Cadmium: An Implementation of ACD Term Rewriting

Gregory J. Duck[1], Leslie De Koninck[2] *, and Peter J. Stuckey[1]

[1] National ICT Australia (NICTA) **
Department of Computer Science and Software Engineering
University of Melbourne
{gjd,pjs}@cs.mu.oz.au
[2] Department of Computer Science, K.U.Leuven, Belgium
Leslie.DeKoninck@cs.kuleuven.be

Abstract. Cadmium is a rule based programming language for compiling solver independent constraint models to various solver dependent back-ends. Cadmium is based on a hybrid between Constraint Handling Rules (CHR) and term rewriting modulo Associativity, Commutativity and a restricted form of Distributivity (ACD) called Conjunctive Context (CC). Experience with using Cadmium in the G12 project shows that CC is a powerful language feature, as local model mapping can depend on some non-local context, such as variable declarations or other constraints. However, CC significantly complicates the Cadmium normalisation algorithm, since the normal form of a term may depend on what context it appears in. In this paper we present an implementation of Cadmium based on classic bottom-up evaluation, but modified to handle CC matching. We evaluate the performance of the new implementation compared to earlier prototype normalisation algorithms. We show that the resulting system is fast enough to run "real-world" Cadmium applications.

1 Introduction

Cadmium is high-level rule based programming language based on ACD Term Rewriting (ACDTR) [4] – a generalisation of Constraint Handling Rules (CHR) [5] and Associative Commutative (AC) term rewriting systems [1]. Cadmium's main application is the G12 project [10], where it is used to map high-level models of satisfaction and optimisation problems to low-level executable models. The flexibility and expressiveness of Cadmium allows us to map the same high-level model to different low-level models with very succinct programs (see e.g. [6, 2]).

* Research funded by a Ph.D. grant of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen)

** NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

Associative Commutative (AC) term rewriting allows implicit reordering of AC operators before applying rules. An AC operator $\oplus$ satisfies the axioms:

$$\begin{array}{rc} (\mathit{associativity}) & (X \oplus Y) \oplus Z = X \oplus (Y \oplus Z) \\ (\mathit{commutativity}) & X \oplus Y = Y \oplus X \end{array}$$

For example, the rewrite rule $r = (X \wedge \neg X \ \rightarrow \ false)$ will not match the term $T = \neg a \wedge a$ under non-AC term rewriting because the order of the conjunction is different. However, since $\wedge$ is *commutative*, T is equivalent to $T' = a \wedge \neg a$. Since $T \equiv_{AC} T'$ and T' matches r (i.e. AC matching), T can be rewritten to $false$ under AC term rewriting.

ACD term rewriting [4] extends AC term rewriting with $\wedge$-Distributivity using the following axiom for all functors f:

$$(\mathit{distribution}) \ \ P \wedge f(Q_1, \ldots, Q_i, \ldots Q_n) \ = \ P \wedge f(Q_1, \ldots, P \wedge Q_i, \ldots, Q_n)$$

It represents the fact that if some property P holds in the context of a term $f(Q_1, \ldots, Q_i, \ldots Q_n)$ it also holds in the context of all the subterms. We can then define the *conjunctive context* (CC)[3] of a term T as the conjunction of all terms that appear conjoined with a parent of that term, i.e. all terms that can $\wedge$-distribute to T.

Example 1. The CC of the boxed occurrence of x in the term

$$(x = 3) \wedge (x^2 > y \ \vee \ (\boxed{x} = 4) \wedge U \ \vee \ V) \wedge W$$

is $(x = 3) \wedge U \wedge W$. □

We introduce CC matching rules of the form $(C \setminus H \iff B)$ which say we can rewrite the term H to B if H appears in a position where its conjunctive context is $C \wedge D$ for some D. Thus we can match on any term appearing in the conjunctive context of H.

Example 2. For example, CC matching can be used to specialise constraints based on variable types.

$$\begin{array}{rc} \mathit{int}(X) \wedge \mathit{int}(Y) \ \setminus \ X \leq Y & \iff \mathit{intleq}(X, Y) \\ \mathit{real}(X) \wedge \mathit{real}(Y) \ \setminus \ X \leq Y & \iff \mathit{realleq}(X, Y) \\ \mathit{pair}(X, A, B) \wedge \mathit{pair}(Y, C, D) \ \setminus \ X \leq Y & \iff (A \leq C \vee (A = C \wedge B \leq D)) \end{array}$$

Given the term $x \leq y \wedge \mathit{int}(b) \wedge \mathit{int}(d) \wedge \mathit{pair}(x, a, b) \wedge \mathit{pair}(y, c, d)$ the conjunctive context of $x \leq y$ is the remainder of the conjunction. Therefore the term $\mathit{pair}(x, a, b) \wedge \mathit{pair}(y, c, d)$ appears in the conjunctive context of $x \leq y$ so the last rule is applicable. In the resulting term, $x \leq y$ is replaced by the right hand side of the rule obtaining:

$$\mathit{int}(b) \wedge \mathit{int}(d) \wedge \mathit{pair}(x, a, b) \wedge \mathit{pair}(y, c, d) \wedge (a \leq c \vee (a = c \wedge \mathit{intleq}(b, d))$$

[3] We will use CC as shorthand for "conjunctive context" in the rest of the paper.

Now $int(b) \wedge int(d)$ appears in the CC of $b \leq d$ and hence the first rule applies to that term:

$$int(b) \wedge int(d) \wedge pair(x, a, b) \wedge pair(y, c, d) \wedge (a \leq c \vee (a = c \wedge intleq(b, d))$$

This term is now in *normal form*, i.e. no more rules are applicable. □

One simple normalisation algorithm is *strict evaluation*, i.e. to normalise a term $f(T_1, ..., T_n)$, we first normalise each $T_1, ..., T_n$ to $U_1, ..., U_n$, and then test rules against $f(U_1, ..., U_n)$. If a rule $(f(H_1, ..., H_n) \iff B)$ matches $f(U_1, ..., U_n)$, then any variable V in $H_1, ..., H_n$ must be bound to a normalised term. This is an important property, since it means V's value can be copied to the rule body without the need for further work.

Example 3. Consider the rule $(f(X) \iff g(X))$. To ensure the body $g(X)$ is normalised it is sufficient to only check the rules for $g/1$, rather than normalise X first. Under strict evaluation X must already be in normal form. □

A normalisation algorithm for Cadmium is more complex because of CC matching. It is possible that terms matched in the CC are not in normal form.

Example 4. Consider the following Cadmium program consisting of three rules:

```
X = Y \ X <=> var(X) | Y.    pass <=> true.    eq(X,Y) <=> X = Y.
```

This is an example of actual Cadmium code. Cadmium term syntax follows Prolog term syntax: any name starting with a capital letter represents a variable. The first rule implements substitution using CC matching – i.e. given an `X` where `X=Y` holds, then substitute `X` with `Y`. Like Prolog, Cadmium allows variables to appear in the goal. We will examine distinct problems with two different goals.

Early Application : Consider the goal $G_1 = (\texttt{A} \wedge \texttt{A=pass})$. Under left-to-right strict evaluation, conjunct `A` with `A=pass` in its CC will be normalised first. Since `A=pass` can be rewritten to `A=true`, the CC is not in normal form. If we apply the first rule to `A`, and copy variables from the matching to the body as per Example 3, then the result is the unnormalised term `pass`. This is called *early application* since the result is unnormalised because a rule was *applied* with an unnormalised CC.

Early Failure : Consider the goal $G_2 = (\texttt{A} \wedge \texttt{eq(A,true)})$. Conjunct `A` with `eq(A,true)` in its CC is normalised first. Again, the CC is unnormalised, since `eq(A,true)` can be rewritten to `A=true`. In this case the first rule does not match, since the substitution rule expects a `=`-term, not an `eq`-term. If `eq(A,true)` was normalised first, then the rule would match. This is called *early failure* since a rule *failed* to match because the CC was unnormalised. □

So why not simply normalise the CC before it is used? In general it is impossible to force the CC to be normalised before it is used.

Example 5. Consider the program from Example 4 and the goal $\texttt{X=Y} \wedge \texttt{Y=X}$. Under the ACDTR semantics, only two rule applications are possible:

1. variable `Y` (inside conjunct `X=Y`) with CC `Y=X` rewrites to `X`; or
2. variable `X` (inside conjunct `Y=X`) with CC `X=Y` rewrites to `Y`.

The CC for (1) is unnormalised, because by (2) we have that `Y=X` can be rewritten to `Y=Y`. Likewise the CC for (2) is unnormalised because of (1). Either way a rule is applied with unnormalised CC. Therefore, in general it is impossible to guarantee a normalised CC. □

A prototype basic normalisation algorithm that accounts for unnormalised CC first appeared in [4]. The main contributions of this paper are:

- we show that the basic normalisation algorithm, whilst simple to implement, is too inefficient to be practical on some "real-world" applications;
- we analyse the causes for incomplete normalisation (e.g. Example 4) and show how the basic algorithm handles these cases;
- we use this information to derive a more efficient normalisation algorithm used in the G12 Cadmium implementation of ACDTR;
- we also show how the information can be used to compile the bodies of rules into more efficient executable code.

2 Preliminaries

The syntax of Cadmium closely resembles that of Constraint Handling Rules [5]. There are two[4] types of rules; they have the following form:

$$\begin{array}{ll} \textit{(simplification)} & H \iff g \mid B \\ \textit{(simpagation)} & C \setminus H \iff g \mid B \end{array}$$

where *head* H, *conjunctive context* C, *guard* g, and *body* B are arbitrary terms. A program P is a set of rules. Essentially a rule works as follows: given a term $t[h]$ and matching substitution θ where $h = H\theta$ such that $g\theta$ is true, and $C\theta$ appears in the conjunctive context of h, then we obtain the term $t[B\theta]$. ACDTR rules can be applied to any subterm of the goal, unlike in CHR.

For space reasons, we refer the reader to [4] for details about the declarative and operational semantics of Cadmium. A general understanding of term rewriting is sufficient to follow the paper, all important differences w.r.t. standard term rewriting are illustrated by examples.

2.1 Basic Normalisation with Conjunctive Context

In this section we present a version of the basic normalisation algorithm for ACDTR that first appeared in [4]. The basic algorithm is shown in Figure 1. The function normalise_acdtr(T,CC) normalises some term T with respect to the current CC and some compiled version of the program. For the initial goal $CC = \wedge$, i.e. an empty conjunction. Normalisation works in two parts: the first part handles conjunction and the second part handles all other terms.

[4] The original ACDTR [4] semantics also included a generalisation of *propagation rules*. However, these are not implemented in Cadmium.

```
normalise_acdtr(T,CC)
  if T = ∧(...)                /* Conjunction */
    Acc := T
    repeat
      let Acc = ∧(T_1, ..., T_n)
      Acc := ∧
      rulefired := false
      forall 1 ≤ i ≤ n
        CC' := flatten(∧(Acc, T_{i+1}, ..., T_n, CC))
        U_i := normalise_acdtr(T_i, CC')
        if (U_i ≠ T_i) rulefired := true
        Acc := flatten(Acc ∧ U_i)
    until not rulefired
    return call_∧(Acc, CC)
  if T = f(T_1, ..., T_n)      /* Other terms */
    forall 1 ≤ i ≤ n
      U_i := normalise_acdtr(T_i, CC)
    if isAC(f)
      U := flatten(f(U_1, ..., U_n))
      return call_f(U, CC)
    else return call_f(U_1, ..., U_n, CC)
  else
    return T
```

Fig. 1. Basic ACDTR normalisation algorithm.

To begin with, let us consider the second part, which implements basic normalisation for AC term rewriting using a strict evaluation strategy. To normalise a (non-conjunction) term $T = f(T_1, \ldots, T_n)$, we first normalise each argument $T_1, \ldots, T_n$ to $U_1, \ldots, U_n$ respectively, and then normalise $U = f(U_1, \ldots, U_n)$ by calling a compiled procedure call_f that applies any rule that matches `f`$/n$ terms, or returns U if no such rule exists. Details about the compiled procedures can be found in Section 4.

The calling conventions for AC and non-AC operators are different. For the AC case, the term $U = f(U_1, \ldots, U_n)$ must be created and *flattened* before the procedure call_f is called. The idea behind flattening is to represent a nested binary AC expression as a "flat" n-ary term. For example, the equivalent AC terms $(1 + 2) + 3$ and $1 + (2 + 3)$ are represented as $+(1, 2, 3)$ in flattened form. As with the binary-$+$, the n-ary $+$ is also commutative, i.e. $+(\ldots, U_i, \ldots, U_j, \ldots) \equiv +(\ldots, U_j, \ldots, U_i, \ldots)$. The motivation for flattening is to simplify the AC matching implementation (see [8] for a more detailed discussion). For the rest of the paper, we may switch between binary and flattened notation for AC terms whenever convenient.

Conjunction is normalised differently from other (AC) terms. A conjunction $T = \wedge(T_1, ..., T_n)$ is normalised over several passes by the **repeat** loop, which works as follows. The **let** matching is assumed to return all conjuncts T_1,. . . ,T_n of *Acc*, where *Acc* is initially set to T (the input conjunction), and to the accu-

mulated result from the previous pass of the **repeat** loop in all further passes. For each pass, this may be a different set. Next *Acc* and *rulefired* are initialised, followed by the **forall** loop, which normalises each conjunct T_i of T with respect to CC', which is CC extended by all other conjuncts of T excluding T_i. The new conjuncts U_i are accumulated into variable *Acc*. One complication is that each U_i may itself be a conjunction, so flatten is used to ensure *Acc* remains in flattened form. We also compare the old T_i against its normalised version U_i. If there is a difference, then a rule has fired, and *rulefired* is set to *true*, which ensures another pass of the repeat loop. Note that the Cadmium implementation tracks rule firings explicitly rather than actually comparing (potentially large) terms. Finally, once *Acc* has reached a fixed point, i.e. $\neg$*rulefired* holds, the procedure call_$\wedge$ is run. Each pass of the **repeat**-loop is referred to as a *conjunction pass*.

The intuition behind the normalise_acdtr algorithm is as follows. If a T_i changes to a U_i, then the CC of all T_j where $j \neq i$ has also changed – i.e. the CC contained T_i but now contains U_i. The next pass of the repeat loop ensures each T_j is *woken-up* with respect to the up-to-date CC containing U_i. Here, the terminology *wake-up* means a conjunct is renormalised with a new CC in the next conjunction pass.

Example 6 (Early Application). Consider the program and goal G_1 from Example 4. The first pass of the normalise_acdtr algorithm is (1) `A` with CC `A=pass` is rewritten to `pass` (early application), then (2) `A=pass` (with CC `pass`) is rewritten to `A=true`. After the first pass the conjunction *Acc* is `pass` $\wedge$ `A=true`.

Since a rule has fired, the conjunction is normalised again. This time (3) `pass` (with CC `A=true`) is rewritten to `true`, then (4) `A=true` (with CC `true`) remains unchanged. After the second pass the conjunction *Acc* is `true` $\wedge$ `A=true`. Since again a rule has fired, the conjunction is renormalised once more. This time no rule fires, since the conjunction is already in normal form. The $\neg$*rulefired* test succeeds, and `true` $\wedge$ `A=true` is ultimately returned. □

Example 7 (Early Failure). Consider the program and goal G_2 from Example 4. The normalise_acdtr algorithm works as follows: (1) `A` with CC `eq(A,true)` remains unchanged (early failure), then (2) `eq(A,true)` (with CC `A`) is rewritten to `A=true`. After the first pass the conjunction *Acc* is `A` $\wedge$ `A=true`.

Since a rule has fired, the conjunction is normalised again. This time (3) `A` with CC `A=true` is rewritten to `true`, then (4) `A=true` (with CC `true`) remains unchanged. After the second pass the conjunction *Acc* is `true` $\wedge$ `A=true`. Another pass is tried, but since the conjunction is already in normal form, no more rewrites take place, and `true` $\wedge$ `A=true` is returned. □

3 Improved Normalisation

Algorithm normalise_acdtr is relatively simple and was used in earlier versions of the Cadmium implementation. However, the algorithm is still very "coarse" in the sense that *any* change of a conjunct results in the *entire conjunction being processed again*. Clearly this is sub-optimal, as it is probable that some changes

in the CC do not affect the normalisation status of other conjuncts. An extreme example of this situation occurs when all rules in a program are simplification rules, and thus do not depend on the CC at all. In this case, conjunction can be treated the same as any other AC functor, hence only one pass is required to ensure a normal form.

Even if the program contains simpagation rules, the number of conjuncts that need to be woken-up per pass can often be reduced.

Example 8 (Early Failure). For example, consider the following rule that simplifies less-than constraints if the negation greater-than is present in the CC.

```
X > Y \  X < Y <=> false.
```

Suppose that the goal is `A<B` $\wedge$ `f(A,B)`. Clearly the normal form of the `A<B` conjunct only depends on the presence/absence of `A>B` in its CC.

During the initial conjunction pass of normalise_acdtr, conjunct `A<B` will not be rewritten because `A>B` does not appear in the CC (i.e. early failure). Then `A<B` will only need to *wake-up* iff a `A>B` term is subsequently added to the CC. Any other change to the CC can be safely ignored, since this would not affect the applicability of the above rule. □

By definition, early failure means a conjunct C is not rewritten because some term T was not in its CC. Thus, we only need to wake-up C if a suitable T is subsequently added to C's CC. Likewise, a wake-up for early application is sometimes not necessary.

Example 9 (Early Application). Consider the following rules for Zinc expression manipulation in Cadmium.

```
decl(T,X) /\ decl(T,Y) /\ decl(T,Z) \ X*(Y+Z) <=> X*Y+X*Z.
int <=> float.
```

Here, `decl(`T`,`V`)` encodes a Zinc variable declaration, where T is the type and V is the variable. Consider the goal

`A*(B+C)` $\wedge$ `decl(int,A)` $\wedge$ `decl(int,B)` $\wedge$ `decl(int,C)`

During the first conjunction pass, conjunct `A*(B+C)` is rewritten to `A*B+A*C` with the reaminder of the goal as CC. The CC is not in normal form since subterm(s) `int` are not in normal form, hence this is a case of early application.

However, `A*B+A*C` is in normal form and therefore does not need to be woken-up again. This is because the rule body depended only on program variables also appearing in the rule head, i.e. `X`, `Y`, and `Z`, but not `T`. Therefore if `X`, `Y`, and `Z` are in normal form, the new term `A*B+A*C` will also be in normal form. □

The basic idea of the refined algorithm is to only wake up conjuncts if there is actually a need to do so.

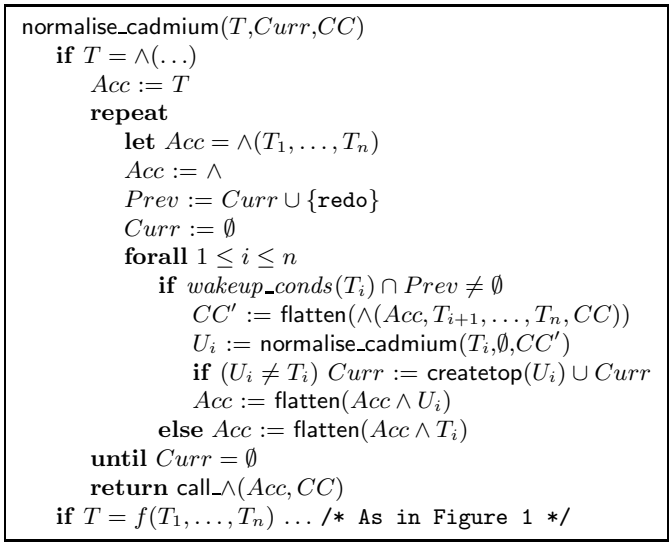

Fig. 2. Improved normalisation algorithm from the Cadmium implementation.

Events and wake-up conditions *Wake-up conditions* are conditions associated to conjuncts in conjunctions. An *event* declares that a wake-up condition has become satisfied. During normalisation, if an event occurs satisfying a wake-up condition, then any associated conjunct will be woken-up during the next conjunction pass.

The Cadmium implementation uses the following wake-up conditions:

Case	*Condition/Event*
Early Application	`redo`
Early Failure	`create(`f/a`)`

Condition `redo` means that the conjunct is always to be woken-up during the next pass. It will be associated to a conjunct C if C is not in normal form due to early application. Early application does not always result in a `redo` condition, as was the case in Example 9. Condition `create(`f/a`)` means that the conjunct will be woken-up if some term with functor/arity f/a is added to the CC. This condition is useful for early failure. For example, in Example 8, the conjunct `A<B` needs to be woken up if a `A>B` term is added to its CC. We can approximate this precise condition with a `create(>/2)` wake-up condition. In general, determining the precise conditions is undecidable, so some approximation is always required.

The improved algorithm is described in Figure 2. This version is called normalise_cadmium because it is the actual normalisation algorithm used by the Cadmium implementation. The main difference between this version and the previous algorithm is the tracking of events and wake-up conditions. $Prev$ and $Curr$ are sets of events. $Prev$ contains all events that occurred during the previous pass and a `redo` event. Each new pass generates this event. $Curr$ accumulates

the events that occur during the current pass. Note that initially $Curr$ is an argument to normalise_cadmium. For now, we can assume that the value passed in through $Curr$ is always the empty set. This will change later in Section 4.

The function $wakeup\_conds(T_i)$ returns the set of wake-up conditions associated to a given conjunct T_i. If T_i has not been normalised yet (i.e. in the initial pass), then its set of wake-up conditions is assumed to be $\{\texttt{redo}\}$. If T_i is subsequently normalised to U_i, then the wake-up conditions for U_i are roughly determined as follows:

1. *Early Application*: If an early application resulted in an unnormalised subterm of U_i, then $\texttt{redo} \in wakeup\_conds(U_i)$.
2. *Early Failure*: If a simpagation rule $(C \backslash H \iff G|B)$ failed to fire on some subterm S of T_i, and S also appears in U_i, then

$$\{\texttt{create}(f_1/a_1), \ldots, \texttt{create}(f_n/a_n)\} \subseteq wakeup\_conds(U_i)$$

   where the f_i/a_i are the functor/arity pairs of the conjuncts in C.

Note that wake-up conditions propagate upwards, i.e. if the normalisation of some subterm S of T_i generates a wake-up condition C, then C is propagated upwards and attached to U_i. For nested conjunctions, C will be propagated upwards to every conjunct S appeared in. The exact mechanism for generating wakeup conditions is the role of the Cadmium compiled code, i.e. in the `call_f` procedures. This will explained in Section 4.2.

Wake-up conditions are used to prevent unnecessary renormalisation during the second or later conjunction passes. Conjunct T_i will only wake-up if there exists an event in $Prev$ that is also present in $wakeup\_conds(T_i)$. Otherwise, the conjunct is already in normal form, and the old value can be used.

If a conjunct T_i is renormalised to U_i where $T_i \neq U_i$, the call $\texttt{createtop}(U_i)$ will generate an appropriate set of `create` events. If $U_i = \wedge(V_1, \ldots, V_n)$, then $\texttt{createtop}(U_i)$ generates $\{\texttt{create}(f_1/a_1), \ldots, \texttt{create}(f_n/a_n)\}$, where $f_1/a_1, \ldots, f_n/a_n$ are the functor/arities of $V_1, \ldots, V_n$. Otherwise, if $U_i = g(W_1, \ldots, W_n)$ where $g \neq \wedge$, then $\texttt{createtop}(U_i)$ generates the singleton set $\{\texttt{create}(g/n)\}$. The generated events are accumulated into $Curr$ and used as $Prev$ during the next pass.

Example 10. Consider the following three rule program.

```
(1)  X > Y \ X < Y <=> false.
(2)  gt(X,Y) <=> X > Y.
(3)  h(_) \ f(X) <=> g(X).
```

Rule (1) is the rule from Example 8. Rule (2) rewrites an auxiliary term into a `>`/2 term. Rule (3) is an artificial rule which depends on the CC.

Consider the execution of the goal term: `h(1)` $\wedge$ `wrap(f(x<y)` $\wedge$ `gt(x,y))`. There are two levels of nested conjunction, with the inner conjunction inside the `wrap`/1 term. Assume execution proceeds from left-to-right. First `h(1)` is normalised and remains unchanged. Next the `wrap`/1 term and the inner conjunction are normalised. The inner conjunct `f(x<y)` is normalised first. As normalisation proceeds bottom-up, the following wake-up conditions are generated:

1. Subterm `x<y` could potentially fire Rule (1), given a `x>y` term in the CC. Thus a `create(>/2)` waking condition is generated for this term.
2. Term `f(x<y)` fires rule (3) to give `g(x<y)`. Since the body is independent of the CC, no `redo` waking condition needs to be generated for this term.

Thus, the set of waking conditions attached to the first inner conjunct `g(x<y)` is {`create(>/2)`}.

Next, the second inner conjunct `gt(x,y)` is normalised to `x>y`. This generates a `create(>/2)` event, which is recorded in $Curr$, but no wake-up conditions are recorded for this term. The intermediate result after the first pass is: `h(1)` $\wedge$ `wrap(g(x<y)` $\wedge$ `x>y)`.

In the second pass, the inner conjunct `g(x<y)` is renormalised to `g(false)`, since the attached wake-up condition `create(>/2)` matches an event that occurred during the previous pass. The second inner conjunct, `x>y`, will not be woken up since it has no wake-up conditions. Normalisation proceeds without any more rule applications, thus the final result is: `h(1)` $\wedge$ `wrap(g(false)` $\wedge$ `x > y)`. □

4 Implementation

In this section we discuss some details about the Cadmium implementation.

The current Cadmium implementation compiles rules of the form ($C \setminus H \iff B$) into a low-level byte-code for a simple virtual machine. There are two parts to compilation: compiling the matching ($C \setminus H$), and compiling the body B.

Matching Matching in Cadmium is similar to matching in any other declarative programming language, such as Prolog. For example, the rule (`f(g(X,Y),h) <=>` ...) can be compiled directly into a Prolog clause (`call_f(g(X,Y),h) :- !,` ...) that uses Prolog unification for matching. The cut is necessary since Cadmium rules are committed choice.

Compiling AC matching is somewhat more complicated, as it involves nondeterministically trying combinations of matchings – i.e. the different permutations of the arguments of an AC term. This can be implemented in Prolog as backtracking search. CC matching is essentially the same as AC matching, except we match against the accumulated CC rather than a term matching the rule head.

Body The simplest version of rule body compilation is to call the Cadmium normalisation procedure. For example, given (`f(X,Y,Z) <=> g(h(X,Y),a(1,Z))`) then the compiled rule in Prolog is

```
call_f(X,Y,Z,Ret) :- !, normalise(g(h(X,Y),a(1,Z)),Ret)
```

where `normalise` implements the Cadmium normalisation algorithm. This will cause each term matching `X`, `Y`, `Z` to be renormalised again, which is inefficient.

A better (and more standard) approach is to eliminate all calls to the normalise procedure by iteratively unfolding its application, and to substitute matching variables directly rather than renormalising them. For example, after one unfolding step we have:

```
normalise(g(h(X,Y),a(1,Z))) ≡ call_g(normalise(h(X,Y)),normalise(a(1,Z)))
```

After completely unfolding `normalise`, the rule code becomes:

```
call_f(X,Y,Z,Ret) :- !, call_h(X,Y,RH), call_a(1,Z,RA), call_g(RH,RA,Ret).
```

4.1 Compiling Conjunction in the Body

Compiling conjunction in the rule body is the same as before, i.e. iterative unfolding of the call to the `normalise` procedure. However, because of CC, the `normalise` procedure cannot be unfolded any deeper than any top-most conjunction appearing in the rule body. For example, consider the rule:

```
f(X,Y) <=> g(X) /\ f(Y).
```

According to the Cadmium normalisation algorithm, `g(X)` and all of its subterms must be (re)normalised with `f(Y)` in the CC, and vice versa. Therefore, unfolding `normalise` directly will not work, i.e.,

```
normalise(g(X)∧f(Y)) ≢ call_∧(normalise(g(X)),normalise(f(Y)))
```

since the latter does not handle CC correctly.

The basic approach for handling conjunctions is to unfold `normalise` *as much as possible*, but stopping at the top-most conjunction. This conjunction is simply constructed, then passed to `normalise` to be executed as if it were a fresh goal. For example, the compiled version of the above rule is:

```
call_f(X,Y,Ret) :- !, C1 = g(X), C2 = f(Y), wakeup_on(redo,C1),
          wakeup_on(redo,C2), normalise(C1 /\ C2,Ret).
```

This clause constructs `g(X) /\ f(Y)` and passes it to `normalise`. The built-in `wakeup_on`/2 attaches a `redo` wake-up condition to each conjunct to force normalisation via wake-up. Without the `redo`, `normalise` will skip each conjunct.

Conjunction collector optimisation Under the basic approach, each conjunct in a rule body is completely (re)normalised again, which in some cases is inefficient. However, sometimes we can avoid wake-up in a rule body. Consider the following rule from the MiniZinc to FlatZinc mapping [6, 2]:

```
cons(X) /\ cons(Y) /\ Z <=> cons(X /\ Y) /\ Z.
```

Here `cons(`X`)` represents a Zinc constraint item (`constraint` X). The body of the rule contains two conjuncts: `cons(X /\ Y)` and `Z`. In this rule, `Z` will always match a conjunction – i.e. the "rest" of the conjunction matching the rule head minus `cons(X)` and `cons(Y)`. Furthermore, thanks to bottom-up evaluation, `Z`

must already be normalised, so each conjunct in `Z` already has a set of wake-up conditions attached to it. We can use these tighter wake-up conditions instead of attaching a `redo` condition as was the case above. This can potentially avoid a lot of unnecessary renormalisation.

Example 11. Consider the constraint item collection rule from above. When this rule is applied, the CC of the conjuncts in `Z` remains unchanged save for the removal/addition of some `cons`/1 terms. Therefore, only the conjuncts with a `create(cons/1)` wake-up condition need be renormalised.

The optimised code of the constraint item collection rule is:

```
call_/\(Conj,Ret) :- /* Code for matching */ !,
          C1 = cons(X /\ Y), wakeup_on(redo,C1),
          normalise(C1 /\ Z,[create(cons/1)],Ret).
```

Notice that (1) there is no `redo` wake-up condition attached to `Z`, and (2) we now pass the initial event `create(cons/1)` to the `normalise` procedure, since a new `cons`/1 term was added to the conjunction. This will cause any conjunct with a `create(cons/1)` in `Z` to be renormalised as expected. □

4.2 Generating Wake-up Conditions

In this section we explain how wake-up conditions are generated in the compiled code. This depends on the type of condition being generated.

Wake-up condition `redo`: The normalise_cadmium algorithm returns either a normalised term, or an unnormalised term because of early application. In the latter case, a `redo` wake-up condition must be generated to ensure overall completeness after subsequent passes of the super conjunctions.[5] A `redo` wake-up condition is therefore needed when the body B from a rule $(C \backslash H \iff G|B)$ contains a variable X such that X is also in C, but not in H. Note that if X also appears in H, then because X was processed before its super term H, we can assume X is in normal form, or `redo` has already been generated.

Example 12. Consider the following rules:

```
X > Y \ X < Y <=> false.
X = Y \ X <=> var(X) | Y.
decl(T,X) /\ decl(T,Y) /\ decl(T,Z) \ X*(Y+Z) <=> X*Y+X*Z.
```

from Examples 8, 4, and 9 respectively. The body of the first rule does not contain any variables, thus is independent of the CC. The body of the second rule does depend on the CC through variable `Y`. The body of the third rule shares variables `X`, `Y`, and `Z` with the CC; however these variables also appear in the rule head. Therefore, only the second rule is required to generate a `redo` wake-up condition. The code for the second rule is therefore:

[5] Early application implies there is at least one super conjunction, since otherwise the CC will be empty.

```
call_var(X,Ret) :- /* Matching */, !, wakeup_lift([redo]), Ret = Y.
```

Here, the call `wakeup_lift(`C`)` lifts wake-up conditions C to any conjunct containing the term matching `X`. □

Wake-up condition `create`: The `create` wake-up conditions are generated after all rules for a particular term fail to match. The compiler assumes that any simpagation rule matching failure is caused by *early failure*.

Example 13. Consider the following program which contains two rules for `f`/2.

```
g(X) \ f(X,Y) <=> i(Y).      g(Y) /\ h(Y,Y) \ f(1,Y) <=> Y.
```

Consider the compiled version of this program, where procedure `call_f` checks these rules. If both rules fail to match, then `call_f` will simply construct the `f`/2 term, but will also generate the appropriate `create` wake-up conditions:

```
···     /* Code for rules 1-2. */
call_f(X,Y,Ret) :- !, wakeup_lift([create(g/1),create(h/2)]), Ret = f(X,Y).
```

In general this approach is an over-approximation. For example, `call_f(2,Y)` will never apply the second rule. However the compiler still assumes early failure has occurred, and generates a `create(h/2)` wake-up condition accordingly. This may result in some unnecessary wake-ups.

5 Experiments

Cadmium is part of the G12 project [10]. Its main application is mapping Zinc models, represented as terms, into various solver-dependent back-ends and/or to FlatZinc [6].

Two sets of benchmarks are tested.[6] The first set in Figure 3(a) compare Cadmium versus the Maude 2.3 system [3]. The second set in Figure 3(b) compare the Cadmium implementation using normalisation with/without events. All timings are an average over 10 runs on an Intel E8400 clocked at 3.6GHz.

The benchmarks from Figure 3(a) originate from the second *Rewriting Engines Competition* [7]. Note that the remaining examples from [7] could not be used for various reasons, e.g. running too fast/slow on both systems, or testing confluence (not supported in Cadmium). The benchmarks show that Cadmium is competitive compared to an established implementation on pure (AC) term rewriting problems. The exception is `taut_hard`, where Cadmium is slower than Maude, because of differences in the implementation of AC indexing. The `taut_hard` causes worst-case behaviour for Cadmium's AC index structures. For the `perm` benchmark, which also uses AC matching, Cadmium improves upon Maude. Note that none of these benchmarks use CC matching, hence the CC optimisations shown in Figure 3(b) are not applicable here.

[6] Benchmarks are available at http://www.cs.mu.oz.au/~gjd/download/iclp2008.tar.gz

Bench.	*Maude*	*Cadmium*
`qsort(216)`	1.12s	1.20s
`qsort(343)`	6.37s	6.74s
`msort(729)`	4.27s	4.28s
`msort(1000)`	12.56s	11.64s
`bsort(240)`	1.65s	1.78s
`bsort(360)`	7.67s	8.52s
`rev`	0.83s	1.25s
`taut_hard(2)`	0.13s	3.84s
`taut_hard(3)`	0.36s	37.82s
`perm(8)`	0.38s	0.27s
`perm(9)`	6.79s	3.71s

(a) Maude vs. Cadmium

Bench.	$-events$	$+events$
`queens(8)`	2.61s	2.74s
`queens(9)`	40.70s	43.69s
`cnf_conversion(19)`	11.07s	9.36s
`cnf_conversion(20)`	15.42s	12.97s
`substitution(22)`	1.54s	0.92s
`substitution(23)`	3.13s	2.04s
`warehouses.mzn`	5.14s	0.57s
`langford.mzn`	>300s	33.42s
`packing.mzn`	0.96s	0.23s
`timetabling.mzn`	9.19s	0.76s
`radiation.mzn`	37.58s	2.47s
Geom. mean[7]	6.57s	36.76%

(b) Cadmium ±events

Fig. 3. Experiments comparing the run-time performance of Cadmium.

To test CC normalisation with/without events the benchmarks are as follows: Benchmark `queens(`n`)` finds all solutions to the n-queens problem. Benchmark `cnf_conversion(`n`)` converts the following Boolean formula into conjunctive normal form $\bigwedge_{i=1}^{n}\bigvee_{j=i+1}^{n} x_i \oplus x_j$. using a generic CNF conversion algorithm. Benchmark `substitution(`n`)` applies the substitution rule (Example 4) to the conjunction: $\bigwedge_{i=1}^{n}$ `X`$_i$ `= [X`$_{i+1}$`,...,X`$_n$`] /\ f(X`$_i$`)`. Finally, the `*.mzn` benchmarks test MiniZinc to FlatZinc flattening in Cadmium [6]. These benchmarks are the most important, since they are a "real-world" Cadmium application doing what Cadmium was intended to do – i.e. rewrite (Mini)Zinc models. Note that the mapping used is further developed than earlier versions appearing in [6, 2].

Figure 3(b) compares Cadmium normalisation without events ($-events$) versus with events ($+events$). Overall, normalisation *with* events is significantly better, with a 63% improvement.[7] The MiniZinc flattening benchmarks showed the largest gains. This is especially true for `langford.mzn`, where the $-events$ version is too slow to be practical. On the other hand, the `queens` benchmarks were better off without events. In this case, the $+events$ version avoided almost no wake-ups, so the extra overhead of tracking events causes a slow-down.

6 Related Work and Conclusions

Cadmium is a powerful rewriting language that implements rewriting based on non-local information in the form of Conjunctive Context. However, CC complicates any potential Cadmium normalisation algorithm, since the CC must be distributed to everywhere it is used. Furthermore, there are no guarantees the context itself is normalised, so traditional bottom-up evaluation strategies

[7] excluding `langford.mzn`.

do not work. We have presented a normalisation algorithm based on waking-up conjuncts whose context may have changed in a way that affects rule application. By tracking wake-up conditions and events, renormalisation because of context changes can be significantly decreased. Experiments show speed-ups in real-world Cadmium applications such as Zinc model flattening.

There exist several other implementations of term rewriting, such as Maude [3], and others. Like Cadmium, matching modulo AC is a standard feature. The main difference between Cadmium and other implementations is the native support for CC normalisation.

Unification (and therefore matching) modulo distribution, i.e. $x * (y + z) = x*y+x*z$, has also been studied, e.g. in [9]. However, this work is not relevant to CC-distribution, which is based on a different axiom, e.g. $x \wedge f(y) = x \wedge f(x \wedge y)$.

For future work we intend to further improve the performance of Cadmium. We believe it is possible to refine the normalisation algorithm further, i.e. to avoid even more wake-ups by refining events, and to specialise the renormalisation that occurs during wake-up.

References

1. F. Baader and T. Nipkow. *Term rewriting and all that.* Cambridge Univ. Press, 1998.
2. S. Brand, G. J. Duck, J. Puchinger, and P. J. Stuckey. Flexible, Rule-based Constraint Model Linearisation. In *10th Intl. Symp. on Practical Aspects of Declarative Languages*, LNCS 4902, pages 68–83. Springer, 2008.
3. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. The Maude 2.0 System. In *14th Intl. Conf. on Rewriting Techniques and Applications*, LNCS 2706, pages 76–87. Springer, 2003.
4. G. J. Duck, P. J. Stuckey, and S. Brand. ACD Term Rewriting. In *22nd Intl. Conf. on Logic Programming*, LNCS 4079, pages 117–131. Springer, 2006.
5. T. Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming*, 37:95–138, 1998.
6. N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack. MiniZinc: Towards a Standard CP Modelling Language. In *13th Intl. Conf. on Principles and Practice of Constraint Programming*, LNCS 4741, pages 529–543. Springer, 2007.
7. Rewriting Engines Competition. http://www.lcc.uma.es/~duran/rewriting_competition/.
8. S. M. Eker. Associative-Commutative Matching Via Bipartite Graph Matching. *Computer Journal*, 38(5):381–399, 1995.
9. M. Schmidt-Schauß. Decidability of Unification in the Theory of One-Sided Distributivity and a Multiplicative Unit. *Journal of Symbolic Computation*, 22(3):315–344, 1997.
10. P. J. Stuckey, M. García de la Banda, M. Maher, K. Marriott, J. Slaney, Z. Somogyi, M. Wallace, and T. Walsh. The G12 project: Mapping solver independent models to efficient solutions. In *21st Intl. Conf. on Logic Programming*, LNCS 3668, pages 9–13. Springer, 2005.