Accurate Functional Dependency Analysis for Constraint Handling Rules

Gregory J. Duck[1] and Tom Schrijvers[2*]

[1] Department of Computer Science, University of Melbourne, Australia
[2] Department of Computer Science, K.U.Leuven, Belgium

Abstract. Information about functional dependencies is used by modern CHR compilers for both optimisation and for further program analysis (e.g. confluence analysis). Before this work, CHR compilers relied on an ad hoc analysis for functional dependencies based on searching for rules of a particular form and the results from late storage analysis.
We present a more formal functional dependency analysis of CHRs based on abstract interpretation. We show, by example, that the new analysis is more accurate than the existing ad hoc analysis.

1 Introduction

The recent appearance of new optimising CHR systems [3, 5] has given rise to the need to communicate and compare between different CHR systems, which has led to the formulation of the more deterministic refined operational semantics [2] shared among CHR compilers.

Apart from the common formal semantics, there is also a need to communicate and compare program analyses. As the complexity of CHR compilers increases we need a better understanding of current analyses and ways to extend and combine them. Most of the currently existing analyses have been formulated in an ad hoc way and no formal proofs of correctness exist.

In [6] a formal framework for program analysis of CHR, based on abstract interpretation [1], has been presented. This framework provides a remedy for the current difficulties in correctly analysing CHR programs, and should enable optimising CHR compilers to reach a new level of complexity and correctness.

In this paper we apply the abstract interpretation framework of CHR to the ad hoc analysis of the functional dependency property presented in [4]. The functional dependency property is useful for program optimisations such as join ordering, index selection and never-stored optimisation. Moreover, functional dependency information allows for more accurate confluence analysis [2] with respect to the refined operational semantics.

The new functional dependency analysis is an improvement over the ad hoc version for several reasons. Firstly, it provides a formal description of the analysis, as an abstraction of the refined operational semantics. This allows for better

* Research Assistant of the Fund for Scientific Research - Flanders (Belgium)(F.W.O. - Vlaanderen)

insight into the analysis, its correctness, possible improvements and integration with other analyses. Secondly, functional dependency information is associated with every "program point", rather than for the program as a whole. This allows for more localised optimisations. Thirdly, the analysis keeps track of constraint multiplicity information for the constraint store. This allows for more accurate approximation of the actual control flow.

Overview First, in Section 2 we summarize the operational semantics that the functional dependency property is based on. Then Section 3 formally defines the property itself and illustrates it on a number of practical examples. Next, Section 4 briefly explains the existing ad hoc analysis. In Section 5 the new analysis, its abstract domain and abstract semantics function are presented and illustrated. We discuss the implementation and evaluation of our new analysis in Section 6. Finally, Section 7 concludes.

2 Operational Semantics of Constraint Handling Rules

Functional dependency analysis relies on the refined call-based operational semantics of CHRs, which we briefly introduce here. Although this paper assumes strong familiarity with the call-based semantics of CHRs, we must omit a proper introduction for space reasons. We direct the reader to [6, 2] for an extensive treatment.

The call-based semantics for CHRs is defined as a transition system between *execution states*. An execution state is a tuple $\langle G, A, S, B, T\rangle_n$, where G is the *goal*, A is the *activation stack*, S is the *CHR store*, B is the *built-in store*, T is the *propagation history* and n the *next free constraint identifier*. Execution states are represented by the symbol σ, and $\mathbf{\Sigma}$ is the set of all execution states.

The goal G is either a *built-in constraint*, *CHR constraint*, *identified constraint* or *active constraint*. Built-in constraints are handled by an underlying built-in solver, whereas the other kinds of constraints are handled by the CHR execution algorithm. A CHR constraint c may be associated with a (unique) integer identifier i to become an *identified constraint* $c\#i$. Furthermore, $c\#i$ may be associated with an *occurrence number* j to become an *active constraint* $c\#i:j$. The identifier i distinguishes multiple copies of c, and the occurrence number j is essentially a program counter (which decides the rule to match against).

The CHR store S is a set of identified constraints. The built-in store B is an accumulated conjunction of the built-in constraints passed to the built-in solver during solving. The meaning of the built-in constraints is decided by a *constraint domain* $\mathcal{D}$. For example, if $\mathcal{D} \models B \rightarrow (x = y)$, then x and y are equal w.r.t. the constraints in B.

Operationally, a constraint c is *called* by making it the goal in an execution state. If c is a built-in constraint, then c is added to the built-in store (i.e. passed to the built-in solver). Otherwise, if c is a CHR constraint, then it is *activated* to become $c\#i:1$ (for new number i) and $c\#i$ is added to S. An active constraint will search through all occurrences (by incrementing the occurrence number) until it is either deleted by a rule, or all occurrences and matchings have been already been tried.

3 The Functional Dependency Property

In this section we formally define the functional dependency property.

A functional dependency is a relationship between the arguments of a CHR constraint. The notation we use for functional dependencies is

$$p(x_1, \ldots, x_n) :: \{x_{i_0}, \ldots, x_{i_l}\} \rightsquigarrow \{x_{j_0}, \ldots, x_{j_m}\}$$

which indicates that the arguments $\{x_{i_0}, \ldots, x_{i_l}\}$ functionally determine the value of arguments $\{x_{j_0}, \ldots, x_{j_m}\}$, where both $\{x_{i_0}, \ldots, x_{i_l}\}$ and $\{x_{j_0}, \ldots, x_{j_m}\}$ are subsets of $\{x_1, \ldots, x_n\}$. We sometimes refer the domain $\{x_{i_0}, \ldots, x_{i_l}\}$ as the *key* for the functional dependency.

The key to detecting functional dependencies is the following utility function, which counts the number of constraints satisfying a particular form in the given CHR store.

Definition 1. *Given a functor/arity of a constraint p/n, a set of positive integers $\{i_0, \ldots, i_j\}$, CHR store S and a built-in store B, we define function $\mathsf{count}(p, n, \{i_0, \ldots, i_j\}, S, B)$ to be the following. Let $S' \subseteq S$ be the maximal subset of S such that all $p(x_1, \ldots, x_n)\#i \in S'$ and $p(y_1, \ldots, y_n)\#i' \in S'$ satisfy*

$$\mathcal{D} \models B \rightarrow (x_{i_0} = y_{i_0} \wedge \ldots \wedge x_{i_j} = y_{i_j})$$

Then $\mathsf{count}(p, n, \{i_0, \ldots, i_j\}, S, B) = |S'|$.

Example 1. For example, given the following CHR store

$$S = \{p(1,2)\#1, p(1,3)\#2, p(1,4)\#3\}$$

then $\mathsf{count}(p, 2, \{1\}, S, true) = 3$ since there is at most three constraints which share the same first argument. Similarly, $\mathsf{count}(p, 2, \{2\}, S, true) = 1$ since there is at most one constraint which share the same second argument. □

We can formally define a functional dependency in terms of the count function as follows.

Definition 2 (Set Semantic Functional Dependency). *Given a CHR store S, and built-in store B, we say constraint $\mathtt{p}/n$ has a* set semantic functional dependency

$$p(x_1, \ldots, x_n) :: \{x_{i_0}, \ldots, x_{i_j}\} \rightsquigarrow \{x_1, \ldots, x_n\}$$

if $\mathsf{count}(p, n, \{i_0, \ldots, i_j\}, S, B) \leq 1$.

Example 2. Consider the CHR store S from Example 1. Then the set semantic functional dependency $p(x, y) :: \{y\} \rightsquigarrow \{x, y\}$ holds since each possible value of y is associated with at most one value of $\{x, y\}$ in the store. For example, $y = 3$ is only associated with $\{1, 3\}$ from the constraint $p(1, 3)\#2$, etc.

On the other hand, the set semantic functional dependency $p(x, y) :: \{x\} \rightsquigarrow \{x, y\}$ does not hold since there exists a value for x, namely $x = 1$, which is associated with multiple values for $\{x, y\}$, i.e., $\{1, 2\}$, $\{1, 3\}$, etc. $\square$

This definition is only concerned with *full* functional dependencies, i.e. the domain of the functional dependency determines all other arguments. A more general functional dependency only needs to determine some subset of all arguments greater than the domain, e.g., $p(x, y, z) :: \{x\} \rightsquigarrow \{x, y\}$ is a non-full functional dependency. Currently, we only analyse for full functional dependencies. Some optimisations, such as indexing [4], require full functional dependencies.

Also, our definition differs from the usual mathematical definition of a functional dependency. Strictly speaking, a functional dependency is a relationship between arguments of constraints, namely what arguments determine the values of other arguments. In a *set semantic* functional dependency, we require both a traditional functional dependency, and the requirement that there is at most one copy of a constraint with the same key in the CHR store. For example, the CHR store $\{p(1, 2)\#1, p(1, 2)\#2\}$ has a functional dependency between the first and second arguments of the `p` constraint, because given the value to the first argument we can determine the value of the second. However, there is no set semantic functional dependency, because two constraints with the same first argument simultaneously appear in the store at once. Set semantic functional dependencies are stronger than the traditional functional dependencies. CHR optimisations specifically rely on set semantic functional dependencies, hence the distinction. For brevity, we will often refer "set semantic functional dependencies" as simply "functional dependencies" from now on.

3.1 Examples of Functional Dependency

Example 3. Our first example is based on using CHRs as a queryable data structure. This is a common CHR idiom. It supports insertion and lookups of key-value $(K - V)$ pairs.

```
insert(K,V) <=> entry(K,V).
entry(K,_) \ entry(K,_) <=> true.
entry(K,V) \ lookup(K,V0) <=> V0 = V.
lookup(_,_) <=> fail.
```

The second rule ensures that a functional dependency exists between the key and value for a given `entry` constraint in the store. This means that the key K determines the value V for any `entry(`K`,`V`)` appearing in the store. Essentially, if two entries exist with the same key, then one will be deleted, thus preserving the functional dependency.

The other constraints, namely `insert` and `lookup` are both *never-stored*. This is because these constraints are always deleted by either the first or fourth rule. □

The ad hoc analysis detects all functional dependencies and never-stored constraints for the database program. However there are examples of programs where such information cannot be detected by current implementations.

Example 4. The following CHR program sorts the list $[x_1, x_2, x_3, \ldots, x_n]$ represented as the CHR constraints $link(x_1, x_2), link(x_2, x_3), \ldots, link(x_{n-1}, x_n)$.

```
link(X,Y) <=> X > Y | link(Y,X).
link(X,Y) \ link(X,Z) <=> Y =< Z | link(Y,Z).
link(X,Y) \ link(T,Y) <=> X >= T | link(T,X).
```

For example, the goal $link(9,5), link(5,7), link(7,6), link(6,8)$ representing the list '$[9,5,7,6,8]$' results in $link(5,6), link(6,7), link(7,8), link(8,9)$ representing the list '$[5,6,7,8,9]$'. □

Both the second and third rule enforce functional dependencies between arguments. For example, the second rule enforces that the first argument determines the value of the second in a `link/2` constraint.

Under current implementations of functional dependency analysis, neither functional dependency will be detected. This is because the compiler's analysis is too weak to decide the meaning of inequalities in either guard. One solution is to strengthen the existing analysis by allowing the compiler to reason about inequalities, as is done in [8] to detect redundant code. However, in some systems such as HAL there are no guarantees about the meaning of the inequality predicate (e.g. `=</2` could be an arbitrarily defined user predicate in HAL).

Another solution is for the programmer to add redundant rules to "declare" the functional dependency. In this case the programmer will define the following rules.

```
link(X,_) \ link(X,_) <=> true.
link(_,Y) \ link(_,Y) <=> true.
```

Note that the operational behaviour of the rules are not important, since the programmer never intends for the rules to be executed anyway. The question remains of where to insert the rules into the program such that the rules are never executed. Clearly, the rules must be inserted after the first three rules in Example 4.

Unfortunately, under the ad hoc functional dependency analysis the new rules provide no additional benefit. The reason is because the ad hoc analysis relies on the results of *late storage analysis* [4, 6]. Although rules 4 and 5 are of the appropriate form for functional dependencies, an active `link/2` constraint will have already been stored thanks to rules 2 and 3.

Our new functional dependency analysis is independent of late storage analysis. As a result, we will show that we can still derive benefit from the addition of the new rules for this program.

4 Ad Hoc Functional Dependency Analysis

In this section we give a brief overview of the ad hoc functional dependency analysis in [4].

Essentially, the analysis relies on detecting rules of the following form in the program.

$$p(x_1,\ldots,x_n)[\backslash,]p(y_1,\ldots,y_n)_j \Longleftrightarrow x_{i_0} = y_{i_0} \wedge \ldots \wedge x_{i_m} = y_{i_m} \mid C$$

Here $x_1,\ldots,x_n,y_1,\ldots,y_n$ are distinct variables. For example, the following rule from Example 3 fits the required form.

```
entry(K,_) \ entry(K,_) <=> true.
```

Note that shared variables in the rule head indicate equation guards, e.g., the above rule is equivalent to

```
entry(K1,_) \ entry(K2,_) <=> K1 = K2 | true.
```

If such a rule exists, then generally the functional dependency $p(x_1,\ldots,x_n) :: \{x_{i_0},\ldots,x_{i_m}\} \rightsquigarrow \{x_1,\ldots,x_n\}$ holds. To see this, assume that for a given key, e.g. $K = cat$, there already exists a constraint, say `entry(`cat`,`dog`)` in the store. If a new constraint with the same key, e.g. `entry(`cat`,`pig`)`, becomes active, then we can statically determine that the above mentioned rule will always fire, deleting the new constraint with the same key. Thus we maintain the invariant that only one constraint exists in the store per key

There is a slight complication, the refined semantics requires that the new constraint be immediately stored when it first becomes active, before any rules can fire. This means that until the rule fires, the functional dependency is being violated. Thus the ad hoc functional dependency analysis relies on *late storage analysis* [4, 6], which determines the latest point in the program where a constraint needs to be added into the store. If the rule enforcing the functional dependency appears before this point, then the functional dependency is valid, otherwise the analysis cannot infer any functional dependency.

In addition to functional dependencies, there is an ad hoc *never-stored* analysis [4], which determines if a CHR constraint is never present in the CHR store. This analysis is similar to the previous, except this time we look for rules of the from

$$p(x_1, ..., x_n) \Longleftrightarrow C$$

where $x_1, ..., x_n$ are distinct variables. If such a rule appears before an active `p`/n is stored, then we say that `p`/n is never-stored. For example, `insert` and `lookup` from Example 3 are all never-stored.

5 Accurate Functional Dependency Analysis

In this section we present the more accurate functional dependency analysis for CHRs based on the abstract interpretation framework for CHRs [6]. In particular we formulate our analysis in terms of the denotational semantics-based formulation of the framework [7].

5.1 Abstract Domain $\mathbf{\Sigma_a}$

In this section we describe the abstract domain $\mathbf{\Sigma_a}$ for functional dependency analysis. In our abstraction we preserve information about two components: the goal and the constraint store.

The abstraction of the former retains the necessary information about the program point as required by the abstract interpretation framework. We define function α_{fd-c} that abstracts the goal component of a state.

$$\begin{array}{lll}
\alpha_{fd-c}(\Box) & = \Box & \\
\alpha_{fd-c}(c) & = \texttt{builtin} & (c \text{ built-in}) \\
\alpha_{fd-c}(p(x_1,\ldots,x_n)) & = p & \\
\alpha_{fd-c}(p(x_1,\ldots,x_n)\#i) & = p\# & \\
\alpha_{fd-c}(p(x_1,\ldots,x_n)\#i\!:\!j) & = p\!:\!j & \\
\alpha_{fd-c}([c|G]) & = [\alpha_{fd-c}(c)|\alpha_{fd-c}(G)] &
\end{array}$$

The abstraction of the latter captures patterns in the constraint store S that are essential for the analysis. Therefore, the abstract store is a set of *lookups*, which are defined by the following *lookup function.*

Definition 3 (Lookup function). *We define the* lookup function $\mathsf{lookup}(p,n,K)$, *where* p *is a predicate symbol,* n *is the arity of* p *and* $K \subseteq \{1,\ldots,n\}$ *a set of integers, as follows.*

$$\begin{array}{lll}
\mathsf{lookup}(p,n,K) & = p(\mathsf{lookup}(1,K),\ldots,\mathsf{lookup}(n,K)) & \\
\mathsf{lookup}(i,K) & = * & i \in K \\
\mathsf{lookup}(i,K) & = \_ & i \notin K
\end{array}$$

Consider the constraint $\texttt{p}/2$, then the set of possible lookups are

$$\{p(*,*), p(*,\_), p(\_,*), p(\_,\_)\}$$

Note that $\mathsf{lookup}(p,n,K)$ is isomorphic to K, and its usage is mainly syntactic. The concept of a *lookup* will become relevant to CHR optimisation. The set of arguments represented by the $*$s are referred to as the *key* of the lookup.

We can now define an abstraction function α_{fd-S} over the CHR store.

Definition 4. *Let* S *be a CHR store, and let* $\texttt{p}/n$ *be the functor/arity of a CHR constraint of interest (e.g. any CHR constraint appearing in program* P*). Let* $K \subseteq \{1,\ldots,n\}$ *be a set of integers, let* $c = \mathsf{lookup}(p,n,K)$*, let count* $= \mathsf{count}(p,n,K,S,B)$ *and let count'* $= \mathsf{count}(p,n,K,S',B)$ *where* S' *is defined as follows. Let* $p(x_1,\ldots,x_n)\#i\!:\!j$ *be the first active constraint in* A *with predicate* p*, then* $S' = S - \{p(x_1,\ldots,x_n)\#i\}$*, otherwise (if no such active constraint exists)* $S' = S$*. Then*

$$\begin{array}{lll}
\mathit{count} = 0 & \Rightarrow & c^0 \in \alpha_{fd-S}(A,S,B) \\
\mathit{count} = 1 \wedge \mathit{count}' = 0 & \Rightarrow & c^{1a} \in \alpha_{fd-S}(A,S,B) \\
\mathit{count} = 1 \wedge \mathit{count}' = 1 & \Rightarrow & c^{1} \in \alpha_{fd-S}(A,S,B) \\
\mathit{count} = 2 \wedge \mathit{count}' = 1 & \Rightarrow & c^{2a} \in \alpha_{fd-S}(A,S,B) \\
\mathit{count} = 2 \wedge \mathit{count}' = 2 & \Rightarrow & c^{*} \in \alpha_{fd-S}(A,S,B) \\
\mathit{count} \geq 2 & \Rightarrow & c^{*} \in \alpha_{fd-S}(A,S,B)
\end{array}$$

We define $\alpha_{fd-S}(A, S, B)$ to be the smallest possible set satisfying the above conditions.

We refer the superscript associated with each lookup as the *counter* for that lookup. The counters 0, 1 and $*$ are fairly intuitive, as they mean that there is at most 0, 1 or many constraints in the CHR store with the same key as the lookup. The special counters, $1a$ and $2a$ are slightly more complicated. The counter $1a$ is equivalent to 1 except that if we were to remove the top-most active constraint of the same functor/arity from the concrete store, then the new counter will be 0. Similarly, the counter $2a$ is equivalent to 2 (although we treat 2 the same as $*$), but if the top-most active constraint were to be removed, then the counter will be 1. We shall refer to these special counters as *marked counters*, and other counters as *unmarked counters*. Marked counters are necessary since functional dependency analysis relies on improving the counters (i.e. moving to a lower counter) if possible.

Example 5. Consider the following CHR store from the `database` program from Example 3.

$$S = \{\texttt{entry}(key, cat)\#2, \texttt{entry}(key, dog)\#1\}$$

Assume that the built-in store is trivial, i.e. $B = true$, and the given execution stack is $A = [\texttt{entry}(key, cat)\#2{:}1]$, then

$$S' = \{\texttt{entry}(key, dog)\#1\}$$

hence

$$\alpha_{fd-S}(A, S, B) = \{\texttt{entry}(*, *)^1, \texttt{entry}(*, \_)^{2a}, \texttt{entry}(\_, *)^1, \texttt{entry}(\_, \_)^{2a}\}$$

Both `entry`$(*, *)$ and `entry`$(\_, *)$ have the counter 1, since there is at most one `entry` that shares the same key $\{1, 2\}$ or $\{2\}$. One the other hand, both `entry`$(*, \_)$ and `entry`$(\_, \_)$ are have the marked counter $2a$. This is because there are two `entry` constraints that share the same key $\{1\}$ or $\emptyset$, however there would be only one such constraint if we were to remove `entry`(key, cat) (the current active constraint) from consideration.

If the given activation stack had been empty, i.e. $A = []$, then $S = S'$ hence

$$\alpha_{fd-S}(A, S, B) = \{\texttt{entry}(*, *)^1, \texttt{entry}(*, \_)^{*}, \texttt{entry}(\_, *)^1, \texttt{entry}(\_, \_)^{*}\}$$

There are still 2 constraints sharing the same key $\{1\}$ or $\emptyset$. However there is no current active constraint to remove from consideration, thus the counter for `entry`$(*, \_)$ and `entry`$(\_, \_)$ is now $*$. □

Example 6. Consider the following execution state σ for the `database` program in Example 3.

$$\langle \texttt{entry}(key, cat)\#2{:}1, [], \{\texttt{entry}(key, cat)\#2, \texttt{entry}(key, dog)\#1\} true, \emptyset\rangle_3$$

The CHR and built-in stores are the same as in Example 5. Then

$$\alpha_{fd}(\sigma) = \langle \texttt{entry}{:}1, \{\texttt{entry}(*, *)^1, \texttt{entry}(*, \_)^{2a}, \texttt{entry}(\_, *)^1, \texttt{entry}(\_, \_)^{*}\}\rangle$$

□

Finally, abstraction function α_{fd} that maps concrete states $\sigma = \langle c, A, S, B, T\rangle_n$ of the concrete domain $\mathbf{\Sigma}$ onto abstract states.

Definition 5 (Abstraction Function). *We define function α_{fd} as follows*

$$\alpha_{fd}(\langle c, A, S, B, T\rangle_n) = \langle \alpha_{fd-c}(c), \alpha_{fd-S}([c|A], S, B)\rangle$$

For any given abstract state σ we can easily determine which functional dependencies exist by interpreting the counts on the lookups. In general, a lookup with a count of 1 represents a functional dependency. For example, the lookup $\mathtt{p}(*, \_)^1$ indicates the functional dependency $p(x, y) :: \{x\} \rightsquigarrow \{x, y\}$ holds for the given abstract state, and hence all corresponding program points.

The abstract interpretation framework requires a partial order to be defined over abstract states.

Definition 6 (Partial Ordering). *Let $s_0 = \langle G_0, S_0\rangle$ and $s_1 = \langle G_1, S_1\rangle$, then $s_0 \preceq_{fd} s_1$ iff $G_0 = G_1$ and $S_0 \preceq_{fd_S} S_1$. The partial order $\preceq_{fd_S}$ over abstract CHR stores is defined as follows. If for all $c^n \in S_0$, there exists a $c^m \in S_1$ (with the same c), and $c^n \preceq c^m$, then $S_0 \preceq_{fd_S} S_1$. Here we define*

$$c^0 \prec c^{1a} \prec c^1 \prec c^{2a} \prec c^*$$

if S_0 and S_1 contain different lookups, $\preceq_{fd_S}$ is undefined .[3]

We can use the definition of the partial ordering to define the concretisation function of this abstract domain as $\gamma_{fd}(s) = \{\sigma \mid \alpha_{fd}(\sigma) \preceq_{fd} s\}$. For the sake of completeness[4], we add a top element $\top_{fd}$ to the abstract domain, with $\gamma(\top_{fd}) = \mathbf{\Sigma}$ and $\forall s \in \mathbf{\Sigma_a} : s \preceq_{fd} \top_{fd}$.

Clearly the abstract domain forms a lattice with the ordering relation $\preceq_{fd}$. The least upper bound operation over abstract stores is defined as follows.

Definition 7 (Least Upper Bound). *Let $s_0 = \langle G_0, S_0\rangle$ and $s_1 = \langle G_1, S_1\rangle$, then $s_0 \sqcup_{fd} s_1 = \langle G_0, S_0 \sqcup_{fd-S} S_1\rangle$ if $G_0 = G_1$ and S_0 and S_1 contain the same lookups, otherwise it is $\top_{fd}$. The operator $\sqcup_{fd-S}$ over abstract CHR stores is defined as follows. If for all $c^n \in S_0$, there exists a $c^m \in S_1$ (with the same c), then $\mathsf{max}_{\preceq_{fd-S}}(c^n, c^m) \in S_0 \sqcup_{fd-S} S_1$. The set $S_0 \sqcup_{fd-S} S_1$ must be the minimal set satisfying the above condition. Here, function $\mathsf{max}_{\preceq_{fd-S}}$ is a maximum function using the ordering $\preceq_{fd-S}$ given in Definition 6. Otherwise $\sqcup_{fd-S}$ is undefined if S_0 and S_1 contain different lookups.*

5.2 Abstract Semantic Function $\mathcal{AS}$

The abstract semantic function $\mathcal{AS}$ for the functional dependency analysis is defined below.

[3] In the abstract interpretation, S_0 and S_1 will always have the same set of lookups.
[4] Our analysis never produces $\top_{fd}$.

Definition 8 (Abstract Semantic Function).
1. $\mathsf{AbstractSolve}$

$$\mathcal{AS}[\![\mathcal{P}]\!](\langle \mathtt{builtin}, S\rangle) = \langle \Box, S_k\rangle$$

Let S_g be the maximal subset of S such that for all $p(x_1, \ldots, x_n)^c \in S_g$ we have that p is ground[5]. Let $S_{ng} = S \setminus S_g$, then

$$\begin{aligned} &S_0 = S_g \uplus \mathsf{multi}(S_{ng}) \\ &s_j = \langle \Box, S_j\rangle = \sqcup_{fd}\{s_j^i |\ \mathcal{AS}[\![\mathcal{P}]\!](\langle p_i\#, S_{j-1}\rangle) = s_j^i \wedge 1 \leq i \leq n\}, j \geq 1 \end{aligned}$$

where p_i are predicates of all potentially nonground constraints. Let k be the smallest positive integer such that $s_k = s_{k-1}$.

Adding a built-in constraint, e.g. an equation to the built-in store has the potential to increase the counts of lookups for nonground constraints arbitrarily. Therefore, we use function multi on the nonground lookups, which overwrites any count by $*$.

$$\mathsf{multi}(S) = \{c^* \mid c^i \in S\}$$

In effect, we are assuming the weakest possible information for these lookups.
2. $\mathsf{AbstractActivate}$

$$\mathcal{AS}[\![\mathcal{P}]\!](\langle p, S\rangle) = \langle p\!:\!1, \mathsf{increase}(p, S)\rangle$$

A new constraint is added to the store, hence we must *increase* the counts for p. Here we define

$$\mathsf{increase}(p, S) = \left\{ c' \;\middle|\; c \in S \wedge \begin{array}{l} \text{if } \mathsf{functor}(c) = p \text{ then } c' = \mathsf{increase}(c) \\ \text{else } c' = c \end{array} \right\}$$

where

$$\begin{array}{lll} \mathsf{increase}(c^0) \ = c^{1a} & \mathsf{increase}(c^1) \ = c^{2a} & \mathsf{increase}(c^*) = c^* \\ \mathsf{increase}(c^{1a}) = c^{2a} & \mathsf{increase}(c^{2a}) = c^* & \end{array}$$

Notice that the resulting counts are always marked (except for $*$).
3. $\mathsf{AbstractReactivate}$

$$\mathcal{AS}[\![\mathcal{P}]\!](\langle p\#, S\rangle) = \langle p\!:\!1, S\rangle$$

Unlike $\mathsf{AbstractActivate}$, the new active constraint is already present in the store, hence there is no need to call function $\mathsf{increase}$ on the abstract store.
4. $\mathsf{AbstractDrop}$

$$\mathcal{AS}[\![\mathcal{P}]\!](\langle p\!:\!j, S\rangle) = \langle \Box, \mathsf{unmark}(p, S)\rangle$$

There is no occurrence j for predicate p.

Because the active constraint for p no longer exists, we must *unmark* all of the counters for p.

$$\mathsf{unmark}(p, S) = \left\{ c' \;\middle|\; c \in S \wedge \begin{array}{l} \text{if } \mathsf{functor}(c) = p \text{ then } c' = \mathsf{unmark}(c) \\ \text{else } c' = c \end{array} \right\}$$

[5] This information may either be derived through groundness analysis [6] or mode declarations.

where

$$\begin{array}{lll} \mathsf{unmark}(c^0) \ = c^0 & \mathsf{unmark}(c^1) \ = c^1 & \mathsf{unmark}(c^*) = c^* \\ \mathsf{unmark}(c^{1a}) = c^1 & \mathsf{unmark}(c^{2a}) = c^* & \end{array}$$

In effect, the (former) active constraint is now treated the same as any other constraint in the store.

5. AbstractSimplify

$$\mathcal{AS}[\![\mathcal{P}]\!](\langle p\!:\!j, S\rangle) = s$$

Let r be the rule which contains the j^{th} occurrence of predicate p. If we assume that the rule fired, then

$$\mathcal{AS}[\![\mathcal{P}]\!](\langle \alpha_{fd-c}(C), \mathsf{decrease}(p, S)\rangle) = \langle \Box, S_1\rangle = s_1$$

Because the active constraint has been deleted, we must *decrease* all of the counters for p.

$$\mathsf{decrease}(p, S) = \left\{ c' \;\middle|\; c \in S \wedge \begin{array}{l} \text{if } \mathsf{functor}(c) = p \text{ then } c' = \mathsf{decrease}(c) \\ \text{else } c' = c \end{array} \right\}$$

where

$$\begin{array}{lll} \mathsf{decrease}(c^0) \ = c^0 & \mathsf{decrease}(c^1) \ = c^1 & \mathsf{decrease}(c^*) = c^* \\ \mathsf{decrease}(c^{1a}) = c^0 & \mathsf{decrease}(c^{2a}) = c^1 & \end{array}$$

Note that we can only alter the *marked* counters. Also, the resulting counts are unmarked, since the active constraint has been deleted.

We consider the following three cases for deriving the resulting state s.

1. If rule r is of the form

$$p(x_1, \ldots, x_n)[\backslash,]p(y_1, \ldots, y_n)_j \Longleftrightarrow x_{i_0} = y_{i_0} \wedge \ldots \wedge x_{i_m} = y_{i_m} \mid C$$

(where occurrence j is shown).
If we assume the rule did not fire, then the resulting state is

$$\mathcal{AS}[\![\mathcal{P}]\!](\langle p\!:\!(j+1), \mathsf{decrease\_2a}(p, n, \{i_0, \ldots, i_m\}, S)\rangle) = s_2$$

We use function decrease_2a to improve the counts of lookups which contain key $\{i_0, \ldots, i_m\}$, where

$$\mathsf{decrease\_2a}(p, n, K, S) = \left\{ c' \;\middle|\; \begin{array}{l} c \in S \wedge \\ \text{if } \exists K' \supseteq K \text{ such that } c = \mathsf{lookup}(p, n, K') \\ \quad \text{then } c' = \mathsf{decrease\_2a}(c) \\ \text{else } c' = c \end{array} \right\}$$

and

$$\begin{array}{lll} \mathsf{decrease\_2a}(c^0) \ = c^0 & \mathsf{decrease\_2a}(c^1) \ = c^1 & \mathsf{decrease\_2a}(c^*) = c^* \\ \mathsf{decrease\_2a}(c^{1a}) = c^{1a} & \mathsf{decrease\_2a}(c^{2a}) = c^1 & \end{array}$$

We can make this improvement since if c^{2a} were in the abstract store for some lookup c with a key containing $\{i_0, \ldots, i_m\}$, then the rule *must* have fired. This kind of improvement is the essential part of functional dependency analysis.

For the resulting state we have $s = s_1 \sqcup_{fd} s_2$.

2. If rule r is an unconditional simplification rule (i.e. the guard is *true*) of the form

$$p(x_1, \ldots, x_n)_j \Longleftrightarrow C$$

then $s = s_1$.

3. Otherwise (r is not in any of the above forms), then

$$s = s_1 \sqcup_{fd} \mathcal{AS}[\![\mathcal{P}]\!](\langle p\colon(j+1), S\rangle)$$

6. AbstractPropagate

$$\mathcal{AS}[\![\mathcal{P}]\!](\langle p\colon j, S\rangle) = \mathcal{AS}[\![\mathcal{P}]\!](\langle p\colon(j+1), S_k\rangle)$$

1. If the following conditions hold:
   - Rule r is of the form

$$p(x_1, \ldots, x_n)_j[\backslash,]p(y_1, \ldots, y_n) \Longleftrightarrow x_{i_0} = y_{i_0} \wedge \ldots \wedge x_{i_m} = y_{i_m} \wedge g \mid C$$

   (where occurrence j is shown). Here g represents an arbitrary guard.[6]
   - For $c = lookup(p, n, \{i_0, ..., i_m\})$ we have that $c^{2a} \in S$.

   Then

$$\mathcal{AS}[\![\mathcal{P}]\!](\langle \alpha_{fd-c}(C), \mathsf{decrease\_2a}(p, n, \{i_0, \ldots, i_m\}, S)\rangle) = \langle \Box, S_k\rangle$$

   defines S_k.

2. Otherwise, let

$$\begin{aligned} &S_0 = S \\ &s_j = \langle \Box, S_j\rangle = \mathcal{AS}[\![\mathcal{P}]\!](\langle \alpha_{fd-c}(C), S_{j-1}\rangle) \end{aligned}$$

   and let k be the smallest positive integer such that $s_k = s_{k-1}$.

7. AbstractGoal

$$\begin{aligned} \mathcal{AS}[\![\mathcal{P}]\!](\langle \Box, S\rangle) &= \langle \Box, S'\rangle \\ \mathcal{AS}[\![\mathcal{P}]\!](\langle [c|G], S\rangle) &= \mathcal{AS}[\![\mathcal{P}]\!](\langle G, S'\rangle) \end{aligned}$$

where

$$\mathcal{AS}[\![\mathcal{P}]\!](\langle c, S\rangle) = \langle \Box, S'\rangle$$

[6] Ideally g contains (nor entails) no equations of the from $x_k = y_k$ for $1 \leq k \leq n$. If g does entail such an equation, then the analysis will be weaker than it could be.

5.3 Example Analysis

Consider the following rule on the `entry` constraint from Example 3.

```
entry(K,_) \ entry(K,_) <=> true.
```

The abstract derivations for executing a single `entry` constraint are shown in Figure 1. For brevity, we have abbreviated `entry` to `e`, and we have omitted the subcomputations for AbstractPropagate. In each instance, the AbstractPropagate does not change the abstract store. We are assuming that the `entry` constraint is a ground constraint.

After three iterations we arrive at a fixed point for the final state. The resulting abstract store is

$$\{\texttt{entry}(*,*)^1, \texttt{entry}(*,\_)^1, \texttt{entry}(\_,*)^*, \texttt{entry}(\_,*)^*\}$$

This indicates that after an `entry` constraint has finished being active, the set semantic functional dependencies $\texttt{entry}(X,Y) :: \{X\} \rightsquigarrow \{X,Y\}$ and $\texttt{entry}(X,Y) :: \{X,Y\} \rightsquigarrow \{X,Y\}$ hold.

6 Implementation and Evaluation

Information about functional dependencies and never-stored is used in further program analysis (e.g. confluence analysis [2]) and for join ordering, index selection and never-stored optimisations. We have implemented a simple prototype functional dependency analysis in SWI-Prolog, and use the results of the analysis to improve the compilation of HAL CHR programs. Note that currently the analysis is not integrated in the HAL compiler itself, so (in some cases) the optimisations have been performed by hand. Full integration remains future work.

In Table 1 we compare several CHR programs with/without functional dependency (and never-stored) optimisations. The source code and description of the programs can be downloaded from `http://www.cs.mu.oz.au/~gjd/chr05/`. The $-fd$ ($+fd$) version has the optimisation disabled (enabled). Overall we see a 17% improvement for the $+fd$ version, which shows that functional dependency analysis is beneficial.

For most of the programs the ad hoc analysis gives equivalent results to the new analysis. The exceptions are the `sort` example (as explained in Section 3.1) and the `game` program, where one of the CHR constraints is locally never-stored for some occurrences (rather than all occurrences as required by the ad hoc analysis). In both of these programs, the ad hoc analysis is equivalent to the $-fd$ no analysis version. This shows that the new analysis is an improvement over the ad hoc version.

7 Conclusion

In this paper we have replaced the old ad hoc functional dependency analysis of [4] with a more powerful analysis. We have provided a full formal specification of

$$\begin{aligned}
&\mathcal{AS}[\![\mathcal{P}]\!](\langle \mathsf{e}, \{\mathsf{e}(*,*)^0, \mathsf{e}(*,\_)^0, \mathsf{e}(\_,*)^0, \mathsf{e}(\_,\_)^0\}\rangle)\\
&\qquad = \mathcal{AS}[\![\mathcal{P}]\!](\langle \mathsf{e}\!:\!1, \{\mathsf{e}(*,*)^{1a}, \mathsf{e}(*,\_)^{1a}, \mathsf{e}(\_,*)^{1a}\mathsf{e}(\_,\_)^{1a}\}\rangle) && (\textbf{Activate})\\
&\qquad = \mathcal{AS}[\![\mathcal{P}]\!](\langle \square, \{\mathsf{e}(*,*)^0, \mathsf{e}(*,\_)^0, \mathsf{e}(\_,*)^0\mathsf{e}(\_,\_)^0\}\rangle)\\
&\qquad\quad \sqcup_{fd} \mathcal{AS}[\![\mathcal{P}]\!](\langle \mathsf{e}\!:\!2, \{\mathsf{e}(*,*)^{1a}, \mathsf{e}(*,\_)^{1a}, \mathsf{e}(\_,*)^{1a}\mathsf{e}(\_,\_)^{1a}\}\rangle) && (\textbf{Simplify})\\
&\qquad = \langle \square, \{\mathsf{e}(*,*)^0, \mathsf{e}(*,\_)^0, \mathsf{e}(\_,*)^0\mathsf{e}(\_,\_)^0\}\rangle \sqcup_{fd} \langle \square, \{\mathsf{e}(*,*)^1, \mathsf{e}(*,\_)^1, \mathsf{e}(\_,*)^1\mathsf{e}(\_,\_)^1\}\rangle\\
&\qquad = \langle \square, \{\mathsf{e}(*,*)^1, \mathsf{e}(*,\_)^1, \mathsf{e}(\_,*)^1\mathsf{e}(\_,\_)^1\}\rangle
\end{aligned}$$

$$\begin{aligned}
&\mathcal{AS}[\![\mathcal{P}]\!](\langle \mathsf{e}\!:\!2, \{\mathsf{e}(*,*)^{1a}, \mathsf{e}(*,\_)^{1a}, \mathsf{e}(\_,*)^{1a}\mathsf{e}(\_,\_)^{1a}\}\rangle)\\
&\qquad = \mathcal{AS}[\![\mathcal{P}]\!](\langle \mathsf{e}\!:\!3, \{\mathsf{e}(*,*)^{1a}, \mathsf{e}(*,\_)^{1a}, \mathsf{e}(\_,*)^{1a}\mathsf{e}(\_,\_)^{1a}\}\rangle) && (\textbf{Propagate})\\
&\qquad = \langle \square, \{\mathsf{e}(*,*)^1, \mathsf{e}(*,\_)^1, \mathsf{e}(\_,*)^1\mathsf{e}(\_,\_)^1\}\rangle && (\textbf{Drop})
\end{aligned}$$

$$\begin{aligned}
&\mathcal{AS}[\![\mathcal{P}]\!](\langle \mathsf{e}, \{\mathsf{e}(*,*)^1, \mathsf{e}(*,\_)^1, \mathsf{e}(\_,*)^1, \mathsf{e}(\_,\_)^1\}\rangle)\\
&\qquad = \mathcal{AS}[\![\mathcal{P}]\!](\langle \mathsf{e}\!:\!1, \{\mathsf{e}(*,*)^{2a}, \mathsf{e}(*,\_)^{2a}, \mathsf{e}(\_,*)^{2a}\mathsf{e}(\_,\_)^{2a}\}\rangle) && (\textbf{Activate})\\
&\qquad = \mathcal{AS}[\![\mathcal{P}]\!](\langle \square, \{\mathsf{e}(*,*)^1, \mathsf{e}(*,\_)^1, \mathsf{e}(\_,*)^1\mathsf{e}(\_,\_)^1\}\rangle)\\
&\qquad\quad \sqcup_{fd} \mathcal{AS}[\![\mathcal{P}]\!](\langle \mathsf{e}\!:\!2, \{\mathsf{e}(*,*)^1, \mathsf{e}(*,\_)^1, \mathsf{e}(\_,*)^{2a}\mathsf{e}(\_,\_)^{2a}\}\rangle) && (\textbf{Simplify})\\
&\qquad = \langle \square, \{\mathsf{e}(*,*)^1, \mathsf{e}(*,\_)^1, \mathsf{e}(\_,*)^1\mathsf{e}(\_,\_)^1\}\rangle \sqcup_{fd} \langle \square, \{\mathsf{e}(*,*)^1, \mathsf{e}(*,\_)^1, \mathsf{e}(\_,*)^*\mathsf{e}(\_,\_)^*\}\rangle\\
&\qquad = \langle \square, \{\mathsf{e}(*,*)^1, \mathsf{e}(*,\_)^1, \mathsf{e}(\_,*)^*\mathsf{e}(\_,\_)^*\}\rangle
\end{aligned}$$

$$\begin{aligned}
&\mathcal{AS}[\![\mathcal{P}]\!](\langle \mathsf{e}\!:\!2, \{\mathsf{e}(*,*)^1, \mathsf{e}(*,\_)^1, \mathsf{e}(\_,*)^{2a}\mathsf{e}(\_,\_)^{2a}\}\rangle)\\
&\qquad = \mathcal{AS}[\![\mathcal{P}]\!](\langle \mathsf{e}\!:\!3, \{\mathsf{e}(*,*)^1, \mathsf{e}(*,\_)^1, \mathsf{e}(\_,*)^{2a}\mathsf{e}(\_,\_)^{2a}\}\rangle) && (\textbf{Propagate})\\
&\qquad = \langle \square, \{\mathsf{e}(*,*)^1, \mathsf{e}(*,\_)^1, \mathsf{e}(\_,*)^*\mathsf{e}(\_,\_)^*\}\rangle && (\textbf{Drop})
\end{aligned}$$

$$\begin{aligned}
&\mathcal{AS}[\![\mathcal{P}]\!](\langle \mathsf{e}, \{\mathsf{e}(*,*)^1, \mathsf{e}(*,\_)^1, \mathsf{e}(\_,*)^*, \mathsf{e}(\_,\_)^*\}\rangle)\\
&\qquad = \mathcal{AS}[\![\mathcal{P}]\!](\langle \mathsf{e}\!:\!1, \{\mathsf{e}(*,*)^{2a}, \mathsf{e}(*,\_)^{2a}, \mathsf{e}(\_,*)^*\mathsf{e}(\_,\_)^*\}\rangle) && (\textbf{Activate})\\
&\qquad = \mathcal{AS}[\![\mathcal{P}]\!](\langle \square, \{\mathsf{e}(*,*)^1, \mathsf{e}(*,\_)^1, \mathsf{e}(\_,*)^*\mathsf{e}(\_,\_)^*\}\rangle)\\
&\qquad\quad \sqcup_{fd} \mathcal{AS}[\![\mathcal{P}]\!](\langle \mathsf{e}\!:\!2, \{\mathsf{e}(*,*)^1, \mathsf{e}(*,\_)^1, \mathsf{e}(\_,*)^*\mathsf{e}(\_,\_)^*\}\rangle) && (\textbf{Simplify})\\
&\qquad = \langle \square, \{\mathsf{e}(*,*)^1, \mathsf{e}(*,\_)^1, \mathsf{e}(\_,*)^*\mathsf{e}(\_,\_)^*\}\rangle \sqcup_{fd} \langle \square, \{\mathsf{e}(*,*)^1, \mathsf{e}(*,\_)^1, \mathsf{e}(\_,*)^*\mathsf{e}(\_,\_)^*\}\rangle\\
&\qquad = \langle \square, \{\mathsf{e}(*,*)^1, \mathsf{e}(*,\_)^1, \mathsf{e}(\_,*)^*\mathsf{e}(\_,\_)^*\}\rangle
\end{aligned}$$

$$\begin{aligned}
&\mathcal{AS}[\![\mathcal{P}]\!](\langle \mathsf{e}\!:\!2, \{\mathsf{e}(*,*)^1, \mathsf{e}(*,\_)^1, \mathsf{e}(\_,*)^*\mathsf{e}(\_,\_)^*\}\rangle)\\
&\qquad = \mathcal{AS}[\![\mathcal{P}]\!](\langle \mathsf{e}\!:\!3, \{\mathsf{e}(*,*)^1, \mathsf{e}(*,\_)^1, \mathsf{e}(\_,*)^*\mathsf{e}(\_,\_)^*\}\rangle) && (\textbf{Propagate})\\
&\qquad = \langle \square, \{\mathsf{e}(*,*)^1, \mathsf{e}(*,\_)^1, \mathsf{e}(\_,*)^*\mathsf{e}(\_,\_)^*\}\rangle && (\textbf{Drop})
\end{aligned}$$

Fig. 1. Example abstract execution

Table 1. Timings (ms) showing the benefit of functional dependency based optimisations

Program	$-fd$	$+fd$
`database(100000,5)`	1694	1518
`union(160)`	1583	853
`cycle(14)`	664	595
`queue(100000)`	1204	1006
`queens(15)`	1117	914
`game(30000)`	1887	1761
`sort(100000)`	2624	2486
geom. mean	1424	83%

our analysis, in terms of an instantiation of the abstract interpretation framework for CHR of [6]. Even though the formulation of our analysis is relatively simple, evaluation shows that fairly strong results are obtained.

7.1 Future Work

In future work we would like to investigate a number of improvements of the functional dependency analysis, on account of both its efficiency and strength. Firstly, we would like to experiment with more compact representations of the abstract domain. If $\mathsf{lookup}(p, n, K)^i \in S$ with S an abstract constraint store, then $\forall K \subset K' : \mathsf{lookup}(p, n, K')^j \in S \wedge j \leq i$. If $i = j$ for some K, K', then we could simply omit $\mathsf{lookup}(p, n, K')^j$ from S. When looking for the counter of a K', we take $\min_{K \subset K'} \mathsf{lookup}(p, n, K)$.

Secondly, there is some information in the abstract domain that is currently not exploited in the abstract transition function. Namely, if the counter of a particular $\mathsf{lookup}(p, n, K)^j$ is smaller than the number of copies of a constraints p/n in a CHR rule, then this rule cannot be applied. This provides richer control flow information.

Thirdly, the approach of reasoning about guards, as done in [8], could prove a useful improvement to the current analysis domain.

In addition we would also like to combine the functional dependency analysis with other analyses formulated in terms of the abstract interpretation framework. In particular, the late storage analysis of [6] provides relevant information concerning the storage of the active constraint. The other way around, the functional dependency abstract domain provides useful information about the constraint store and the number of candidates for the active constraint.

References

1. Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unifed Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In

POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, pages 238–252, Los Angeles, California, 1977. ACM Press.

2. G. Duck, P. Stuckey, M. Garcia de la Banda, and C. Holzbaur. The refined operational semantics of constraint handling rules. In *20th International Conference on Logic Programming (ICLP'04)*, pages 90–104, Saint-Malo, France, September 2004.
3. C. Holzbaur, M. Garcia de la Banda, P. Stuckey, and G. Duck. Optimizing Compilation of Constraint Handling Rules in HAL. *Special Issue of Theory and Practice of Logic Programming on Constraint Handling Rules*, 2005. To appear.
4. C. Holzbaur, P. Stuckey, M. Garcia de la Banda, and D. Jeffery. Optimizing compilation of constraint handling rules. In P. Codognet, editor, *Logic Programming: Proceedings of the 17th International Conference*, LNCS, pages 74–89. Springer-Verlag, 2001.
5. T. Schrijvers and B. Demoen. The K.U.Leuven CHR system: Implementation and application. In *First workshop on constraint handling rules: selected contributions*, 2004. Published as technical report: Ulmer Informatik-Berichte Nr. 2004-01, ISSN 0939-5091, http://www/informatik.uni-ulm.de/epin/pw/10481.
6. T. Schrijvers, P. Stuckey, and G. Duck. Abstract Interpretation for Constraint Handling Rules. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, 2005. (to appear).
7. Tom Schrijvers. *Analyses, Optimizations and Extensions of Constraint Handling Rules*. PhD thesis, K.U.Leuven, Leuven, Belgium, June 2005.
8. Jon Sneyers, Tom Schrijvers, and Bart Demoen. Guard reasoning for CHR optimization. Report CW 411, Department of Computer Science, K.U.Leuven, Leuven, Belgium, May 2005.