Congruence Types*

Gilles Barthe[1,3] and Herman Geuvers[2,3]

[1] Centrum voor Wiskunde en Informatica (CWI),
Amsterdam, The Netherlands, gilles@cwi.nl

[2] Faculty of Mathematics and Informatics,
Technical University of Eindhoven, The Netherlands, herman@win.tue.nl

[3] Faculty of Mathematics and Informatics, University of Nijmegen, The Netherlands

Abstract. *We introduce a type-theoretical framework in which canonical term rewriting systems can be represented faithfully both from the logical and the computational points of view. The framework is based on congruence types, a new syntax which combines inductive, algebraic and quotient types. Congruence types improve on existing work to combine type theories with algebraic rewriting by making explicit the fact that the term-rewriting systems under consideration are initial models of an equational theory. As a result, the interaction gustavo:thesisween the type theory and the algebraic types (rewriting systems) is much more powerful than in previous work. Congruence types can be used (i) to introduce initial models of canonical term-rewriting systems (ii) to obtain a suitable computational behavior of a definable operation (iii) to provide an elegant solution to the problem of equational reasoning in type theory.*

1 Introduction

The combination of type systems with algebraic rewriting systems has given rise to algebraic-functional languages, a class of very powerful programming languages (see for example [4, 9, 12, 22]). Yet these frameworks only allow for a limited interaction between the algebraic rewriting systems and the type theory. For example, if $\mathbb{Z}$ is defined as an algebraic type, one cannot define the absolute value or prove that every integer is either positive or negative. This serious objection to algebraic-functional languages is in fact due to the absence of induction principles for algebraic types and so one might be tempted to formulate such principles. However, the task is not so easy if we want to have:

- *dependent elimination principles:* the naive approach which consists in adding the elimination principle directly to the algebraic type, as done in Clean ([27]), is limited to non-dependent elimination principles. For example, one could not prove from such an induction principle on $\mathbb{Z}$ that every integer is either positive or negative.

* This work was partially supported by the Esprit project 'Types: types for programs and proofs'.

- *confluence of the reduction relations on legal terms:* the computations attached to induction principles and those attached to algebraic types do not interact satisfactorily. What is usually required in programming languages is that the induction principle can only be applied to canonical values (i.e. closed algebraic terms in normal form). Without this restriction the reduction relation fails to be locally confluent.

To solve these problems, we opt for a two-level approach, in which every algebraic type is accompanied by the inductive type of its signature and related to it by suitable axioms for quotients[4]. For the case of $\mathbb{Z}$, this amounts to having an inductive type $\underline{Z}$ with constructors $\underline{0}, \underline{s}$ and $\underline{p}$ (the type of terms of the signature of $\mathbb{Z}$) and an algebraic type Z with constants $0 : Z$, $s : Z \to Z$ and $p : Z \to Z$ and rewrite rules $p(sx) \to x$ and $s(px) \to x$. The interaction between the types $\underline{Z}$ and Z is axiomatised by two maps: a 'class' map $[-] : \underline{Z} \to Z$ and a 'representant' map $\mathsf{rep} : Z \to \underline{Z}$, some reduction rules which specify the computational behavior of these maps (in particular, rep is forced to be the unique map which assigns to every 'class' a representant in normal form and [.] is forced to be the unique morphism of algebras from $\underline{Z}$ to Z) and a logical axiom (which states that there is no confusion, i.e. that the [.] map does identify exactly those terms which are provably equal for the theory of integers). In this way, one can transfer both the non-dependent and dependent induction principles (of $\underline{Z}$) to the algebraic type (Z) without affecting the confluence of the system. We claim that such a formalism, which we call *congruence types*, is suited for representing canonical term-rewriting systems in a faithful way (both from the logical and the computational points of view).

We see three important uses of congruence types.

- Represent initial models of term-rewriting systems, such as $\mathbb{Z}$. (They cannot be defined as inductive types, because they arise as a quotient of an inductive type). In this case we are mainly interested in the quotient type (Z) and we use the inductive type ($\underline{Z}$) to reason over the quotient type.
- Obtain a better computational behavior of a definable operation on an inductive type. This is achieved by defining an inductive type with 'extra' constructors and adding rewrite rules to specify the behavior of the extra constructor so that it represents the function we have in mind. How this works is best illustrated by an example. Consider the inductive type of natural numbers and the addition function $+$ on it. Then one has $+\ (sx)\ y \longrightarrow s(+\ x\ y)$ but (in general) not $+\ x\ (sy) \longrightarrow s(+\ x\ y)$. Hence $+$ has an unsatisfactory computational behavior. Now, consider the rewriting system (N, R), where N is the signature with constant 0, unary function s and binary symbol $+$ and the set of rewrite rules R consists of $+\ x\ 0 \to x$, $+\ 0\ x \to x$, $+\ (+\ x\ y)\ z \to +\ x\ (+\ y\ z)$, $+\ sx\ y \to s(+\ x\ y)$ and $+\ x\ sy \to s(+\ x\ y)$. The congruence type defined from this set of rewrite rules gives rise to an inductive type $\underline{N}$ with constructors $\underline{0}$, $\underline{s}$ and $\underline{+}$ and an algebraic type with the reduction rules R. In this framework, $+$ has a suitable computational

[4] The reader is refered to [6, 13, 20, 21] for a type-theoretic account of quotients.

behavior and N gives indeed a suitable representation of $\mathbb{N}$. Note that in this case we are again interested in the quotient type N.

- Use the quotient structure to prove properties of the algebra of terms (the inductive type). In this case the quotient structure acts as an *oracle* to derive a statement about the algebra of terms. Consider the congruence type associated to the theory of groups: the inductive type corresponds to the set of terms of the theory of groups and the quotient type corresponds to the free group over infinitely many elements. To know whether an equation (s, t) is a theorem of the theory of groups, it is enough to know whether $[s] = [t]$. The gain here is that if $[s]$ and $[t]$ have a common reduct, then the conclusion is immediate. This use of congruence types is very important in proof-checking and is the basis of lean proof-checking, a two-level approach to formal mathematics for efficient equational reasoning introduced in [7] and further developed in [10].

In this paper we want to emphasize especially the usefulness of congruence types and therefore we discuss three examples in quite some detail. Furthermore, we give a definition of the general syntax and an overview of the meta-theory of the system. The paper is organised as follows. In section 2, we discuss related work. In section 3, the more technical motivations of congruence types are discussed and we treat the integers as a motivating example of the syntax. In section 4, the syntax is given in detail (for the calculus of constructions) and we give some of the meta-theory (without proof). In section 5 we give two further examples of congruence types and their possible applications to programming and proof-checking. In the final section we suggest some extensions of the framework.

Related work

Congruence types are at the junction of several fundamental concepts and programming paradigms. They combine features of inductive ([25, 26, 24]), algebraic ([12, 4, 22]) and quotient types ([6, 13, 21]). Congruence types arise as a special form of quotient type where the underlying type is inductively defined and where the equivalence relation is given by a canonical term-rewriting system.

Congruence types and inductive types Congruence types are more expressive than inductive types because they allow to introduce initial models of canonical term-rewriting systems instead of initial models of signatures. They can be seen as a variant of the congruence types of Backhouse *et al.* which allow the introduction of initial models of arbitrary specifications ([2, 3]). Their work differs from ours in two respects; first, they focus on specifications and not on canonical term-rewriting systems, so there is no question of giving a computationally faithful representation of the rewrite rules. Second, their formalism requires a very strong form of equality as it is present for example in ITT.

Congruence types and pattern-matching It is possible to use congruence types to give a computationally faithful representation of definable operators on inductive

types. In effect, congruence types share some of the power of pattern-matching as introduced by Coquand in [14]. See section 4.1.

Congruence types and algebraic rewriting Congruence types are also more expressive than algebraic rewriting because of the presence of elimination principles. They are closely related to Jouannaud and Okada's algebraic functional paradigm ([4, 22]). In algebraic functional languages, (higher-order) constants are defined by rewrite rules, whereas they are defined inductively in the framework of congruence types. An advantage of congruence types is that the elimination principles can be used to reason over the data structures, a possibility which is ruled out in algebraic-functional languages. See section 2.

Applications of congruence types to proof-checking Congruence types provide a suitable framework to ease the problem of equational reasoning in proof-checking. As argued in [10], they also lay the foundations for a theoretical study of the interaction between computer algebra systems and proof-checkers. See section 4.2.

Prerequisites and terminology

The paper assumes some familiarity with pure type systems ([5, 17]), inductive types (see for example [26]) and first-order term-rewriting ([15, 23]). A signature is a pair $\Sigma = (F_\Sigma, \mathsf{Ar})$ where F_Σ is a set (the set of function symbols) and $\mathsf{Ar} : F_\Sigma \to \mathbb{N}$ is the arity map. Term-rewriting systems are defined as usual. By canonical term-rewriting system, we mean confluent and terminating term-rewriting system. An algebraic type is a type corresponding to a term-rewriting system.

2 Motivation

For every term-rewriting system $\mathcal{S} = (\Sigma, \mathcal{R})$, one can reason on the initial model $T_\mathcal{S}$ of $\mathcal{S}$ by induction on the structure of the terms. This form of reasoning implicitly uses the universality of $T_\mathcal{S}$ as a quotient of T_Σ and the initiality of T_Σ. In type theory (or any formal system), such a reasoning is only possible if the relationship between T_Σ and $T_\mathcal{S}$ is made explicit. Congruence types provide an axiomatic framework in which the relationship between the initial Σ-algebra and the initial $\mathcal{S}$-model is described axiomatically. The idea is to introduce two types Σ and $\mathcal{S}$ simultaneously; these types should respectively correspond to T_Σ and $T_\mathcal{S}$ (so we will confuse Σ with T_Σ and $\mathcal{S}$ with $T_\mathcal{S}$). Every function symbol f of arity n induces two maps, $\underline{f}$ and f such that:

- if $q_1{:}\Sigma, \ldots, q_n{:}\Sigma$, then $\underline{f} q_1 \cdots q_n : \Sigma$,
- if $a_1{:}\mathcal{S}, \ldots, a_n{:}\mathcal{S}$, then $f a_1 \cdots a_n : \mathcal{S}$.

Hence every Σ-term t induces two terms $\underline{t}$ and t of respective type Σ and $\mathcal{S}$. Equality in $\mathcal{S}$ is forced by the rewriting rules of $\mathcal{R}$. Now the crucial step is to

relate $\mathcal{S}$ and Σ by suitable axioms. As $T_{\mathcal{S}}$ is a quotient of T_{Σ}, we can inspire ourselves from the standard rules for quotients ([6]). First, there must be a canonical 'class' map $[-]$ from Σ to $\mathcal{S}$; it is the unique morphism of Σ-algebras and satisfies for every function symbol f of arity n and $t_1, \ldots, t_n$ elements of Σ,

$$[\underline{f}(t_1, \ldots, t_n)] = f([t_1], \ldots, [t_n])$$

Type theory is a computational framework, so it is natural to see this equality as a computation rule (from the left to the right). In a second instance, we must ensure that the two standard criterions for quotients hold:

- *no junk:* the map $[-]$ from Σ to $\mathcal{S}$ is surjective;
- *no confusion:* for every two terms s and t, $\mathcal{S} \vdash s \doteq t \quad \Leftrightarrow \quad [s] = [t]$, where the first equality $\mathcal{S} \vdash - \doteq -$ is the deductive closure of the rewrite rules.

In the syntax for quotient types, there are two alternatives to ensure the no junk condition: by the introduction of a map rep from $\mathcal{S}$ to Σ which picks a representative for each equivalence class or by adding a logical axiom that enforces the surjectivity of $[-]$. We prefer the first alternative over the more traditional second approach, because it can be given a computational meaning; the idea is that rep should assign to every equivalence class c the unique term t in 'normal form'[5] such that $[t] = c$. Note that the behavior of rep is completely specified on closed terms by the above requirement, hence rep is *not* a choice operator and does *not* alter the constructive character of type theory. The behavior of rep is forced by several rewrite rules. First, one must have the computation rule $[\mathsf{rep}\ x] = x$ for every x in $\mathcal{S}$. Second, we must impose the further computation rule

$$\mathsf{rep}(f(t_1, \ldots, t_n)) = \underline{f}(\mathsf{rep}\ t_1, \ldots, \mathsf{rep}\ t_n)$$

provided $f(t_1, \ldots, t_n)$ is a closed term in normal form (this corresponds exactly to our intuition of rep). The restriction to closed terms is necessary to preserve confluence.

As for the no confusion rule, it is ensured axiomatically. The rule expresses the fact that, if two elements of Σ are in the same class, then they are in the least equivalence relation that contains the rewrite relation (seen as a relation on Σ). This is achieved by adding a constant noconf that takes a proof p of $[a] =_{\mathcal{S}} [b]$ and returns a proof noconf p of $R_{\mathcal{S}}\ a\ b$, where $R_{\mathcal{S}}$ is the (impredicatively defined) least equivalence relation containing the rewrite relation.

A worked out example: the integers One of the starting points of our investigation was the representation of the set $\mathbb{Z}$ of integers in type theory. Despite being a fairly simple data type, it has no direct representation in type theory; it can either be defined as a "quotient" of $\mathbb{N} \times \mathbb{N}$, where $\mathbb{N}$ is the inductively defined type of natural numbers, or as an inductive type using some encoding ([11]), or as an algebraic type, i.e. a term-rewriting system (without induction principle).

[5] The rewriting relation is defined on $\mathcal{S}$ so the notion of a term (in Σ) in 'normal form' is an informal one.

However, none of these solutions captures adequately the structure of $\mathbb{Z}$. If we see $\mathbb{Z}$ as a canonical term-rewriting system, then the first two definitions are not computationally faithful. On the other hand, if $\mathbb{Z}$ is represented as an algebraic type, the representation of $\mathbb{Z}$ is unsatisfactory from a logical point of view; for example, one cannot prove that every integer is either positive or negative nor define the absolute value of an integer.

On the other hand, congruence types provide a suitable representation of $\mathbb{Z}$. $\mathbb{Z}$ can be defined with congruence types by introducing simultaneously an algebraic type Z corresponding to $\mathbb{Z}$ and an inductive type $\underline{Z}$ corresponding to the signature of $\mathbb{Z}$ and by relating them by suitable rules for quotient types. In this formalism, the representation of $\mathbb{Z}$ is computationally faithful and it is possible to derive from the induction principle on $\underline{Z}$ several standard induction principles on Z. The rules are as follows.

The inductive type $\underline{Z}$ of ground terms of the theory of integers with constructors $\underline{0}$, $\underline{s}$ and $\underline{p}$. $\underline{Z}$ is given by the standard rules for an inductive type

$$\frac{}{\vdash \underline{Z} : \Box} \qquad \frac{}{\vdash \underline{0} : \underline{Z}} \qquad \frac{\Gamma \vdash t : \underline{Z}}{\Gamma \vdash \underline{s}t : \underline{Z}} \quad \frac{\Gamma \vdash t : \underline{Z}}{\Gamma \vdash \underline{p}t : \underline{Z}}$$

with the elimination rules

$$\frac{\Gamma \vdash C : \underline{Z} \to * \qquad \Gamma \vdash a : \underline{Z} \qquad \Gamma \vdash f_0 : C\underline{0} \qquad \Gamma \vdash f_s : \Pi x : \underline{Z}.Cx \to C(\underline{s}x) \qquad \Gamma \vdash f_p : \Pi x : \underline{Z}.Cx \to C(\underline{p}x)}{\Gamma \vdash \epsilon\,[f_0, f_s, f_p]\; a : C\; a}$$

$$\frac{\Gamma \vdash C : \Box \qquad \Gamma \vdash a : \underline{Z} \qquad \Gamma \vdash f_0 : C \qquad \Gamma \vdash f_s : \underline{Z} \to C \to C \qquad \Gamma \vdash f_p : \underline{Z} \to C \to C}{\Gamma \vdash \epsilon\,[f_0, f_s, f_p]\; a : C}$$

The term-rewriting system Z is introduced via the rules

$$\frac{}{\vdash Z : \Box} \qquad \frac{}{\vdash 0 : Z} \qquad \frac{\Gamma \vdash t : Z}{\Gamma \vdash st : Z} \quad \frac{\Gamma \vdash t : Z}{\Gamma \vdash pt : Z}$$

The axioms for quotients, that relate Z and $\underline{Z}$, are represented by the rules

$$\frac{\Gamma \vdash a : \underline{Z}}{\Gamma \vdash [a] : Z} \quad \frac{\Gamma \vdash a : Z}{\Gamma \vdash \mathsf{rep}\; a : \underline{Z}} \quad \frac{\Gamma \vdash p : [a] =_Z [b]}{\Gamma \vdash \mathsf{noconf}\, p : R_Z\; a\; b.}$$

Here, R_Z is the least equivalence relation on $\underline{Z}$ that is closed under the rewrite rules. More precisely: for $a, b{:}\underline{Z}$,

$$R_Z a\; b := \Pi S{:}\underline{Z}{\to}\underline{Z}{\to}{*}.\mathrm{eqrel}\,(S){\to}(\Pi x{:}\underline{Z}.S\;(\underline{p}(\underline{s}x))\;x){\to}(\Pi x{:}\underline{Z}.S\;(\underline{s}(\underline{p}x))\;x){\to}S\;a\;b$$

where $\mathrm{eqrel}\,(S)$ denotes that S is an equivalence relation. There is a new conversion rule, which extends the reduction-expansion rule to take into account the new reduction relations[6].

[6] Note that in pure type systems, this rule is equivalent to the standard conversion rule; the equivalence follows from the subject reduction lemma and the Church-Rosser property of β-reduction on pseudo-terms ([5]). One consequence of the equivalence

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash A' : */\Box \quad A \twoheadrightarrow_{\beta\chi\iota\rho} A' \text{ or } A' \twoheadrightarrow_{\beta\chi\iota\rho} A}{\Gamma \vdash a : A'}$$

The computational behavior of the system is specified by β-reduction and three other reduction relations:

- ι-reduction (The computational meaning of the elimination principles over the inductive type $\underline{Z}$.)

$$\begin{aligned} \epsilon\ [f_0, f_s, f_p]\ \underline{0} &\to_\iota f_0 \\ \epsilon\ [f_0, f_s, f_p]\ (\underline{s}x) &\to_\iota f_s\ x\ (\epsilon\ [f_0, f_s, f_p]\ x) \\ \epsilon\ [f_0, f_s, f_p]\ (\underline{p}x) &\to_\iota f_p\ x\ (\epsilon\ [f_0, f_s, f_p]\ x) \end{aligned}$$

  These reduction rules are the standard ones for inductive types.
- ρ-reduction (Given by the term-rewriting system defining $\mathbb{Z}$.)

$$s(px) \to_\rho x \quad p(sx) \to_\rho x$$

- χ-reduction (The computational meaning of quotients.)

$$\begin{array}{rrr} [\mathsf{rep}\ x] \to_\chi x & & \\ [\underline{0}] \to_\chi 0 & [\underline{s}x] \to_\chi s[x] & [\underline{p}x] \to_\chi p[x] \\ \mathsf{rep}\ 0 \to_\chi \underline{0} & \mathsf{rep}\ (st) \to_\chi \underline{s}\ (\mathsf{rep}\ t) & \mathsf{rep}\ (pt) \to_\chi \underline{p}\ (\mathsf{rep}\ t) \end{array}$$

  where in the last two rules it is respectively assumed that st and pt are closed algebraic terms (i.e built from 0, s and p) in normal form.

One of the main advantages of our definition is that it suppresses the burden of providing equality proofs when reasoning about integers. Indeed, the equality between integers is computational and handled by the reduction relations. It makes them very attractive to use in proof-checking. Furthermore, our definition also captures the logical content of $\mathbb{Z}$ as one can prove that all the standard induction principles for $\mathbb{Z}$ hold for Z. The first induction principle is proof by induction, which stipulates that for every predicate P on Z,

if $P0$ and $\forall x \in Z.\mathsf{pos}\ x, Px \to P(sx)$ and $\forall x \in Z.\mathsf{neg}\ x, Px \to P(px)$ then $\forall x \in Z.Px$

where being positive (pos) and being negative (neg) are suitably defined predicates. A similar non-dependent elimination principle over $\Box$ can be defined. For $P : \Box$, one can build from

$$f_0 : P,\ f_s : \Pi x : Z.(\mathsf{pos}\ x) \to P \to P \text{ and } f_p : \Pi x : Z.(\mathsf{neg}\ x) \to P \to P \text{ a term } F(f_0, f_s, f_p) \text{ of type } Z \to P.$$

---

is that for every two convertible legal types A and B, there exists a conversion path through legal types; this property is called soundness in [19]. Soundness is a very desirable property of the system because it ensures that non-typable terms do not play any role in derivations.

In presence of ρ-reduction, one cannot rely on subject reduction or confluence of the combined reduction relation on the set of pseudo-terms to prove soundness. The solution is to replace the conversion rule by the reduction-expansion rule (see [4]).

The construction of these terms is rather intricate and involves the definition of a normal form map $\mathsf{nf} : \underline{Z} \to \underline{Z}$ with suitable properties. The construction will be reported elsewhere.

The term F behaves as a kind of 'primitive recursor for the integers'. Indeed, one can check that the following equalities hold:

$$\begin{aligned} F\ f_0\ f_s\ f_p\ 0 &=_{\beta\iota\chi\rho} f_0 \\ F\ f_0\ f_s\ f_p\ (s\ t) &=_{\beta\iota\chi\rho} f_s\ t\ q\ (F\ f_0\ f_s\ f_p\ t) \\ F\ f_0\ f_s\ f_p\ (p\ t) &=_{\beta\iota\chi\rho} f_p\ t\ q\ (F\ f_0\ f_s\ f_p\ t) \end{aligned}$$

where in the second rule, st is a closed term in normal form and q a proof of $\mathsf{pos}\ t$ and in the last rule, pt is a closed term in normal form and q a proof that $\mathsf{neg}\ t$. In contrast, the dependent elimination principle over $*$ does not have such a clear computational meaning. It seems to emphasize the necessity to separate between propositions and objects, as it is done in the present system by putting the sets on the *kind*-level. Our view is that only inhabitation is central to propositions, so that the computational meaning of the elimination principle over propositions is not crucial. On the contrary, both inhabitation and computational behavior of the inhabitants are important in the case of objects, so the computational meaning of the elimination principle over objects must be clear. Still, one can get an elimination principle for $*$ which is computationally meaningful by strengthening mildly the induction hypotheses. (So, this elimination principle is logically weaker). Indeed, one can easily construct a term G of type

$$\forall P : Z \to *.P0 \to (\forall x : Z.Px \to P(sx)) \to (\forall x : Z.Px \to P(px)) \to \forall x : Z.Px$$

that satisfies reductions that are similar to the ones for F above.

3 The calculus of constructions with congruence types

3.1 Syntax

We start from a (finite) collection $\mathcal{S}_1 = (\Sigma_1, \mathcal{R}_1), \ldots, \mathcal{S}_1 = (\Sigma_n, \mathcal{R}_n)$ of canonical term-rewriting systems. We let $F = \bigcup_{i=1,\ldots,n} F_{\Sigma_i}$ and $\underline{F} = \{\underline{f} | f \in F\}$. The set of pseudo-terms is defined by the abstract syntax:

$$T = V \mid * \mid \Box \mid TT \mid \Pi V : T.T \mid \lambda V : T.T \mid \mathcal{S}_i \mid \Sigma_i \mid F\mathbf{T} \mid \underline{F}\mathbf{T} \mid [T] \mid \mathsf{rep}\ T \mid \mathsf{noconf}\ T \mid \epsilon_i[\mathbf{T}]\ T$$

The rules for derivation are those of the Calculus of Constructions (see Appendix) extended by the rules for congruence types. The rules are divided in four categories.

- *formation and introduction rules:* these rules introduce the congruence types and all the constructors. As motivated earlier, congruence types are introduced as kinds.

$$\frac{}{\vdash \mathcal{S}_i : \Box} \qquad \frac{}{\vdash \Sigma_i : \Box} \qquad \frac{\Gamma \vdash a : \Sigma_i}{\Gamma \vdash [a] : \mathcal{S}_i} \qquad \frac{\Gamma \vdash a : \Sigma_i}{\Gamma \vdash \mathsf{rep}\ a : \mathcal{S}_i}$$

$$\frac{\Gamma \vdash a_1 : \mathcal{S}_i \quad \ldots \quad \Gamma \vdash a_m : \mathcal{S}_i}{\Gamma \vdash f\ a_1\ \ldots\ a_m : \mathcal{S}_i} \qquad \frac{\Gamma \vdash a_1 : \Sigma_i \quad \ldots \quad \Gamma \vdash a_m : \Sigma_i}{\Gamma \vdash \underline{f}\ a_1\ \ldots\ a_m : \Sigma_i}$$

where it is assumed that $f \in \mathcal{F}_i$ has arity m;

- *elimination rules:* these are the standard elimination rules for inductive types; let Σ_i have constructors $f_1, \ldots, f_{n_i}$ of respective arity $m_1, \ldots, m_{n_i}$.

$$\frac{\Gamma \vdash a : \Sigma_i \qquad \Gamma \vdash P : \Sigma_i \to *}{\Gamma \vdash E_j : \Pi x_1 \ldots x_{m_j} : \Sigma_i . P x_1 \to \cdots \to P x_{m_j} \to P(\underline{f_j}\ x_1 \cdots x_{m_j})\ [1 \le j \le n_i]}{\epsilon_i[E_1, \ldots, E_{n_i}]a : Pa}$$

$$\frac{\Gamma \vdash a : \Sigma_i \qquad \Gamma \vdash P : \Box}{\Gamma \vdash E_j : \Pi x_1 \ldots x_{m_j} : \Sigma_i . P \to \cdots \to P \to P\ [1 \le j \le n_i]}{\epsilon_i[E_1, \ldots, E_{n_i}]\ a : P}$$

- *logical rule:* the no confusion rule is formalised by defining the closure of $\mathcal{R}_i$ as a relation on Σ_i. The relation is defined impredicatively and denoted by *abus de language* by $\mathcal{R}_i$.

$$\frac{\Gamma \vdash p : [a] =_{\mathcal{S}_i} [b]}{\Gamma \vdash \mathsf{noconf}\ p : \mathcal{R}_i\ a\ b}$$

- *reduction rule:* the reduction rule has to be extended so as to take into account the new reduction relations associated to congruence types.

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash A' : */\Box \qquad A \twoheadrightarrow_{\beta\chi\iota\rho} A' \text{ or } A' \twoheadrightarrow_{\beta\chi\iota\rho} A}{\Gamma \vdash a : A'}$$

The new reduction relations are ι-reduction (which specifies the computational behavior of the elimination principles), χ-reduction (which specifies the computational behavior of quotient types) and ρ-reduction (which embeds the reduction relation of the term-rewriting systems into the type theory). The rules are:

- *ι-reduction:* if $f_j \in \mathcal{F}_i$ is of arity m_j, $\epsilon_i[\mathbf{E}](\underline{f_j} a_1 \cdots a_{m_j}) \to_\iota E_j a_1 \cdots a_{m_j} (\epsilon_i[\mathbf{E}]a_1) \cdots (\epsilon_i[\mathbf{E}]a_{m_j})$,
- *ρ-reduction:* for every rewriting rule $l \to r$, there is a rule $l \to_\rho r$,
- *χ-reduction:* the rules are

$$[\mathsf{rep}\ x] \to_\chi x$$
$$[\underline{f}\ t_1\ \ldots\ t_m] \to_\chi f\ [t_1]\ \ldots\ [t_m]$$
$$\mathsf{rep}\ (f\ t_1\ \ldots\ t_m) \to_\chi \underline{f}\ (\mathsf{rep}\ t_1)\ \ldots\ (\mathsf{rep}\ t_m)$$

In the last rule, it is assumed that $f\ t_1\ \ldots\ t_m$ is a closed algebraic term in (ρ-)normal form or that f is a *fundamental constructor*, i.e. for all Σ-terms $t_1, \ldots, t_m$, the normal form of $f(t_1, \ldots, t_m)$ is $f(t'_1, \ldots, t'_m)$ where the t'_i's are the normal forms of the t_i's. In section 4.1, we will justify this slight weakening of the proviso.

3.2 Meta-Theory

There are some important properties to be established before we can safely use the extension of CC with Congruence Types. These are the *Church-Rosser property* for the well-typed terms, *subject-reduction* (which ensures that reduction preserves typing), *consistency* (as a logical system, saying that not all types are inhabited by a closed term) and *decidability of typing* (it is decidable if in a given context Γ, a pseudo-term M has type A). These properties will of course depend on the specific algebraic rewrite rules that we have added, but remember that we only consider canonical (i.e. Church-Rosser and strongly normalizing) term-rewriting systems.

It turns out that all the standard results for the Calculus of Constructions hold for its extension with congruence types. Note however that proofs are complicated by the fact that $\beta\chi\iota\rho$-reduction is not confluent on pseudo-terms (see [12] for a counterexample). A relatively easy fact, but nevertheless a key observation is the following.

Lemma 1 *The $\beta\chi\iota\rho$-reduction is Weak Church-Rosser (WCR) on the set of pseudoterms. (That is, if $M \longrightarrow_{\beta\chi\iota\rho} M_1$ and $M \longrightarrow_{\beta\chi\iota\rho} M_2$, then there is a term Q such that $M_1 \twoheadrightarrow_{\beta\chi\iota\rho} Q$ $M_2 \twoheadrightarrow_{\beta\chi\iota\rho} Q$.)*

The subject reduction property (SR) can also be proved. Because $\beta\chi\iota\rho$-reduction is *not* Church-Rosser on the pseudo-terms, this involves some extra technicalities that were developped in [4] for the addition of algebraic rewriting to CC.

Proposition 2 (Subject Reduction) *If $\Gamma \vdash a : A$ and $a \rightarrow_{\beta\chi\iota\rho} a'$, then $\Gamma \vdash a' : A$.*

Termination is a modular property of CC with congruence types, under the mild restriction that the term-rewriting systems are non-duplicating[7]. We do not know whether strong normalisation pertains if the restriction to non-duplicating term-rewriting systems is dropped.

Theorem 3 (Strong Normalization) *Let $\mathcal{S}_1, \ldots, \mathcal{S}_n$ be canonical, non-duplicating term-rewriting systems. Then* CC *extended with the congruence types associated to $\mathcal{S}_1, \ldots, \mathcal{S}_n$ is strongly normalising.*

The proof is an adaptation of the semantical proof of strong normalisation for CC with (first-order) inductive types given in [18].

Corollary 4 *1. CC with Congruence Types satisfies the Church-Rosser property. (If M is well-typed and $M \twoheadrightarrow_{\beta\chi\iota\rho} M_1$ and $M \twoheadrightarrow_{\beta\chi\iota\rho} M_2$, then there is a term Q such that $M_1 \twoheadrightarrow_{\beta\chi\iota\rho} Q$ $M_2 \twoheadrightarrow_{\beta\chi\iota\rho} Q$.)*

[7] A term-rewriting system is non-duplicating if for every rule $l \rightarrow r$ and variable x, $\mathrm{occ}(x, l) \leq \mathrm{occ}(x, r)$ where for every term t, $\mathrm{occ}(x, t)$ denotes the number of occurences of x in t.

2. *CC with Congruence Types is consistent. (There is no closed term M with $\vdash M : \bot$.)*
3. *CC with Congruence Types has decidable typing.*

The first is due to Newman's Lemma (SN & WCR imply CR). The second follows by showing that a closed term of type $\bot$ ($:= \Pi\alpha{:}*.\alpha$) can never be in normal form in our system. This involves some more technical facts about the possible structure of inhabitants of types of a specific form. (Note that in presence of congruence types, this reasoning is slightly more complicated then for the Calculus of Constructions, because of the no confusion rules.) The consistency can also be proved in a more direct way by extending the proof-irrelevance model or the realisability model for CC to the case for congruence types. The third follows because for two well-typed terms it is decidable whether they have a common reduct.

4 Examples

4.1 The natural numbers with addition

Traditionally, the natural numbers are defined as an inductive type N with two constructors, zero and successor. Then addition, multiplication and other primitive recursive functions can all be defined inductively. One of the problems of this approach is that the computational behavior of these operations can be quite unsatisfactory. For example, if we define addition inductively on the first component, we have the reduction rule $sx + y \to s(x + y)$ but in general not $x + sy \to s(x + y)$ (if x and y are variables, then the reduction does not hold). Hence $+$ has not the expected computational behavior. This fact was already pointed out in [14] as a motivating example to introduce pattern-matching in type theory. Congruence types offer an alternative approach to define a type of natural numbers with well-behaved arithmetical operations. Consider the term-rewriting system $N = (\underline{N}, \mathcal{R}_N)$ where $\underline{N}$ is the signature consisting of one constant 0, one unary function symbol s and one binary function symbol $+$ and $\mathcal{R}_N$ is the term-rewriting system given by the reduction rules

$$\begin{aligned} +\ x\ 0 &\to x \\ +\ 0\ x &\to x \\ +\ (+\ x\ y)\ z &\to +\ x\ (+\ y\ z) \\ +\ sx\ y &\to s(+\ x\ y) \\ +\ x\ sy &\to s(+\ x\ y) \end{aligned}$$

We claim that N gives a suitable representation of $\mathbb{N}$. In particular, one can prove the standard induction principles for natural numbers. However, the weakening of the proviso in the rules for χ-reduction ($\mathsf{rep}\,(f\,\mathsf{t}) \longrightarrow \underline{f}(\mathsf{rep}\,\mathsf{t})$ if f is a fundamental constructor) is essential to derive the standard elimination principles for N. The key fact is that in the present example s and 0 are fundamental

constructors, hence $\mathsf{rep}\,(st) \longrightarrow \underline{s}(\mathsf{rep}\ t)$ for an arbitrary term t. Note that, as N is inductively definable, every closed algebraic term reduces to a *fundamental algebraic term*, i.e. one built from the fundamental constructors.

Congruence types and pattern-matching It is particularly interesting to compare our syntax with pattern-matching as introduced in [14]. Both offer a means to give a computationally adequate representation of definable operations on inductive types. Technically, this is achieved by different means. The most important differences between pattern-matching and congruence types are summarised below.

- Pattern-matching is schematic and can be used repeatedly to define new operators in the same way as the elimination principle. In contrast, congruence types are specific: they only provide a faithful representation of those operators introduced as constructors. For example, substraction will not have the expected computational behavior in the above definition of N. Moreover, pattern-matching can be used to define (for example) predicates, which is not possible with congruence types.
- The structure of rewrite rules allowed is more liberal in the syntax of congruence types than in the syntax of pattern-matching. For example, the rule $+\ (+\ x\ y)\ z \rightarrow +\ x\ (+\ y\ z)$ does not satisfy the criterion given in [14].

4.2 The free group over a set of atoms

Oracle types is another syntax for introducing term-rewriting systems in type theory, obtained from congruence types by forgetting the rep constructor and its associated reduction rules. In [10, 8], Barthe *et al.* have proposed oracle types as a theoretical framework to study the combination of proof-checkers and computer algebra systems. Indeed, oracle types can be viewed as an interface between a logical system (type theory with inductive types) and a calculational system (the computer algebra system, modelled by ρ-reduction). The two systems are correlated by the no confusion rule, which can be seen as some kind of soundness result. In this context, the no confusion rule can be read as follows.

> Let $(\Sigma, \mathcal{R})$ be a canonical term-rewriting system and let s, t be two Σ-terms. Every computation on $[s]$ and $[t]$ (the computer algebra representations of s and t) which yields a common reduct can be lifted to a proof that s and t are in the deductive closure of $\mathcal{R}$ (viewed as an equational theory).

In the remaining of this section, we illustrate how Barthe *et al.* have used congruence/oracle types to give a partial solution to the problem of equational reasoning in proof-checking.Consider the term-rewriting system $G = (\underline{G}, \mathcal{R}_G)$ where $\underline{G}$ is the signature of groups extended with infinitely many constants and $\mathcal{R}_G$ is the Knuth-Bendix completion of the axioms of the theory of groups. That is, $\mathcal{R}_G$ consists of the rules

$$
\begin{array}{rr}
o\ e\ x \to_\rho x & i\ e \to_\rho e \\
o\ x\ e \to_\rho x & o\ (o\ x\ (i\ z))\ z \to_\rho x \\
o\ x\ (o\ y\ z) \to_\rho o\ (o\ x\ y)\ z & o\ (o\ x\ z)\ (i\ z) \to_\rho x \\
o\ (i\ x)\ x \to_\rho e & i\ (i\ x) \to_\rho x \\
o\ x\ (i\ x) \to_\rho e & i\ (o\ x\ y) \to_\rho o\ (i\ y)\ (i\ x)
\end{array}
$$

The congruence type generated by G consists of two parts: the free group G over infinitely many elements and the inductive set of terms of the theory of groups (the infinite collection of constants serves as the set of variables). The interaction between the two types allows a simple solution to equational problems of the theory of groups. Assume we can derive

$$\Gamma \vdash H : \Box \quad \Gamma \vdash o_H : H \to H \to H \quad \Gamma \vdash e_H : H \quad \Gamma \vdash i_H : H \to H \to H$$

and we have a proof of the fact that (H, o_H, e_H, i_H) satisfies the axioms of groups (we work with Leibniz equality). Assume that we want to decide whether $a =_H b$. One possible way to solve the problem is to find two inhabitants s, t of $\underline{G}$ and an assignment[8] α such that $[\![s]\!]_\alpha \twoheadrightarrow a$ and $[\![t]\!]_\alpha \twoheadrightarrow b$ (in fact, there are optimal such s and t). By the conversion rule, the problem can be reduced to $[\![s]\!]_\alpha =_H [\![t]\!]_\alpha$. But, by definition of $\mathcal{R}_G$, this is an immediate consequence of $\mathcal{R}_G\ s\ t$. (Note that we are implicitely using the soundness theorem for equational logic, which is an easy consequence of the impredicative definition of $\mathcal{R}_G$.) Now congruence types offer a decision procedure for solving $\mathcal{R}_G\ s\ t$, simply by checking whether $[s] = [t]$ (because of the no confusion rule).

5 Final remarks

We have presented a new syntax of congruence types and shown how the syntax can be used to give a faithful representation of canonical term-rewriting systems in type theory. In this paper, we have restricted our attention to unsorted term-rewriting systems. In the future, it seems natural to extend the framework to cover other case of term-rewriting systems such as:

- *many-sorted term-rewriting systems:* the extension would allow to introduce strongly normalising type theories (with explicit substitutions) as congruence types;
- *higher-order term-rewriting systems:* the extension of our framework to higher-order specifications would allow to consider congruence types generated by first-order languages (quantification has to be introduced as a higher-order constructor).
- *non-standard term-rewriting systems:* many theories, such as commutative theories, fall out of the scope of this paper because they do not yield canonical

[8] Assignments and their extension to interpretations of terms are defined as usual.

term-rewriting systems. It would be interesting to investigate the theory of congruence types when the term-rewriting systems under consideration are conditional or priority rewriting systems or are defined modulo a set of equations.

Another important direction for research is the application of congruence and oracle types in proof-checking. Extending the framework of oracle types to cover many forms of rewriting would enable the two-level approach of [8, 10] to be extended to a significant class of problems, including for example a decision procedure to detect logical equivalence of formulae of propositional logic.

References

1. S. Abramsky, D. Gabbay, and T. Maibaum, editors. *Handbook of Logic in Computer Science*. Oxford Science Publications, 1992.
2. R. Backhouse, P. Chisholm, and G. Malcolm. Do-it-yourself type theory (part I). *BEATCS: Bulletin of the European Association for Theoretical Computer Science*, 34:68–110, 1988.
3. R. Backhouse, P. Chisholm, and G. Malcolm. Do-it-yourself type theory (part II). *BEATCS: Bulletin of the European Association for Theoretical Computer Science*, 35:205–244, 1988.
4. F. Barbanera, M. Fernandez, and H. Geuvers. Modularity of strong normalisation and confluence in the algebraic λ-cube. In *Proceedings of LICS'94*, pages 406–415. IEEE Computer Society Press, 1994.
5. H.P. Barendregt. Lambda calculi with types. In Abramsky et al. [1], pages 117–309. Volume 2.
6. G. Barthe. Extensions of pure type systems. In Dezani-Ciancaglini and Plotkin [16], pages 16–31.
7. G. Barthe. Formalising algebra in type theory: fundamentals and applications to group theory. Manuscript. An earlier version appeared as technical report CSI-R9508, University of Nijmegen, under the title 'Formalising mathematics in type theory: fundamentals and case studies', 1995.
8. G. Barthe and H. Elbers. Towards lean proof checking. Manuscript, 1996.
9. G. Barthe and H. Geuvers. Modular properties of algebraic pure type systems. In G. Dowek, J. Heering, K. Meinke, and B. Möller, editors, *Proceedings of HOA'95*, Lecture Notes in Computer Science. Springer-Verlag, 1996. To appear.
10. G. Barthe, M. Ruys, and H. Barendregt. A two-level approach towards lean proof-checking. In S. Berardi and M. Coppo, editors, *Proceedings of TYPES'95*, Lecture Notes in Computer Science. Springer-Verlag, 1996. To appear.
11. G. Betarte. A machine-assisted proof that the integers form an integral domain. Master's thesis, Department of Computer Science, Chalmers University, 1993.
12. V. Breazu-Tannen. Combining algebra and higher-order types. In *Proceedings of LICS'88*, pages 82–90. IEEE Computer Society Press, 1988.
13. R.L. Constable, S.F. Allen, H.M. Bromley, W.R. Cleaveland, J.F. Cremer, R.W. Harper, D.J. Howe, T.B. Knoblock, N.P. Mendler, P. Panangaden, J.T. Sasaki, and S.F. Smith. *Implementing Mathematics with the NuPrl Development System*. Prentice-Hall, inc., Englewood Cliffs, New Jersey, first edition, 1986.

14. T. Coquand. Pattern matching in type theory. In B. Nordström, editor, *Informal proceedings of LF'92*, pages 66–79, 1992. Available from http://www.dcs.ed.ac.uk/lfcsinfo/research/types-bra/proc/index.html.
15. N. Dershowitz and J-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Formal models and semantics. Handbook of Theoretical Computer Science*, volume B, pages 243–320. Elsevier, 1990.
16. M. Dezani-Ciancaglini and G. Plotkin, editors. *Proceedings of TLCA'95*, volume 902 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.
17. H. Geuvers. *Logics and type systems.* PhD thesis, University of Nijmegen, 1993.
18. H. Geuvers. A short and flexible proof of strong normalisation for the calculus of constructions. In P. Dybjer, B. Nordström, and J. Smith, editors, *Proceedings of TYPES'94*, volume 996 of *Lecture Notes in Computer Science*, pages 14–38. Springer-Verlag, 1995.
19. H. Geuvers and B. Werner. On the Church-Rosser property for expressive type systems and its consequence for their metatheoretic study. In *Proceedings of LICS'94*, pages 320–329. IEEE Computer Society Press, 1994.
20. M. Hofmann. A simple model for quotient types. In Dezani-Ciancaglini and Plotkin [16], pages 216–234.
21. B. Jacobs. *Categorical logic and type theory.* 199-. Book. In preparation.
22. J-P. Jouannaud and M. Okada. Executable higher-order algebraic specification languages. In *Proceedings of LICS'91*, pages 350–361. IEEE Computer Society Press, 1991.
23. J-W. Klop. Term-rewriting systems. In Abramsky et al. [1], pages 1–116. Volume 2.
24. Z. Luo. *Computation and Reasoning: A Type Theory for Computer Science.* Number 11 in International Series of Monographs on Computer Science. Oxford University Press, 1994.
25. B. Nordström, K. Petersson, and J. Smith. *Programming in Martin-Löf's Type Theory. An Introduction.* Number 7 in International Series of Monographs on Computer Science. Oxford University Press, 1990.
26. C. Paulin-Mohring. Inductive definitions in the system Coq. Rules and properties. In M. Bezem and J-F. Groote, editors, *Proceedings of TLCA'93*, volume 664 of *Lecture Notes in Computer Science*, pages 328–345. Springer-Verlag, 1993.
27. M.J. Plasmeijer and M.C.J.D. van Eekelen. Clean 1.0 reference manual. Technical report, Department of Computer Science, University of Nijmegen, 1996. In preparation.

The Calculus of Constructions

We now give a precise definition of the Calculus of Constructions and at the same time we fix some terminology. See for example [5, 17] for more information.

In CC there are two specific constants, $*$ and $\Box$. The first represents the universe of *types* (so we shall say that σ *is a type* if $\sigma : *$) and the second represents the universe of *kinds* (so we shall say that A *is a kind* if $A : \Box$). The universe $*$ is a specific example of a kind, so it will be the case that $* : \Box$. To present the derivation rules for CC we first fix the set of *pseudoterms* from which the derivation rules select the (typable) terms.

Definition 5 *The set T of pseudoterms is defined by the following abstract syntax*

$$T = V \mid * \mid \Box \mid TT \mid \Pi V : T.T \mid \lambda V : T.T$$

where V is a countable set of variables. Both Π and λ bind variables and we have the usual notions of free variable *and* bound variable. *The substitution of N for v in M is denoted by $M[N/v]$. On T we have the usual notion of β-reduction, denoted by $\longrightarrow_\beta$. We also adopt from the untyped λ calculus the conventions of denoting the transitive reflexive closure of $\longrightarrow_\beta$ by $\twoheadrightarrow_\beta$ and the transitive symmetric closure of $\twoheadrightarrow_\beta$ by $=_\beta$.*

The typing of terms is done under the assumption of specific types for the free variables that occur in the term. These are listed in a *context*, which is a sequence of declarations $v_1{:}T_1, \ldots, v_n{:}T_n$, where the v_i are distinct variables and the T_i are pseudoterms. Contexts are denoted by the symbol Γ. For Γ a context and v a variable, v is said to be *Γ-fresh* if it is not among the variables that are declared in Γ.

Definition 6 *The Calculus of Constructions (CC) is the typed λ-calculus with the following deduction rules.*

Axiom	$\vdash * : \Box$	
Start	$\dfrac{\Gamma \vdash A : */\Box}{\Gamma, x : A \vdash x : A}$	*if* $x \notin \Gamma$
Weakening	$\dfrac{\Gamma \vdash t : A \quad \Gamma \vdash B : */\Box}{\Gamma, x : B \vdash t : A}$	*if* $x \notin \Gamma$
Product	$\dfrac{\Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash \Pi x : A.B : s_2}$	$s_1, s_2 \in \{*, \Box\}$
Application	$\dfrac{\Gamma \vdash t : \Pi x : A.B \quad \Gamma \vdash u : A}{\Gamma \vdash tu : B[u/x]}$	
Abstraction	$\dfrac{\Gamma, x : A \vdash t : B \quad \Gamma \vdash \Pi x : A.B : */\Box}{\Gamma \vdash \lambda x : A.t : \Pi x : A.B}$	
Conversion	$\dfrac{\Gamma \vdash u : A \quad \Gamma \vdash B : */\Box}{\Gamma \vdash u : B}$	*if* $A \twoheadrightarrow_\beta B$ *or* $B \twoheadrightarrow_\beta A$

The set of terms of CC *is defined by* $\mathsf{Term} = \{A \mid \exists \Gamma, B[\Gamma \vdash A : B \vee \Gamma \vdash B : A]\}$.