ACD Term Rewriting

Gregory J. Duck, Peter J. Stuckey, and Sebastian Brand

NICTA Victoria Laboratory
Department of Computer Science & Software Engineering,
University of Melbourne, Australia
{gjd, pjs, sbrand}@cs.mu.oz.au

Abstract. In this paper we introduce Associative Commutative Distributive Term Rewriting (ACDTR), a rewriting language for rewriting logical formulae. ACDTR extends AC term rewriting by adding *distribution* of conjunction over other operators. Conjunction is vital for expressive term rewriting systems since it allows us to require that multiple conditions hold for a term rewriting rule to be used. ACDTR uses the notion of a "conjunctive context", which is the conjunction of constraints that must hold in the context of a term, to enable the programmer to write very expressive and targeted rewriting rules. ACDTR can be seen as a general logic programming language that extends Constraint Handling Rules and AC term rewriting. In this paper we define the semantics of ACDTR and describe our prototype implementation.

1 Introduction

Term rewriting is a powerful instrument to specify computational processes. It is the basis of functional languages; it is used to define the semantics of languages and it is applied in automated theorem proving, to name only a few application areas.

One difficulty faced by users of term rewriting systems is that term rewrite rules are *local*, that is, the term to be rewritten occurs in a single place. This means in order to write precise rewrite rules we need to gather all relevant information in a single place.

Example 1. Imagine we wish to "program" an overloaded ordering relation for integers variables, real variables and pair variables. In order to write this the "type" of the variable must be encoded in the term[1] as in:

$$\begin{aligned} int(x) \leq int(y) &\rightarrow intleq(int(x), int(y)) \\ real(x) \leq real(y) &\rightarrow realleq(real(x), real(y)) \\ pair(x_1, x_2) \leq pair(y_1, y_2) &\rightarrow x_1 \leq y_1 \vee x_1 = y_1 \wedge x_2 \leq y_2 \end{aligned}$$

In a more standard language, the type information for variables (and other information) would be kept separate and "looked up" when required. □

[1] Operator precedences used throughout this paper are: ∧ binds tighter than ∨, and all other operators, e.g. ¬, =, bind tighter than ∧.

S. Etalle and M. Truszczyński (Eds.): ICLP 2006, LNCS 4079, pp. 117–131, 2006.

Term rewriting systems such as constraint handling rules (CHRs) [5] and associative commutative (AC) term rewriting [3] allow "look up" to be managed straightforwardly for a single conjunction.

Example 2. In AC term rewriting the above example could be expressed as:

$$\begin{array}{rcl} \mathit{int}(x) \wedge \mathit{int}(y) \wedge x \leq y & \rightarrow & \mathit{int}(x) \wedge \mathit{int}(y) \wedge \mathit{intleq}(x, y) \\ \mathit{real}(x) \wedge \mathit{real}(y) \wedge x \leq y & \rightarrow & \mathit{real}(x) \wedge \mathit{real}(y) \wedge \mathit{realleq}(x, y) \\ \mathit{pair}(x, x_1, x_2) \wedge \mathit{pair}(y, y_1, y_2) \wedge x \leq y & \rightarrow & \mathit{pair}(x, x_1, x_2) \wedge \mathit{pair}(y, y_1, y_2) \wedge \\ & & (x_1 \leq y_1 \vee x_1 = y_1 \wedge x_2 \leq y_2) \end{array}$$

where each rule replaces the $x \leq y$ by an appropriate specialised version, in the conjunction of constraints. The associativity and commutativity of $\wedge$ is used to easily collect the required type information from a conjunction. □

One difficulty remains with both AC term rewriting and CHRs. The "look up" is restricted to be over a single large conjunction.

Example 3. Given the term $\mathit{int}(x_1) \wedge \mathit{int}(y_1) \wedge \mathit{pair}(x, x_1, x_2) \wedge \mathit{pair}(y, y_1, y_2) \wedge x \leq y$. Then after rewriting $x \leq y$ to $(x_1 \leq y_1 \vee x_1 = y_1 \wedge x_2 \leq y_2)$ we could not rewrite $x_1 \leq y_1$ since the types for x_1, y_1 appear in a different level.

In order to push the type information inside the disjunction we need to distribute conjunction over disjunction. □

Simply adding distribution rules like

$$A \wedge (B \vee C) \rightarrow A \wedge B \vee A \wedge C \quad (1)$$

$$A \wedge B \vee A \wedge C \rightarrow A \wedge (B \vee C) \quad (2)$$

does not solve the problem. Rule (1) creates two copies of term A, which increases the size of the term being rewritten. Adding Rule (2) to counter this effect results in a non-terminating rewriting system.

1.1 Conjunctive Context

We address the non-termination vs. size explosion problem due to distributivity rewrite rules in a similar way to how commutativity is dealt with: by handling distributivity on the language level. We restrict ourselves to dealing with expanding distributivity of conjunction $\wedge$ over any other operator, and we account for idempotence of conjunction.[2] Thus we are concerned with distribution rules of the form

$$P \wedge f(Q_1, \ldots, Q_n) \rightarrow P \wedge f(P \wedge Q_1, \ldots, P \wedge Q_n). \quad (3)$$

Let us introduce the conjunctive context of a term and its use in rewrite rules, informally for now. Consider a term T and the conjunction $\mathcal{C} \wedge T$ modulo

[2] This means that conjunction is distributive over any function f in presence of a redundant copy of P, i.e. $P \wedge (P \wedge f(Q_1, \ldots, Q_n)) \rightarrow P \wedge f(P \wedge Q_1, \ldots, P \wedge Q_n)$. We use idempotence to simplify the RHS and derive (3).

idempotence of $\wedge$ that would result from exhaustive application of rule (3) to the superterm of T. By the *conjunctive context* of T we mean the conjunction $\mathcal{C}$.

Example 4. The conjunctive context 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 allow a rewrite rule $P \ \rightarrow \ T$ to refer to the conjunctive context $\mathcal{C}$ of the rule head P. We use the following notation:

$$\mathcal{C} \setminus P \Longleftrightarrow T.$$

This facility provides $\wedge$-distributivity without the undesirable effects of rule (3) on the term size.

Example 5. We can express that an equality can be used anywhere "in its scope" by viewing the equality as a conjunctive context:

$$x = a \setminus x \Longleftrightarrow a.$$

Using this rule on the term of Example 4 results in

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

without dissolving the disjunction. □

1.2 Motivation and Applications

Constraint Model Simplification. Our concrete motivation behind associative commutative distributive term rewriting (ACDTR) is *constraint model mapping* as part of the G12 project [7]. A key aim of G12 is the mapping of solver independent models to efficient solver dependent models. We see ACDTR as the basis for writing these mappings. Since models are not flat conjunctions of constraints we need to go beyond AC term rewriting or CHRs.

Example 6. Consider the following simple constraint model inspired by the Social Golfers problem. For two groups g_1 and g_2 playing in the same week there can be no overlap in players: $maxOverlap(g_1, g_2, 0)$ The aim is to maximise the number of times the overlap between two groups is less than 2; in other words minimise the number of times two players play together in a group.

$$\text{constraint} \quad \bigwedge_{\substack{\forall w \in Weeks \\ \forall g_1, g_2 \in weeks[w] \\ g_1 < g_2}} maxOverlap(g_1, g_2, 0)$$

$$\text{maximise} \quad \sum_{\substack{\forall w_1, w_2 \in Weeks \\ \forall g_1 \in weeks[w_1] \\ \forall g_2 \in weeks[w_2] \\ g_1 < g_2}} holds(maxOverlap(g_1, g_2, 1))$$

Consider the following ACDTR program for optimising this constraint model.

$$maxOverlap(a, b, c_1) \setminus maxOverlap(a, b, c_2) \Longleftrightarrow c_2 \geq c_1 \mid true$$
$$holds(true) \Longleftrightarrow 1$$
$$holds(false) \Longleftrightarrow 0$$

The first rule removes redundant *maxOverlap* constraints. The next two rules implement partial evaluation of the *holds* auxiliary function which coerces a Boolean to an integer.

By representing the constraint model as a giant term, we can optimise the model by applying the ACDTR program. For example, consider the trivial case with one week and two groups G_1 and G_2. The model becomes

$$maxOverlap(G_1, G_2, 0) \wedge maximise(holds(maxOverlap(G_1, G_2, 1))).$$

The subterm $holds(maxOverlap(G_1, G_2, 1))$ simplifies to 1 using the conjunctive context $maxOverlap(G_1, G_2, 0)$. □

It is clear that pure CHRs are insufficient for constraint model mapping for at least two reasons, namely

- a constraint model, e.g. Example 6, is typically not a flattened conjunction;
- some rules rewrite functions, e.g. rules (2) and (3) rewriting function *holds*, which is outside the scope of CHRs (which rewrite constraints only).

Global Definitions. As we have seen conjunctive context matching provides a natural mechanism for making global information available. In a constraint model, structured data and constraint definitions are typically global, i.e. on the top level, while access to the data and the use of a defined constraint is local, e.g. the type information from Example 1. Another example is partial evaluation.

Example 7. The solver independent modelling language has support for arrays. Take a model having an array a of given values. It could be represented as the top-level term $array(a, [3, 1, 4, 1, 5, 9, 2, 7])$. Deeper inside the model, accesses to the array a occur, such as in the constraint $x > y + lookup(a, 3)$. The following rules expand such an array lookup:

$$array(A, Array) \setminus lookup(A, Index) \Longleftrightarrow list\_element(Array, Index)$$
$$list\_element([X|Xs], 0) \Longleftrightarrow X$$
$$list\_element([X|Xs], N) \Longleftrightarrow N > 0 \mid list\_element(Xs, N-1)$$

Referring to the respective array of the lookup expression via its conjunctive context allows us to ignore the direct context of the lookup, i.e. the concrete constraint or expression in which it occurs. □

Propagation Rules. When processing a logical formula, it is often useful to be able to specify that a new formula Q can be derived from an existing formula P *without consuming* P. In basic term rewriting, the obvious rule $P \Longleftrightarrow P \wedge Q$ causes trivial non-termination. This issue is recognised in CHRs, which provide support for inference or *propagation* rules. We account for this fact and use rules of the form $P \Longrightarrow Q$ to express such circumstances.

Example 8. The following is the classic CHR `leq` program reimplemented for ACD term rewriting (we omit the basic rules for logical connectives):

$$\begin{array}{rll} leq(X,X) \Longleftrightarrow true & & (reflexivity) \\ leq(X,Y) \setminus leq(Y,X) \Longleftrightarrow X = Y & & (antisymmetry) \\ leq(X,Y) \setminus leq(X,Y) \Longleftrightarrow true & & (idempotence) \\ leq(X,Y) \wedge leq(Y,Z) \Longrightarrow leq(X,Z) & & (transitivity) \end{array}$$

These rules are almost the same as the CHR version, with the exception of the second and third rule (*antisymmetry* and *idempotence*) which generalise its original by using conjunctive context matching. □

Propagation rules are also used for adding redundant information during model mapping.

The rest of the paper is organised as follows. Section 2 covers the standard syntax and notation of term rewriting. Section 3 defines the declarative and operational semantics of ACDTR. Section 4 describes a prototype implementation of ACDTR as part of the G12 project. Section 5 compares ACDTR with related languages. Finally, in Section 6 we conclude.

2 Preliminaries

In this section we briefly introduce the notation and terminology used in this paper. Much of this is borrowed from term rewriting [3].

We use $\mathcal{T}(\Sigma, X)$ to represent the set of all terms constructed from a set of function symbols Σ and set of variables X (assumed to be countably infinite). We use $\Sigma^{(n)} \subseteq \Sigma$ to represent the set of function symbols of arity n.

A *position* is a string (sequence) of integers that uniquely determines a subterm of a term T, where ϵ represents the empty string. We define function $T|_p$, which returns the subterm of T at position p as

$$\begin{aligned} T|_\epsilon &= T \\ f(T_1, \ldots, T_i, \ldots, T_n)|_{ip} &= T_i|_p \end{aligned}$$

We similarly define a function $T[S]_p$ which replaces the subterm of T at position p with term S. We define the set $\mathcal{P}os(T)$ to represent the set of all *positions* of subterms in T.

An *identity* is a pair $(s, t) \in \mathcal{T}(\Sigma, X) \times \mathcal{T}(\Sigma, X)$, which is usually written as $s \approx t$. Given a set of identities E, we define $\approx_E$ to be the set of identities closed under the axioms of *equational logic* [3], i.e. symmetry, transitivity, etc.

We define the congruence class $[T]_{\approx_E} = \{S \in \mathcal{T}(\Sigma, X) | S \approx_E T\}$ as the set of terms equal to T with respect to E.

Finally, we define function $vars(T)$ to return the set of variables in T.

3 Syntax and Semantics

The syntax of ACDTR closely resembles that of CHRs. There are three types of rules of the following form:

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

where r is a *rule identifier*, and *head* H, *conjunctive context* C, *guard* g and *body* B are arbitrary terms. The rule identifier is assumed to uniquely determine the rule. A program P is a set of rules.

We assume that $vars(g) \subseteq vars(H)$ or $vars(g) \subseteq vars(H) \cup vars(C)$ (for simpagation rules). The rule identifier can be omitted. If $g = true$ then the guard can be omitted.

We present the declarative semantics of ACDTR based on equational logic.

First we define the set of operators that ACDTR treats specially.

Definition 1 (Operators). *We define the set of* associate commutative *operators as* AC. *The set* AC *must satisfy* $AC \subseteq \Sigma^{(2)}$ *and* $(\wedge) \in AC$.

For our examples we assume that $AC = \{\wedge, \vee, +, \times\}$. We also treat the operator $\wedge$ as *distributive* as explained below.

ACDTR supports a simple form of guards.

Definition 2 (Guards). *A* guard *is a term. We denote the set of all "true" guards as* $\mathcal{G}$, *i.e. a guard* g *is said to* hold *iff* $g \in \mathcal{G}$. *We assume that* $true \in \mathcal{G}$ *and* $false \notin \mathcal{G}$.

We can now define the declarative semantics for ACDTR. In order to do so we employ a special binary operator *where* to explicitly attach a conjunctive context to a term. Intuitively, the meaning of T *where* C is equivalent to that of T provided C is *true*, otherwise the meaning of T *where* C is unconstrained. For Boolean expressions, it is useful to interpret *where* as conjunction $\wedge$, therefore *where*-distribution, i.e. identity (6) below, becomes equivalent to $\wedge$-distribution (3). The advantage of distinguishing *where* and $\wedge$ is that we are not forced to extend the definition of $\wedge$ to arbitrary (non-Boolean) functions.

We denote by $\mathcal{B}$ the following set of *built-in* identities:

$$A \circ B \approx B \circ A \tag{1}$$

$$(A \circ B) \circ C \approx A \circ (B \circ C) \tag{2}$$

$$T \approx (T \ where \ true) \tag{3}$$

$$A \wedge B \approx (A \ where \ B) \wedge B \tag{4}$$

$$T \ where \ (W_1 \wedge W_2) \approx (T \ where \ W_1) \ where \ W_2 \tag{5}$$

$$f(A_1, ..., A_i, ..., A_n) \ where \ W \approx f(A_1, ..., A_i \ where \ W, ..., A_n) \ where \ W \tag{6}$$

for all $\circ \in AC$, functions $f \in \Sigma^{(n)}$, and $i \in \{1, \ldots, n\}$.

Definition 3 (Declarative Semantics for ACDTR). *The* declarative semantics *for an ACDTR program P (represented as a multiset of rules) is given by the function* $[\![\,]\!]$ *defined as follows:*

$$
\begin{array}{ll}
[\![P]\!] & = \{[\![\theta(R)]\!] \mid \forall R, \theta \,.\, R \in P \wedge \theta(\mathsf{guard}(R)) \in \mathcal{G}\} \cup \mathcal{B} \\
[\![H \Longleftrightarrow g \mid B]\!] & = \exists_{vars(B)-vars(H)}(H \approx B) \\
[\![C \setminus H \Longleftrightarrow g \mid B]\!] & = \exists_{vars(B)-vars(C,H)}(H \ where\ C \approx B \ where\ C) \\
[\![H \Longrightarrow g \mid B]\!] & = \exists_{vars(B)-vars(H)}(H \approx H \wedge B)
\end{array}
$$

where function $\mathsf{guard}(R)$ *returns the guard of a rule.*

The function $[\![\,]\!]$ maps ACDTR rules to identities between the head and the body terms, where body-only variables are existentially quantified.[3] Note that there is a new identity for each possible binding of $\mathsf{guard}(R)$ that holds in $\mathcal{G}$. A propagation rule is equivalent to a simplification rule that (re)introduces the head H (in conjunction with the body B) in the RHS. This is analogous to propagation rules under CHRs.

A simpagation rule is equivalent to a simplification rule provided the conjunctive context is satisfied.

The built-in rules $\mathcal{B}$ from Definition 3 contain identities for creating/destroying (3) and (4), combining/splitting (5), and distributing downwards/upwards (6) a conjunctive context in terms of the *where* operator.

The set $\mathcal{B}$ also contains identities (1) and (2) for the associative/commutative properties of the AC operators.

Example 9. Consider the following ACDTR rule and the corresponding identity.

$$[\![X = Y \setminus X \Longleftrightarrow Y]\!] \quad = \quad (Y \ where\ X = Y) \approx (X \ where\ X = Y) \qquad (7)$$

Under this identity and using the rules in $\mathcal{B}$, we can show that $f(A) \wedge (A = B) \approx f(B) \wedge (A = B)$, as follows.

$$
\begin{array}{ll}
f(A) \wedge (A = B) & \approx_{(4)} \\
(f(A) \ where\ (A = B)) \wedge (A = B) & \approx_{(6)} \\
(f(A \ where\ (A = B)) \ where\ (A = B)) \wedge (A = B) & \approx_{(7)} \\
(f(B \ where\ (A = B)) \ where\ (A = B)) \wedge (A = B) & \approx_{(6)} \\
(f(B) \ where\ (A = B)) \wedge (A = B) & \approx_{(4)} \\
f(B) \wedge (A = B) &
\end{array}
$$

3.1 Operational Semantics

In this section we describe the operational semantics of ACDTR. It is based on the theoretical operational semantics of CHRs [1,4]. This includes support for identifiers and propagation histories, and conjunctive context matching for simpagation rules.

[3] All other variables are implicitly universally quantified, where the universal quantifiers appear outside the existential ones.

Propagation History. The CHR concept of a *propagation history*, which prevents trivial non-termination of propagation rules, needs to be generalised over arbitrary terms for ACDTR. A propagation history is essentially a record of all propagation rule applications, which is checked to ensure a propagation rule is not applied twice to the same (sub)term.

In CHRs, each constraint is associated with a unique *identifier.* If multiple copies of the same constraint appear in the CHR store, then each copy is assigned a different identifier. We extend the notion of identifiers to arbitrary terms.

Definition 4 (Identifiers). *An* identifier *is an integer associated with each (sub)term. We use the notation $T\#i$ to indicate that term T has been associated with identifier i. A term T is* annotated *if T and all subterms of T are associated with an identifier. We also define function* $\mathsf{ids}(T)$ *to return the set of identifiers in T, and* $\mathsf{term}(T)$ *to return the non-annotated version of T.*

For example, $T = f(a\#1, b\#2)\#3$ is an annotated term, where $\mathsf{ids}(T) = \{1, 2, 3\}$ and $\mathsf{term}(T) = f(a, b)$.

Identifiers are considered separate from the term.We could be more precise by separating the two, i.e. explicitly maintain a map between $\mathcal{P}os(T)$ and the identifiers for T. We do not use this approach for space reasons. We extend and overload all of the standard operations over terms (e.g. from Section 2) to annotated terms in the obvious manner. For example, the subterm relation $T|_p$ over annotated terms returns the annotated term at position p. The exception are elements of the congruence class $[T]_{\approx_{AC}}$, formed by the AC relation $\approx_{AC}$, which we assume satisfies the following constraints.

$$A\#i \circ B\#j \approx_{AC} B\#j \circ A\#i$$
$$A\#i \circ (B\#j \circ C\#k) \approx_{AC} (A\#i \circ B\#j) \circ C\#k$$

We have neglected to mention the identifiers over AC operators. These identifiers will be ignored later, so we leave them unconstrained.

A propagation history is a set of entries defined as follows.

Definition 5 (Entries). *A* propagation history entry *is of the form $(r\ @\ E)$, where r is a propagation rule identifier, and E is a string of identifiers. We define function* $\mathsf{entry}(r, T)$ *to return the propagation history entry of rule r for annotated term T as follows.*

$$\begin{array}{lll} \mathsf{entry}(r,T) & = (r\ @\ \mathsf{entry}(T)) & \\ \mathsf{entry}(T_1 \circ T_2) & = \mathsf{entry}(T_1)\ \mathsf{entry}(T_2) & \circ \in AC \\ \mathsf{entry}(f(T_1,...,T_n)\#i) & = i\ \mathsf{entry}(T_1)\ ...\ \mathsf{entry}(T_n) & \textit{otherwise} \end{array}$$

This definition means that propagation history entries are unaffected by associativity, but are effected by commutativity.

Example 10. Consider the annotated term $T = f((a\#1 \wedge b\#2)\#3)\#4$. We have that $T \in [T]_{\approx_{AC}}$ and $T' = f((b\#2 \wedge a\#1)\#3)\#4 \in [T]_{\approx_{AC}}$. Although T and T' belong to $[T]_{\approx_{AC}}$ they have different propagation history entries, e.g. $\mathsf{entry}(r, T) = (r\ @\ (4\ 1\ 2))$ while $\mathsf{entry}(r, T') = (r\ @\ (4\ 2\ 1))$. □

When a (sub)term is rewritten into another, the new term is assigned a set of new unique identifiers. We define the auxiliary function $\mathsf{annotate}(\mathcal{P}, T) = T_a$ to map a set of identifiers $\mathcal{P}$ and un-annotated term T to an annotated term T_a such that $\mathsf{ids}(T_a) \cap \mathcal{P} = \emptyset$ and $|\mathsf{ids}(T_a)| = |\mathcal{P}os(T)|$. These conditions ensure that all identifiers are new and unique.

When a rule is applied the propagation history must be updated accordingly to reflect which terms are copied from the matching. For example, the rule $f(X) \iff g(X, X)$ essentially clones the term matching X. The identifiers, however, are not cloned. If a term is cloned, we expect that both copies will inherit the propagation history of the original. Likewise, terms can be merged, e.g. $g(X, X) \iff f(X)$ merges two instances of the term matching X. In this case, the propagation histories of the copies are also merged.

To achieve this we duplicate entries in the propagation history for each occurrence of a variable in the body that also appeared in the head.

Definition 6 (Updating History). *Define function*

$$\mathsf{update}(H, H_a, B, B_a, T_0) = T_1$$

where H and B are un-annotated terms, H_a and B_a are annotated terms, and T_0 and T_1 are propagation histories. T_1 is a minimal propagation history satisfying the following conditions:

- $T_0 \subseteq T_1$*;*
- $\forall p \in \mathcal{P}os(H)$ *such that* $H|_p = V \in \mathcal{X}$ *(where $\mathcal{X}$ is the set of variables), and* $\exists q \in \mathcal{P}os(B)$ *such that* $B|_q = V$*, then define identifier renaming ρ such that $\rho(H_a|_p)$ and $B_a|_q$ are identical annotated terms. Then if $E \in T_0$ we have that $\rho(E) \in T_1$.*

Example 11. Consider rewriting the term $H_a = f((a\#1 \wedge b\#2)\#3)\#4$ with a propagation history of $T_0 = \{(r \ @ \ (1\ 2))\}$ using the rule $f(X) \iff g(X, X)$. The resulting term is $B_a = g((a\#5 \wedge b\#6)\#7), (a\#8 \wedge b\#9)\#10\#11$ and the new propagation history is $T_1 = \{(r \ @ \ (1\ 2)), (r \ @ \ (5\ 6)), (r \ @ \ (8\ 9))\}$.

Conjunctive Context. According to the declarative semantics, a term T with conjunctive context C is represented as (T *where* C). Operationally, we will never explicitly build a term containing a *where* clause. Instead we use the following function to compute the conjunctive context of a subterm on demand.

Definition 7 (Conjunctive Context). *Given an (annotated) term T and a position $p \in \mathcal{P}os(T)$, we define function $\mathsf{cc}(T, p)$ to return the conjunctive context at position p as follows.*

$$\begin{aligned}
&\mathsf{cc}(T, \epsilon) &&= true \\
&\mathsf{cc}(A \wedge B, 1p) &&= B \wedge \mathsf{cc}(A, p) \\
&\mathsf{cc}(A \wedge B, 2p) &&= A \wedge \mathsf{cc}(B, p) \\
&\mathsf{cc}(f(T_1, \ldots, T_i, \ldots, T_n), ip) &&= \mathsf{cc}(T_i, p) && (f \neq \wedge)
\end{aligned}$$

States and Transitions. The operational semantics are defined as a set of transitions on execution states.

Definition 8 (Execution States). *An* execution state *is a tuple of the form* $\langle G, T, \mathcal{V}, \mathcal{P} \rangle$*, where* G *is a term (the* goal*),* T *is the* propagation history*,* $\mathcal{V}$ *is the set of variables appearing in the initial goal and* $\mathcal{P}$ *is a set of identifiers.*

We also define initial and final states as follows.

Definition 9 (Initial and Final States). *Given an initial goal* G *for program* P*, the* initial state *of* G *is*

$$\langle G_a, \emptyset, vars(G), \mathsf{ids}(G_a) \rangle$$

where $G_a = \mathsf{annotate}(\emptyset, G)$*. A* final state *is a state where no more rules are applicable to the goal* G*.*

We can now define the operational semantics of ACDTR as follows.

Definition 10 (Operational Semantics)

$$\langle G_0, T_0, \mathcal{V}, \mathcal{P}_0 \rangle \rightarrowtail \langle G_1, T_1, \mathcal{V}, \mathcal{P}_1 \rangle$$

1. Simplify: *There exists a (renamed) rule from* P

$$H \Longleftrightarrow g \mid B$$

such that there exists a matching substitution θ *and a term* G_0' *such that*

- $G_0 \approx_{AC} G_0'$
- $\exists p \in \mathcal{P}os(G_0') \ . \ G_0'|_p = \theta(H)$
- $\theta(g) \in \mathcal{G}$
- $B_a = \mathsf{annotate}(\mathcal{P}_0, \theta(B))$

Then $G_1 = G_0'[B_a]_p$*,* $\mathcal{P}_1 = \mathcal{P}_0 \cup \mathsf{ids}(G_1)$ *and* $T_1 = \mathsf{update}(H, G_0'|_p, B, B_a, T_0)$*.*

2. Propagate: *There exists a (renamed) rule from* P

$$r \ @ \ H \Longrightarrow g \mid B$$

such that there exists a matching substitution θ *and a term* G_0' *such that*

- $G_0 \approx_{AC} G_0'$
- $\exists p \in \mathcal{P}os(G_0') \ . \ G_0'|_p = \theta(H)$
- $\theta(g) \in \mathcal{G}$
- $\mathsf{entry}(r, G_0'|_p) \notin T_0$
- $B_a = \mathsf{annotate}(\mathcal{P}_0, \theta(B))$

Then $G_1 = G_0'[G_0'|_p \wedge B_a]_p$*,* $T_1 = \mathsf{update}(H, G_0'|_p, B, B_a, T_0) \cup \{\mathsf{entry}(r, G_0'|_p)\}$ *and* $\mathcal{P}_1 = \mathcal{P}_0 \cup \mathsf{ids}(G_1)$*.*

3. Simpagate: *There exists a (renamed) rule from* P

$$C \setminus H \Longleftrightarrow g \mid B$$

$\langle (leq(X_1, Y_2)_3 \wedge_4 leq(Y_5, Z_6)_7 \wedge_8 \neg_9 leq(X_{10}, Z_{11})_{12}), \emptyset \rangle \rightarrowtail_{trans}$
$\langle (leq(X_1, Y_2)_3 \wedge_4 leq(Y_5, Z_6)_7 \wedge_{13} leq(X_{15}, Z_{16})_{14} \wedge_8 \neg_9 leq(X_{10}, Z_{11})_{12}), T \rangle \rightarrowtail_{idemp}$
$\langle (leq(X_1, Y_2)_3 \wedge_4 leq(Y_5, Z_6)_7 \wedge_{13} leq(X_{15}, Z_{16})_{14} \wedge_8 \neg_9 true_{17}), T \rangle \rightarrowtail_{simplify}$
$\langle (leq(X_1, Y_2)_3 \wedge_4 leq(Y_5, Z_6)_7 \wedge_{13} leq(X_{15}, Z_{16})_{14} \wedge_8 false_{18}), T \rangle \rightarrowtail_{simplify}$
$\langle (leq(X_1, Y_2)_3 \wedge_4 leq(Y_5, Z_6)_7 \wedge_{13} false_{19}), T \rangle \rightarrowtail_{simplify}$
$\langle (leq(X_1, Y_2)_3 \wedge_4 false_{20}), T \rangle \rightarrowtail_{simplify}$
$\langle (false_{21}), T \rangle$

Fig. 1. Example derivation for the *leq* program

such that there exists a matching substitution θ *and a term* G_0' *such that*

- $G_0 \approx_{AC} G_0'$
- $\exists p \in \mathcal{P}os(G_0')\ .\ G_0'|_p = \theta(H)$
- $\exists D.\theta(C) \wedge D \approx_{AC} \mathsf{cc}(G_0', p)$
- $\theta(g) \in \mathcal{G}$
- $B_a = \mathsf{annotate}(\mathcal{P}_0, \theta(B))$

Then $G_1 = G_0'[B_a]_p$, $T_1 = \mathsf{update}(H, G_0'|_p, B, B_a, T_0)$ *and* $\mathcal{P}_1 = \mathcal{P}_0 \cup \mathsf{ids}(G_1)$.

Example. Consider the *leq* program from Example 8 with the goal

$$leq(X, Y) \wedge leq(Y, Z) \wedge \neg leq(X, Z)$$

Figure 1 shows one possible derivation of this goal to the final state representing *false*. For brevity, we omit the $\mathcal{V}$ and $\mathcal{P}$ fields, and represent identifiers as subscripts, i.e. $T\#i = T_i$. Also we substitute $T = \{\texttt{transitivity} @ (3\ 2\ 1\ 7\ 5\ 6)\}$.

We can state a soundness result for ACDTR.

Theorem 1 (Soundness). *If* $\langle G_0, T_0, \mathcal{V}, \mathcal{P} \rangle \rightarrowtail^* \langle G', T', \mathcal{V}, \mathcal{P} \rangle$ *with respect to a program* P*, then* $[\![P]\!] \models \exists_{vars(G')-\mathcal{V}}\ G_0 \approx G'$

This means that for all algebras $\mathcal{A}$ that satisfy $[\![P]\!]$, G_0 and G' are equivalent for some assignment of the fresh variables in G'.

4 Implementation

We have implemented a prototype version of ACDTR as part of the mapping language of the G12 project, called Cadmium. In this section we give an overview of the implementation details. In particular, we will focus on the implementation of conjunctive context matching, which is the main contribution of this paper.

Cadmium constructs *normalised* terms from the bottom up. Here, a *normalised* term is one that cannot be reduced further by an application of a rule. Given a goal $f(t_1, ..., t_n)$, we first must recursively normalise all of $t_1, ..., t_n$ (to say $s_1, ..., s_n$), and then attempt to find a rule that can be applied to the top-level of $f(s_1, ..., s_n)$. This is the standard execution algorithm used by many TRSs implementations.

This approach of normalising terms bottom up is complicated by the consideration of conjunctive context matching. This is because the conjunctive context of the current term appears "higher up" in the overall goal term. Thus conjunctive context must be passed top down, yet we are normalising bottom up. This means there is no guarantee that the conjunctive context is normalised.

Example 12. Consider the following ACDTR program that uses conjunctive context matching.

$$\begin{aligned} X = V \setminus X &\iff var(X) \land nonvar(V) \mid V. \\ one(X) &\iff X = 1. \\ not\_one(1) &\iff false. \end{aligned}$$

Consider the goal $not\_one(A) \land one(A)$, which we expect should be normalised to $false$. Assume that the sub-term $not\_one(A)$ is selected for normalisation first. The conjunctive context for $not\_one(A)$ (and its subterm A) is $one(A)$. No rule is applicable, so $not\_one(A)$ is not reduced.

Next the subterm $one(A)$ is reduced. The second rule will fire resulting in the new term $A = 1$. Now the conjunctive context for the first term $not\_one(A)$ has changed to $A = 1$, so we expect that A should be rewritten to the number 1. However $not\_one(A)$ has already being considered for normalisation. □

The current Cadmium prototype solves this problem by re-normalising terms when and if the conjunctive context "changes". For example, when the conjunctive context $one(A)$ changes to $A = 1$, the term $not\_one(X)$ will be renormalised to $not\_one(1)$ by the first rule.

The general execution algorithm for Cadmium is shown in Figure 2. Function normalise takes a term T, a substitution θ, a conjunctive context CC and a Boolean value Ch which keeps track of when the conjunctive context of the current subterm has changed. If $Ch = true$, then we can assume the substitution θ maps variables to normalised terms. For the initial goal, we assume θ is empty, otherwise if we are executing a body of a rule, then θ is the matching substitution.

Operationally, normalise splits into three cases depending on what T is. If T is a variable, and the conjunctive context has changed (i.e. $Ch = true$), then $\theta(T)$ is no longer guaranteed to be normalised. In this case we return the result of renormalising $\theta(T)$ with respect to CC. Otherwise if $Ch = false$, we simply return $\theta(T)$ which must be already normalised. If T is a conjunction $T_1 \land T_2$, we repeatedly call normalise on each conjunct with the other added to the conjunctive context. This is repeated until a fixed point (i.e. further normalisation does not result in either conjunct changing) is reached, and then return the result of apply_rule on the which we will discuss below. This fixed point calculation accounts for the case where the conjunctive context of a term changes, as shown in Example 12. Otherwise, if T is any other term of the form $f(T_1, ..., T_n)$, construct the new term T' by normalising each argument. Finally we return the result of apply_rule applied to T'.

The function call apply_rule(T',CC) will attempt to apply a rule to normalised term T' with respect to conjunctive context CC. If a matching rule is found,

```
normalise(T,θ,CC,Ch)
  if is_var(T)
    if Ch
      return normalise(θ(T),θ,CC,false)
    else
      return θ(T)
  else if T = T1 ∧ T2
    do
      T1' := T1
      T2' := T2
      T1 := normalise(T1',θ,T2' ∧ CC,true)
      T2 := normalise(T2',θ,T1' ∧ CC,true)
    while T1 ≠ T1' ∧ T2 ≠ T2'
    return apply_rule(T1' ∧ T2',CC)
  else
    T = f(T1, ..., Tn)
    T' := f(normalise(T1,θ,CC,Ch), ..., normalise(Tn,θ,CC,Ch))
  return apply_rule(T',CC)
```

Fig. 2. Pseudo code of the Cadmium execution algorithm

then the result of normalise(B,θ,CC,$false$) is returned, where B is the (renamed) rule body and θ is the matching substitution. Otherwise, T' is simply returned.

5 Related Work

ACDTR is closely related to both TRS and CHRs, and in this section we compare the three languages.

5.1 AC Term Rewriting Systems

The problem of dealing with associative commutative operators in TRS is well studied. A popular solution is to perform the rewriting modulo some permutation of the AC operators. Although this complicates the matching algorithm, the problem of trivial non-termination (e.g. by continually rewriting with respect to commutativity) is solved.

ACDTR subsumes ACTRS (Associative Commutative TRS) in that we have introduced distributivity (via simpagation rules), and added some “CHR-style” concepts such as identifiers and propagation rules.

Given an ACTRS program, we can map it to an equivalent ACDTR program by interpreting each ACTRS rule $H \rightarrow B$ as the ACDTR rule $H \iff B$. We can now state the theorem relating ACTRS and ACDTR.

Theorem 2. *Let P be an ACTRS program and T a ground term, then $T \rightarrow^* S$ under P iff $\langle T_a, \emptyset, \emptyset, \mathsf{ids}(T_a)\rangle \rightarrowtail^* \langle S_a, \emptyset, \emptyset, \mathcal{P}\rangle$ under $\alpha(P)$ (where $T_a =$ annotate $(\emptyset, T)$) for some $\mathcal{P}$ and $\mathsf{term}(S_a) = S$.*

5.2 CHRs and CHR$^\vee$

ACDTR has been deliberately designed to be an extension of CHRs. Several CHR concepts, e.g. propagation rules, etc., have been adapted.

There are differences between CHRs and ACDTR. The main difference is that ACDTR does not have a "built-in" or "underlying" solver, i.e. ACDTR is not a constraint programming language. However it is possible to encode solvers directly as rules, e.g. the simple *leq* solver from Example 8. Another important difference is that CHRs is based on predicate logic, where there exists a distinction between predicate symbols (i.e. the names of the constraints) and functions (used to construct terms). ACDTR is based on equational logic between terms, hence there is no distinction between predicates and functions (a predicate is just a Boolean function). To overcome this, we assume the existence of a set $\mathcal{P}red$, which contains the set of function symbols that are Boolean functions. We assume that $AC \cap \mathcal{P}red = \{\wedge^{(2)}\}$.

The mapping between a CHR program and an ACDTR program is simply $\alpha(P) = P \cup \{X \wedge true \iff X\}$. [4] However, we assume program P is restricted as follows:

- rules have no guards apart from implicit equality guards; and
- the only built-in constraint is *true*

and the initial goal G is also restricted:

- G must be of the form $G_0 \wedge ... \wedge G_n$ for $n > 0$;
- Each G_i is of the form $f_i(A_0, ..., A_m)$ for $m \geq 0$ and $f_i \in \mathcal{P}red$;
- For all $p \in \mathcal{P}os(A_j), 0 \leq j \leq m$ we have that if $A_j|_p = g(B_0, ..., B_q)$ then $g^{(q)} \notin AC$ and $g^{(q)} \notin \mathcal{P}red$.

These conditions disallow predicate symbols from appearing as arguments in CHR constraints.

Theorem 3. *Let P be a CHR program, and G an initial goal both satisfying the above conditions, then $\langle G, \emptyset, true, \emptyset\rangle_1^{\mathcal{V}} \rightarrowtail \langle \emptyset, S, true, T\rangle_i^{\mathcal{V}}$ (for some T, i and $\mathcal{V} = vars(G)$) under the theoretical operational semantics [4] for CHRs iff $\langle G_a, \emptyset, \mathcal{V}, \mathsf{ids}(G_a)\rangle \rightarrowtail \langle S_a, T', \mathcal{V}, \mathcal{P}\rangle$ (for some T', $\mathcal{P}$) under ACDTR, where $\mathsf{term}(S_a) = S_1 \wedge ... \wedge S_n$ and $S = \{S_1\#i_1, ..., S_n\#i_n\}$ for some identifiers $i_1, ..., i_n$.*

We believe that Theorem 3 could be extended to include CHR programs that extend an underlying solver, provided the rules for handling tell constraints are added to the ACDTR program. For example, we can combine rules for rational tree unification with the *leq* program from Example 8 to get a program equivalent to the traditional *leq* program under CHRs.

ACDTR generalises CHRs by allowing other operators besides conjunction inside the head or body of rules. One such extension of CHRs has been studied before, namely CHR$^\vee$ [2] which allows disjunction in the body. Unlike ACDTR,

[4] There is one slight difference in syntax: CHRs use '**,**' to represent conjunction, whereas ACDTR uses '$\wedge$'.

which manipulates disjunction syntactically, $CHR^{\vee}$typically finds solutions using backtracking search.

One notable implementation of $CHR^{\vee}$is [6], which has an operational semantics described as an and/or ($\wedge/\vee$) tree rewriting system. A limited form of conjunctive context matching is used, similar to that used by ACDTR, based on the knowledge that conjunction $\wedge$ distributes over disjunction $\vee$. ACDTR generalises this by distributing over all functions.

6 Future Work and Conclusions

We have presented a powerful new rule-based programming language, ACDTR, that naturally extends both AC term rewriting and CHRs. The main contribution is the ability to match a rule against the conjunctive context of a (sub)term, taking advantage of the distributive property of conjunction over all possible functions. We have shown this is a natural way of expressing some problems, and by building the distributive property into the matching algorithm, we avoid non-termination issues that arise from naively implementing distribution (e.g. as rewrite rules).

We intend that ACDTR will become the theoretical basis for the Cadmium constraint mapping language as part of the G12 project [7]. Work on ACDTR and Cadmium is ongoing, and there is a wide scope for future work, such as confluence, termination and implementation/optimisation issues.

References

1. S. Abdennadher. Operational semantics and confluence of constraint propagation rules. In Gert Smolka, editor, *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming*, LNCS 1330, pages 252–266. Springer-Verlag, 1997.
2. S. Abdennadher and H. Schütz. $CHR^{\vee}$: A flexible query language. In *International conference on Flexible Query Answering Systems*, number 1495 in LNCS, pages 1–14, Roskilde, Denmark, 1998. Springer-Verlag.
3. F. Baader and T. Nipkow. *Term rewriting and all that*. Cambridge Univ. Press, 1998.
4. G. Duck, P. Stuckey, M. Garcia de la Banda, and C. Holzbaur. The refined operational semantics of constraint handling rules. In B. Demoen and V. Lifschitz, editors, *Proceedings of the 20th International Conference on Logic Programming*, LNCS 3132, pages 90–104. Springer-Verlag, September 2004.
5. T. Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming*, 37:95–138, 1998.
6. L. Menezes, J. Vitorino, and M. Aurelio. A High Performance $CHR^{\vee}$ Execution Engine. In *Second Workshop on Constraint Handling Rules*, Sitges, Spain, 2005.
7. P.J. Stuckey, M. Garcia 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 M. Gabrielli and G. Gupta, editors, *Proceedings of the 21st International Conference on Logic Programming*, number 3668 in LNCS, pages 9–13. Springer-Verlag, 2005.