A Lazy Approach to Partial Algebras *

M. Cerioli

DISI–Dipartimento di Informatica e Scienze dell'Informazione, Università di Genova
Viale Benedetto XV, 3 – 16132 Genova – Italy
e-mail: cerioli@disi.unige.it

Abstract. Starting from the analysis of which features are required by an algebraic formalism to describe at least the more common data types used in imperative and functional programming, a framework is proposed, collecting many techniques and ideas from the algebraic community, with the capability for an immediate representation of partiality and error-recovery. This formalism, of so called lazy algebras, inherits mainly from two parents: partial and label algebras; from the former especially on a technical side and from the later from a philosophical point of view. But, as all children, it has its own individuality and in particular an original mechanism to represent lazy evaluation in an algebraic framework has been introduced.

Introduction

This paper aims at the definition of an algebraic formalism providing tools for the immediate representation of the most common data types used in imperative and functional programming. In order to decide which features are needed by such an algebraic framework, the starting point is the analysis of the steps a user of a specification language goes through, toward the formal description of the required data type. First of all names for the types and operations are decided; moreover for every operation the number (and usually the type) of the arguments are fixed. Then among the strings on that alphabet a subset is chosen, of the "well-formed" terms, that are usually (a subset of) the many-sorted terms on those functions; this corresponds to the static check and should get rid of the real junk, i.e. of all those sequences of symbols that are meaningless and whose nonsense can be decided. Notice that if the data type under construction has only total functions, then static check suffices to eliminate all meaningless terms. Otherwise among the well-formed terms there are still strings that are not intended to represent correct values, but that could be needed as error messages or in order to apply some kind of error-recovery strategy. Thus the following design step is the division of well-formed terms into "ok" and "possibly errors" and then equivalences onto the two classes are defined to describe the semantics. Therefore a "user-friendly" formalism should provide tools to concisely describe

* This work has been partially supported by ESPRIT BRA WG n. 6112 COMPASS, HCM-Medicis and MURST 40% Modelli della computazione e dei linguaggi di programmazione

well-formed terms, i.e. some kind of signature, to partition terms into "ok" and errors, to give semantics to the "ok" ones and to deal with errors in a twofold way: on one side error terms should disappear, in the sense that they have no place in the carriers of the models, but on the other side they are needed for error-recovery.

Trying to collect bits of known formalisms to build a powerful framework, it is easy to find tools for the syntactical analysis; indeed it is immediate to check that standard many-sorted features are easy to use and allow to eliminate most of the junk. It is worth noting that pure order-sorted approaches are, from this point of view too restrictive, as the "errors" are (or should be in principle) avoided, narrowing the domain of "partial" functions, and hence errors are prevented at a syntactical level. On the other side using one-sorted formalisms is too weak, as even trivially uncorrect strings are considered correct terms and must be taken in account; for example search space for inference systems dramatically increases.

Using partiality it is also easy to manage error terms that do not appear in the carriers, but this seems contradictory w.r.t. error recovery; indeed if an error does not exist how can it be recovered? The innovative idea of the framework proposed here is exactly a way to conciliate the partiality of function interpretation in the models with an error-recovery purely syntactic, with the flavor of lazy valuation.

The main results of the paper, besides the introduction of the *lazy partial algebra* framework (Section 1), are the existence of initial and free objects (Section 2) and an alternative characterizations of such objects through a sound and complete inference system (Section 4). In Section 3 a few examples, showing the great usage facility of the proposed formalisms, are given. Proofs are omitted, but a large use of lemmas should suggest the structure of the more relevant theorems.

1 The Lazy framework

1.1 Motivations

Analyzing the most common data types present in computer science, (at least) three features appear to be mandatory for a specification language and hence for an algebraic formalism designed to give semantics to the language.

Typing. After a long time of many-sorted frameworks, recently homogeneous approaches have been presented (see e.g. [13, 11, 12]), where the freedom given by the lack of typing is used to enlarge the expressive power of the language. In most of these approaches sorts and elements float together in the carrier of any model, so that, for example, functions apply to sorts too and hence dependent types are available and operations between sorts, like intersection or sum, can be defined; moreover an operation on elements can result in a sort, or better a subsort, so that non-determinism and (if the sort is empty) non-termination are immediately at hand. But this power has a price: first of all static check, i.e. well formedness of terms, is limited to the number of arguments and this means a smaller help for the language users in "debugging" their specifications; moreover the carrier of any model has to be very large. This can be thought irrelevant, as most users could be interested only on the logical side, especially if a tool for prototyping would be available; but the proliferation of elements, all denoted by terms, makes the search space unmanageable and hence prevents a concrete use of such tools. Therefore we prefer to have a typing system, forfeiting the extra-power.

On the other side a too strict typing mechanism can be as dangerous as its absence. Indeed think for example of a pure order sorted approach to the specification of natural numbers with a predecessor operation; as the predecessor of 0 is undefined, a subsort of non-zero natural numbers is introduced, built by the successor applied to any natural argument. Then terms with two predecessors in a row are not well-formed, as the result of a predecessor is a (possibly zero) natural number and hence has not the correct type for another predecessor to be applied. Moreover if possibly incorrect terms are prevented, then error-recovery is impossible, as there is nothing to be recovered. Note that, using the retract mechanism (see [10]), an implicit and unavoidable error-recovery strategy is imposed and the only improvement w.r.t. standard many-sorted approaches is the uniformity of the error-detection, as a term represent an error iff it cannot be reduced to a form without retracts.

On the base of this analysis (for a longer and deeper discussion on the use of sorts to classify elements and restrain errors see [14]), in the sequel a standard many-sorted typing is adopted.

Partiality. As the data types used in programming languages are inherently partial, partiality must be taken in account. Note that the partiality introduced to allow a top-down development, where reasonably many details are not decided at the top level, but left free for the designer of lower modules to fix, is a feature of the language that does not need, in principle, to have a corresponding feature at the semantic level; indeed it is a tool for helping the design of the project, whose use disappears at the last stage, where the system is built bottom-up and every detail has been fixed. But the partiality due to the need for representing a semicomputable (and non computable) function persists to the last moment and cannot be eliminated, although in many cases (but not in all interesting ones) can be dealt with in an indirect way. As the theory of partial algebras is nowadays well established ([16, 7, 19]), using an indirect representation of partiality seems unreasonable.

Although, having partial functions, semidecidable boolean relations can be easily simulated (at least if their falsity is never tested), by representing every relation as a "boolean" function, with a special boolean sort, (see [8] and [9] for a similar approach in a total framework), predicates have been considered necessary, because of their recently increasing use in specifications, for example to represent transitions in concurrency, or the typing relation. In particular the use of predicates allows specifications of many simple data types, where boolean functions are the unique partial functions, whose truth can be decided, while the falsity cannot, to be described as total specification, avoiding the proliferation of errors.

Non strictness. Roughly speaking there are two kinds of non-strict functions widely used in programming languages: the "don't care", like the famous **if then else** or boolean **and**, **or** with lazy valuation, and the "error handling". "Don't care" is characterized by resulting in the same value whatever it is substituted for the missing value, while "error handling" corresponds to recover a value, after an erroneous computation, mainly by means of projections, as for example by instantiating the equations $x * 0 = 0$ or $pop(push(x, s)) = s$ for "erroneous" values of x.

Analyzing the examples of both kind of non-strictness it is easy to note that a non-strict function is needed only to represent some kind of *lazy valuation* (call by need, call by name ...) or simplification and hence its standard representation using totality plus a labeling of values to describe that some states (or computations or values) are erroneous is unsatisfactory, because it makes the models full of junk and does not meet the intuition. If just the "don't care" kind of non-strictness is needed, then there are approaches that avoid to introduce values to represent the missing arguments of the non-strict function (see e.g. [1]), although the implicit monotonicity condition raises some troubles. But if "error recovery" is needed, then apparently there is no way to avoid the "erroneous" values; indeed, for example, if positive integers are considered, then to have that the evaluation of $t = pred(zero) * zero$ is 0, the evaluation of $pred(zero)$ should be a value a such that the interpretation of the $*$ operation on a and 0 yields 0, as the interpretation in a model A of t is $pred^A(zero^A) *^A zero^A$, by compositionality.

But this approach does not meet the intuition that a simplification of t to $zero$, by the rule $x * zero = zero$, as been performed *on terms*, so that the evaluation of t reduces to the evaluation of $zero$. To stay close to this notion, algebras are endowed with a congruence on terms, representing the preprocessing (or compilation, or simplification) that transforms a term into a simpler one, so that the evaluation of a term becomes the easier evaluation of its simplification.

1.2 Lazy Signatures and Structures

Lazy signatures are usual many sorted signatures with predicates.

Definition 1. A *signature* Σ consists of a set S of *sorts*, an $S^* \times S$-indexed family F of *operation symbols* and an S^+-indexed family P of *predicate symbols*. If $f \in F_{(s_1 \ldots s_n, s)}$, then we write $f: s_1 \times \ldots \times s_n \rightarrow s$ and say that $s_1 \times \ldots \times s_n$ is the *arity* of f and s is the *type* of f; analogously if $p \in P_{s_1 \ldots s_n}$, then we write $p: s_1 \times \ldots \times s_n$ and say that $s_1 \times \ldots \times s_n$ is the *arity* of p.

Given a signature $\Sigma = (S, F, P)$, a family X of variables for Σ in an S-sorted family of disjoint sets X_s of new symbols, i.e. s.t. $X_s \cap (\cup_{w \in S^*, s' \in S} F_{w,s'} \cup \cup_{w \in S^*} P_w) = \emptyset$ for all $s \in S$. □

As in the sequel we will use the definition of congruence both in the total and in the partial frameworks, let us recall these notions in order to fix the notation.

Definition 2. Given a signature $\Sigma = (S, F, P)$ and a total algebra A over (S, F), a *partial congruence* $\equiv^A$ on A is an S-indexed family of symmetric and transitive relations $\{\equiv_s^A \subseteq s^A \times s^A\}_{s \in S}$ s.t. if $f \in F_{(s_1 \ldots s_n, s)}$ and $t_i \equiv_{s_i}^A u_i$ for $i = 1, \ldots, n$, then either $f(t_1, \ldots, t_n) \equiv_s^A f(u_1, \ldots, u_n)$ or both $f(t_1, \ldots, t_n)$ and $f(u_1, \ldots, u_n)$ do not belong to the *domain* of $\equiv^A$, where, given a partial congruence $\equiv^A$, its *domain*, denoted by $Dom(\equiv^A)$, consists of all elements τ s.t. $\tau \equiv^A \tau$.

Moreover a partial congruence $\equiv^A$ on A is called a *total congruence* iff its domain is A, i.e. if every $\equiv_s^A$ is reflexive. □

Let us introduce the model theoretic ingredients of the lazy framework.

Definition 3. Given a signature $\Sigma = (S, F, P)$, a *lazy algebra* $\mathcal{A}$ on Σ consists of:

- a partial algebra A on the signature Σ, i.e. $A = (\{s^A\}_{s \in S}, \{f^A\}_{f \in F})$, where s^A is a set for every $s \in S$, said the *carrier of sort* s in A and $f^A : {s_1}^A \times \ldots \times {s_n}^A \rightarrow s^A$ is a partial function for every $f \in F_{(s_1 \ldots s_n, s)}$; in particular if $n = 0$, i.e. if $f \in F_{(\Lambda, s)}$, then f^A is either undefined or an element of s^A.
- a total congruence $\equiv^A$ on $T_\Sigma(A)$, the algebra ofΠ terms built on the variable family $\{s^A\}_{s \in S}$, *compatible* with the interpretation of function symbols in A, i.e. s.t. if $t^A, u^A \in s^A$, then $t \equiv^A u$ iff $t^A = u^A$, where for a term $\tau \in T_\Sigma(A)$ we denote by τ^A the standard evaluation of the term τ in the partial algebra A w.r.t. the identity evaluation of variables in A.
- an interpretation $\mathcal{P}^A$, associating every predicate symbol $p \in P_{s_1 \ldots s_n}$ with its *truth-set* $\mathcal{P}^A(p) \subseteq {s_1}^{T_\Sigma(A)} \times \ldots \times {s_n}^{T_\Sigma(A)}$; moreover the interpretation $\mathcal{P}^A$ is required to be *sound* w.r.t. $\equiv^A$, i.e. if $t_i \equiv_{s_i}^A u_i$ for $i = 1 \ldots n$ then $(t_1, \ldots, t_n) \in \mathcal{P}^A(p)$ iff $(u_1, \ldots, u_n) \in \mathcal{P}^A(p)$, for every $p \in P_{s_1 \ldots s_n}$.

Given lazy algebras $\mathcal{A} = (A, \equiv^A, \mathcal{P}^A)$ and $\mathcal{B} = (B, \equiv^B, \mathcal{P}^B)$ on Σ, a *lazy homomorphism* $h: \mathcal{A} \rightarrow \mathcal{B}$ is a homomorphism $h: A \rightarrow B$ between partial algebras, i.e. an S-indexed family of total functions $h_s: s^A \rightarrow s^B$ s.t. if $f^A(a_1, \ldots, a_n) = a \in s^A$, then $f^B(h_{s_1}(a_1), \ldots, h_{s_n}(a_n)) = h_s(a)$, preserving simplification and predicates, i.e. s.t. $t \equiv^A u$ implies $h(t) \equiv^B h(u)$ and $(t_1, \ldots, t_n) \in \mathcal{P}^A(p)$ implies $(h_{s_1}(t_1), \ldots, h_{s_n}(t_n)) \in \mathcal{P}^B(p)$, where h is extended by freeness on terms in $T_\Sigma(A)$. □

Note that predicates are interpreted as their truth-set, but apply to *terms* instead of values; this is a slight generalization of the ideas behind label algebras (see e.g. [6]), where only unary predicates, called label indeed, where allowed. A relevant difference w.r.t. label algebra approach is that term evaluating to the same value are indistinguishable by predicates, while labeling disregards the equalities between terms. Thus predicates here are a bit more general than in standard approaches, because properties of undefined terms can be stated too, but less flexible than labeling; this restriction has been introduced in order to avoid (pathological) examples as the following.

Example 1. Let us consider the following label specification, with just one label b.

spec Sp =
sorts s, s'
opns $a\colon \to s$
$f\colon s \times s \times s \to s'$
axioms $x = a$
$f(x, x, y) : b$
$f(x, y, x) : b$
$f(y, x, x) : b$

Although labeling disregards equalities between values, and hence should in some sense be unaffected by the requirement $x = a$, in all models $f(x, y, z) : b$ holds for lack of values to instantiate x, y and z. Indeed, because of the first axiom, all models have a one-point carrier of sort s, so, recalling that evaluations are made on terms built from the elements of the algebra as variables, there are two elements to evaluate x, y and z on, that are the element of the carrier (seen as a variable) and the constant a; hence for every valuation for x, y and z (at least) one among the last three axioms applies so that $f(x, y, z) : b$ holds. But note that if another constant of sort s is introduced, whose interpretation must be the same value as that of a by the first axiom, then three terms of sort s exist and hence none of the axioms applies, so that $f(x, y, z) : b$ does not hold anymore, even if the models have the same carriers and function interpretation as before. In other words adding syntax, even if in every model such new terms reduce to values already denoted by old terms, can change the validity of labeling. □

Since the definition of lazy homomorphism is obviously satisfied by the identity and by the composition of lazy homomorphisms, we can define the category of lazy algebras and inherit the notion of initial (free) object in a class.

Definition 4. Given a signature Σ, the category $\mathbf{CAlg}(\Sigma)$ has lazy algebras as objects and lazy homomorphisms as arrows; identities are the families of identity maps $\{I_s \mid s \in S\}$ and composition is defined componentwise.

Let C be a subclass of $\mathbf{CAlg}(\Sigma)$ objects and X be an S-sorted family of variables; then a pair $(\mathcal{F}, E)$ is *free* for X in C iff $\mathcal{F} = (F, \equiv^F, \mathcal{P}^F) \in \mathsf{C}$, with $E\colon X \to F$, and for every $\mathcal{A} = (A, \equiv^A, \mathcal{P}^A)$ in C and every $V\colon X \to A$ there exists a unique homomorphism $h_V\colon \mathcal{F} \to \mathcal{A}$ s.t. $h_V \cdot E = V$.

If X is empty (i.e. $X_s = \emptyset$ for every $s \in S$), then a free pair is called *initial*.
□

Note that, since the definition of free (initial) coincides with the standard one in category theory, the properties of free (initial) objects applies to our framework too; in particular any two free (initial) objects are isomorphic.

Notation. In the sequel for every lazy algebra $\mathcal{A} = (A, \equiv^A, \mathcal{P}^A)$ and every $\tau \in T_\Sigma(A)$ the evaluation τ^A is inductively defined by $a^A = a$ for all $a \in A$ and $f(t_1, \ldots, t_n)^A = f^A(t_1^A, \ldots, t_n^A)$, i.e. is the standard evaluation for the identical valuation of variables in A as elements of A.

Moreover if $(\sigma\colon X \to T_\Sigma(Y))$ $\sigma\colon T_\Sigma(X) \to T_\Sigma(Y)$ and t is a term on X, then $\sigma(t)$ denotes the image of t along (the free extension of) σ.

Finally for every valuation $V: X \to T_\Sigma(A)$ and every term $t \in T_\Sigma(X)$, we will denote by $t^{A,V}$ the evaluation $V(t)^A$, that is the standard evaluation for the valuation $id_A \cdot V$. $\square$

Note that $t^{A,V}$ is a value (or is undefined), while $V(t)$ is a term (and it is always defined).

As in more standard algebraic frameworks, an initial (free) object in a class, if any, is minimally defined and satisfies as few identities between defined terms as possible; moreover the interpretation of predicates and the simplification relation (i.e. the error-recovery) are minimal too.

Theorem 5. *Given a signature Σ, a subclass of* $\mathbf{CAlg}(\Sigma)$ *objects* C *and an* S*-sorted family of variables* X*, if* $(\mathcal{F}, E)$ *is free for* X *in* C*, then for every* $\mathcal{A} = (A, \equiv^A, \mathcal{P}^A) \in \mathsf{C}$*, every valuation* $V: X \to T_\Sigma(A)$ *and every* $t, t', t_1, \ldots, t_n \in T_\Sigma(X)$ *the following conditions hold:*

minimal definedness *if* $t^{F,E}$ *is defined, then* $t^{A,V}$ *is defined too;*
no-confusion *if* $t^{F,E} = t'^{F,E} = a \in F$*, then* $t^{A,V} = t'^{A,V}$*;*
minimal simplification *if* $E(t) \equiv^F E(t')$*, then* $V(t) \equiv^A V(t')$*;*
minimal truth *if* $(E(t_1), \ldots, E(t_n)) \in \mathcal{P}^F(p)$*, then* $(V(t_1), \ldots, V(t_n)) \in \mathcal{P}^A(p)$*.* $\square$

2 A logic of simplifications

The main interest is on the definition of basic specifications taking advantage of the tools introduced so far. As usual in algebraic specification, we consider Horn-Clauses, built starting from three different kinds of atoms: (existential) equality between values, equivalence between terms and predicate application. As in our framework the result of an operation depends not only on the values of its arguments, but also on the history of such values, the valuations for variables have to be made not in the carriers of the models, but on *terms*; in other words we substitute computations, instead of values, for variables.

2.1 Formulas and Specifications

Definition 6. Let $\Sigma = (S, F, P)$ be a signature and X be a family of variables for Σ. Then the set $Atoms(\Sigma, X)$ of *atoms* over Σ and X consists of

Existential equalities: $t =_e t'$, where t, t' are terms on X of the same sort;
Simplifications: $t \equiv t'$, where t, t' are terms on X of the same sort;
Predicates: $p(t_1, \ldots, t_n)$, where t_i are terms of sort s_i on X and $p \in P_{s_1 \ldots s_n}$.

Moreover the set $HC(\Sigma, X)$ of *positive conditional formulas* over Σ and X consists of all formulas of the form $\Delta \supset \epsilon$, where $\Delta \subseteq Atoms(\Sigma, X)$ and $\epsilon \in Atoms(\Sigma, X)$, too. If $\Delta = \emptyset$, then $\Delta \supset \epsilon$ is simply written as ϵ, so that $Atoms(\Sigma, X) \subseteq HC(\Sigma, X)$.

In the sequel for every formula $\phi = \Delta \supset \epsilon$ we will denote by $Var(\phi)$ the set of variables that appear in ϕ, by $Prem(\phi)$ the set Δ of the *premises* of ϕ and by $Cons(\phi)$ the atom ϵ, called the *consequence* of ϕ. Moreover the existential equalities of the form $t{=_e}t$, where both sides are the same term, are denoted by $D(t)$.

Let $\mathcal{A} = (A, \equiv^A, \mathcal{P}^A)$ be a lazy algebra, ϕ be a positive conditional formula over Σ and X and $V\colon Y \to T_\Sigma(A)$ be a valuation, with $Var(\phi) \subseteq Y$; then $\mathcal{A}$ *satisfies* ϕ w.r.t. V, denoted by $\mathcal{A} \models_V \phi$, according with the following conditions:

- $\mathcal{A} \models_V t{=_e}t'$ iff $t^{A,V}$ and $t'^{A,V}$ are the same value in A;
- $\mathcal{A} \models_V t \equiv t'$ iff $V(t) {\equiv^A} V(t')$;
- $\mathcal{A} \models_V p(t_1, \ldots, t_n)$ iff $(V(t_1), \ldots, V(t_n)) \in \mathcal{P}^A(p)$;
- $\mathcal{A} \models_V \Delta \supset \epsilon$ iff $\mathcal{A} \models_V \epsilon$ or $\mathcal{A} \not\models_V \delta$ for some $\delta \in \Delta$.

Then $\mathcal{A}$ *satisfies* ϕ, denoted by $\mathcal{A} \models \phi$, iff $\mathcal{A} \models_V \phi$ for all $V\colon Var(\phi) \to T_\Sigma(A)$. $\square$

Remark. Lazy signature, algebras and formulas do not form an institution. Indeed, since variable valuations range on terms and not on values, the satisfaction condition does not hold, not even for signature inclusions, as the forgotten syntax can increase the number of possible instantiations; the problems are the same as for label algebras. However the lazy framework forms an rps preinstitution (see e.g. [17]) and this property suffices to guarantee that the models of a larger specification, hierarchically built on a smaller one, can be restricted to models of the smaller, so that modular constructions are meaningful. Thus, although the theory of institution independent specification languages ([18, 3, 4]) does not apply directly to the lazy framework, the most relevant part of such theory can be rephrased for rps preinstitutions and hence, in particular, for the lazy formalism. $\square$

Lemma 7. *Let $\Sigma = (S, F, P)$ be a signature and X be a family of variables for Σ; for every lazy algebra $\mathcal{A} = (A, \equiv^A, \mathcal{P}^A)$ on Σ, every positive conditional formula ϕ over Σ and X and all valuations $V\colon X \to T_\Sigma(A)$, $\sigma\colon Y \to T_\Sigma(X)$ we have that $\mathcal{A} \models_{V \cdot \sigma} \phi$ iff $\mathcal{A} \models_V \sigma(\phi)$.* $\square$

Definition 8. A *specification* Sp consists of a signature Σ and a set of positive conditional formulas over Σ and any family X of variables, called the *axioms* of Sp.

Given a specification $Sp = (\Sigma, Ax)$, the *model class* of Sp, denoted by $\mathbf{Mod}(Sp)$, consists of all lazy algebras over Σ satisfying all formulas in Ax. $\square$

2.2 Initial and free models

As more standard frameworks, lazy specifications admit initial and free models, characterized by properties like "no-junk" and "no-confusion" (see e.g. Theorem 5). Moreover the proof follows a classical pattern: the quotient of the term algebra w.r.t. the *minimal congruence* is shown to be free.

Definition 9. Let $\Sigma = (S, F, P)$ be a signature and X be a family of variables for Σ; then a *lazy congruence* over $T_\Sigma(X)$ consists of

- a *partial* congruence $\approx$ on $T_\Sigma(X)$ containing $X \times X$.
- a *total* congruence $\cong$ on $T_\Sigma(X)$ s.t. $\approx \subseteq \cong$ and if $t \cong u$ with t and u belonging to the domain of $\approx$, then $t \approx u$.
- for every $p \in P_{s_1 \ldots s_n}$ a set $\bar{p} \subseteq T_\Sigma(X)_{s_1} \times \ldots \times T_\Sigma(X)_{s_n}$ s.t. if $t_i \cong_{s_i} u_i$ for $i = 1 \ldots n$ and $(t_1, \ldots, t_n) \in \bar{p}$, then $(u_1, \ldots, u_n) \in \bar{p}$.

Given a lazy congruence $\Re = (\approx, \cong, \{\bar{p}\})$ over $T_\Sigma(X)$, the *quotient lazy algebra* $\mathcal{A} = T_\Sigma(X)/\Re$ consists of:

- for every $s \in S$, denoting by $[t]$ the equivalence class of t in $\approx$, $s^A = \{[t] \mid t \in Dom(\approx)\}$;
- for every $f \in F_{s_1 \ldots s_n, s}$, and every t_i in the domain of $\approx$, $f^A([t_1], \ldots, [t_n])$ is $[f(t_1, \ldots, t_n)]$, if $f(t_1, \ldots, t_n)$ belongs to the domain of $\approx$, is undefined otherwise.
- $t \equiv^A t'$ iff $\tau \in \rho(t)$ and $\tau' \in \rho(t')$ exist s.t. $\tau \cong \tau'$, where ρ is defined below.
- for every $p \in P_{s_1 \ldots s_n}$, $(t_1, \ldots, t_n) \in \mathcal{P}^A(p)$ iff $\tau_i \in \rho(t_i)$, for $i = 1 \ldots n$, exist s.t. $(\tau_1, \ldots, \tau_n) \in \bar{p}$, where ρ is defined below.

For every term $t \in T_\Sigma(A)$ let $\rho(t)$ denote the set of terms on $T_\Sigma(X)$ obtained by removing brackets in t, i.e. $\rho(t)$ is inductively defined by $\rho([t]) = \{t' \mid t' \in [t]\}$ if $t \in A$ and $\rho(f(t_1, \ldots t_n)) = \{f(t'_1, \ldots t'_n) \mid t'_1 \in \rho(t_1) \ldots t'_n \in \rho(t_n)\}$;

Moreover in the sequel $\overline{V}: X \to T_\Sigma(A)$ will denote the evaluation defined by $\overline{V}(x) = [x]$ for all $x \in X$. □

The evaluation of terms in a quotient is strongly related to the evaluation within the term algebra.

Lemma 10. *Let $\Re = (\approx, \cong, \{\bar{p}\})$ be a lazy congruence over $T_\Sigma(X)$, $\mathcal{A}$ be the lazy algebra $T_\Sigma(X)/\Re$ and V: $Var(\phi) \to T_\Sigma(A)$ be a valuation. Then for every valuation U: $Var(\phi) \to T_\Sigma(X)$ s.t. $U(x) \in \rho(V(x))$ for all $x \in X$ we have:*

- $t^{A,V} = U(t)^{A,\overline{V}}$ *for every* $t \in T_\Sigma(Var(\phi))$;
- $\overline{V} \cdot U(t) \equiv^A V(t)$ *for every* $t \in T_\Sigma(Var(\phi))$. □

Definition 11. Let $\Sigma = (S, F, P)$ be a signature, X be a family of variables for Σ, $\mathcal{A} = (A, \equiv^A, \mathcal{P}^A)$ be a lazy algebra and $V: X \to A$ be a valuation. Then the *kernel* $k(\mathcal{A}, V)$ of the evaluation w.r.t. $\mathcal{A}$ and V is the congruence $(\approx, \cong, \{\bar{p}\})$ over $T_\Sigma(X)$ defined by:

- $t \approx_s u$ iff $t^{A,V} = u^{A,V} \in s^A$.
- $t \cong_s u$ iff $V(t) \equiv^A V(u)$.
- for every $p \in P_{s_1 \ldots s_n}$, $(t_1, \ldots, t_n) \in \bar{p}$ iff $(V(t_1), \ldots, V(t_n)) \in \mathcal{P}^A(p)$.

Let Sp be a specification over Σ and X be a family of variables for Σ; then the lazy congruence $K(Sp, X)$ is the (componentwise) intersection of $k(\mathcal{A}, V)$ for all $\mathcal{A} \in \mathbf{Mod}(Sp)$ and all valuations $V: X \to A$. □

It is immediate to check that the kernels are congruences and that the intersection of congruences is a congruence too.

As quite common in algebraic frameworks, the free object is the quotient of the corresponding term algebra w.r.t. the "minimal" kernel.

Theorem 12. *Let Sp be a specification over $\Sigma = (S, F, P)$ and X be a family of variables for Σ. Then $(T_\Sigma(X)/K(Sp, X), \overline{V})$ is free for X in $\mathbf{Mod}(Sp)$.* $\square$

3 Using Lazy Specifications

Let us first of all note that every total and partial specification can be immediately and automatically translated[2] into a lazy specification, by adding the definedness of variables in the premises, to make their instantiation range on values, and axioms of the form $p(x_1, \dots, x_n) \supset D(x_i)$ to restrict predicates to work on values; moreover, for total specifications, axioms of the form $D(x_1) \wedge \dots \wedge D(x_n) \supset D(f(x_1, \dots, x_n))$ have to be added, too, to have that the interpretation of function symbols are total functions in all models. As most specifications are total or partial, a specification language based on the lazy framework should provide facilities to concisely describe those requirements. It is interesting to note that, from a practical point of view, stating the definedness of variables in the premises does not increase the number of checks; indeed the typing verification needed in any framework is here replaced by the definedness check.

But lazy specifications also allow the definition of evaluation strategies. For instance, let us assume given a specification Sp_B for the boolean expressions of a programming language, including the constants **true** and **false**, but possibly with other (partial) operations, and enrich it by an **and** construct. Then many different evaluation strategies can be defined. Each of the following groups of axioms presents one of the more usual strategies, assuming that all defined terms reduces either to **true** or to **false**; but it is easy to see that others (e.g. "right to left") could be axiomatized as well.

$$\left.\begin{array}{l} D(x) \wedge y = \mathbf{true} \supset x \;\mathbf{and}\; y = x \\ D(x) \wedge y = \mathbf{false} \supset x \;\mathbf{and}\; y = \mathbf{false} \end{array}\right\} \text{strict evaluation}$$

$$\left.\begin{array}{l} x = \mathbf{true} \supset x \;\mathbf{and}\; y \equiv y \\ x = \mathbf{false} \supset x \;\mathbf{and}\; y \equiv \mathbf{false} \end{array}\right\} \text{left to right evaluation, strict on the first argument}$$

$$\left.\begin{array}{l} \mathbf{true} \;\mathbf{and}\; x \equiv x \\ \mathbf{false} \;\mathbf{and}\; x \equiv \mathbf{false} \end{array}\right\} \text{non-strict left to right evaluation}$$

$$\left.\begin{array}{l} \mathbf{true} \;\mathbf{and}\; x \equiv x \\ \mathbf{false} \;\mathbf{and}\; x \equiv \mathbf{false} \\ x \;\mathbf{and}\; \mathbf{true} \equiv x \\ x \;\mathbf{and}\; \mathbf{false} \equiv \mathbf{false} \end{array}\right\} \text{parallel non-strict evaluation}$$

[2] This indeed is a particular case of logical simulation (see e.g. [8]) and its existence not only guarantees that the lazy framework is at least as expressive as the total and partial ones, but also allows the use of multiparadigm specification languages (see e.g. [4])

Notice the difference between $x = \mathbf{true} \supset x \mathbf{\ and\ } y \equiv y$ and $\mathbf{true\ and\ } y \equiv y$; indeed in the first case x is required to be defined, while the later allows the simplification of $x \mathbf{\ and\ } y$ to y even if x is undefined, provided that x simplifies to **true** (as from $x \equiv \mathbf{true}$, also $x \mathbf{\ and\ } y \equiv \mathbf{true\ and\ } y$ follows, by congruence, and hence $x \mathbf{\ and\ } y \equiv y$ is required by transitivity).

Another interesting point is the interaction between error-recovery and modularity (see [15]). A classical example of this problem is the definition of the stacks (see [5] for a full discussion on the possible specifications of stacks). Indeed the stack data type is parametric w.r.t. the specification of its elements; thus *a priori* there is no way to define the value of the top of the empty stack without destroying the sufficient completeness property; indeed it should be a new error of the primitive type of elements. In many total approaches the solution is to reduce it to any previously existing error value, but there are no guarantees that such value exists. Using partial specifications, it is easy to have that `top` and `pop` are defined only on non-empty stacks, saving the sufficient completeness; consider indeed the following standard partial specification, parametric on the specification of the elements, that is supposed to have a principal sort, `elem`.

$$
\begin{array}{lll}
\mathbf{spec}\ Sp_1 = & \mathbf{enrich}\ \ \mathtt{Elem}\ \ \mathbf{by} & \\
\quad \mathbf{sorts} & \mathtt{stack} & \\
\quad \mathbf{opns} & \mathtt{empty}\colon \to \mathtt{stack} & \\
 & \mathtt{push}\colon \mathtt{elem} \times \mathtt{stack} \to \mathtt{stack} & \\
 & \mathtt{top}\colon \mathtt{stack} \to \mathtt{elem} & \\
 & \mathtt{pop}\colon \mathtt{stack} \to \mathtt{stack} & \\
\quad \mathbf{axioms} & D(\mathtt{empty}) & \\
 & D(x) \wedge D(s) \supset D(\mathtt{push}(x,s)) & \\
 & D(x) \wedge D(s) \supset \mathtt{top}(\mathtt{push}(x,s)) = x & \\
 & D(x) \wedge D(s) \supset \mathtt{pop}(\mathtt{push}(x,s)) = s &
\end{array}
$$

If error recovery is required, then the only way to deal with it in partial (as well as in order-sorted) frameworks is to add the definedness (well-formedness) of "errors", destroying the sufficient completeness, together with recovery axioms, so that the framework is in no way different w.r.t. the standard many-sorted approach. Using lazy algebras, instead, it is possible to introduce simplification on terms.

$$
\begin{array}{ll}
\mathbf{spec}\ Sp_2 = & \mathbf{enrich}\ \ Sp_1\ \ \mathbf{by} \\
\quad \mathbf{axioms} & \mathtt{top}(\mathtt{push}(x,s)) \equiv x \\
 & \mathtt{pop}(\mathtt{push}(x,s)) \equiv s
\end{array}
$$

Notice that in Sp_2 for example the term $\tau = \mathtt{top}(\mathtt{push}(t, \mathtt{pop}(\mathtt{empty})))$ simplifies, in all models, to t, but, even if t is defined, τ is undefined; indeed τ represents a "recovered" computation. The strictness of functions prevents τ to be defined, unless $\mathtt{pop}(\mathtt{empty})$ is defined too. In other words the initial model of Sp_2 consists of the same partial algebra as the initial model of Sp_1, but the simplification relation has been enlarged to recover some errors.

Note that finer error-recovery strategies can be defined as well; for example the following specification corresponds to the recovery of just one level of error due to `pop` and `top`.

$$
\begin{array}{ll}
\textbf{spec } Sp_3 = & \textbf{enrich } Sp_1 \textbf{ by} \\
\quad \textbf{axioms} & D(s) \wedge D(x) \supset \mathtt{top}(\mathtt{push}(x, \mathtt{pop}(s))) \equiv x \\
& D(s_1) \wedge D(s_2) \supset \mathtt{pop}(\mathtt{push}(\mathtt{top}(s_1), s_2)) \equiv s_2
\end{array}
$$

A third kind of problem comes from the definition of *limited* data types, for example limited stacks. The intuition is that after a phase of top-down design, during which the stacks were regarded as their "ideal" model, in the bottom-up development stacks should be replaced by a more "real" and limited model. But in all algebraic approaches this step cannot be done painlessly. Indeed terms that were seen as perfectly correct (i.e., depending on the framework, as defined, well-formed, labeled by "ok" and so on) should be moved in the "incorrect" part of the type, loosing so, in some sense, a property, while all frameworks are incremental and only allow to increase the properties of any term. This reflects in the need for a heavy modification of the original specification (e.g. by decorating the axioms in order to apply them only to those arguments that do not raise an overflow), against any notion of modularity.

Analyzing this phenomenon, it is easy to see that the original specification is overdefined; indeed if the knowledge about which terms represent values and which errors is reached only at the last stage of design, then it is incorrect, and indeed it is a source of troubles, to define `push` as a total function, because it is not total in limited models. But the standard approaches require the terms to be defined (well formed/labeled with "ok"...) in order to apply simplifying axioms to them, so that the contradiction between the need for delaying the decision about definedness and the capability to state equalities among terms is inescapable. Using lazy specification, instead, the specification Sp_1 can be rephrased with equality replaced by the simplification relation, capturing in this way the intuition that the axioms state an equivalence on computation and that the definedness is only added at the last possible moment.

$$
\begin{array}{ll}
\textbf{spec } Sp'_1 = & \textbf{enrich } \mathtt{Elem} \textbf{ by} \\
\quad \textbf{sorts} & \mathtt{stack} \\
\quad \textbf{opns} & \mathtt{empty}\colon \to \mathtt{stack} \\
& \mathtt{push}\colon \mathtt{elem} \times \mathtt{stack} \to \mathtt{stack} \\
& \mathtt{top}\colon \mathtt{stack} \to \mathtt{elem} \\
& \mathtt{pop}\colon \mathtt{stack} \to \mathtt{stack} \\
\quad \textbf{axioms} & \mathtt{top}(\mathtt{push}(x, s)) \equiv x \\
& \mathtt{pop}(\mathtt{push}(x, s)) \equiv s
\end{array}
$$

Now the definition of the stacks large at the most `max` elements, with the convention that $\mathtt{push}^n(x_1, \ldots, x_n, s)$ is the term inductively defined by $\mathtt{push}^0(\lambda, s) = s$ and $\mathtt{push}^{n+1}(x_1, \ldots, x_{n+1}, s) = \mathtt{push}^n(x_1, \ldots, x_n, \mathtt{push}(x_{n+1}, s))$ for every term s of sort stack, is the following enrichment:

$$
\begin{array}{ll}
\textbf{spec } Sp'_2 = & \textbf{enrich } Sp'_1 \textbf{ by} \\
\quad \textbf{axioms} & D(x_1) \wedge \ldots \wedge D(x_{\mathtt{max}}) \supset D(\mathtt{push}^n(x_1, \ldots, x_{\mathtt{max}}, \lambda))
\end{array}
$$

By strictness the definition of all the subterms is given. Notice that the axioms of Sp'_1 define a strong error-recovery strategy as the definedness of variables is not required in the premises; adding such premises, the error-recovery can be weakened, and even, adding the definedness of all subterms instead of variables, completely banished.

4 A Sound and Complete Calculus

It is implicit in the nature of algebraic specifications, seen as tools for reasoning on programs, the need for a calculus to help the understanding of the specification basic properties, possibly with the related definition of a tool. A calculus is doubly needed in this framework, that is based on the concept of simplification.

We are mainly interested in formulas where most variables can only range on *defined* elements, represented in the form $\{D(x_1), \ldots, D(x_n)\} \supset \epsilon$, or on subsets described by predicates. Thus, although atomic deduction would suffice for the definition of the initial object, the calculus presented here is strictly conditional. This, moreover, enables a concise description of variables used in deduction already utilized for the partial approach (see e.g. [2]), in order to avoid well known problems related to empty-carriers.

Definition 13. Let $\Sigma = (S, F, P)$ be a signature, X an S-sorted family of denumerable variable sets and Ax a set of positive conditional formulas over Σ and X.

Then the inference system $CL(Sp)$, where $Sp = (\Sigma, Ax)$ consists of Ax and of the following rules, where $t, t', t'', t_1, \ldots, t_n, t'_i$ are terms on Σ and X:

Existential equality rules

$$\frac{}{t{=_e}t' \supset t'{=_e}t} \qquad \frac{}{t{=_e}t' \wedge t'{=_e}t'' \supset t{=_e}t''} \qquad \frac{}{D(f(t_1, \ldots, t_n)) \supset D(t_i)}$$

$$\frac{}{t_i{=_e}t'_i \wedge D(f(t_1, \ldots, t_n)) \supset f(t_1, \ldots, t_n){=_e}f(t_1, \ldots, t_{i-1}, t'_i, t_{i+1} \ldots, t_n)}$$

Simplification rules

$$\frac{}{t \equiv t} \qquad \frac{}{t \equiv t' \supset t' \equiv t} \qquad \frac{}{t \equiv t' \wedge t' \equiv t'' \supset t \equiv t''}$$

$$\frac{}{t_1 \equiv t'_1 \wedge \ldots \wedge t_n \equiv t'_n \supset f(t_1, \ldots, t_n) \equiv f(t'_1, \ldots, t'_n)}$$

$$\frac{}{t \equiv t' \wedge D(t) \wedge D(t') \supset t{=_e}t'} \qquad \frac{}{t{=_e}t' \supset t \equiv t'}$$

Predicate rule

$$\frac{}{t_1 \equiv t'_1 \wedge \ldots \wedge t_n \equiv t'_n \wedge p(t_1, \ldots, t_n) \supset p(t'_1, \ldots, t'_n)}$$

Substitution rule

$$\frac{\phi}{\sigma(\phi)} \qquad \sigma \text{ term substitution}$$

Modus Ponens

$$\frac{\Delta \supset \epsilon, \Gamma \supset \delta}{\Delta - \{\delta\} \cup \Gamma \cup \Theta \supset \epsilon} \qquad \Theta = \{x \equiv x \mid x \in Var(\delta) - (Var(\Delta - \{\delta\} \cup \Gamma \supset \epsilon))\}$$

If ϕ is inferred by $CL(Sp)$, then we write $CL(Sp) \vdash \phi$. □

It is worth noting that the side condition of the Modus Ponens rule guarantees that variables can be eliminated, during a deduction, only if a substitution takes place. This suffices for inconsistent deductions to be avoided even in the case of empty carriers and non-sensible signatures.

The above calculus is sound, as the following proposition will show.

Proposition 14. *Let $Sp = (\Sigma, Ax)$ be a specification; then $CL(Sp) \vdash \phi$ implies that $\mathcal{A} \models_V \phi$ for all $\mathcal{A} \in \mathbf{Mod}(Sp)$ and all $V\colon Var(\phi) \to T_\Sigma(A)$.* $\square$

We are mainly interested in atomic completeness, as an atomically-complete calculus gives the free (initial) model, that satisfies as few atoms as possible, but, since in our framework the variables used in the deduction that cannot be eliminated (as values for their instantiation are missing) are kept track of by atoms of the form $x \equiv x$ (if the variable can be instantiated on every term) or $x =_e x$ (if the variable can be instantiated only on defined term) in the premises, we should regard

$$x_1 \equiv x_1 \wedge \ldots \wedge x_n \equiv x_n \wedge y_1 =_e y_1 \wedge \ldots \wedge y_m =_e y_m \supset \epsilon$$

as an *atom* in the notation $\forall x_1 \ldots x_n : T_\Sigma(A); \forall y_1 \ldots y_m : A.\epsilon$. But if the x have to be instantiated on terms built from the values of A (and hence from the y), their presence in the premises does not increase the number of elements needed in the carriers in order to make the deduction possible. Hence we are interested in an easier form of formulas, those that in the premises have only definedness assertions.

Notation. Every ϕ of the form $x_1 =_e x_1 \wedge \ldots \wedge x_n =_e x_n \supset \epsilon$ is called *basically atomic* and if $CL(Sp) \vdash \phi$, then we say that $CL(Sp) \vdash_X \epsilon$ holds for each $X \supseteq \{x_1 \ldots x_n\}$. $\square$

The above calculus is complete w.r.t. basically atomic sentences; the proof is done by constructing a model satisfying only the deduced atoms.

Definition 15. Let $Sp = (\Sigma, Ax)$ be a specification and X be a family of variables; then $K(CL(Sp), X)$ is the lazy congruence $(\approx, \cong, \{\bar{p}\})$ defined by:

- $t \approx t'$ iff $CL(Sp) \vdash_X t =_e t'$ for all $t, t' \in T_\Sigma(X)$;
- $t \cong t'$ iff $CL(Sp) \vdash_X t \equiv t'$ for all $t, t' \in T_\Sigma(X)$;
- $(t_1, \ldots, t_n) \in \bar{p}$ iff $CL(Sp) \vdash_X p(t_1, \ldots, t_n)$ for all $t_i \in T_\Sigma(X)$ and $p \in P$;

Moreover let $\mathcal{G}(Sp, X)$ be the quotient lazy algebra $T_\Sigma(X)/K(CL(Sp), X)$ and let $E\colon X \to G(Sp, X)$ be the evaluation defined by $E(x) = [x]$ for all $x \in X$. $\square$

Let us remark that $K(CL(Sp), X)$ is a lazy congruence. Indeed, because of modus ponens and of the rules for existential equality, $\approx$ is a partial congruence, whose domain includes X; moreover, because of modus ponens and of the rules for simplification, $\cong$ is a total congruence, $\approx \subseteq \cong$ and if $t \cong t'$, with $t \approx t$ and $t' \approx t'$, then $t \approx t'$. Finally, the predicate interpretation is well defined w.r.t. simplification, because of the rules for predicate and modus ponens.

Proposition 16. *Let* $Sp = (\Sigma, Ax)$ *be a specification,* X *be a family of variables and* $V\colon Var(\phi) \to T_\Sigma(G(Sp, X))$ *be a valuation. Then* $\mathcal{G}(Sp, X) \models_E \epsilon$ *iff* $CL(Sp) \vdash_X \epsilon$*, for every atom* ϵ *over* X. $\square$

Note that, in particular, $\mathcal{G}(Sp, X) \models_E x{=_e}x$ for all $x \in X$, because of the symmetry rule for existential equality.

Proposition 17. *Let* $Sp = (\Sigma, Ax)$ *be a specification and* X *be a family of variables. Then* $\mathcal{G}(Sp, X)$ *is a model of* Sp. $\square$

We are finally able to show the completeness of the calculus w.r.t. basically atomic formulas.

Theorem 18. *Let* $Sp = (\Sigma, Ax)$ *be a specification and* ϕ *be* $\Delta \supset \epsilon$*, where* $\Delta = \{D(x_1), \ldots, D(x_m)\}$*; if* $\mathcal{A} \models_V \phi$ *for all* $\mathcal{A} \in \mathbf{Mod}(Sp)$ *and all* $V\colon X \to T_\Sigma(A)$*, then* $CL(Sp) \vdash_X \epsilon$*, where* $X = Var(\phi)$. $\square$

The calculus gives the free models.

Theorem 19. *Let* Sp *be a specification over* $\Sigma = (S, F, P)$ *and* X *be a family of variables for* Σ*. Then* $(\mathcal{G}(Sp, X), E)$ *is free for* X *in* $\mathbf{Mod}(Sp)$. $\square$

Conclusions and Further Developments

Having proposed an algebraic framework to easily deal with the features of an imperative or functional programming language, the next step should be the definition of a user-friendly specification language, whose semantics will be defined in terms of lazy specifications. This will lead to investigate about rewriting techniques for the lazy framework and, possibly, to an implementation. To this aim it could be useful to modify the simplification relation, dropping the requirement that it is a congruence, and making it a "rewrite" relation.

Acknowledgments. I would like to thank my office-mate, Elena Zucca, for patiently debating with me the original, vague and confused intuitions that brought to the definition of the lazy frame. Moreover I'm indebted to Gianna Reggio for carefully reading a draft of this paper and to Egidio Astesiano for a continuous stimulus to research.

References

1. E. Astesiano and M. Cerioli. Non-strict don't care algebras and specifications. In S. Abramsky and T.S.E. Maibaum, editors, *Proceedings of TAPSOFT'91*, number 493 in Lecture Notes in Computer Science, pages 121–142, Berlin, 1992. Springer Verlag.
2. E. Astesiano and M. Cerioli. Free objects and equational deduction for partial conditional specifications. *Theoretical Computer Science*, 1995. To appear.
3. E. Astesiano and M. Cerioli. Relationships between logical frames. In *Recent Trends in Data Type Specification*, number 655 in Lecture Notes in Computer Science, pages 126–143, Berlin, 1993. Springer Verlag.

4. E. Astesiano and M. Cerioli. Multiparadigm specification languages: a first attempt at foundations. In D.J. Andrews, J.F. Groote, and C.A. Middelburg, editors, *Semantics of Specification Languages (SoSL'93)*, Workshops in Computing, pages 168–185. Springer Verlag, 1994.
5. J.A. Bergstra and J.V. Tucker. The inescapable stack: an exercize in algebraic specification with total functions. Technical Report P8804, University of Amsterdam; Programming Research Group, 1988.
6. G. Bernot and P. Le Gall. Label algebras: a systematic use of terms. In *Recent Trends in Data Type Specification*, number 655 in Lecture Notes in Computer Science, pages 144–163, Berlin, 1993. Springer Verlag.
7. P. Burmeister. *A Model Theoretic Oriented Approach to Partial Algebras.* Akademie Verlag, Berlin, 1986.
8. M. Cerioli. *Relationships between Logical Formalisms.* PhD thesis, Universities of Genova, Pisa and Udine, 1993. Available as internal report of Pisa University, TD-4/93.
9. R. Diaconescu. The logic of Horn clauses is equational. Submitted for publication, 1992.
10. J.A. Goguen and R. Diaconescu. A survey of order sorted algebra. Draft, 1992.
11. V. Manca, A. Salibra, and G. Scollo. Equational type logic. *Theoretical Computer Science*, 77:131–159, 1990. Special Issue dedicated to AMAST'89.
12. A. Mégrelis. *Algèbre galactique - Un procédé de calcul formel, relatif aux semi-functions, à l'inclusion et à légalité.* PhD thesis, University of Nancy I, 1990.
13. P. Mosses. Unified algebras and institutions. In *Proceedings of 4th Annual IEEE Symposium on Logic in Computer Science*, pages 304–312, 1989.
14. P. Mosses. The use of sorts in algebraic specifications. In *Recent Trends in Data Type Specification*, number 655 in Lecture Notes in Computer Science, pages 66–92, Berlin, 1993. Springer Verlag.
15. A. Poigné. Partial algebras, subsorting, and dependent types: Prerequisites of error handling in algebraic specifications. In *Recent Trends in Data Type Specification*, number 332 in Lecture Notes in Computer Science, pages 208–234, Berlin, 1987. Springer Verlag.
16. H. Reichel. *Initial Computability, Algebraic Specifications, and Partial Algebras.* Akademie Verlag, 1986.
17. A. Salibra and G. Scollo. A soft stairway to institutions. In *Recent Trends in Data Type Specification*, number 655 in Lecture Notes in Computer Science, pages 310–329, Berlin, 1992. Springer Verlag.
18. D. Sannella and A. Tarlecki. Specifications in an arbitrary institution. *Information and Computation*, 76:165–210, 1988.
19. M. Wirsing. Algebraic specification. In *Handbook of Theoretical Computer Science.* North Holland, 1990.

This article was processed using the LaTeX macro package with LLNCS style