure is not distinguished from non-termination. For example, the empty program and the program consisting only of the clause $q \leftarrow q$ have the same denotation. This reflects the fact that the statements we want to produce from a dataflow analysis are of the form “whenever execution reaches this point, so and so holds.” But in saying so, we do not actually say that execution does reach the point. In particular, “whenever the computation terminates, so and so holds” concludes nothing about termination. As non-termination is not distinguished from finite failure, we can assume a parallel search rule, rather than the customary depth-first rule—this simplifies our task. Owing to the use of a parallel search rule, the execution of a program naturally yields a set of answer constraints, as opposed to a sequence. However, note that the usual left-to-right computation rule is assumed.

The semantic definitions make use of renamings. A renaming is a mapping $\rho \in \text{Ren} \subset \text{Var} \rightarrow \text{Var}$ such that $\rho \circ \rho$ is the identity. Note that this
is not the standard definition of renaming, but it simplifies the following presentation. A renaming is not distinguished from its natural extension to atoms, clauses, constraints or sets of constraints. We make use of the function $ren : \mathcal{P}\text{Var} \rightarrow \mathcal{P}\text{Var} \rightarrow \text{Ren}$ for renaming where $(ren U W)$ is some renaming, such that the variables in $W$ are renamed away from those in $U$. For instance, $ren \{x, y\} \{x\}$ might be $\{x \mapsto z, z \mapsto x\}$.

The definitions also make use of the functions $\text{add} : \text{Con} \rightarrow \mathcal{P}\text{Con} \rightarrow \mathcal{P}\text{Con}$ which adds a constraint to a set of constraints and $\text{exist} : \mathcal{P}\text{Var} \rightarrow \mathcal{P}\text{Con} \rightarrow \mathcal{P}\text{Con}$ which existentially quantifies a set of constraints. They are defined by:

\[
\begin{align*}
\text{add } \pi \Pi & = \{\pi \land \pi' \mid \pi' \in \Pi\} \setminus \{\text{false}\} \\
\text{exist } W \Pi & = \{\exists_W \pi \mid \pi \in \Pi\}.
\end{align*}
\]

where $\exists_W \pi$ is the existential closure of $\pi$ for all variables not in $W$.

We now give a definition of the operational semantics of CLP. It should be clear that this definition is equivalent to the usual operational semantics [8]. The main differences to the standard definition are: it is explicit that the operational semantics is a fixpoint definition; and the semantics operates over sets of constraints rather than single constraints. Note that the test $\Pi' = \emptyset$ corresponds to the usual test for (un)satisfiability. We let $::$ denote concatenation of sequences.Clauses are implicitly universally quantified, so each call of a clause should produce new incarnations of its variables. This is done by means of the function $ren$, so that the generated clause instance will contain no variables previously met during execution (those in $\Pi$) or to be met later (those in $A : G$). The function $\text{vars}$ takes a syntactic object and returns the set of variables occurring in it.

**Definition.** The *operational semantics* has semantic domains

\[
\text{Sem} = \text{Atom}^* \rightarrow \mathcal{P}\text{Con} \rightarrow \mathcal{P}\text{Con}
\]

and semantic functions

\[
\begin{align*}
O : \text{Prog} \rightarrow \text{Sem} \\
O' : \text{Prog} \rightarrow \text{Sem} \rightarrow \text{Sem}.
\end{align*}
\]

The semantic equations are

\[
\begin{align*}
O [P] & = \text{lfp} (O' [P]) \\
O' [P] S \text{ nil } \Pi & = \Pi \\
O' [P] S (A : G) \Pi & = \\
& \quad \text{let } U = (\text{vars } \Pi) \cup (\text{vars } (A : G)) \text{ in} \\
& \quad \text{let } [H \leftarrow \pi \land B] = \text{ren } U \text{ (vars } C) \text{ } C \text{ in} \\
& \quad \bigcup_{C \in \mathcal{P}} \text{ let } \Pi' = \text{add } (\pi \land A = H) \Pi \text{ in} \\
& \quad \text{if } \Pi' = \emptyset \text{ then } \emptyset \text{ else } \text{exist } U \text{ (S } (B :: G) \text{ } \Pi').
\end{align*}
\]
Example 3.1 Consider the program $P$ from Example 1.1 and the query $Q_3 = \text{mortgage}(x_p, 120, 12, x_r, x_b)$. We have that $O[P] Q_3$ is $\{x_p = 0.303* x_b + 69.70* x_r\}$. ■

Actually the operational semantics does not really give the type of information a compiler requires for the generation of efficient code. This is because we are primarily interested in removing unnecessary tests for satisfiability and unnecessary choicepoints. Therefore, we need information about the constraints at call time. It is straightforward to modify the operational semantics so that it returns the “call constraints” rather than the “answer constraints”. However, to aid clarity and brevity we will ignore this modification and only consider the operational semantics. It should be remembered though, that correctness of an analysis which gives information about call constraints can only be proved with respect to the “call semantics”.

There is still another problem with using the operational semantics as the basis for dataflow analysis. The problem is that the meaning of a goal may depend on the meaning of an infinite number of other goals; thus termination of analyses based on the operational semantics cannot easily be guaranteed. For instance, assume we are interested in determining which variables are bound to ground terms. As descriptions we choose sets $U$ of variables, with the intention that $V \in U$ means that the current constraint definitely grounds $V$. Now consider the following program.

\[
\text{ancestor}(x, z) \leftarrow \text{parent}(x, z).
\]
\[
\text{ancestor}(x, z) \leftarrow \text{ancestor}(x, y), \text{ancestor}(y, z).
\]

Assume we are given the query $\leftarrow \text{ancestor}(a, z)$. An analysis based on the operational semantics must compute an infinite number of query/description tuples, namely (among others)

\[
\langle \text{ancestor}(x, z), \{x\} \rangle
\]
\[
\langle \text{ancestor}(x, y) \land \text{ancestor}(y, z), \{x\} \rangle
\]
\[
\langle \text{ancestor}(x, w) \land \text{ancestor}(w, y) \land \text{ancestor}(y, z), \{x\} \rangle
\]
\[
\vdots
\]

Clearly such a dataflow analysis will not terminate. What we need is a semantic definition that somehow “merges” information about all incarnations of a variable that appears in a program and is compositional in the sense that the meaning of a goal is given in terms of the meaning of its subgoals.

We now develop such a definition in two stages. The first stage is obtained by observing that

\[
O[P] \ G \ \Pi = \rho \ (O[P] (\rho \ G) \ (\rho \ \Pi))
\]

and that

\[
O[P] (A : G) \ \Pi = O[P] \ G (O[P] \ A \ \Pi).
\]
Equation (4) means that we can modify the definition so that the calling atom and constraint set are renamed away from the clause rather than the reverse. Equation (5) means that we can rewrite the definition so that the denotation of a goal or clause body is given in terms of the denotation of the constituent atoms. This means that the denotation of a program need only be a mapping from atoms and constraint sets to the answer constraints rather than a mapping from goals and constraint sets.

**Definition.** The *standard semantics* has semantic domain

\[ \text{Den} = \text{Atom} \rightarrow \mathcal{P} \text{Con} \rightarrow \mathcal{P} \text{Con} \]

and semantic functions

\[
\begin{align*}
P_{\text{std}} & : \text{Prog} \rightarrow \text{Den} \\
C_{\text{std}} & : \text{Clause} \rightarrow \text{Den} \rightarrow \text{Den}.
\end{align*}
\]

The semantic equations are

\[
\begin{align*}
P_{\text{std}} [P] &= \text{lfp} \left( \bigsqcup_{C \in P} C_{\text{std}} [\llbracket C \rrbracket] \right) \\
C_{\text{std}} [H \leftarrow \pi \land B] D A \Pi &= \\
&\quad \text{let } \rho = \text{ren} (\text{vars } H \cup \text{vars } B \cup \text{vars } \pi) (\text{vars } A \cup \text{vars } \Pi) \text{ in} \\
&\quad \text{let } W = \text{vars } (\rho A) \cup \text{vars } (\rho \Pi) \text{ in} \\
&\quad \rho (\text{exist } W (\Sigma D B (\text{add } (\pi \land H = \rho A) (\rho \Pi))))
\end{align*}
\]

where \( \Sigma F \) is the *folded* version of a function \( F \) defined by

\[
\Sigma F \text{ nil } z = z \Sigma F \text{ (x : y) } z = \Sigma F \text{ y } (F \text{ x } z).
\]

**Proposition 3.1** For every program \( P \), \( O \llbracket P \rrbracket = \Sigma (P_{\text{std}} \llbracket P \rrbracket) \).

We now tailor this definition even closer to the sort of dataflow analyses we are interested in. The motivation for the next change is that in a dataflow analysis we wish to limit the number of atom and abstract constraint pairs whose denotation will need to be computed for a particular abstract query. The idea is to modify the definition so that only atom and constraint set pairs in which the constraint set is restricted to the variables in the atom need be considered. Justification for this modification comes from the observation that:

\[
O \llbracket P \rrbracket G \Pi \subseteq \text{meet} \Pi (O \llbracket P \rrbracket G (\text{exist } (\text{vars } G) \Pi))
\]

where the function \( \text{meet} : \mathcal{P} \text{Con} \rightarrow \mathcal{P} \text{Con} \rightarrow \mathcal{P} \text{Con} \) is defined by

\[
\text{meet} \Pi \Pi' = \{ \pi \land \pi' \mid \pi \in \Pi \land \pi' \in \Pi' \} \setminus \{ \text{false} \}.
\]
**Definition.** The *lax semantics* has semantic domain

\[ Den = \text{Atom} \rightarrow \mathcal{P} \text{Con} \rightarrow \mathcal{P} \text{Con} \]

and semantic functions

\[
\begin{align*}
P_{\text{lax}} &: \text{Prog} \rightarrow Den \\
C_{\text{lax}} &: \text{Clause} \rightarrow Den \rightarrow Den.
\end{align*}
\]

The semantic equations are

\[
\begin{align*}
P_{\text{lax}} [P] &= \text{lfp} \left( \bigsqcup_{C \in P} C_{\text{lax}} [C] \right) \\
C_{\text{lax}} [H \leftarrow \pi \land B] D \ A \ \Pi &= \text{ret} A \ \Pi \ H \ (\Sigma \ D \ B \ (\text{call} \ A \ \Pi \ H \ \pi))
\end{align*}
\]

where

\[
\begin{align*}
call &= \text{Atom} \rightarrow \mathcal{P} \text{Con} \rightarrow \text{Atom} \rightarrow \text{Con} \rightarrow \mathcal{P} \text{Con} \\
\text{ret} &= \text{Atom} \rightarrow \mathcal{P} \text{Con} \rightarrow \text{Atom} \rightarrow \mathcal{P} \text{Con} \rightarrow \mathcal{P} \text{Con}
\end{align*}
\]

are defined by

\[
\begin{align*}
call A \ \Pi \ H \ \pi &= \text{ret} H \ \{\pi\} \ A \ \Pi \\
\text{ret} A \ \Pi \ H \ \Pi' &= \\
&\quad \text{let } \rho = \text{ren} (\text{vars} A \cup \text{vars} \ \Pi) \ (\text{vars} H \cup \text{vars} \ \Pi') \text{ in} \\
&\quad \text{meet} \ \Pi \ (\text{exist} (\text{vars} A) \ (\text{add} (A = \rho \ H) \ (\rho \ \Pi'))) \quad \blacksquare
\end{align*}
\]

**Proposition 3.2** Letting the concretization function be the identity function, \( P_{\text{lax}} \) appr \( P_{\text{std}} \). \( \blacksquare \)

We now turn to dataflow analyses. To extract runtime properties of pure CLP programs one can develop a variety of non-standard interpretations of the preceding semantics. To clarify the presentation, we extract from the lax semantics a *dataflow* semantics which contains exactly those features that are common to all the non-standard interpretations that we want. It leaves the interpretation of one dynamic type, \( A_{\text{con}} \), and two base functions, \( c \) and \( r \), unspecified. These missing details of the dataflow semantics are to be filled in by the interpretations. In the standard interpretation of this semantics, \( I_{\text{lax}} \), \( A_{\text{con}} \) is \( \mathcal{P} \text{Con} \), \( c \) is *call* and \( r \) is *ret*. In a non-standard interpretation, \( A_{\text{con}} \) is assigned whatever “descriptions” we choose in approximating sets of constraints. \( A_{\text{con}} \) should thus be a complete lattice which corresponds to \( \mathcal{P} \text{Con} \) in the standard semantics in a way laid down by some concretization function from \( A_{\text{con}} \) to \( \mathcal{P} \text{Con} \). The interpretation of \( c \) and \( r \) should approximate *call* and *ret* respectively. The types *Prog*, *Clause*, and *Atom* are static and have the obvious fixed interpretation.
Definition. The dataflow semantics has dynamic type Acon and semantic domain

\[ \text{Den} = \text{Atom} \to \text{Acon} \to \text{Acon}, \]

semantic functions

\[
\begin{align*}
P &: \text{Prog} \to \text{Den} \\
C &: \text{Clause} \to \text{Den} \to \text{Den}
\end{align*}
\]

and base functions

\[
\begin{align*}
c &: \text{Atom} \to \text{Acon} \to \text{Atom} \to \text{Con} \to \text{Acon} \\
r &: \text{Atom} \to \text{Acon} \to \text{Atom} \to \text{Acon} \to \text{Acon}.
\end{align*}
\]

The semantic equations are

\[
\begin{align*}
P[\mathcal{P}] &= \text{lfp} \left( \bigsqcup_{\mathcal{C} \in \mathcal{P}} \mathcal{C} \left[ \mathcal{C} \right] \right) \\
C \left[ \mathcal{H} \leftarrow \pi \land \mathcal{B} \right] \ D \ A \ II &= r \ A \ II \ H \left( \Sigma \ D \ B \left( c \ A \ II \ H \pi \right) \right).
\end{align*}
\]

For convenience we may specify an interpretation \( I_Y \) of the dataflow semantics by the triple \( \langle I_Y \text{Acon}, I_Y c, I_Y r \rangle \) and denote \( I_Y [P] \) by \( P_Y \). For example, \( I_{\text{lax}} \) is given by \( \langle P \text{Con}, \text{call}, \text{ret} \rangle \) and the corresponding semantics is denoted by \( P_{\text{lax}} \).

Since the interpretations of \( c \) and \( r \) are required to be monotonic functions we have:

**Proposition 3.3** For every interpretation \( I_X \), \( P_X \) is well-defined.

**Definition.** Let \( I = \langle \text{Acon}, c, r \rangle \) and \( I' = \langle \text{Acon}', c', r' \rangle \) be interpretations. We say \( I' \) approximates \( I \) iff for some concretization function from \( \text{Acon}' \) to \( \text{Acon} \), \( c' \) appr \( c \) and \( r' \) appr \( r \).

It follows immediately from Proposition 2.1 that:

**Proposition 3.4** If \( I_X \) approximates \( I_Y \), \( P_X \) appr \( P_Y \).

And then from Proposition 2.2 and Proposition 3.2 it follows that:

**Theorem 3.5** If \( I_X \) approximates \( I_{\text{lax}} \), \( P_X \) appr \( P_{\text{std}} \).

Developing a dataflow analysis in this framework is therefore a matter of choosing the description domain \( \text{Acon} \) so that it captures the information required from the analysis and defining suitable functions to approximate \( \text{call} \) and \( \text{ret} \). We give two examples of this in the next section.
4 Example Dataflow Analyses

In this section we show that the dataflow semantics given in the last section is a suitable basis for the analysis of CLP programs by sketching two example dataflow analyses based on it. The first is a freeness analysis, the second is a definiteness analysis.

4.1 Freeness Analysis

In the freeness analysis, descriptions are sets of variables which are read as being “possibly” constrained. The complement of such a variable set are those variables which are “definitely” not constrained, that is, they are free.

**Definition.** Let $A_{\text{con}} \rightarrow \mathcal{P} \ Var$ with subset ordering and define $A_{\text{free}} : A_{\text{con}} \rightarrow \mathcal{P} \ \mathcal{C}$ by

$$\gamma_{\text{free}} W = \{ \pi \in \mathcal{C} \mid \forall V \notin W. V \ \text{free} \ \text{in} \ \pi \}$$

where $V \ \text{free} \ \text{in} \ \pi$ iff $\pi$ does not constrain $V$, that is, there is a constraint $\pi'$ which is equivalent to $\pi$ and $V \notin (\text{vars} \ \pi')$.

**Lemma 4.1** The function $\gamma_{\text{free}}$ is a concretization function from $A_{\text{con}}$ to $A_{\text{free}}$.

We can approximate $\text{call}$ and $\text{ret}$ as follows.

**Definition.** Define $\text{call}_{\text{free}} : \text{Atom} \rightarrow A_{\text{con}} \rightarrow \text{Atom} \rightarrow \mathcal{C} \rightarrow A_{\text{con}}$ and $\text{ret}_{\text{free}} : \text{Atom} \rightarrow A_{\text{con}} \rightarrow \text{Atom} \rightarrow A_{\text{con}} \rightarrow A_{\text{con}}$ by

$$\text{call}_{\text{free}} A W H \pi = \text{ret}_{\text{free}} H (\text{vars} \ \pi) A W$$

$$\text{ret}_{\text{free}} A W H W' =$$

let $\rho = \text{ren} (\text{vars} \ A \cup W) (\text{vars} \ H \cup W')$ in

$W \cup \{ V \in (\text{vars} \ A) \mid \neg (\exists U \subseteq \text{Var. alias} \ U \ (A = \rho H) \land U \cap (\rho W') = \land U \cap (\text{vars} \ A) = \{ V \}) \}$

where $\text{alias} \ U \ \pi$ is true iff the only constraint $\pi$ places on the variables in $U$ is that they are aliases.

The reason for this definition of $\text{ret}_{\text{free}} A W H W'$ is that $V$ is $\text{not}$ possibly constrained, that is, free, if:

- $(A = \rho H)$ only constrains $V$ to be aliased to other variables, that is, for some $U \subseteq \text{Var}$, $\text{alias} \ U \ (A = \rho H)$ and $V \in U$ are true;
- $V$ is not aliased to any possibly constrained variable, that is, $U \cap (\rho W') = \emptyset$; and
• $V$ is not aliased to any other variables in $(\text{vars } A)$, that is, $U \cap (\text{vars } A) = \{V\}$.

**Lemma 4.2** The following hold:

• $\text{call}_{\text{free}} \text{ appr call}$, and

• $\text{ret}_{\text{free}} \text{ appr ret}$. ■

**Definition.** Let $I_{\text{free}}$ be the interpretation given by $\langle A_{\text{con}}_{\text{free}}, \text{call}_{\text{free}}, \text{ret}_{\text{free}} \rangle$.

It follows from Lemma 4.2 that $I_{\text{free}}$ approximates $I_{\text{lax}}$. Thus from Theorem 3.5:

**Theorem 4.3** $P_{\text{free}} \text{ appr } P_{\text{std}}$. ■

The call version of the freeness analysis can be used to determine that the query description

$$\text{mortgage}(x_p, x_t, x_i, x_r, x_b) \{x_p, x_t, x_i, x_r\}$$

leads to call constraints of only the same form. It follows that if $x_b$ is not constrained in the initial query, then it will not be constrained in the subsequent calls to mortgage. As indicated in Section 1, this information allows a compiler to generate code for mortgage without a redundant test for satisfiability in the first clause.

### 4.2 Definiteness Analysis

In the definiteness analysis, descriptions are sets of variables that are read as being “definitely” constrained to a unique value. This analysis corresponds to a groundness analysis for logic programs.

**Definition.** Let $A_{\text{con}}_{\text{def}} = \mathcal{P} \text{ Var}$ with superset ordering and define $\gamma_{\text{def}} : A_{\text{con}}_{\text{def}} \rightarrow \mathcal{P} \text{ Cons}$ by

$$\gamma_{\text{def}} W = \{\pi \in \text{Con} | \forall V \in W. V \text{ definite in } \pi\}$$

where $V \text{ definite in } \pi$ iff $\pi$ constrains $V$ to a unique value. ■

**Lemma 4.4** The function $\gamma_{\text{def}}$ is a concretization function from $A_{\text{con}}_{\text{def}}$ to $\mathcal{P} \text{ Con}$. ■

The functions $\text{call}$ and $\text{ret}$ are approximated as follows.
Definition. Define

\[
\begin{align*}
\text{call}_{\text{def}} &: \text{Atom} \rightarrow \text{Acon}_{\text{def}} \rightarrow \text{Atom} \rightarrow \text{Con} \rightarrow \text{Acon}_{\text{def}} \\
\text{ret}_{\text{def}} &: \text{Atom} \rightarrow \text{Acon}_{\text{def}} \rightarrow \text{Atom} \rightarrow \text{Acon}_{\text{def}} \rightarrow \text{Acon}_{\text{def}}
\end{align*}
\]

by

\[
\begin{align*}
\text{call}_{\text{def}} A W H \pi &= \begin{cases}
\text{let } \rho = \text{ren} \left( \text{vars } H \cup \text{vars } \pi \right) \left( \text{vars } A \cup W \right) \text{ in }
\{ V \in (\text{vars } H) \mid \text{funct } (\rho W) (\pi \land \rho A = H) V \} & \\
\text{ret}_{\text{def}} A W H \pi &= \begin{cases}
\text{let } \rho = \text{ren} \left( \text{vars } A \cup W \right) \left( \text{vars } H \cup W' \right) \text{ in }

W \cup \{ V \in (\text{vars } A) \mid \text{funct } (\rho W') (A = \rho H) V \}
\end{cases}
\end{cases}
\]

where \( \text{funct } U \pi V \) iff given unique values for the variables in \( U \), the constraint \( \pi \) constrains \( V \) to have a unique value.

Lemma 4.5 The following hold:

- \( \text{call}_{\text{def}} \text{ appr } \text{call} \), and
- \( \text{ret}_{\text{def}} \text{ appr } \text{ret} \).

Definition. Let \( I_{\text{def}} \) be the interpretation given by \( \langle \text{Acon}_{\text{def}}, \text{call}_{\text{def}}, \text{ret}_{\text{def}} \rangle \).

It follows from Lemma 4.5 that \( I_{\text{def}} \) approximates \( I_{\text{lax}} \). Thus from Theorem 3.5:

Theorem 4.6 \( P_{\text{def}} \text{ appr } P_{\text{std}} \).

The call version of the definiteness analysis can be used to determine that the query description

\[
mortgage(x_p, x_t, x_i, x_r, x_b) \{ x_t \}
\]

leads to call constraints of only the same form. It follows that if \( x_t \) is uniquely constrained in the initial query, then it will be uniquely constrained in the subsequent calls to \( mortgage \). This information indicates that \( mortgage \) is locally deterministic because if \( x_t \) has a unique value then for a given \( x_t \) both of the constraints \( x_t > 1 \) and \( x_t = 1 \) cannot be satisfiable. As indicated in Section 1, this information allows a compiler to generate code for \( mortgage \) without an unnecessary choicepoint in the first clause.

4.3 Other Dataflow Analyses

We have sketched two simple, generic dataflow analyses. We note that more accurate analyses are often constraint domain specific; descriptions which capture properties of constraint sets over the reals are not necessarily going to
capture interesting properties of constraint sets over the Herbrand universe. Thus a major topic of future research is to find useful descriptions for specific constraint domains, such as the real numbers.

We feel that it is still useful to classify descriptions into one of two main classes: downward-closed and upward-closed. Not all descriptions are downward- or upward-closed but many are, and often description domains can be decomposed into a product of downward and upward-closed description domains.

**Definition.** A constraint description domain $A_{con}$ is **downward-closed** iff

$$\forall a \in A_{con}. \forall \pi, \pi' \in Con. (a \text{ appr } \pi \land (\pi \Rightarrow \pi')) \Rightarrow a \text{ appr } \pi'.$$

We define **upward-closed** in the dual manner. ■

The definiteness analysis exemplifies a downward-closed description domain while the freeness analysis exemplifies an upward-closed description domain. In a sense, downward-closed descriptions approximate constraints by weaker constraints and upward-closed descriptions approximate constraints by stronger constraints. In general, downward-closed description domains are useful for analyses which remove redundant choicepoints while upward-closed description domains are useful for analyses which remove redundant tests for satisfiability.

5 Conclusion

We have identified a need for the dataflow analysis of CLP programs and presented a framework for their development. This framework is simple, much simpler than the less general frameworks previously given for the top-down abstract interpretation of definite logic programs. There are two reasons for this. The first is the use of a definition-independent meta-language given by Marriott, Søndergaard and Jones [15] which has allowed results concerning abstract interpretation to be pushed into the metalanguage and facilitated a stepwise, derivational approach to the development of dataflow analyses. The second reason is that constraints are conceptually simpler than most general unifiers and the basic operations required in the semantic definitions may be expressed naturally in terms of logical operations on the constraints.

Acknowledgements

The authors would like to thank Joxan Jaffar who participated in several useful discussions.
References


