On the implementation of construction functions for non-free concrete data types

Frédéric Blanqui[1], Thérèse Hardin[2], and Pierre Weis[3]

[1] INRIA & LORIA, BP 239, 54506 Villers-lès-Nancy Cedex, France
[2] UPMC, LIP6, 104, Av. du Pr. Kennedy, 75016 Paris, France
[3] INRIA, Domaine de Voluceau, BP 105, 78153 Le Chesnay Cedex, France

Abstract. Many algorithms use concrete data types with some additional invariants. The set of values satisfying the invariants is often a set of representatives for the equivalence classes of some equational theory. For instance, a sorted list is a particular representative wrt commutativity. Theories like associativity, neutral element, idempotence, etc. are also very common. Now, when one wants to combine various invariants, it may be difficult to find the suitable representatives and to efficiently implement the invariants. The preservation of invariants throughout the whole program is even more difficult and error prone. Classically, the programmer solves this problem using a combination of two techniques: the definition of appropriate construction functions for the representatives and the consistent usage of these functions ensured via compiler verifications. The common way of ensuring consistency is to use an abstract data type for the representatives; unfortunately, pattern matching on representatives is lost. A more appealing alternative is to define a concrete data type with private constructors so that both compiler verification and pattern matching on representatives are granted. In this paper, we detail the notion of private data type and study the existence of construction functions. We also describe a prototype, called Moca, that addresses the entire problem of defining concrete data types with invariants: it generates efficient construction functions for the combination of common invariants and builds representatives that belong to a concrete data type with private constructors.

1 Introduction

Many algorithms use data types with some additional invariants. Every function creating a new value from old ones must be defined so that the newly created value satisfy the invariants whenever the old ones so do.

One way to easily maintain invariants is to use abstract data types (ADT): the implementation of an ADT is hidden and construction and observation functions are provided. A value of an ADT can only be obtained by recursively using the construction functions. Hence, an invariant can be ensured by using appropriate construction functions. Unfortunately, abstract data types preclude pattern matching, a very useful feature of modern programming languages [10, 11, 16, 15]. There have been various attempts to combine both features in some way.

In [23], P. Wadler proposed the mechanisms of *views*. A view on an ADT α is given by providing a concrete data type (CDT) γ and two functions $in : \alpha \to \gamma$ and $out : \gamma \to \alpha$ such that $in \circ out = id_\gamma$ and $out \circ in = id_\alpha$. Then, a function on α can be defined by matching on γ (by implicitly using *in*) and the values of type γ obtained by matching can be injected back into α (by implicitly using *out*). However, by leaving the applications of *in* and *out* implicit, we can easily get inconsistencies whenever *in* and *out* are not inverses of each other. Since it may be difficult to satisfy this condition (consider for instance the translations between cartesian and polar coordinates), these views have never been implemented. Following the suggestion of W. Burton and R. Cameron to use the *in* function only [3], some propositions have been made for various programming languages but none has been implemented yet [4, 17].

In [3], W. Burton and R. Cameron proposed another very interesting idea which seems to have attracted very little attention. An ADT must provide construction and observation functions. When an ADT is implemented by a CDT, they propose to also export the constructors of the CDT but only for using them as patterns in pattern matching clauses. Hence, the constructors of the underlying CDT can be used for pattern matching but not for building values: only the construction functions can be used for that purpose. Therefore, one can both ensure some invariants and offer pattern matching. These types have been introduced in OCaml by the third author [24] under the name of *concrete data type with private constructors*, or *private data type* (PDT) for short.

Now, many invariants on concrete data types can be related to some equational theory. Take for instance the type of *list* with the constructors [] and ::. Given some elements $v_1..v_n$, the sorted list which elements are $v_1..v_n$ is a particular representative of the equivalence class of $v_1{::}..{::}v_n{::}[]$ modulo the equation $x{::}y{::}l{=}y{::}x{::}l$. Requiring that, in addition, the list does not contain the same element twice is a particular representative modulo the equation $x{::}x{::}l{=}x{::}l$.

Consider now the type of join lists with the constructors *empty*, *singleton* and *append*, for which concatenation is of constant complexity. Sorting corresponds to associativity and commutativity of *append*. Requiring that no argument of *append* is *empty* corresponds to neutrality of *empty* wrt *append*. We have a structure of commutative monoid.

More generally, given some equational theory on a concrete data type, one may wonder whether there exists a representative for each equivalence class and, if so, whether a representative of $C(t_1 \ldots t_n)$ can be efficiently computed knowing that $t_1 \ldots t_n$ are themselves representatives.

In [21, 22], S. Thompson describes a mechanism introduced in the Miranda functional programming language for implementing such non-free concrete data types without precluding pattern matching. The idea is to provide conditional rewrite rules, called *laws*, that are implicitly applied as long as possible on every newly created value. This can also be achieved by using a PDT which construction functions (primed constructors in [21]) apply as long as possible each of the laws. Then, S. Thompson studies how to prove the correctness of functions defined by pattern matching on such *lawful types*. However, few hints are given on how to check whether the laws indeed implement the invariants one has in mind. For this reason and because reasoning on lawful types is difficult, the law mechanism was removed from Miranda.

In this paper, we propose to specify the invariants by unoriented equations (instead of rules). We will call such a type a *relational data type* (RDT). Sections 2 and 3 introduce private and relational data types. Then, we study when an RDT can be implemented by a PDT, that is, when there exist construction functions computing some representative for each equivalence class. Section 4 provides some general existence theorem based on rewriting theory. But rewriting may be inefficient. Section 5 provides, for some common equational theories, construction functions more efficient than the ones based on rewriting. Section 6 presents Moca, an extension of OCaml with relational data types whose construction functions are automatically generated. Finally, Section 7 discusses some possible extensions.

2 Concrete data types with private constructors

We first recall the definition of a first-order term algebra. It will be useful for defining the values of concrete and private data types.

Definition 1 (First-order term algebra) A *sorted term algebra definition* is a triplet $\mathcal{A} = (\mathcal{S}, \mathcal{C}, \Sigma)$ where $\mathcal{S}$ is a non-empty set of *sorts*, $\mathcal{C}$ is a non-empty set of *constructor symbols* and $\Sigma : \mathcal{C} \to \mathcal{S}^+$ is a *signature* mapping a non-empty sequence of sorts to every constructor symbol. We write $C : \sigma_1 \dots \sigma_n \sigma_{n+1} \in \Sigma$ to denote the fact that $\Sigma(C) = \sigma_1 \dots \sigma_n \sigma_{n+1}$. Let $\mathcal{X} = (\mathcal{X}_\sigma)_{\sigma \in \mathcal{S}}$ be a family of pairwise disjoint sets of *variables*. The sets $\mathcal{T}_\sigma(\mathcal{A}, \mathcal{X})$ of *terms of sort* σ are inductively defined as follows:

- If $x \in \mathcal{X}_\sigma$, then $x \in \mathcal{T}_\sigma(\mathcal{A}, \mathcal{X})$.
- If $C : \sigma_1 \dots \sigma_{n+1} \in \Sigma$ and $t_i \in \mathcal{T}_{\sigma_i}(\mathcal{A}, \mathcal{X})$, then $C(t_1, \dots, t_n) \in \mathcal{T}_{\sigma_{n+1}}(\mathcal{A}, \mathcal{X})$.

Let $\mathcal{T}_\sigma(\mathcal{A})$ be the set of terms of sort σ containing no variable.

In the following, we assume given a set $\mathcal{S}_0$ of primitive types like `int`, `string`, ... and a set $\mathcal{C}_0$ of primitive constants `0`, `1`, `"foo"`, ... Let Σ_0 be the corresponding signature ($\Sigma_0(\mathtt{0}) = \mathtt{int}, \dots$).

In this paper, we call *concrete data type* (CDT) an inductive type *à la* ML defined by a set of *constructors*. More formally:

Definition 2 (Concrete data type) A *concrete data type definition* is a triplet $\Gamma = (\gamma, \mathcal{C}, \Sigma)$ where γ is a sort, $\mathcal{C}$ is a non-empty set of *constructor symbols* and $\Sigma : \mathcal{C} \to (\mathcal{S}_0 \cup \{\gamma\})^+$ is a *signature* such that, for all $C \in \mathcal{C}$, $\Sigma(C) = \sigma_1..\sigma_n\gamma$. The set $Val(\gamma)$ of *values of type* γ is the set of terms $\mathcal{T}_\gamma(\mathcal{A}_\Gamma)$ where $\mathcal{A}_\Gamma = (\mathcal{S}_0 \cup \{\gamma\}, \mathcal{C}_0 \cup \mathcal{C}, \Sigma_0 \cup \Sigma)$.

This definition of CDTs corresponds to a small but very useful subset of all the possible types definable in ML-like programming languages. For the purpose of this paper, it is not necessary to use a more complex definition.

Example 1 The following type[4] `cexp` is a CDT definition with two constant constructors of sort `cexp` and a binary operator of sort `cexp cexp cexp`.

```
type cexp =  Zero | One | Opp of cexp | Plus of cexp * cexp
```

Now, a private data type definition is like a CDT definition together with construction functions as in abstract data types. Constructors can be used as patterns as in concrete data types but they *cannot* be used for value creation (except in the definition of construction functions). For building values, one must use construction functions as in abstract data types. Formally:

Definition 3 (Private data type) A *private data type definition* is a pair $\Pi = (\Gamma, \mathcal{F})$ where $\Gamma = (\pi, \mathcal{C}, \Sigma)$ is a CDT definition and $\mathcal{F}$ is a family of *construction functions* $(f_C)_{C \in \mathcal{C}}$ such that, for all $C : \sigma_1..\sigma_n\pi \in \Sigma$, $f_C : \mathcal{T}_{\sigma_1}(\mathcal{A}_\Gamma) \times \ldots \times \mathcal{T}_{\sigma_n}(\mathcal{A}_\Gamma) \to \mathcal{T}_\pi(\mathcal{A}_\Gamma)$. Let $Val(\pi)$ be the set of the *values of type* π, that is, the set of terms that one can build by using the construction functions only. The function $f : \mathcal{T}_\pi(\mathcal{A}_\Gamma) \to \mathcal{T}_\pi(\mathcal{A}_\Gamma)$ such that, for all $C : \sigma_1..\sigma_n\pi \in \Sigma$ and $t_i \in \mathcal{T}_{\sigma_i}(\mathcal{A}_\Gamma)$, $f(C(t_1..t_n)) = f_C(f(t_1)..f(t_n))$, is called the *normalization function associated to* $\mathcal{F}$.

This is quite immediate to see that:

Lemma 1. *$Val(\pi)$ is the image of f.*

PDTs have been implemented in OCaml by the third author [24]. Extending a programming language with PDTs is not very difficult: one only needs to modify the compiler to parse the PDT definitions and check that the conditions on the use of constructors are fulfilled.

Note that construction functions have no constraint in general: the full power of the underlying programming language is available to define them.

It should also be noted that, because the set of values of type π is a subset of the set of values of the underlying CDT γ, a function on π defined by pattern matching may be a total function even though it is not defined on all the possible cases of γ. Defining a function with patterns that match no value of type π does not harm since the corresponding code will never be run. It however reveals that the developer is not aware of the distinction between the values of the PDT and those of the underlying CDT, and thus can be considered as a programming error. To avoid this kind of errors, it is important that a PDT comes with a clear identification of its set of possible values. To go one step further, one could provide a tool for checking the completeness and usefulness of patterns that takes into account the invariants, when it is possible. We leave this for future work.

Example 2 Let us now start our running example with the type `exp` describing operations on arithmetic expressions.

[4] Examples are written with OCaml [10], they can be readily translated in any programming language offering pattern-matching with textual priority, as Haskell, SML, etc.

```
type exp = private Zero | One | Opp of exp | Plus of exp * exp
```

This type `exp` is indeed a PDT built upon the CDT `cexp`. Prompted by the keyword `private`, the OCaml compiler forbids the use of `exp` constructors (outside the module `my_exp.ml` containing the definition of `exp`) except in patterns. If `Zero` is supposed to be neutral by the writer of `my_exp.ml`, then he/she will provide construction functions as follows:

```
let rec zero = Zero and one = One and opp x = Opp x
and plus = function
| (Zero,y) ->  y
| (y,Zero) -> y
| (x,y) -> Plus(x,y)
```

3 Relational data types

We mentioned in the introduction that, often, the invariants upon concrete data types are such that the set of values satisfying them is indeed a set of representatives for the equivalence classes of some equational theory. We therefore propose to specify invariants by a set of unoriented equations and study to which extent such a specification can be realized with an abstract or private data type. In case of a private data type however, it is important to be able to describe the set of possible values.

Definition 4 (Relational data type) A *relational data type (RDT) definition* is a pair $(\Gamma, \mathcal{E})$ where $\Gamma = (\pi, \mathcal{C}, \Sigma)$ is a CDT definition and $\mathcal{E}$ is a finite set of equations on $\mathcal{T}_\pi(\mathcal{A}_\Gamma, \mathcal{X})$. Let $=_\mathcal{E}$ be the smallest congruence relation containing $\mathcal{E}$. Such an RDT is *implementable* by a PDT $(\Gamma, \mathcal{F})$ if the family of construction functions $\mathcal{F} = (f_C)_{C \in \mathcal{C}}$ is *valid wrt* $\mathcal{E}$:

(Correctness) For all $C : \sigma_1..\sigma_n\pi$ and $v_i \in Val(\sigma_i)$, $f_C(v_1..v_n) =_\mathcal{E} C(v_1..v_n)$.

(Completeness) For all $C : \sigma_1..\sigma_n\sigma$, $v_i \in Val(\sigma_i)$, $D : \tau_1..\tau_p\sigma \in \Sigma$ and $w_i \in Val(\tau_i)$, $f_C(v_1..v_n) = f_D(w_1..w_p)$ whenever $C(v_1..v_n) =_\mathcal{E} D(w_1..w_p)$.

We are going to see that the existence of a valid family of construction functions is equivalent to the existence of a valid normalization function:

Definition 5 (Valid normalization function) A map $f : \mathcal{T}_\pi(\mathcal{A}_\Gamma) \rightarrow \mathcal{T}_\pi(\mathcal{A}_\Gamma)$ is a *valid normalization function* for an RDT $(\Gamma, \mathcal{E})$ with $\Gamma = (\pi, \mathcal{C}, \Sigma)$ if:

(Correctness) For all $t \in \mathcal{T}_\pi(\mathcal{A}_\Gamma)$, $f(t) =_\mathcal{E} t$.

(Completeness) For all $t, u \in \mathcal{T}_\pi(\mathcal{A}_\Gamma)$, $f(t) = f(u)$ whenever $t =_\mathcal{E} u$.

Note that a valid normalization function is idempotent ($f \circ f = f$) and provides a decision procedure for $=_\mathcal{E}$ (the boolean function $\lambda xy.f(x) = f(y)$).

Theorem 6 The normalization function associated to a valid family is a valid normalization function.

Proof.

- Correctness. We proceed by induction on the size of $t \in \mathcal{T}_\pi$. We have $C : \sigma_1..\sigma_n\pi \in \Sigma$ and t_i such that $t = C(t_1..t_n)$. By definition, $f(t) = f_C(f(t_1).. f(t_n))$. By induction hypothesis, $f(t_i) =_\mathcal{E} t_i$. Since the family is valid and $f(t_1)..f(t_n)$ are values, $f_C(f(t_1)..f(t_n)) =_\mathcal{E} C(f(t_1)..f(t_n))$. Thus, $f(t) =_\mathcal{E} t$.
- Completeness. Let $t, u \in \mathcal{T}_\pi$ such that $t =_\mathcal{E} u$. We have $t = C(t_1..t_n)$ and $u = D(u_1..u_p)$. By definition, $f(t) = f_C(f(t_1)..f(t_n))$ and $f(u) = f_D(f(u_1)..f(u_p))$. By correctness, $f(t_i) =_\mathcal{E} t_i$ and $f(u_j) =_\mathcal{E} u_j$. Hence, $C(f(t_1)..f(t_n)) =_\mathcal{E} D(f(u_1)..f(u_p))$. Since the family is valid and $f(t_1)..f(t_n)$ are values, $f_C(f(t_1) ..f(t_n)) = f_D(f(t_1)..f(t_n))$. Thus, $f(t) = f(u)$. ■

Conversely, given $f : \mathcal{T}_\pi(\mathcal{A}_\Gamma) \rightarrow \mathcal{T}_\pi(\mathcal{A}_\Gamma)$, one can easily define a family of construction functions that is valid whenever f is a valid normalization function.

Definition 7 (Associated family of constr. functions) Given a CDT $\Gamma = (\pi, \mathcal{C}, \Sigma)$ and a function $f : \mathcal{T}_\pi(\mathcal{A}_\Gamma) \rightarrow \mathcal{T}_\pi(\mathcal{A}_\Gamma)$, the *family of construction functions associated to* f is the family $(f_C)_{C\in\mathcal{C}}$ such that, for all $C : \sigma_1..\sigma_n\pi \in \Sigma$ and $t_i \in \mathcal{T}_{\sigma_1}(\mathcal{A}_\Gamma)$, $f_C(t_1, \ldots, t_n) = f(C(t_1, \ldots, t_n))$.

Theorem 8 The family of construction functions associated to a valid normalization function is valid.

Example 3 We can choose `cexp` as the underlying CDT and $\mathcal{E} = \{$ `Plus x Zero = x`$\}$ to define a RDT implementable by the PDT `exp`, with the valid family of construction functions `zero`, `one`, `opp`, `plus`.

4 On the existence of construction functions

In this section, we provide a general theorem for the existence of valid families of construction functions based on rewriting theory. We recall the notions of rewriting and completion. The interested reader may find more details in [8].

Standard rewriting. A *rewrite rule* is an ordered pair of terms (l, r) written $l \rightarrow r$. A rule is *left-linear* if no variable occurs twice in its left hand side l.

As usual, the set $\mathrm{Pos}(t)$ of *positions in* t is defined as a set of words on positive integers. Given $p \in \mathrm{Pos}(t)$, let $t|_p$ be the subterm of t at position p and $t[u]_p$ be the term t with $t|_p$ replaced by u.

Given a finite set $\mathcal{R}$ of rewrite rules, the *rewriting relation* is defined as follows: $t \rightarrow_\mathcal{R} u$ iff there are $p \in \mathrm{Pos}(t)$, $l \rightarrow r \in \mathcal{R}$ and a substitution θ such that $t|_p = l\theta$ and $u = t[r\theta]_p$. A term t is an *$\mathcal{R}$-normal form* if there is no u such that $t \rightarrow_\mathcal{R} u$. Let $=_\mathcal{R}$ be the symmetric, reflexive and transitive closure of $\rightarrow_\mathcal{R}$.

A *reduction ordering* $\succ$ is a well-founded ordering (there is no infinitely decreasing sequence $t_0 \succ t_1 \succ \ldots$) stable by context ($C(..t..) \succ C(..u..)$ whenever $t \succ u$) and substitution ($t\theta \succ u\theta$ whenever $t \succ u$). If $\mathcal{R}$ is included in a reduction ordering, then $\rightarrow_\mathcal{R}$ is well-founded (terminating, strongly normalizing).

We say that $\rightarrow_{\mathcal{R}}$ is *confluent* if, for all terms t, u, v such that $u \leftarrow^*_{\mathcal{R}} t \rightarrow^*_{\mathcal{R}} v$, there exists a term w such that $u \rightarrow^*_{\mathcal{R}} w \leftarrow^*_{\mathcal{R}} v$. This means that the relation $\leftarrow^*_{\mathcal{R}}\rightarrow^*_{\mathcal{R}}$ is included in the relation $\rightarrow^*_{\mathcal{R}}\leftarrow^*_{\mathcal{R}}$ (composition of relations is written by juxtaposition).

If $\rightarrow_{\mathcal{R}}$ is confluent, then every term has at most one normal form. If $\rightarrow_{\mathcal{R}}$ is well-founded, then every term has at least one normal form. Therefore, if $\rightarrow_{\mathcal{R}}$ is confluent and terminating, then every term has a unique normal form.

Standard completion. Given a finite set $\mathcal{E}$ of equations and a reduction ordering $\succ$, the standard Knuth-Bendix completion procedure [2] tries to find a finite set $\mathcal{R}$ of rewrite rules such that:

- $\mathcal{R}$ is included in $\succ$,
- $\rightarrow_{\mathcal{R}}$ is confluent,
- $\mathcal{R}$ and $\mathcal{E}$ have same theory: $=_{\mathcal{E}} \; = \; =_{\mathcal{R}}$.

Note that completion may fail or not terminate but, in case of successful termination, $\mathcal{R}$-normalization provides a decision procedure for $=_{\mathcal{E}}$ since $t =_{\mathcal{E}} u$ iff the $\mathcal{R}$-normal forms of t and u are syntactically equal.

However, since permutation theories like commutativity or associativity and commutativity together (written AC for short) are included in no reduction ordering, dealing with them requires to consider rewriting with pattern matching modulo these theories and completion modulo these theories. In this paper, we restrict our attention to AC.

Definition 9 (Associative-commutative equations) Let Com be the set of commutative constructors, *i.e.* the set of constructors C such that $\mathcal{E}$ contains an equation of the form $C(x, y) = C(y, x)$. Then, let $\mathcal{E}_{AC}$ be the subset of $\mathcal{E}$ made of the commutativity and associativity equations for the commutative constructors, $=_{AC}$ be the smallest congruence relation containing $\mathcal{E}_{AC}$ and $\mathcal{E}_{\neg AC} = \mathcal{E} \setminus \mathcal{E}_{AC}$.

Rewriting modulo AC. Given a set $\mathcal{R}$ of rewrite rules, *rewriting with pattern matching modulo AC* is defined as follows: $t \rightarrow_{\mathcal{R},AC} u$ iff there are $p \in \mathrm{Pos}(t)$, $l \rightarrow r \in \mathcal{R}$ and a substitution θ such that $t|_p =_{AC} l\theta$ and $u = t[r\theta]_p$. A reduction ordering $\succ$ is *AC-compatible* if, for all terms t, t', u, u' such that $t =_{AC} t'$ and $u =_{AC} u'$, $t' \succ u'$ iff $t \succ u$. The relation $\rightarrow_{\mathcal{R},AC}$ is *confluent modulo AC* if $(\leftarrow^*_{\mathcal{R},AC}=_{AC}\rightarrow^*_{\mathcal{R},AC}) \subseteq (\rightarrow^*_{\mathcal{R},AC}=_{AC}\leftarrow^*_{\mathcal{R},AC})$.

Completion modulo AC. Given a finite set $\mathcal{E}$ of equations and an AC-compatible reduction ordering $\succ$, completion modulo AC [18] tries to find a finite set $\mathcal{R}$ of rules such that:

- $\mathcal{R}$ is included in $\succ$,
- $\rightarrow_{\mathcal{R},AC}$ is confluent modulo AC,
- $\mathcal{E}$ and $\mathcal{R} \cup \mathcal{E}_{AC}$ have same theory: $=_{\mathcal{E}} \; = \; =_{\mathcal{R}\cup\mathcal{E}_{AC}}$.

Definition 10 A theory $\mathcal{E}$ has a *complete presentation* if there is an AC-compatible reduction ordering for which the AC-completion of $\mathcal{E}_{\neg AC}$ successfully terminates.

Many interesting systems have a complete presentation: (commutative) monoids, (abelian) groups, rings, etc. See [13, 5] for a catalog. Moreover, there are automated tools implementing completion modulo AC. See for instance [6, 12].

A term may have distinct $\mathcal{R}, AC$-normal forms but, by confluence modulo AC, all normal forms are AC-equivalent and one can easily define a notion of normal form for AC-equivalent terms [13]:

Definition 11 (*AC*-normal form) Given an associative and commutative constructor C, *C-left-combs* (resp. *C-right-combs*) and their *leaves* are inductively defined as follows:

- If t is not headed by C, then t is both a C-left-comb and a C-right-comb. The *leaves* of t is the one-element list $\mathit{leaves}(t) = [t]$.
- If t is not headed by C and u is a C-right-comb, then $C(t, u)$ is a C-right-comb. The *leaves* of $C(t, u)$ is the list $t :: \mathit{leaves}(u)$.
- If t is not headed by C and u is a C-left-comb, then $C(u, t)$ is a C-left-comb. The *leaves* of $C(u, t)$ is the list $\mathit{leaves}(u)@[t]$, where @ is the concatenation.

Let *orient* be a function associating a kind of combs (left or right) to every AC-constructor. Let $\leq$ be a total ordering on terms. Then, a term t is in *AC-normal form wrt orient and $\leq$* if:

- Every subterm of t headed by an AC-constructor C is an $\mathit{orient}(C)$-comb whose leaves are in increasing order wrt $\leq$.
- For every subterm of t of the form $C(u, v)$ with C commutative but non-associative, we have $u \leq v$.

As it is well-known, one can put any term in AC-normal form:

Theorem 12 Whatever the function *orient* and the ordering $\leq$ are, every term t has an AC-normal form $t{\downarrow_{AC}}$ wrt *orient* and $\leq$, and $t =_{AC} t{\downarrow_{AC}}$.

Proof. Let $\mathcal{A}$ be the set of rules obtained by choosing an orientation for the associativity equations of $\mathcal{E}_{AC}$ according to *orient*:

- If $\mathit{orient}(C)$ is "left", then take $C(x, C(y, z)) \rightarrow C(C(x, y), z)$.
- If $\mathit{orient}(C)$ is "right", then take $C(C(x, y), z) \rightarrow C(x, C(y, z))$.

$\rightarrow_{\mathcal{A}}$ is a confluent and terminating relation putting every subterm headed by an AC-constructor into a comb form according to *orient*. Let *comb* be a function computing the $\mathcal{A}$-normal form of a term. Let now *sort* be a function permuting the leaves of combs and the arguments of commutative but non-associative constructors to put them in increasing order wrt $\leq$. Then, the function $\mathit{sort} \circ \mathit{comb}$ computes the AC-normal form of any term and $\mathit{sort}(\mathit{comb}(t)) =_{AC} t$. ■

This naturally provides a decision procedure for AC-equivalence: the function $\lambda xy.\mathit{sort}(\mathit{comb}(x)) = \mathit{sort}(\mathit{comb}(y))$. It follows that $\mathcal{R}, AC$-normalization together with AC-normalization provides a valid normalization function, hence the existence of a valid family of construction functions:

Theorem 13 If $\mathcal{E}$ has a complete presentation, then there exists a valid family of construction functions.

Proof. Assume that $\mathcal{E}$ has a complete presentation $\mathcal{R}$. We define the computation of normal forms as it is generally implemented in rewriting tools. Let *step* be a function making an $\mathcal{R}, AC$-rewrite step if there is one, or failing if the term is in normal form. Let *norm* be the function applying *step* until a normal form is reached. Since $\mathcal{R}$ is a complete presentation of $\mathcal{E}$, by definition of the completion procedure, $sort \circ comb \circ norm$ is a valid normalization function. Thus, by Theorem 8, the associated family of construction functions is valid. ■

The construction functions described in the proof are not very efficient since they are based on rewriting with pattern matching modulo AC, which is NP-complete [1], and do not take advantage of the fact that, by definition of PDTs, they are only applied to terms already in normal form. We can therefore wonder whether they can be defined in a more efficient way for some common equational theories like the ones of Figure 1.

Fig. 1. Some common equations on binary constructors

Name	Abbrev	Definition	Example
associativity	$Assoc(C)$	$C(C(x,y),z) = C(x,C(y,z))$	$(x+y)+z = x+(y+z)$
commutativity	$Com(C)$	$C(x,y) = C(y,x)$	$x+y = y+x$
neutrality	$Neu(C,E)$	$C(x,E) = x$	$x+0 = x$
inverse	$Inv(C,I,E)$	$C(x,I(x)) = E$	$x+(-x) = 0$
idempotence	$Idem(C)$	$C(x,x) = x$	$x \wedge x = x$
nilpotence	$Nil(C,A)$	$C(x,x) = A$	$x \oplus x = \bot$ (exclusive or)

Rewriting provides also a way to check the validity of construction functions:

Theorem 14 If $\mathcal{E}$ has a complete presentation $\mathcal{R}$ and $\mathcal{F} = (f_C)_{C \in \mathcal{C}}$ is a family such that, for all $C : \sigma_1..\sigma_n\pi \in \Sigma$ and terms $v_i \in Val(\sigma_i)$, $f_C(v_1..v_n)$ is an $\mathcal{R}, AC$-normal form of $C(v_1..v_n)$ in AC-normal form, then $\mathcal{F}$ is valid.

Proof.

- Correctness. Let $C : \sigma_1..\sigma_n\pi \in \Sigma$ and $v_i \in Val(\sigma_i)$. Since $f_C(v_1..v_n)$ is an $\mathcal{R}, AC$-normal form of $C(v_1..v_n)$, we clearly have $f_C(v_1..v_n) =_{\mathcal{E}} C(v_1..v_n)$.
- Completeness. Let $C : \sigma_1..\sigma_n\pi \in \Sigma$, $v_i \in Val_{\mathcal{F}}(\sigma_i)$, $D : \tau_1..\tau_p\pi \in \Sigma$, and $w_i \in Val_{\mathcal{F}}(\tau_i)$ such that $C(v_1..v_n) =_{\mathcal{E}} D(w_1..w_p)$. Since $\mathcal{R}$ is a complete presentation of $\mathcal{E}$, $norm(C(v_1..v_n)) =_{AC} norm(D(w_1..w_p))$. Thus, $f_C(v_1..v_n) = f_D(w_1..w_p)$. ■

It follows that rewriting provides a natural way to explain what are the possible values of an RDT: values are AC-normal forms matching no left hand side of a rule of $\mathcal{R}$.

5 Towards efficient construction functions

When there is no commutative symbol, construction functions can be easily implemented by simulating innermost rewriting as follows:

Definition 15 (Linearization) Let $\mathrm{VPos}(t)$ be the set of positions $p \in \mathrm{Pos}(t)$ such that $t|_p$ is a variable $x \in \mathcal{X}$. Let $\rho : \mathrm{VPos}(t) \to \mathcal{X}$ be an injective mapping and $\mathit{lin}(t)$ be the term obtained by replacing in t every subterm at position $p \in \mathrm{VPos}(t)$ by $\rho(p)$. Let now $\mathit{Eq}(t)$ be the conjunction of `true` and of the equations $\rho(p) = \rho(q)$ such that $t|_p = t|_q$ and $p, q \in \mathrm{VPos}(t)$.

Definition 16 Given a set $\mathcal{R}$ of rewrite rules, let $\mathcal{F}(\mathcal{R})$ be the family of construction functions $(f_C)_{C \in \mathcal{C}}$ defined as follows:

- For every rule $l \to r \in \mathcal{R}$ with $l = C(l_1, \ldots, l_n)$, add to the definition of f_C the clause $\mathit{lin}(l_1), \ldots, \mathit{lin}(l_n)$ `when` $\mathit{Eq}(l)$ `->` $\widehat{\mathit{lin}(r)}$, where $\widehat{t}$ is the term obtained by replacing in t every occurrence of a constructor C by a call to its construction function f_C.
- Terminate the definition of f_C by the *default clause* `x -> C(x)`.

Theorem 17 Assume that $\mathcal{E}_{AC} = \emptyset$ and $\mathcal{E}$ has a complete presentation $\mathcal{R}$. Then, $\mathcal{F}(\mathcal{R})$ is valid wrt $\mathcal{E}$ (whatever the order of the non-default clauses is).

We now consider the case of commutative symbols. We are going to describe a modular way of defining the construction functions by pursuing our running example, with the type `exp`. Assume that `Plus` is declared to be associative and commutative only. The construction functions can then be defined as follows:

```
let zero = Zero and one = One and opp x = Opp x

and plus = function
| Plus(x,y), z -> plus (x, plus (y,z))
| x, y -> insert_plus x y

and insert_plus x = function
| Plus(y,_) as u when x <= y -> Plus(x,u)
| Plus(y,t) -> Plus (y, insert_plus x t)
| u when x > u -> Plus(u,x)
| u -> Plus(x,u)
```

One can easily see that `plus` does the same job as the function *sort* $\circ$ *comb* used in Theorem 12 but in a slightly more efficient way since $\mathcal{A}$-normalization and sorting are interleaved.

Assume moreover that `Zero` is neutral. The AC-completion of $\{$ `Plus(Zero,`x`)` $= x\}$ gives $\{$ `Plus(Zero,`x`)` $\to x\}$. Hence, if x and y are terms in normal form, then `Plus(`x, y`)` can be rewritten modulo AC only if $x =$ `Zero` or $y =$ `Zero`. Thus, the function `plus` needs to be extended with two new clauses only:

```
and plus = function
| Zero, y -> y
| x, Zero -> x
| Plus(x,y), z -> plus (x, plus (y,z))
| x, y -> insert_plus x y
```

Assume now that `Plus` is declared to have `Opp` as inverse. Then, the completion modulo AC of { $\texttt{Plus}(\texttt{Zero}, x) = x$, $\texttt{Plus}(\texttt{Opp}(x), x) = \texttt{Zero}$} gives the following well known rules for abelian groups [13]: { $\texttt{Plus}(\texttt{Zero}, x) \to x$, $\texttt{Plus}(\texttt{Opp}(x), x) \to \texttt{Zero}$, $\texttt{Plus}(\texttt{Plus}(\texttt{Opp}(x), x), y) \to y$, $\texttt{Opp}(\texttt{Zero}) \to \texttt{Zero}$, $\texttt{Opp}(\texttt{Opp}(x)) \to x$, $\texttt{Opp}(\texttt{Plus}(x, y)) \to \texttt{Plus}(\texttt{Opp}(y),\texttt{Opp}(x))$ }.

The rules for `Opp` are easily translated as follows:

```
and opp = function
| Zero -> Zero
| Opp(x) -> x
| Plus(x,y) -> plus (opp y, opp x)
| _ -> Opp(x)
```

The third rule of abelian groups is called an *extension* of the second one since it is obtained by first adding the context $Plus([], y)$ on both sides of this second rule,then normalizing the right hand side. Take now two terms x and y in normal form and assume that (x, y) matches none of the three clauses previously defining `plus`, that is, x and y are distinct from `Zero`, and x is not of the form $\texttt{Plus}(x_1, x_2)$. To get the normal form of $\texttt{Plus}(x, y)$, we need to check that x and the normal form of its opposite $\texttt{Opp}(x)$ do not occur in y. The last clause defining `plus` needs therefore to be modified as follows:

```
and plus = function
| Zero, y -> y
| x, Zero -> x
| Plus(x,y), z -> plus (x, plus (y,z))
| x, y -> insert_opp_plus (opp x) y

and insert_opp_plus x y =
  try delete_plus x y
  with Not_found -> insert_plus (opp x) y

and delete_plus x = function
| Plus(y,_) when x < y -> raise Not_found
| Plus(y,t) when x = y -> t
| Plus(y,t) -> Plus (y, delete_plus x t)
| y when y = x -> Zero
| _ -> raise Not_found
```

Forgetting about `Zero` and `Opp`, suppose now that `Plus` is declared associative, commutative and idempotent. The function `plus` is kept but the `insert` function is modified as follows:

```
and insert_plus x = function
| Plus(y,_) as u when x = y -> u
| Plus(y,_) as u when x < y -> Plus(x,u)
| Plus(y,t) -> Plus (y,insert_plus x t)
| u when x > u -> Plus(u,x)
| u when x = u -> u
| u -> Plus(x,u)
```

Nilpotence can be dealt with in a similar way.

In conclusion, for various combinations of the equations of Figure 1, we can define in a nice modular way construction functions that are more efficient than the ones based on rewriting modulo AC. We summarize this as follows:

Definition 18 A set of equations $\mathcal{E}$ is a theory of type:
(1) if $\mathcal{E}_{AC} = \emptyset$ and $\mathcal{E}$ has a complete presentation,
(2) if $\mathcal{E}$ is the union of $\{Assoc(C), Com(C)\}$ with either $\{Neu(C,E), Inv(C,I,E)\}$, $\{Idem(C)\}$, $\{Neu(C,E), Idem(C)\}$ $\{Nil(C,A)\}$ or $\{Neu(C,E), Nil(C,A)\}$.
Two theories are disjoint if they share no symbol.

Let us give schemes for construction functions for theories of type 2. A clause is generated only if the conditions `Neu(C,E)`, `Inv(C,I,E)`, etc. are satisfied. These conditions are not part of the generated code.

```
let f_C = function
| E, x when Neu(C,E) -> x
| x, E when Neu(C,E) -> x
| C(x,y), z when Assoc(C) -> f_C(x,f_C(y,z))
| x, y when Inv(C,I,E) -> insert_inv_C (f_I x) y
| x, y -> insert_C x y

and f_I = function
| E -> E
| I(x) -> x
| C(x,y) -> f_C(f_I y, f_I x)
| x -> I x

and insert_inv_C x y =
  try delete_C x y
  with Not_found -> insert_C (f_I x) y

and delete_C x = function
| Plus(y,_) when x < y -> raise Not_found
| Plus(y,t) when x = y -> t
| Plus(y,t) -> C(y, delete_C x t)
| y when y = x -> E
| _ -> raise Not_found
```

```
and insert_C x = function
| C(y,_) as u when x = y & idem -> u
| C(y,t) when x = y & nil -> f_C(A,t)
| C(y,_) as u when x <= y & com -> C(x,u)
| C(y,t) when Com(C) -> C(y, insert_C x t)
| u when x > u & Com(C) -> C(u,x)
| u when x = u & Idem(C) -> u
| u when x = u & Nil(C,A) -> A
| u -> C(x,u)
```

Theorem 19 Let $\mathcal{E}$ be the union of pairwise disjoint theories of type 1 or 2. Assume that, for all constructor C which theory is of type k, f_C is defined as in Definition 16 if $k = 1$, and as above if $k = 2$. Then, $(f_C)_{C \in \mathcal{C}}$ is valid wrt $\mathcal{E}$.

Proof. Assume that $\mathcal{E} = \bigcup_{i=1}^{n} \mathcal{E}_i$ where $\mathcal{E}_1, \ldots, \mathcal{E}_n$ are pairwise disjoint theories of type 1 or 2. Whatever the type of $\mathcal{E}_i$ is, we saw that $\mathcal{E}_i$ has a complete presentation $\mathcal{R}_i$. Therefore, since $\mathcal{E}_1, \ldots, \mathcal{E}_n$ share no symbol, by definition of completion, the AC-completion of $\mathcal{E}$ successfully terminates with $\mathcal{R} = \bigcup_{i=1}^{n} \mathcal{R}_i$. Thus, $\rightarrow_{\mathcal{R},AC}$ is terminating and AC-confluent. Since $\mathcal{F} = (f_C)_{C \in \mathcal{C}}$ computes $\mathcal{R}, AC$-normal forms in AC-normal forms, by Theorem 14, $\mathcal{F}$ is valid. ■

The construction functions of type 2 can be easily extended to deal with ring or lattice structures (distributivity and absorbance equations).

More general results can be expected by using or extending results on the modularity of completeness for the combination of rewrite systems. The completeness of hierarchical combinations of non-AC-rewrite systems is studied in [19]. Note however that the modularity of confluence for AC-rewrite systems has been formally established only recently in [14].

Note that the construction function definitions of type 1 or 2 provide the same results with call-by-value, call-by-name or lazy evaluation strategy.

The detailed study of the complexity of theses definitions (compared to AC-rewriting) is left for future work.

6 The Moca system

We now describe the Moca prototype, a program generator that implements an extension of OCaml with RDTs. Moca parses a special ".mlm" file containing the RDT definition and produces a regular OCaml module (interface and implementation) which provides the construction functions for the RDT. Moca provides a set of keywords for specifying the equations described in Figure 1.

For instance, the RDT `exp` can be defined in Moca as follows:

```
type exp = private Zero | One | Opp of exp | Plus of exp * exp
  begin associative commutative neutral(Zero) opposite(Opp) end
```

Moca also features user's arbitrary rules with the construction: `rule` *pattern* `->` *pattern*. These rules add extra clauses in the definitions of construction functions generated by Moca: the LHS *pattern* is copied verbatim as the pattern of a clause which returns the RHS *pattern* considered as an expression where constructors are replaced by calls to the corresponding construction functions. Of course, in the presence of such arbitrary rules, we cannot guarantee the termination or completeness of the generated code. This construction is thus provided for expert users that can prove termination and completeness of the corresponding set of rules. That way, the programmer can describe complex RDTs, even those which cannot be described with the set of predefined equational invariants.

Moca also accepts polymorphic RDTs and RDTs mutually defined with record types (but equations between record fields are not yet available).

The equations of Figure 1 also support n-ary constructor, implemented as unary constructors of type `t list -> t`. In this case, `Plus` gets a single argument of type `exp list`. Normal forms are modified accordingly and use lists instead of combs. For instance, associative normal forms get flat lists of arguments: in a `Plus`(l) expression, no element of l is a `Plus`(l') expression. The corresponding data structure is widely used in rewriting.

Finally, Moca offers an important additional feature: it can generate construction functions that provide maximally shared representatives. To fire maximal sharing, just add the `-sharing` option when compiling the ".mlm" file. In this case, the generated type is slightly modified, since every functional constructor gets an extra argument to keep the hash code of the term. Maximally shared representatives have a lot of good properties: not only data size is minimal and user's memoized functions can be light speed, but comparison between representatives is turned from a complex recursive term comparison to a pointer comparison – a single machine instruction. Moca heavily uses this property for the generation of construction functions: when dealing with non-linear equations, the maximal sharing property allows Moca to replace term equality by pointer equality.

7 Future work

We plan to integrate Moca to the development environment Focal [20]. Focal units contain declarations and definitions of functions, statements and proofs as first-class citizens. Their compilation produces both a file checkable by the theorem prover Coq [7] and a OCaml source code. Proofs are done either within Coq or via the automatic theorem prover Zenon [9], which issues a Coq file when it successes. Every Focal unit has a special field, giving the type of the data manipulated in this unit. Thus, it would be very interesting to do a full integration of private/relational data types in Focal, the proof of correctness of construction functions being done with Zenon or Coq and then recorded as a theorem to be used for further proofs. This should be completed by the integration of a tool on rewriting and equational theories able to complete equational presentations, to generate and prove the corresponding lemmas and to show some termination properties. Some experiments already done within Focal on coupling CiME [6] and Zenon give a serious hope of success.

Acknowledgments. The authors thank Claude Kirchner for his comments on a previous version of the paper.

References

1. D. Benanav, D. Kapur, and P. Narendran. Complexity of matching problems. *J. of Symbolic Computation*, 3(1-2):203–216, 1987.
2. P. Bendix and D. Knuth. *Computational problems in abstract algebra*, chapter Simple word problems in universal algebra. Pergamon Press, 1970.
3. F. Burton and R. Cameron. Pattern matching with abstract data types. *J. of Functional Programming*, 3(2):171–190, 1993.
4. W. Burton, E. Meijer, P. Sansom, S. Thompson, and P. Wadler. Views: An extension to Haskell pattern matching. `http://www.haskell.org/extensions/views.html`, 1996.
5. P. Le Chenadec. *Canonical forms in finitely presented algebras.* Research notes in theoretical computer science. Pitman, 1986.
6. E. Contejean, C. Marché, B. Monate, and X. Urbain. *CiME version 2.02.* LRI, CNRS UMR 8623, Université Paris-Sud, France, 2004. `http://cime.lri.fr/`.
7. Coq Development Team. *The Coq Proof Assistant Reference Manual, Version 8.0.* INRIA, France, 2006. `http://coq.inria.fr/`.
8. N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 6. North Holland, 1990.
9. D. Doligez. Zenon, version 0.4.1. `http://focal.inria.fr/zenon/`, 2006.
10. D. Doligez, J. Garrigue, X. Leroy, D. Rémy, and J. Vouillon. *The Objective Caml system release 3.09, Documentation and user's manual.* INRIA, France, 2005. `http://caml.inria.fr/`.
11. S. P. Jones (editor). *Haskell 98 Language and Libraries, The revised report.* Cambridge University Press, 2003.
12. J.-M. Gaillourdet, T. Hillenbrand, B. Löchner, and H. Spies. The new Waldmeister loop at work. In *Proc. of CADE'03*, LNCS 2741. `http://www.waldmeister.org/`.
13. J.-M. Hullot. *Compilation de formes canoniques dans les théories équationnelles.* PhD thesis, Université Paris 11, France, 1980.
14. J.-P. Jouannaud. Modular church-rosser modulo. In *Proc. of RTA'06*, LNCS 4098.
15. P.-E. Moreau, E. Balland, P. Brauner, R. Kopetz, and A. Reilles. *Tom Manual version 2.3.* INRIA & LORIA, Nancy, France, 2006. `http://tom.loria.fr/`.
16. P.-E. Moreau, C. Ringeissen, and M. Vittek. A pattern matching compiler for multiple target languages. In *Proc. of CC'03*, LNCS 2622.
17. C. Okasaki. Views for standard ML. In *Proc. of ML'98.*
18. G. Peterson and M. Stickel. Complete sets of reductions for some equational theories. *J. of the ACM*, 28(2):233–264, 1981.
19. K. Rao. Completeness of hierarchical combinations of term rewriting systems. In *Proc. of FSTTCS'93*, LNCS 761.
20. R. Rioboo, D. Doligez, T. Hardin, and all. *FoCal Reference Manual, version 0.3.1.* Université Paris 6, CNAM & INRIA, 2005. `http://focal.inria.fr/`.
21. S. Thompson. Laws in Miranda. In *Proc. of LFP'86.*
22. S. Thompson. Lawful functions and program verification in Miranda. *Science of Computer Programming*, 13(2-3):181–218, 1990.
23. P. Wadler. Views: a way for pattern matching to cohabit with data abstraction. In *Proc. of POPL'87.*
24. P. Weis. Private constructors in OCaml. `http://alan.petitepomme.net/cwn/2003.07.01.html#5`, 2003.