Definiteness Analysis for CLP($\mathcal{R}$)

Naomi Baker and Harald Sandergaard
Dept. of Computer Science, The University of Melbourne
Parkville Vic. 3052, Australia
Email: njb@cs.mu.oz.au, harald@cs.mu.oz.au

Abstract

Constraint logic programming (CLP) languages generalise logic programming languages, amalgamating logic programming and constraint programming. Combining the best of two worlds, they provide powerful tools for wide classes of problems. As with logic programming languages, code optimization by compilers is an important issue in the implementation of CLP languages. A compiler needs sophisticated global information, collected by dataflow analyses, to generate competitive code.

One kind of useful dataflow information concerns the point at which variables become definite, that is, constrained to take a unique value. In this paper we present a very precise dataflow analysis to determine definiteness, and we discuss its applications. By separating the two concerns: correctness and implementation techniques, abstract interpretation enables us to develop a sophisticated dataflow analysis in a straightforward manner, in fact in a framework where the correctness of the analysis is easily established—a feature which is uncommon when complex analyses are developed in an ad hoc way.

We use a class of Boolean functions, the positive functions, to represent the definiteness relationship between variables. A Boolean function is interpreted as expressing a relation which holds not simply at the given point in an evaluation, but in fact during the rest of the evaluation branch. The nature of variables in a CLP language makes this treatment both possible and natural.

1 Introduction

Constraint logic programming (CLP) combines the best features of constraint programming and logic programming and provides a simple and powerful tool for a large class of problems. Constraint based reasoning is natural and powerful, and pure logic programming offers convenience for programmers and clean, declarative semantics. Logic also has obvious qualities as a database language, and it may even serve as a compact formalism for concurrent programming.

CLP can be considered a generalisation of logic programming. Instead of using substitutions and unification when evaluating queries, CLP is concerned with constraints and constraint solving. Unification can be seen as one particular kind of constraint solving, in the case where constraints are term equations. But in general, CLP languages allow constraints over other domains and offer special constraint solving capabilities. For example, CHIP [8] supports constraint solving over finite domains and handles linear arithmetic over natural numbers, and Prolog III [2] supports reasoning about rational numbers, Boolean values, lists and infinite trees. In this paper we consider CLP($\mathcal{R}$) [10] which supports real number arithmetic and also allows for uninterpreted function symbols.
Example 1.1 The following is a CLP($\mathcal{R}$) program to calculate mortgage repayments. The mortgage predicate relates the following parameters: The principal $p$, the life of the mortgage (in months) $t$, the annual interest rate $i$, the monthly repayment $r$, and the outstanding balance $b$.

\[
\text{mortgage}(p, t, i, r, b) \leftarrow t = 1, b = p \times (1 + i/1200) - r.
\]

\[
\text{mortgage}(p, t, i, r, b) \leftarrow t > 1, \text{mortgage}(p \times (1 + i/1200) - r, t - 1, i, r, b).
\]

CLP languages are relational. The program in Example 1.1 may be queried in many different ways. For example, the query $q_1$: \text{mortgage}(80000, 12, 9.5, 1000, b)$ asks what the balance will be after one year, assuming a given principal, interest rate and monthly repayment. It yields the answer $b = 76656.9$. (In general a query may yield several answers.) The query \text{mortgage}(80000, 120, 9.5, r, 0)$ asks what the monthly repayment should be if the loan is to be paid off in ten years and yields the answer $r = 1035.18$. The query \text{mortgage}(p, 120, 9.5, r, 0)$ asks what relation there is between original loan and monthly repayment, given a fixed life and interest and yields the answer $p = 77.2812 \times r$.

The dominating cost of query evaluation comes from constraint solving. However, for many queries (such as $q_1$ above), the power of a general constraint solver is not really needed. A compiler could translate many or all of the encountered constraints into simple tests or assignments, thus generating efficient code by avoiding runtime calls to a general constraint solving routine. Even if conjunctions of basic constraints do need to be sent to a constraint solver, it is worthwhile trying to detect redundancy in the constraints, so as to maintain as small conjunctions as possible.

Exactly which optimizations of constraint logic programs are possible depends on how runtime structures are handled by an implementation. However, many (and powerful) optimizations can be understood as source-level transformations and require only limited knowledge about an implementation. In the case of CLP($\mathcal{R}$), Jaffar et al. [9] consider code generation for an abstract machine, CLAM, whose instruction set includes instructions designed specifically with an optimizing compiler in mind.

Since CLP generalises logic programming, many optimization techniques from that area can be adapted to CLP languages in a straightforward way. However, it is likely that more powerful transformations for CLP languages will be language-specific and have no counterpart in traditional logic programming languages. Jørgensen, Marriott and Michaylov [11] consider four possible optimizations for CLP($\mathcal{R}$), based on various types of dataflow information: (a) detection of trivial constraints (those that can be replaced by tests or assignments), (b) recognition of mutually exclusive clauses, (c) code motion, and (d) detection of “future redundant” constraints (those that will be implied by constraints subsequently added, and which, in the meantime, cannot cause unsatisfiability, that is, affect the flow of control). In the cases (a), (b), and (c) it is possible to customise analyses known from logic programming, but (d) does not have an obvious analogue.

Let us call a variable definite if it is fixed by the current constraint, that is, constraining it further can only lead to unsatisfiability. In Section 4 we shall be more precise about this notion, so let an example suffice: In the conjunction $x = 5 \land x < y$, the variable $x$ is definite, but $y$ is not.

Jørgensen, Marriott and Michaylov exemplify the optimizations (a)-(d) in terms of the program in Example 1.1:

1. For queries with definite $t$, the constraints $t = 1$ and $t > 1$ are trivial and can be replaced by tests, thus bypassing the constraint solver.
2. For queries with definite \( t \), the two clauses are mutually recursive, and code for backtracking can be avoided.

3. The expression \( (1 + i/1200) \) is recursive-invariant and can be moved outside the recursive clause.

4. Even for queries where \( t \) is not definite, the constraint \( t > 1 \) is future redundant and can be replaced by a simple test.

Jaffar et al. [9] also consider how to minimize redundancy. In a naïve scheme, a basic constraint may be added to the current conjunction, even when it is already implied by the conjunction. More commonly a constraint may become uninteresting in that it is concerned only with variables that will never be used again. Removing such constraints and variables may lead to considerable savings in both time and space.

In addition to such removals, Marriott and Stuckey [15] consider optimization based on the reordering of constraints. The important observation is that a constraint should generally be introduced as late as possible in a computation, to keep the conjunction of constraints small. However, if the constraint may cause unsatisfiability, moving it to a later stage can clearly affect program behaviour. The problem is therefore to determine the earliest point at which a constraint could possibly cause unsatisfiability.

The optimizations discussed above require a large variety of dataflow information. One kind of information which is useful for all of them concerns the definiteness of variables. In this paper we discuss how precise definiteness information can be generated by an analysis based on abstract interpretation. We let the definiteness information be represented by Boolean functions which also capture how the definiteness of one variable may depend on that of other variables.

In Section 2 we discuss the use of Boolean functions to capture dataflow information in CLP languages. In Section 3 we present a generic semantic definition suitable as a basis for developing provably correct dataflow analyses for CLP(\( R \)). In Section 4 we show how a very precise definiteness analysis is obtained by instantiating the generic semantics to manipulate Boolean functions that represent definiteness dependencies. Section 5 contains a brief discussion of the handling of non-linear constraints and complexity. To get the most out of Sections 3 and 4, readers should have a basic understanding of denotational semantics.

2 Boolean Functions for Dependency Analysis

A variable in a CLP language is very different from a variable in an imperative or functional programming language. It is sometimes referred to as a “logical variable” and characterised as “constrain-only.” Execution of a (constraint) logic program proceeds by steps that continually narrow the set of possible values that a variable may take.

Let us use the notation \( \phi|_V \) to denote the constraint \( \phi \) restricted to the set \( V \) of variables (we shall later be more precise about this). Let a query \( Q \) be given and let \( W \) be the set of variables in \( Q \). Consider now one branch of the evaluation tree for \( Q \). If at some point the current constraint is \( \phi \) and later along the same branch it is \( \phi' \), then we must have that \( \phi'|_W \models \phi|_W \).

In addition to this, a characteristic of CLP languages is that they are in principle non-deterministic so that we should think of query evaluation branching into (possibly failing)
sub-evaluations. Failure occurs when a variable becomes “over-constrained,” so that the current conjunction of constraints no longer is satisfiable.

These characteristics of the execution of constraint logic programs makes dataflow analyses harder in some ways, but it also opens up new views of some analysis problems. For example it suggests the possibility of propagating conditional invariants of the form “From this point on, if \( x \) has (ever gets) property \( p \), then \( y \) has (will have) property \( r \).”

Conditional invariants can perhaps also make sense for imperative or functional programs. For example we may be interested in automatically establishing invariants such as

- “in every state that may obtain at program point \( P \), if integer variable \( m \) is positive, so is \( n \)” (sign dependency), or
- “in all calls \( f(e_1, e_2, e_3) \), if \( e_2 \) needs to be evaluated, \( e_3 \) does not, and vice versa” (strictness dependency).

However, the important difference between these examples on the one hand and the analysis of (constraint) logic programs on the other is that the former lack the ability to “project invariants into the future.”

Example 2.1 will make it clearer what we mean by “projecting into the future,” but first we need to explain how dependency information can be represented. A statement such as “\( x \) is definite” may be represented by a propositional variable \( x \). Definiteness (or other) dependencies may then be represented by Boolean functions. For example, the function denoted by \( y \rightarrow x \) expresses that \( y \) cannot be definite without \( x \) being definite as well. (We call a formula such as \( y \rightarrow x \) a dependency formula.)

Definiteness and other interesting properties are not decidable so a dataflow analysis which operates in finite time can only give approximate information, in general. So the statements that it produces will carry a modality, as in “\( x \) is inevitably definite” or “\( x \) is possibly definite.” For this reason, only the positive Boolean functions are useful. If we associate the meaning “\( x \) is inevitably definite” with the formula \( x \), then \( \neg x \) would mean “\( x \) may not always be definite” and this conveys no information at all, that is, exactly the information conveyed by the function \( true \).

**Definition.** A Boolean function is a function \( F : Bool^n \rightarrow Bool \). We call the set of all \( n \)-ary Boolean functions \( Bfun_n \) and let it be ordered by logical consequence (\( \models \)). The function \( F \) is positive iff \( F(true, \ldots, true) = true \). We denote the set of positive Boolean functions of \( n \) variables by \( Pos_n \). ■

For simplicity we will assume that we have some fixed number \( n \) of variables and leave out subscripts and the phrase “of \( n \) variables.” The set of propositional variables \( \{ x_1, \ldots, x_n \} \) will be referred to as \( Pvar \). We shall also use propositional formulas as representations of Boolean functions without worrying about the distinction. Thus we may speak of a formula as if it were a function and in any case denote it by \( F \). The Hasse diagram in Figure 1 shows the ordering of the formulas in \( Pos_2 \).

It is well known that \( Bfun \) is a Boolean lattice and in fact \( Pos \) forms a Boolean sublattice of \( Bfun \). The complementation operation on \( Pos \) is given by \( \neg F = F \leftrightarrow \bigwedge Pvar \), see [3]. Logicians have of course studied \( Pos \) under several different names. However, the history of dependency clauses for dataflow analysis is rather short. Dart used a class of dependency
formulas in his work in the area of deductive databases [5, 6]. Dart’s class is strictly less expressive than Pos, which was suggested by Marriott and Søndergaard [12, 13] (under the less suggestive name ‘Prop’) and further studied by Cortesi et al. [3].

One can imagine many types of properties of dataflow information for which dependency formulas is a useful formalism. Here we are concerned with definiteness. The definiteness dependencies for some constraints in CLP(R) are shown in Table 1. For example, if the constraint \( x = y + 6 \) is reached during query evaluation, the relationship \( x \leftrightarrow y \) is deduced. Informally the formula says that if \( x \) is (becomes) definite then so is (does) \( y \), and vice versa. Example 2.1 shows how this kind of information may be useful.

**Example 2.1** Consider the CLP(R) program to compute the Fibonacci relation:

\[
\begin{align*}
&fib(0, 1). \\
&fib(1, 1). \\
&fib(n, f) \leftarrow f = f1 + f2, \text{fib}(n - 1, f1), @\text{fib}(n - 2, f2).
\end{align*}
\]

Assume the program is queried \( fib(x, y) \). The dataflow analysis defined in Section 4 computes a certain fixpoint in a lattice consisting of Boolean functions. A first approximation, \( x \land y \), is obtained by considering the two facts. That is, so far it looks as if the query will result only in answers with \( x \) and \( y \) definite, but this may well be revised in a subsequent approximation step.

In processing the recursive clause, we use the already obtained approximation. First, the constraint \( f = f1 + f2 \) gives the formula \( (f1 \land f2 \rightarrow f) \land (f1 \land f \rightarrow f2) \land (f2 \land f \rightarrow f1) \).
Conjoining this with the formula expressing what we know (or rather, currently assume) about answers from \( \text{fib} \), we get
\[
(f_1 \land f_2 \rightarrow f) \land (f_1 \land f \rightarrow f_2) \land (f_2 \land f \rightarrow f_1) \land n \land f_1,
\]
which simplifies to
\[
F : (f \leftrightarrow f_2) \land n \land f_1.
\]
This is the formula which approximates the constraints that may obtain at the program point marked @. It expresses that \( n \) and \( f_1 \) will both be definite, and furthermore, should \( f_2 \) be (or ever become) definite then so is (will) \( f \), and vice versa. This relation between \( f \) and \( f_2 \) is what we had in mind when saying that invariants may be “projected into the future.”

We immediately see the value of this: The (current) formula for the last atom in the clause is \( n \land f_2 \), and conjoining this with \( F \) we get, after simplification, \( f \land f_1 \land f_2 \land n \). In terms of the variables in the query this translates (after a projection) to \( x \land y \).

We conclude that \( x \land y \) is a fixpoint, that is, \( x \) and \( y \) will indeed become definite. This information in turn can be translated into call pattern information. For example, if \( \text{fib} \) is queried with a second argument which is definite, it follows that \( f_2 \) must be definite at point @. ■

In the next two sections we sketch how these ideas can be put on a firm formal ground by employing techniques from abstract interpretation.

3 Semantics-Based Static Analysis

The type of reasoning employed in Example 2.1 is an example of what has been called “pseudo-evaluation.” The evaluation of a query is mimicked by manipulating approximate descriptions of constraints, rather than constraints proper. An infinite number of constraints may obtain at point @ in Example 2.1, including
\[
\begin{align*}
f & = 1 + f_2 \land n = 1 \land f_1 = 1, \\
f & = 1 + f_2 \land n = 2 \land f_1 = 1, \\
f & = 2 + f_2 \land n = 3 \land f_1 = 2,
\end{align*}
\]
but the formula \( F : (f \leftrightarrow f_2) \land n \land f_1 \) gives a finite description covering all the cases. One can think of \( F \) as a constraint in a (meta–)CLP language which is sufficiently rich to be used for inferring precise definiteness information, and sufficiently curtailed for the desired information to be finitely computable.

Abstract interpretation formalises the idea of a (correctly) mimicking evaluation. The theory was developed by P. and R. Cousot who in a recent paper recapitulate the ideas and their applications for logic programming [4]. That paper also includes an extensive bibliography.

The idea is that, to be provably correct, static analysis should be semantics-based. Assume that we are given a semantic definition of a programming language in the form of a fixpoint characterisation, that is, the meaning of a program is somehow modelled by a fixpoint in a semantic domain \( D \). We then want to understand the result of a dataflow analysis as an approximation of extreme fixpoints in \( D \). This point should become clearer when we get to Example 4.1.

So our first task is to decide on a semantic definition of CLP(\( \mathcal{R} \)). We follow Marriott and Søndergaard [14] in using a denotational definition. (Marriott and Søndergaard call
the semantics “lax” because it may sometimes (rarely) yield too large a set of constraints. Fortunately this has no impact on correctness or precision of a dataflow analysis based on it. The advantage of the definition is that it is highly reusable, as we shall see.)

**Definition.** The *generic semantics* has domain parameter $\Pi$, semantic domain

\[
Den = Atom \rightarrow \Pi \rightarrow \Pi,
\]

and semantic functions

\[
\begin{align*}
P & : Prog \rightarrow Den \\
C & : Clause \rightarrow Den \rightarrow Den \\
B & : Body \rightarrow Den \rightarrow \Pi \rightarrow \Pi.
\end{align*}
\]

The semantic equations are

\[
\begin{align*}
P [P] & = \text{lfp} (\bigcup_{C \in \mathcal{F}} C [C]) \\
C [H \leftarrow \phi \land B] d A \pi & = \text{return } A \pi H (B [B] d (\text{call } A \pi H \phi)) \\
B [nil] d \pi & = \pi \\
\end{align*}
\]

Notice that the definition is not complete until the domain $\Pi$ and the auxiliary functions

\[
\begin{align*}
call & : Atom \rightarrow \Pi \rightarrow Atom \rightarrow Con \rightarrow \Pi \\
return & : Atom \rightarrow \Pi \rightarrow Atom \rightarrow \Pi \rightarrow \Pi
\end{align*}
\]

are provided. We require that $\Pi$ be a complete lattice and that call and return be monotonic—this guarantees that the semantic functions are well-defined.

In the case of CLP$(\mathcal{R})$, the interesting semantic domain is the set $Con$ of (equivalence classes of) conjunctions of basic constraints over real numbers. This domain has extreme points false and true and is ordered by logical consequence (|=). In fact, since the language may be considered nondeterministic, it is natural to consider the powerset $\mathcal{P}(Con)$ of constraints, ordered by subset ordering. This makes particularly good sense in the context of dataflow analysis, since at a given program point, different constraints may occur over time, and an analysis should therefore be collecting (approximations to) increasing sets of constraints.

Thus we set $\Pi = \mathcal{P}(Con)$ and define the auxiliary functions by

\[
\begin{align*}
call A \Phi H \phi & = \text{return } H \{\phi\} A \Phi \\
return A \Phi H \Phi' & = \text{meet } \Phi (\text{project } (\text{vars } A) (\text{meet } \{A = \rho H\} (\rho \Phi))).
\end{align*}
\]

Here the functions $\text{meet} : \mathcal{P}(Con) \rightarrow \mathcal{P}(Con) \rightarrow \mathcal{P}(Con)$ and $\text{project} : \mathcal{P}(P\text{var}) \rightarrow \mathcal{P}(Con) \rightarrow \mathcal{P}(Con)$ are defined by

\[
\begin{align*}
\text{meet} \Phi \Phi' & = \{\phi \land \phi' \mid \phi \in \Phi \land \phi' \in \Phi'\} \\
\text{project } V \Phi & = \{\exists \overline{V}. \phi \mid \phi \in \Phi\}
\end{align*}
\]

where $\overline{V}$ denotes the complement of $V$. Finally $\rho$ denotes an appropriate renaming of variables, with the property that $\rho = \rho^{-1}$.

Notice that the meaning of a program assigns a constraint transformer to each atom. The denotational definition has the following operational reading: If $A$ calls clause $H \leftarrow \phi \land B$ and the current constraint is $\phi_m$, then (disregarding renaming) the constraint $H = A \land
\( \phi_{in} \) is projected onto the set of variables in \( H \). Semantically, this projection is simply an appropriate existential quantification. The body of the clause is processed recursively, and any constraint \( \phi' \) obtained is used to form \( A = H \land \phi' \), which is projected onto the variables in \( A \). Calling the result \( \phi_{out} \), the clause has determined a partial constraint transformer \( \phi_{in} \rightarrow \phi_{in} \land \phi_{out} \) for \( A \), and the full transformer for \( A \) is found by appropriate combination (a join, \( \sqcup \)) of the transformations determined by the individual clauses.

**Example 3.1** Consider the following program.

\[
q(x, y) \leftarrow p(x, y), r(x, y).
p(u, 1).
p(1, v).
r(u, v) \leftarrow u = v + 1.
\]

To find the result of evaluating (instances of) the query \( q(x, y) \), we compute the Kleene sequence \( d_0, d_1, \ldots, d \) for the semantic function \( \sqcup \{C \in F | C \} \). Notice that elements in the sequence are functions and that we need only compute a small subset of the function graph.

\[
\begin{align*}
d_0 A \Phi &= \emptyset \\
d_1 q(x, y) \{\text{true}\} &= \emptyset \\
d_1 p(x, y) \{\text{true}\} &= \{\text{false}, x = 1, y = 1\} \\
d_1 r(x, y) \{\text{false}, x = 1, y = 1\} &= \{\text{false}, x = 1 \land y = 0, x = 2 \land y = 1\} \\
d_2 q(x, y) \{\text{true}\} &= \{\text{false}, x = 1 \land y = 0, x = 2 \land y = 1\}
\end{align*}
\]

In this case \( d_2 \) turns out to be a fixpoint. Notice that \( d_1 p(x, y) \{\text{true}\} \) is found by taking the join (here: union) of the contributions from the four clauses. Readers may want to try out the definition on recursive programs such as those in Example 1.1 or Example 2.1.

**4 Definiteness Analysis**

We now define a dependency analysis based on the generic semantics from Section 3. It gives information about definiteness dependencies that hold in the call patterns that arise during evaluation of a given query.

It is convenient to include the (non-positive) Boolean function \( \text{false} \) as an approximation to the empty set of constraints. Thus this section deals with the domain \( \text{Pos} \perp = \text{Pos} \cup \{\text{false}\} \), ordered by logical consequence.

The first question is: Exactly which constraints are approximated by a given Boolean function? To answer this precisely, we define the so-called concretization function.

**Definition.** For a constraint \( \phi \), let \( \text{def} \ \phi \) be the truth assignment which maps a variable to true if \( \phi \) fixes the variable and false otherwise. That is, \( \text{def} : \text{Con} \rightarrow \text{Pvar} \rightarrow \text{Bool} \) is defined by

\[
\text{def} \ \phi \ x \Leftrightarrow \begin{cases} 
\phi |_x \text{ is minimal in } \{\phi' |_x \mid \phi' \in \text{Con} \setminus \{\text{false}\}\} & \text{if } \phi \neq \text{false} \\
\text{true} & \text{if } \phi = \text{false}.
\end{cases}
\]

**Definition.** For \( \phi \in \text{Con} \), let \( \Delta_\phi = \{\phi' \in \text{Con} \mid \phi' \models \phi\} \). The concretization function \( \gamma : \text{Pos} \perp \rightarrow \mathcal{P} (\text{Con}) \) is defined by

\[
\gamma(F) = \{\phi \in \text{Con} \mid \forall \phi' \in \Delta_\phi. \text{def} \ \phi' \models F\}.
\]
This concretization function maps \( F \in \text{Pos}_\perp \) to the set of constraints \( \phi \) which have the property that, no matter how they further become instantiated to some \( \phi' \), def \( \phi' \) will satisfy \( F \).

It should be clear that \( \gamma \text{Pos}_\perp \) is closed under intersection. It is well-known [4] that this is a sufficient criterion for the existence of a so-called abstraction function \( \alpha : \mathcal{P}(\text{Con}) \rightarrow \text{Pos}_\perp \) which is an adjoint to \( \gamma \). Given a set of constraints, \( \alpha \) yields the (unique) best \( F \in \text{Pos}_\perp \) which approximates every constraint in the set. The function \( \alpha \) may be defined in terms of \( \gamma \) as follows: \( \alpha \Phi = \bigwedge \{ F \in \text{Pos}_\perp \mid \Phi \subset \gamma F \} \). This is a fairly indirect definition, but it should be clear that \( \alpha \{ \phi \} \) can readily be obtained from Table 1. (Constraints involving more than three variables can be written in three-variable form by introducing temporary variables, and temporary variables can subsequently be projected away.)

The definition of the definiteness analysis can now be given simply as the instance of the generic semantics in which \( \Pi = \text{Pos}_\perp \) and the auxiliary functions

\[
\begin{align*}
\text{call}_d & : \text{Atom} \rightarrow \text{Pos}_\perp \rightarrow \text{Atom} \rightarrow \text{Pos}_\perp \rightarrow \text{Pos}_\perp \text{ and} \\
\text{return}_d & : \text{Atom} \rightarrow \text{Pos}_\perp \rightarrow \text{Atom} \rightarrow \text{Pos}_\perp \rightarrow \text{Pos}_\perp \\
\end{align*}
\]

are defined by

\[
\begin{align*}
\text{call}_d \ A \ F \ H \ \phi & = \ \text{project}_d \ (\text{vars} \ H \cup \text{vars} \ \phi) \ (\text{meet}_d \ (\rho F) \ (\alpha \{ \phi \wedge H = \rho A\})) \\
\text{return}_d \ A \ F \ H \ F' & = \ \text{meet}_d \ F \ (\text{project}_d \ (\text{vars} \ A) \ (\text{meet}_d \ (\alpha \{ A = \rho H\}) \ (\rho F' )))
\end{align*}
\]

with \( \text{meet}_d : \text{Pos}_\perp \rightarrow \text{Pos}_\perp \rightarrow \text{Pos}_\perp \) and \( \text{project}_d : \mathcal{P}(\text{Pvar}) \rightarrow \text{Pos}_\perp \rightarrow \text{Pos}_\perp \) defined by

\[
\begin{align*}
\text{meet}_d \ F \ F' & = \ F \wedge F' \\
\text{project}_d \ V \ F & = \ \exists \ \forall. \ F
\end{align*}
\]

and \( \rho \) denoting an appropriate renaming of variables, with the property that \( \rho = \rho^{-1} \).

Notice that the lattice \( \text{Pos}_\perp \) does not have infinite chains. This is what guarantees the finiteness of the definiteness analysis.

**Example 4.1** Consider again the program

\[
\begin{align*}
q(x, y) \leftarrow p(x, y), r(x, y) @ \ .
\end{align*}
\]

\[
\begin{align*}
p(u, 1).
p(1, v).
\end{align*}
\]

\[
\begin{align*}
r(u, v) \leftarrow u = v + 1.
\end{align*}
\]

Evaluating the Kleene sequence \( d_0, d_1, \ldots, d \) will give us the information about definiteness of variables. Again, for the query \( q(x, y) \), the relevant part of the function graph is:

\[
\begin{align*}
d_0 \ A \ F & = \ false \\
d_1 \ q(x, y) \ true & = \ false \\
d_1 \ p(x, y) \ true & = \ x \vee y \\
d_1 \ r(x, y) (x \vee y) & = \ x \wedge y \\
d_2 \ q(x, y) \ true & = \ x \wedge y
\end{align*}
\]

It turns out that \( d_2 \) is a fixpoint. The computation tells us that any instance of the query \( q(x, y) \) will, if it succeeds, result in both \( x \) and \( y \) becoming definite. It is instructive to see what happens at the program points marked @ and @. After processing of \( p(x, y) \) the approximation is \( x \vee y \). The contribution from \( r(x, y) \) is \( x \leftrightarrow y \), and the conjunction of these gives the result \( x \wedge y \) at point @. This result is the best possible, and achieved only because \( \text{Pos}_\perp \) is sufficiently expressive to capture the disjunctive information “at least one of \( x \) and \( y \) are definite.” 

\[9\]
The correctness of a sophisticated dataflow analysis is normally far from clear. In this case, however, it is quite easy to show that the definiteness analysis is correct. We need to be precise about the term “approximates”:

**Definition.** For \( F \in Pos \) and \( \Phi \subseteq Con \) we define the relation \( appr \) by

\[
F appr \Phi \text{ iff } \Phi \subseteq \gamma F.
\]

It is convenient to lift the relation to function spaces also. Let \( F : X \to Y \) and \( G : U \to V \) be given. Then \( F appr G \) iff \( (F \ x) appr (G \ u) \) whenever \( x appr u \). ■

**Theorem 4.1** Let \( P_c \) denote the instance of \( P \) defined in Section 3 and let \( P_d \) denote the instance defined above. Then \( P_d appr P_c \).

*Proof sketch:* The proof relies on the following lemmata, all quite simple to establish. Namely, \( meet_d \) is associative and commutative; \( meet_d appr meet \); and \( project_d appr project \). From these one can show that \( return_d appr return \) and \( call_d appr call \). The theorem then follows by fixpoint induction. ■

5 Discussion

Readers may wonder about the treatment of nonlinear constraints. In Table 1 the dependency formula for a constraint \( x = y \ast z \) was given as \( y \land z \to x \), which is much weaker than the formula for \( x = y + z \). To see that we cannot strengthen the formula, consider the query \( x = 0, y = 0, x = y \ast z \). Clearly the definiteness of \( x \) and \( y \) need not imply that of \( z \).

However, an analysis which is useful for various other purposes is a bounds (or interval) analysis. Such an analysis estimates for each variable \( x \) an interval in which the value of \( x \) must lie. Given a bounds analysis, it is possible to find cases where variables that appear in nonlinear constraints can be shown to be non-zero, and to choose an accordingly stronger dependency formula.

Definiteness information can be particularly useful in the presence of nonlinear constraints. In CLP(\( R \)) implementations, the handling of nonlinear constraints is very different from that of linear constraints and incurs a considerable amount of overhead. A compiler can capitalise on knowledge of variables that will be definite by generating simpler code for some nonlinear constraints. For example, if it is known that one of \( y \) and \( z \) will always be definite when a constraint \( x = y \ast z \) is met, then the constraint can be treated as if it were linear.

Debray [7] has shown that the worst-case complexity of exact inference of groundness for a simple class of datalog programs (logic programs without function symbols) is EXPTIME-complete. This is relevant here, because his result would seem to carry over to the case of CLP, and for the corresponding simple class of programs (only constraints of the form \( x = c \) or \( x = y \)), our dataflow analysis is indeed exact. It is hard to say whether Debray’s result will be felt in practice. Debray also shows that much less sophisticated analyses are intractable (assuming \( P \neq N \)) on even simpler classes of programs, and these analyses appear to be quite efficient on typical programs.

The dominating factor in assessing the computational cost of the analysis is the number of variables that may appear in a clause. For example, if all clauses in a program contained no more than five variables, then every Boolean function that we ever construct could
be represented by no more than 32 32-bit words (since it has at most 10 variables), the
operations *meet*, *join*, projection, and testing for equality would all be cheap, and *join* and
testing for equality would in fact only be applied to 32-bit words. The problem is that
this representation is exponential in the (maximum) number of variables in a clause. A
candidate for a data structure with a more reasonable time/space trade-off is the decision
diagram, see for example Bryan [1].

There are proper subsets of $Pos$ that also might be good candidates for representing
various kinds of dataflow dependencies. We plan empirical studies of the precision/efficiency
trade-off on typical programs by employing not only positive functions, but also subclasses
such as Dart’s class, that of monotonic functions, that of symmetric functions, and inter-
sections of some of these.

6 Conclusion

We have presented a query-directed definiteness analysis for CLP($R$). Such an analysis has
numerous applications in program transformation and compilation in particular. It turns
out to be desirable to maintain invariants which state how different variables are related
with respect to definiteness. In fact, to be more useful, such invariants should be statements
that are valid not only at a given program point, but hold in the remainder of that branch
of evaluation. This ability of invariants to “project into the future” is closely connected to
the nature of variables in a CLP language.

Positive Boolean functions were used for representing the dependencies of definiteness
between variables, but subclasses of this class may also be useful for the given purpose.
However, the positive functions lead to a very precise analysis, at least in the absence of
nonlinear constraints.

Semantically, the analysis can be viewed as an approximation to the standard semantics,
in a precise sense, and this in turn allows for verification of the correctness of the analysis,
something which is not generally the case for sophisticated analyses. Where the standard
semantics was defined in terms of operations on a lattice of (sets of) constraints, the analysis
is defined in terms of very similar operations on a lattice of Boolean functions, ordered
by logical consequence. One aspect of this view is that it should be relatively simple
to implement a prototype definiteness analysis in a CLP language which handles Boolean
constraints. We hope to investigate the precision/efficiency trade-off in definiteness analysis,
using a fairly sophisticated implementation currently under way.

Acknowledgements

The first author has been supported by an Australian Postgraduate Research Award and a
CITRI supplementary scholarship. The second author has been supported by a grant from
the Australian Research Council.

References


