An Object Calculus with Algebraic Rewriting

Adriana Compagnoni
University of Edinburgh
Department of Computer Science,
Edinburgh EH9 3JZ, U. K.
abc@dcs.ed.ac.uk

Maribel Fernández
LIENS (CNRS URA 1327)
École Normale Supérieure
45 Rue d'Ulm, 75005 Paris, France
maribel@dmi.ens.fr

Abstract. In trying to use Abadi and Cardelli's object calculi as a foundation for a programming language the addition of algebraic data types arises naturally. This paper defines such an extension, shows a motivating example, and explores the new calculi by establishing properties such as Church-Rosser, subject reduction and uniqueness of types.

Keywords: Object calculi; Rewriting; Combined calculi; Type systems.

1 Introduction

Abadi and Cardelli [2] presented a simple object calculus, *the ς-calculus*, which can be used to model object oriented languages in the same way as functional languages are modelled by the λ-calculus. The untyped ς-calculus supports *method update*. The first-order (typed) version of the calculus, extended with a subtyping relation, models also object subsumption.

The ς-calculus is computationally complete (the λ-calculus can be encoded as shown in [2]). However, it lacks expressiveness for the representation of data types: data types can be encoded (e.g. through encodings in the λ-calculus), but this kind of encoding is inefficient and unnatural. To overcome this, Abadi and Cardelli [1] introduced in the typed ς-calculus free algebras that can be thought of as built-in data structures. In this paper we go one step further, by extending the untyped calculus with constants and object rewrite rules. In this way we can also model other computational features that are not present in the ς-calculus, such as non-determinism (choice operators).

More precisely, we define an extension of the ς-calculus where algebraic operators can be used to build objects. Objects can have algebraic expressions as fields or methods, and conversely algebraic operations or constructors can take objects as arguments. For example, a list can be a field of an object and we can also form lists or trees of objects. Formally, the calculus is defined by extending the syntax of the ς-calculus with a set of *algebraic symbols* which are used to build objects, and with *algebraic rewrite rules on objects.* To illustrate the expressiveness of the calculus, we give an example of a text editor using lists of characters. Alternative choices of syntax are discussed in the conclusions.

The Church-Rosser property of the ς-calculus is not preserved in this extension even if the algebraic rules are confluent (we give a counterexample). But we show that with suitable restrictions on the form of the rewrite rules (namely by requiring linear left-hand sides), it is possible to restore this important property.

We present also a typed version of our extended calculus, which satisfies uniqueness of types and subject reduction. The subtyping rules of [2] can be added in a straightforward way to model subsumption of objects. The extension of this calculus with subtyping rules on algebraic types is beyond the scope of this paper and is under development. Also the recursive types of [2] can be added to give more informative typings for objects whose methods return *self* or an updated version of *self*. With these extensions, the calculus of Gordon and Rees [5], which was used to study the correspondence between operational semantics and equivalence in the equational theory of Abadi and Cardelli, can be seen as a particular case: it corresponds to the typed ς-calculus with recursive types and subtyping, extended with a type Bool, constructors True and False, and a conditional defined by two rewrite rules.

Fisher, Honsell and Mitchell [9] have developed an alternative object calculus: a λ-calculus of extensible objects. We expect that the same ideas can be applied to their calculus as well.

2 The Calculus

In this section we define an untyped object calculus which is an algebraic extension of the pure object calculus of Abadi and Cardelli [2]. This extended calculus will serve as a basis for the definition of a typed algebraic object calculus in the following sections.

2.1 Syntax

Definition 1. The set $O(\mathcal{L}, \mathcal{F}, \mathcal{X})$ of *objects* is defined inductively from denumerable pairwise disjoint sets $\mathcal{L}$ (*labels*), $\mathcal{X}$ (*variables*), and $\mathcal{F}$ (*algebraic symbols* with fixed arities), as follows:

1. $\mathcal{X} \subseteq O(\mathcal{L}, \mathcal{F}, \mathcal{X})$;
2. if $f \in \mathcal{F}$ has arity n, and $a_1, \ldots, a_n$ are objects, then $f(a_1, \ldots, a_n)$ is an object;
3. if $l_1, \ldots, l_n$ are different labels and $b_1, \ldots, b_n$ objects, then $[l_i = \varsigma(x_i) b_i{}^{1 \leq i \leq n}]$ is an object with *method names* $l_1, \ldots, l_n$ and *methods* $\varsigma(x_i)\, b_i$ $(1 \leq i \leq n)$, where the order of these components does not matter;
4. if a is an object with method name l then $a.l$ is an object;
5. if a is an object with method name l then $a.l \Leftarrow \varsigma(x)\, b$ is an object.

In a method $\varsigma(x)\, b$, x is the *self variable* and b the *body*. A method that does not use its self parameter (i.e. x does not appear in b) is called a *field*. The last two object constructors in the previous definition are called *field selection* and *field update* respectively, or *method invocation* and *method update* respectively, depending on whether we speak of a field or a proper method. Objects that are built using only symbols in $\mathcal{F}$ and $\mathcal{X}$ will be called *algebraic objects*.

The symbol ς is a binder. The definitions of free and bound variable, as well as substitution, are standard. We refer the reader to [2] for additional details of these issues. Substitutions will be denoted by $\{x_1 \mapsto o_1, \ldots, x_n \mapsto o_n\}$ where $x_1, \ldots, x_n$ are different variables. We work modulo α-conversion, identifying $\varsigma(x)\, b$ with $\varsigma(y)\, b\{x \mapsto y\}$ when y is not free in b. We will use the following standard abbreviations: O instead of $O(\mathcal{L}, \mathcal{F}, \mathcal{X})$ when $\mathcal{L}$, $\mathcal{F}$, and $\mathcal{X}$ are clear from the context, $a.l := b$ for $a.l \Leftarrow \varsigma(y)\, b$ when y does not appear in b, and $[\ldots, l_i = b_i, \ldots]$ for $[\ldots, l_i = \varsigma(x_i)\, b_i, \ldots]$ when x_i does not occur in b_i.

2.2 Reduction Rules

The reduction relation of our calculus combines *algebraic reductions* and *pure object reductions* (corresponding to method invocation and update). The algebraic reductions are induced by a set R of *object rewrite rules*, which are written $l \to r$ where l and r are algebraic objects in O, $l \notin \mathcal{X}$, and the variables that appear in r appear also in l. We define first the one-step reduction relation. We use $\equiv$ to denote syntactic equality, and $-$ as a distinguished variable in contexts. A *context* is an object e with an occurrence of the free variable $-$. If e is a context and a an object, we write $e[a]$ for the result of replacing $-$ by a in e.

Definition 2. We write $a \to b$, for $a, b \in O$, if there is some object o such that $o \equiv [l_i = \varsigma(x_i)\; b_i \;^{1 \le i \le n}]$, and either

1. $a \equiv o.l_j$ for some $1 \le j \le n$, and $b \equiv b_j\{x_j \mapsto o\}$, or
2. $a \equiv o.l_j \Leftarrow \varsigma(x)\; c$ and $b \equiv [l_j = \varsigma(x)\; c, l_i = \varsigma(x_i)\; b_i \;^{1 \le i \le n, i \ne j}]$, or
3. $a \equiv l\sigma$ for some substitution σ and object rewrite rule $l \to r$, and $b \equiv r\sigma$.

The reduction relation on objects is closed under contexts: $a \to b$ implies that $C[a] \to C[b]$ for any context C. We denote by $\to^*$ the transitive reflexive closure of $\to$.

Parts 1 and 2 of the previous definition give the operational semantics of method invocation and update. A method invocation $o.l$ reduces to the result of the substitution of the object o for the self parameter in the body of the method named l. A method update $o.l_j \Leftarrow \varsigma(x)\; c$ replaces the method l_j with $\varsigma(x)\; c$ (it is an atomic operation). Note that this is a "functional" operation: an update produces a modified copy of the object. These reductions (parts 1 and 2 of the previous definition) coincide with the pure object reductions of the ς-calculus [2], and will be denoted by $\to_\varsigma$. Part 3 defines the algebraic reductions of our calculus, which will be denoted by $\to_R$. Then $\to = \to_\varsigma \cup \to_R$.

Abadi and Cardelli originally defined a particular strategy of rewriting for their calculus: no reductions under binders (this corresponds roughly to weak head reduction in λ-calculus), but in [2] they lift the restriction. We do not fix any strategy, since the properties we are going to show for the extended calculus hold for any strategy.

2.3 An Example

Here we give an example of a text editor where we use the recursive nature of objects that allows methods to refer to each other, and algebraic expressions to represent the text with lists.

$$
\begin{array}{l}
\text{Editor} \equiv [\,\mathit{text} = [\ \mathit{before} = l_1,\\
\qquad\qquad\qquad \mathit{after} = l_2,\\
\qquad\qquad\qquad \mathit{cursor} = \varsigma(s)\mathit{hd}(s.\mathit{after})\ ],\\
\qquad \mathit{input} = \varsigma(s)\lambda x.s.\mathit{text} := (s.\mathit{text}.\mathit{before} := s.\mathit{text}.\mathit{before}@[x]),\\
\qquad \mathit{find} = \varsigma(s)\lambda x.\text{if } s.\mathit{text}.\mathit{cursor} = x\\
\qquad\qquad\qquad \text{then } s.\mathit{message} := \text{“found”}\\
\qquad\qquad\qquad \text{else if } s.\mathit{text}.\mathit{cursor} \neq \star\\
\qquad\qquad\qquad\quad \text{then } s.\mathit{forward}.\mathit{find}(x)\\
\qquad\qquad\qquad\quad \text{else } s.\mathit{message} := \text{“not found”},\\
\qquad \mathit{init} = \varsigma(s)s.\mathit{text} := ((s.\mathit{text}.\mathit{before} := \mathit{nil}).\mathit{after} := [\star]),\\
\qquad \mathit{forward} = \varsigma(s)\ \text{if } s.\mathit{text}.\mathit{cursor} \neq \star\\
\qquad\qquad\qquad \text{then } s.\mathit{text} := ((s.\mathit{text}.\mathit{before} :=\\
\qquad\qquad\qquad s.\mathit{text}.\mathit{before}@[s.\mathit{text}.\mathit{cursor}]).\mathit{after} := \mathit{tl}(s.\mathit{text}.\mathit{after}))\\
\qquad\qquad\qquad \text{else } s.\mathit{message} := \text{“end of text”},\\
\qquad \mathit{backwards} = \varsigma(s)\ \text{if } s.\mathit{text}.\mathit{before} \neq \mathit{nil}\\
\qquad\qquad\qquad \text{then } s.\mathit{text} := ((s.\mathit{text}.\mathit{after} :=\\
\qquad\qquad\qquad \mathit{last}(s.\mathit{text}.\mathit{before})::(s.\mathit{text}.\mathit{after})).\mathit{before} :=\\
\qquad\qquad\qquad \mathit{no\text{-}last}(s.\mathit{text}.\mathit{before}))\\
\qquad\qquad\qquad \text{else } s.\mathit{message} := \text{“beginning of text”},\\
\qquad \mathit{message} = \text{“ ”}\ ]
\end{array}
$$

The text is divided by the *cursor* into two parts: *before* and *after*, and the *cursor* is the first character of the *after* list. The initial values of *before* and *after*, l_1 and l_2, are irrelevant. An empty line is a list with the distinguished character $\star$ only. We write $[\star]$ for $\star::\mathit{nil}$, and $[x]$ for $x::\mathit{nil}$. Lists have the constructors *nil* and *::* as usual, and *hd*, *tl*, @, *no-last*, and *last* are defined with the following rules.

$$
\begin{array}{rlrl}
\mathit{hd}(x::l) & \rightarrow x & \mathit{tl}(x::l) & \rightarrow l\\
\mathit{nil}@y & \rightarrow y & (x::l)@y & \rightarrow x::(l@y)\\
\mathit{no\text{-}last}(x::\mathit{nil}) & \rightarrow \mathit{nil} & \mathit{no\text{-}last}(x::(y::l)) & \rightarrow x::\mathit{no\text{-}last}(y::l)\\
\mathit{last}(x::\mathit{nil}) & \rightarrow x & \mathit{last}(x::(y::l)) & \rightarrow \mathit{last}(y::l)
\end{array}
$$

Similarly, we introduce booleans by defining *true*, *false*, and *if then else*. In the tests, equality (=) means syntactic equality. The lambda abstraction is defined as in [2], by encoding it in the ς-calculus.

There is a choice to represent lists (and booleans). Instead of defining algebraic objects with their rewriting rules, we can write them as objects with methods as follows:

$$
\begin{array}{r}
\mathit{before} = [\mathit{val} = ..., \mathit{hd}\ = ..., \mathit{tl}\ = ..., @ = ..., \mathit{last}\ = ..., \mathit{no\text{-}last}\ = ...]\\
\mathit{after} = [\mathit{val} = ..., \mathit{hd}\ = ..., \mathit{tl}\ = ..., @ = ..., \mathit{last}\ = ..., \mathit{no\text{-}last}\ = ...]
\end{array}
$$

the disadvantage being that the definitions for the different constructors are repeated in each list. The term rewriting option avoids such repetitions.

Abadi and Cardelli [2] presented an example of a calculator which can also benefit from using term rewriting to define the arithmetic operations instead of encoding them.

3 Properties

The ς-calculus is confluent [2], i.e. if $o \to_\varsigma^* o_1$ and $o \to_\varsigma^* o_2$, then $o_1 \to_\varsigma^* o_3$ and $o_2 \to_\varsigma^* o_3$ for some o_3. But we cannot expect to have the same property in our extension, since the algebraic rules are arbitrary. In order to obtain a confluent calculus we need to impose some restrictions on the algebraic rewrite rules.

Even if we consider only algebraic reductions that are confluent we cannot ensure confluence of the combined calculus. The following is a simple counterexample (a similar problem appears in combinations of λ-calculus and algebraic rewriting, see e.g. [4] which inspired our counterexample). Assume we are working with a set $\mathcal{F}$ of symbols that contains the object constants 0 and 1, the unary symbol S, and the binary symbol f. The rewrite rules $\{f(x,x) \to 0,\ f(x,S(x)) \to 1\}$ induce a reduction relation that is confluent for the set of algebraic objects built from $\mathcal{F}$ and a set $\mathcal{X}$ of variables. However, if we consider the whole set O of objects and the full reduction relation $\to_R \cup \to_\varsigma$, the calculus is not confluent: the problem is that the object $o.l$, where $o \equiv [l = \varsigma(x)\ S(x.l)]$, reduces in the object calculus to $S(o.l)$, then the object $f(o.l, o.l)$ can be reduced in two ways, either to 0 or to 1.

To ensure confluence in combinations of λ-calculus and algebraic rewriting the standard solution is to restrict the set of algebraic rules to left-linear rules, or to introduce types to avoid infinite reductions. We will take the first alternative. Using proof techniques similar to the ones used for λ-calculus (see e.g. [10]), we will show that the extended ς-calculus is confluent with respect to $\to_R \cup \to_\varsigma$ when R is left-linear and confluent for algebraic objects. The second alternative, restricting the set of objects to those that are strongly normalizing, will be explored in future work.

We will use two lemmas, the first one shows that the confluence of R extends to the whole set $O(\mathcal{L}, \mathcal{F}, \mathcal{X})$ of objects, and the second one establishes the commutation of $\to_R$ and $\to_\varsigma$ when R is left-linear.

Lemma 3. *If R is left-linear and confluent for algebraic objects, then R is confluent for $O(\mathcal{L}, \mathcal{F}, \mathcal{X})$.*

Proof. Assume $o_1 \,{}_R^*\!\!\leftarrow o \to_R^* o_2$. We will show by structural induction that there exists o_3 such that $o_1 \to_R^* o_3$, $o_2 \to_R^* o_3$. We distinguish the cases:

1. $o \equiv [l_i = \varsigma(x_i) b_i{}^{1 \le i \le n}]$. Then the reductions $\to_R$ take place inside the objects b_i, that is, $o_1 \equiv [l_i = \varsigma(x_i) b_i'{}^{1 \le i \le n}]$, $o_2 \equiv [l_i = \varsigma(x_i) b_i''{}^{1 \le i \le n}]$, and $b_i \to_R^* b_i'$, $b_i \to_R^* b_i''$ $(1 \le i \le n)$. By induction, there exists b_i''' such that $b_i' \to_R^* b_i'''$ and $b_i'' \to_R^* b_i'''$. Hence we can take $o_3 \equiv [l_i = \varsigma(x_i) b_i'''{}^{1 \le i \le n}]$.

2. $o \equiv a.l_j$. We proceed as in the previous case.
3. $o \equiv (a.l_j \Leftarrow \varsigma(x)\ b)$. We proceed as in the previous cases.
4. $o \equiv f(a_1, \ldots, a_n)$. Let u_k, $1 \leq k \leq m$, be the maximal subexpressions of the form $[l_i = \varsigma(x_i)b_i \ ^{1 \leq i \leq n}]$, $a.l_j$, or $(a.l_j \Leftarrow \varsigma(x)\ b)$ in $a_1, \ldots, a_n$. We will replace them by different new variables $w_1, \ldots, w_m$, obtaining in this way an algebraic object o' (since no occurrences of non-algebraic constructors are left). Since R is algebraic and left-linear, the reductions $\rightarrow_R^*$ made in the u_k are independent from those in o', hence:
   - $o'_1\ {}_R^*\!\leftarrow o' \rightarrow_R^* o'_2$,
   - $o_1 = o'_1\{w_1 \mapsto v_1, \ldots, w_m \mapsto v_m\}$ where $u_k \rightarrow_R^* v_k$, $1 \leq k \leq m$, and
   - $o_2 = o'_2\{w_1 \mapsto v'_1, \ldots, w_m \mapsto v'_m\}$ where $u_k \rightarrow_R^* v'_k$, $1 \leq k \leq m$.

   Since o', o'_1, and o'_2 are algebraic objects, and R is confluent, there exists o'_3 such that $o'_1 \rightarrow_R^* o'_3\ {}_R^*\!\leftarrow o'_2$. By induction, u_k is confluent, hence, there exists v''_k such that $v_k \rightarrow_R^* v''_k$ and $v'_k \rightarrow_R^* v''_k$, for $1 \leq k \leq m$. Then we take $o_3 = o'_3\{w_1 \mapsto v''_1, \ldots, w_m \mapsto v''_m\}$.

The assumption of left-linearity is not necessary in the previous lemma, it just makes the proof easier. However, the hypothesis of algebraic rules in R is crucial.

Lemma 4. *If R is left-linear, and confluent for algebraic objects, then*

$$b\ {}_\varsigma^*\!\leftarrow o \rightarrow_R^* a \ \Rightarrow\ \exists c : b \rightarrow_R^* c\ {}_\varsigma^*\!\leftarrow a.$$

Proof. We will use a reduction relation $\Rrightarrow_\varsigma$ that makes parallel $\rightarrow_\varsigma$-reductions in one step. It is defined inductively:

1. $o \Rrightarrow_\varsigma o$,
2. if $o \equiv [l_i = \varsigma(x_i)b_i \ ^{1 \leq i \leq n}]$, then $o.l_j \Rrightarrow_\varsigma b_j\{x_j \mapsto o\}$,
3. if $o \equiv [l_i = \varsigma(x_i)\ b_i \ ^{1 \leq i \leq n}]$, then $(o.l_j \Leftarrow \varsigma(x)\ c) \Rrightarrow_\varsigma [l_j = \varsigma(x)\ c, l_i = \varsigma(x_i)\ b_i \ ^{1 \leq i \leq n, i \neq j}]$,
4. $[l_i = \varsigma(x_i)b_i \ ^{1 \leq i \leq n}] \Rrightarrow_\varsigma [l_i = \varsigma(x_i)b'_i \ ^{1 \leq i \leq n}]$, if $b_i \Rrightarrow_\varsigma b'_i$ $(1 \leq i \leq n)$.
5. $o.l \Rrightarrow_\varsigma o'.l$, if $o \Rrightarrow_\varsigma o'$,
6. $(o.l_j \Leftarrow \varsigma(x)\ c) \Rrightarrow_\varsigma (o'.l_j \Leftarrow \varsigma(x)\ c')$, if $o \Rrightarrow_\varsigma o'$, $c \Rrightarrow_\varsigma c'$,
7. $f(a_1, \ldots, a_n) \Rrightarrow_\varsigma f(a'_1, \ldots, a'_n)$, if $a_i \Rrightarrow_\varsigma a'_i$ $(1 \leq i \leq n)$.

We will prove that $o_2\ {}_\varsigma\!\Lleftarrow o \rightarrow_R o_1$ implies $o_2 \rightarrow_R^* o_3\ {}_\varsigma\!\Lleftarrow o_1$, for some o_3. This allows us to close the diagram $b\ {}_\varsigma^*\!\Lleftarrow o \rightarrow_R^* a$ (by induction on the lengths of the derivations $o \rightarrow_R^* a$ and $o \Rrightarrow_\varsigma^* b$). Note that $\Rrightarrow_\varsigma^* \equiv \rightarrow_\varsigma^*$, which completes the proof.

We proceed by induction on the structure of o, distinguishing cases according to the definition of $o \Rrightarrow_\varsigma o_2$.

1. $o \equiv o_2$. Then we take $o_3 \equiv o_1$.
2. $o \equiv [l_i = \varsigma(x_i)b_i \ ^{1 \leq i \leq n}].l_j$ and $o_2 \equiv b_j\{x_j \mapsto [l_i = \varsigma(x_i)b_i \ ^{1 \leq i \leq n}]\}$. Since R is algebraic, no rewrite rule applies at the root of o, then $b_k \rightarrow_R b'_k$ for some $1 \leq k \leq n$, and $o \rightarrow_R o_1 \equiv [l_i = \varsigma(x_i)b_i \ ^{1 \leq i \leq n, i \neq k}, l_k = b'_k].l_j$. Let b'_j denote either b_j if $j \neq k$, or b'_k if $j = k$. Then $o_1 \Rrightarrow_\varsigma o_3 \equiv b'_j\{x_j \mapsto [l_i = \varsigma(x_i)b_i \ ^{1 \leq i \leq n, i \neq k}, l_k = b'_k]\}$, and $o_2 \rightarrow_R b'_j\{x_j \mapsto [l_i = \varsigma(x_i)b_i \ ^{1 \leq i \leq n}]\} \rightarrow_R^* b'_j\{x_j \mapsto [l_i = \varsigma(x_i)b_i \ ^{1 \leq i \leq n, i \neq k}, l_k = b'_k]\}$ as well, because $o \rightarrow_R o_1$.

3. $o \equiv ([l_i = \varsigma(x_i) b_i{}^{1 \le i \le n}].l_j \Leftarrow \varsigma(x)c)$ and $o_2 \equiv [l_j = \varsigma(x)c, l_i = \varsigma(x_i)b_i{}^{1 \le i \le n, i \ne j}]$. Since R is algebraic, no rewrite rule applies at the root of o, then either $b_k \rightarrow_R b'_k$ for some $1 \le k \le n$, or $c \rightarrow_R c'$. We take $o_3 \equiv [l_j = \varsigma(x)\, c'', l_i = \varsigma(x_i)\, b''_i{}^{1 \le i \le n, i \ne j}]$, where $c'' = c'$ or $c'' = c$, depending on whether there was a reduction on c, and similarly for b''_i. Then $o_2 \rightarrow_R o_3$, and $o_1 \Rrightarrow_\varsigma o_3$.
4. $o \equiv [l_i = \varsigma(x_i) b_i{}^{1 \le i \le n}]$ where $b_i \Rrightarrow_\varsigma b'_i$ $(1 \le i \le n)$, and $o_2 \equiv [l_i = \varsigma(x_i) b'_i{}^{1 \le i \le n}]$. Since R is algebraic, $o_1 \equiv [l_i = \varsigma(x_i) b''_i{}^{1 \le i \le n}]$ where $b_i \rightarrow_R b''_i$ for some i. By induction, there exists b'''_i such that $b'_i \rightarrow^*_R b'''_i \; {}_\varsigma\!\Lleftarrow b''_i$. Then we can take $o_3 \equiv [l_i = \varsigma(x_i) b'''_i{}^{1 \le i \le n}]$.
5. $o \equiv a.l$, where $a \Rrightarrow_\varsigma a'$, and $o_2 \equiv a'.l$. Since R is algebraic, $o_1 \equiv a''.l$, where $a \rightarrow_R a''$. By induction, there exists a''' such that $a' \rightarrow^*_R a''' \; {}_\varsigma\!\Lleftarrow a''$. Then we can take $o_3 \equiv a'''.l$.
6. $o \equiv (a.l_j \Leftarrow \varsigma(x)\, c)$, where $a \Rrightarrow_\varsigma a'$ and $c \Rrightarrow_\varsigma c'$, then $o_2 \equiv (a'.l_j \Leftarrow \varsigma(x)\, c')$. Since no algebraic reduction applies at the root of o, there are two cases:
   - $a \rightarrow_R a_1$ and then $o_1 \equiv (a_1.l_j \Leftarrow \varsigma(x)\, c)$. By induction there exists a_3 such that $a' \rightarrow^*_R a_3 \; {}_\varsigma\!\Lleftarrow a_1$, then we can take $o_3 \equiv (a_3.l_j \Leftarrow \varsigma(x)\, c')$.
   - $c \rightarrow_R c_1$ and then $o_1 \equiv (a.l_j \Leftarrow \varsigma(x)\, c_1)$. By induction there exists c_3 such that $c' \rightarrow^*_R c_3 \; {}_\varsigma\!\Lleftarrow c_1$, then we can take $o_3 \equiv (a'.l_j \Leftarrow \varsigma(x)\, c_3)$.
7. $o \equiv f(a_1, \ldots, a_n)$, where $a_i \Rrightarrow_\varsigma a'_i$ $(1 \le i \le n)$, and $o_2 \equiv f(a'_1, \ldots, a'_n)$. There are two cases:
   - If the algebraic reduction does not take place at the root, then $o_1 \equiv f(a''_1, \ldots, a''_n)$, where for one i, say j, $a_j \rightarrow_R a''_j$, and for the others $a_i \equiv a''_i$. By induction, there exists a'''_j such that $a'_j \rightarrow^*_R a'''_j \; {}_\varsigma\!\Lleftarrow a''_j$. Hence, we take $o_3 \equiv f(a'''_1, \ldots, a'''_n)$, where $a'''_i = a'_i$ for $i \ne j$.
   - If the algebraic reduction takes place at the root, then $o \equiv l\sigma$, for some rewrite rule $l \rightarrow r$, and $o_1 \equiv r\sigma$. The redexes u_i which are reduced in the step $o \Rrightarrow_\varsigma o_2$ must occur under variables x_i in l. These variables are uniquely identified, because l is left-linear. Then we can define a substitution σ' such that $o_2 \equiv l\sigma'$, and $\sigma \Rrightarrow_\varsigma \sigma'$. Then $o_2 \rightarrow_R r\sigma' \; {}_\varsigma\!\Lleftarrow o_1$.

Note the essential use of left-linearity in the last part of the proof.

Theorem 5. *The extended object calculus defined by a set R of left-linear and confluent algebraic rewrite rules is confluent with respect to $\rightarrow_R \cup \rightarrow_\varsigma$.*

Proof. First of all, note that since $\rightarrow_\varsigma$ is confluent for pure objects, it remains confluent in the extension with algebraic symbols (the standard technique of Tait and Martin-Löf can be used). Also, $\rightarrow_R$ is confluent for $O(\mathcal{L}, \mathcal{F}, \mathcal{X})$, by Lemma 3.

By Lemma 4, $\rightarrow_R$ and $\rightarrow_\varsigma$ commute with each other. Hence, using the lemma of Hindley and Rosen [12] which says that if two reduction relations are confluent and commute with each other then their union is confluent, we deduce the confluence of $\rightarrow_R \cup \rightarrow_\varsigma$.

4 Typed Calculus

Abadi and Cardelli defined a typed version of the ς-calculus, developing a first-order type system with object types. In the syntax the main change is that objects come with annotated types, as in $o \equiv [l_i = \varsigma(x_i{:}\,A)\, b_i \ ^{1 \le i \le n}]$, where the associated type is $A \equiv [l_i{:}\, B_i \ ^{1 \le i \le n}]$. We refer to [2] for details. In this section we add to the first-order typed ς-calculus an account for typed algebraic objects.

So far the untyped ς-calculus has been extended with algebraic symbols such as *true*, *false*, *zero*, *succ*, *nil*, etc. The next step is to type them, and to this aim we extend the object types with parametric type constants. With the same philosophy of only considering objects of the calculus to be those algebraic symbols which were applied to all their arguments (see Definition 1), we only want to type algebraic symbols when applied to all their arguments, and only consider types those constants which are given all their parameters. The types are defined with the following syntax

$$\begin{array}{rll} A ::= & [l_i : A \ ^{1 \le i \le n}] & \text{an object type} \\ \mid & C_{A,\cdots,A} & \text{a type indexed constant} \end{array}$$

with $\mathcal{K}$ a set of type constants and $C \in \mathcal{K}$. The set $\mathcal{K}$ contains type constructors, such as a constant *Bool* to type *true* and *false*, and *Nat* to type *zero* and *succ(zero)*, or *Char* to type the letter *a*. In our example of the editor we used lists of characters to represent the text. The situation is slightly more complicated with lists. We would like to have polymorphic lists and yet remain in a first-order calculus. We then declare a parametric constant

$$List :: 1$$

that given a type yields another type. So *List* by itself is not a type, but a type constructor, while $List_{Char}$ is. Then *nil*, which was a constant symbol in the untyped calculus, now needs a type to form a list, in other words, an object that can be typed. Thus, in our environment (which we shall later call Σ) we declare

$$nil :: [1](List_1)$$

indicating that it takes one parameter type as argument and forms a list of that type. Our notation resembles that of command declarations in LaTeX where the number of arguments is declared between square brackets and an ordinal is used to indicate where to use each argument, but we omit the #. Then we can type $nil_{Char} : List_{Char}$. $List_{Char}$ is obtained by $List_1[Char]$ which denotes the substitution of 1 by *Char* in $List_1$.

Similarly, for cartesian products we declare a constructor $\times :: 2$ for which we shall use infix notation. The symbol *pair* is declared as follows:

$$pair :: [2](1, 2 \triangleright 1 \times 2)$$

meaning that *pair* takes two types, an element of the first type, and an element of the second type, and produces an element of the cartesian product of the first and second type. Then, $pair_{A,B}(a, b)$ has type $A \times B$, if a has type A and b has type B, where $A \times B$ is the result of $1 \times 2[A, B]$. Note that in $[2](1, 2 \triangleright 1 \times 2)$ the first 2 is a cardinal while the other numbers are ordinals.

Notation: The expression $A[B_1, \cdots, B_n]$ denotes the simultaneous substitution of $1, \cdots, n$ by $B_1, \cdots, B_n$ in A. We write A for $A_1, \cdots, A_n \triangleright A$ when $n = 0$.

The reader has probably noticed that $[2](1, 2 \triangleright 1 \times 2)$ could be interpreted as $\Lambda X.\Lambda Y.X \rightarrow Y \rightarrow X \times Y$ in System F, but note that we do not need the full power of β-reduction since we only use syntactic substitution (numbers are replaced by expressions containing no numbers).

The calculus we are about to define should be able to detect that $nil.l$ is ill typed. The way it enforces that is by preventing algebraic symbol specifications from containing object types. Hence a declaration

$$nil :: [1][l : B, l_i : B_i\ ^{1 \leq i \leq n}]$$

is not legal. But that doesn't mean that the function hd cannot manipulate lists of objects, for example. This will become clearer once the rules are presented. Take the expression $hd(o{::}l)$ which rewrites to the object o. All we need to do to type it is declare

$$hd :: [1](List_1 \triangleright 1)$$

and instantiate 1 with the corresponding object type in the typing rule for algebraic values (ValAlg) defined below.

The algebraic symbols and the type constructors are declared with a specification in an environment Σ. The environment is reminiscent of that of LF [6].

To formalize declarations in Σ we introduce the notion of *algebraic pre-types.* Intuitively, algebraic pre-types are types, other than object types, which can have natural numbers in them. These numbers behave as place holders and shall be used for syntactic substitution only. For example $List_1$ and 1×2 are pre-types.

We define a set of *algebraic pre-types* with the following grammar:

$$\begin{array}{lll} P ::= \mathbb{N} & & \text{a natural number} \\ \quad\ \mid & C_{\mathbb{N},\cdots,\mathbb{N}} & \text{a parameterised constant} \end{array}$$

Constants (types or algebraic symbols) are declared in Σ with a *specification* of the form:

$$\begin{array}{lll} S ::= \mathbb{N} & & \text{a constructor arity} \\ \quad\ \mid & [\mathbb{N}](P, \cdots, P \triangleright P) & \text{an algebraic symbol specification} \end{array}$$

The sets $\mathcal{K}$ (of type constants), $\mathbb{N}$ (of the natural numbers), $\mathcal{X}$ (of object variables), $\mathcal{L}$ (of object labels), and $\mathcal{F}$ (of algebraic symbols) are pairwise disjoint. Note that $[1](List_1)$ is a specification and not a pre-type.

A type constructor is declared in Σ with specification (or arity) n, written $C :: n$, indicating how many types it needs to be saturated. For example $Bool$ has arity 0 and $List$ has arity 1. An algebraic symbol in $\mathcal{F}$ is declared with a specification $f :: [n](P, \cdots, P \triangleright P)$ indicating its functionality.

4.1 Rules

Typing assumptions for object variables ($x{:}A$) will be given in a context Γ. The expression $\Gamma \vdash_\Sigma \diamond$ means that Γ and Σ are well formed, $\Gamma \vdash_\Sigma A$ means A is a type with respect to Γ and Σ, and $\Gamma \vdash_\Sigma a : A$ means that object a has type A with respect to Γ and Σ. The typing rules are as follows, where we use the notation "$\Gamma \vdash_\Sigma B_i$ for all $1 \le i \le n$" as an abbreviation for n antecedents of the same shape if $n > 0$, and if $n = 0$ for $\Gamma \vdash_\Sigma \diamond$.

$$\frac{\Gamma \vdash_\Sigma B_i \text{ for all } 1 \le i \le n}{\Gamma \vdash_\Sigma [l_i : B_i{}^{1 \le i \le n}]} \qquad (\textsc{TypeObj})$$

$$\frac{\Gamma \vdash_\Sigma A_i \text{ for all } 1 \le i \le n \quad (C :: n) \in \Sigma}{\Gamma \vdash_\Sigma C_{A_1, \cdots, A_n}} \qquad (\textsc{TypeCons})$$

$$\emptyset \vdash_\emptyset \diamond \qquad (\textsc{EnvCtxEmpty})$$

$$\frac{\Gamma \vdash_\Sigma \diamond \quad C \notin dom(\Sigma)}{\Gamma \vdash_{\Sigma, C::n} \diamond} \qquad (\textsc{EnvParType})$$

$$\frac{\begin{array}{l}\Gamma \vdash_\Sigma B_j \text{ for all } 1 \le j \le m \\ \Gamma \vdash_\Sigma P_i[B_1, \cdots, B_m] \text{ for all } 1 \le i \le n \quad f \notin dom(\Sigma)\end{array}}{\Gamma \vdash_{\Sigma, f::[m](P_1, \cdots, P_{n-1} \,\triangleright\, P_n)} \diamond} \qquad (\textsc{EnvAlg})$$

$$\frac{\Gamma \vdash_\Sigma A \quad x \notin dom(\Gamma)}{\Gamma, x : A \vdash_\Sigma \diamond} \qquad (\textsc{CtxVar})$$

$$\frac{\Gamma \vdash_\Sigma \diamond \quad x{:}A \in \Gamma}{\Gamma \vdash_\Sigma x : A} \qquad (\textsc{ValVar})$$

$$\frac{\Gamma, x_i : A \vdash_\Sigma b_i : B_i \text{ for all } 1 \le i \le n \quad (\text{where } A \equiv [l_i : B_i{}^{1 \le i \le n}])}{\Gamma \vdash_\Sigma [l_i = \varsigma(x_i : A)\, b_i{}^{1 \le i \le n}] : A} \qquad (\textsc{ValObj})$$

$$\frac{\Gamma \vdash_\Sigma a : [l_i : B_i{}^{1 \le i \le n}] \quad 1 \le j \le n}{\Gamma \vdash_\Sigma a.l_j : B_j} \qquad (\textsc{ValSel})$$

$$\frac{\Gamma \vdash_\Sigma a : A \quad \Gamma, x : A \vdash_\Sigma b : B_j \quad 1 \le j \le n \quad (\text{where } A \equiv [l_i : B_i{}^{1 \le i \le n}])}{\Gamma \vdash_\Sigma (a.l_j \Leftarrow \varsigma(x : A) b) : A} \qquad (\textsc{ValOv})$$

$$\frac{\begin{array}{l}\Gamma \vdash_\Sigma B_j \text{ for all } 1 \le j \le m \\ \Gamma \vdash_\Sigma a_i : P_i[B_1, \cdots, B_m] \text{ for all } 1 \le i \le n \\ (f :: [m](P_1, \cdots, P_n \triangleright P)) \in \Sigma\end{array}}{\Gamma \vdash_\Sigma f_{B_1, \cdots, B_m}(a_1, \cdots, a_n) : P[B_1, \cdots, B_m]} \qquad (\textsc{ValAlg})$$

In the rule EnvAlg, $B_1, \cdots, B_m$ are arbitrary types (they may be the empty object type, for example), their role is to ensure that if any of the P_i is parametric (i.e. it contains natural numbers), it is well formed in the sense that it yields a proper type when instantiated (See lemma 8.). Observe that natural numbers can only appear in Σ.

4.2 Properties of the Typed Calculus

In this section we highlight some properties of the typed calculus. The omitted proofs are standard and straightforward. We use J to denote arbitrary statements. $\Gamma \subseteq \Delta$ means that if $x{:}A$ is in Γ then it is in Δ.

Lemma 6 Determinism. *Any provable statement, other than $\diamond$, uniquely determines the last rule of its derivation.*

Lemma 7. *If $\Gamma \vdash_\Sigma A$ then A doesn't contain any natural numbers.*

Lemma 8 Substitution. *Given $\Gamma \vdash_\Sigma P[B_1, \cdots, B_n]$ it follows that for all $C_1, \cdots, C_n$ if $\Gamma \vdash_\Sigma C_i$ for every $i \in \{1 \cdots n\}$ then $\Gamma \vdash_\Sigma P[C_1, \cdots, C_n]$.*

Proof. If P has no natural numbers in it, the result is immediate. Otherwise, the proof follows by induction on the derivation of $\Gamma \vdash_\Sigma P[B_1, \cdots, B_n]$, using Determinism 6 and Lemma 7.

Definition 9. An object rewrite rule $l \to r$ is *type preserving* if for every substitution σ, $\Gamma \vdash_\Sigma l\sigma : A$ implies $\Gamma \vdash_\Sigma r\sigma : A$.

The system satisfies the uniqueness of types property (which follows by Determinism 6), the type annotations on constants such as *nil* are needed for this property to hold. Provided the algebraic rewrite rules are type preserving, the system satisfies the subject reduction property as well.

Proposition 10 Uniqueness of Types. *If $\Gamma \vdash_\Sigma a : A$ and $\Gamma \vdash_\Sigma a : A'$, then $A \equiv A'$.*

Lemma 11 Context Lemma.
1. *If $\Gamma \vdash_\Sigma \diamond$ then all variables in Γ are different.*
2. *If $\Gamma \vdash_\Sigma J$ then $\Gamma' \vdash_\Sigma \diamond$ for every prefix Γ' of Γ.*
3. *If $\Gamma_1, x{:}A, \Gamma_2 \vdash_\Sigma J$ then $\Gamma_1 \vdash_\Sigma A$ with a shorter subderivation.*

Lemma 12 Strengthening. *If $\Gamma_1, x{:}A, \Gamma_2 \vdash_\Sigma J$ and x is not free in J then $\Gamma_1, \Gamma_2 \vdash_\Sigma J$.*

If one tries to prove Weakening (Corollary 15) directly by induction on derivations the induction hypothesis is too weak in the case of ValObj. An analogous problem occurs in the lambda calculus for the abstraction rule, and was identified by McKinna and Pollack. We use their idea of *renaming* [8].

Definition 13. Let $\Gamma \vdash_\Sigma \diamond$ and $\Delta \vdash_\Sigma \diamond$. A *renaming* for Γ in Δ is a parallel substitution γ from variables to variables such that for every $x{:}A$ in Γ, $\gamma(x){:}A$ is in Δ. We write $J[\gamma]$ for the result of performing the substitution γ in J. The renaming $\gamma\{x \mapsto y\}$ maps x to y and behaves like γ elsewhere.

Proposition 14 Renaming. *If $\Gamma \vdash_\Sigma J$ and γ is a renaming for Γ in Δ then $\Delta \vdash_\Sigma J[\gamma]$.*

Proof. By induction on the derivation of $\Gamma \vdash_\Sigma J$. Let γ be a renaming for Γ in Δ. Most cases follow easily using the induction hypothesis or the definition of renaming. We consider here the case of ValObj being the last applied rule. Let $z_1, \cdots, z_n \notin dom(\Delta)$. Define n renamings, one for each i, $\gamma_i = \gamma\{x_i \mapsto z_i\}$. To prove that γ_i is a renaming for $\Gamma, x_i{:}A$ in $\Delta, z_i{:}A$ we need to prove that $\Delta, z_i{:}A \vdash_\Sigma \diamond$. By Context Lemma 11(3), there is a subderivation of $\Gamma \vdash_\Sigma A$. Then, by the induction hypothesis, $\Delta \vdash_\Sigma A$, and by CtxVar $\Delta, z_i{:}A \vdash_\Sigma \diamond$. We can now apply the induction hypothesis to obtain $\Delta, z_i{:}A \vdash_\Sigma b_i[\gamma\{x_i \mapsto z_i\}] : B_i$ for all $i \in \{1..n\}$. Then by ValObj $\Gamma \vdash_\Sigma [l_i = \varsigma(z_i : A)\, b_i[\gamma\{x_i \mapsto z_i\}]^{1 \le i \le n}] : A$, and, by the definition of substitution, $\Gamma \vdash_\Sigma [l_i = \varsigma(x_i : A)\, b_i^{1 \le i \le n}][\gamma] : A$.

Weakening now follows as a corollary of renaming taking γ to be the identity substitution.

Corollary 15 Weakening. *If $\Gamma \vdash_\Sigma J$ and $\Gamma \subseteq \Delta$, with $\Delta \vdash_\Sigma \diamond$, then $\Delta \vdash_\Sigma J$.*

Lemma 16 Term Substitution. *If $\Gamma_1, x{:}A, \Gamma_2 \vdash_\Sigma b : B$ and $\Gamma_1 \vdash_\Sigma a : A$ then $\Gamma_1, \Gamma_2 \vdash_\Sigma b\{x \mapsto a\} : B$.*

Proof. By induction on the derivation of $\Gamma_1, x{:}A, \Gamma_2 \vdash_\Sigma b : B$, using Strengthening Lemma 12, Weakening (Corollary 15), and Context Lemma 11(1).

Proposition 17 One-Step Subject Reduction. *Let R be a set of type preserving algebraic rewrite rules. Then if $a \to b$, $\Gamma \vdash_\Sigma a : A$ implies $\Gamma \vdash_\Sigma b : A$*

Proof. By induction on the derivation of $\Gamma \vdash_\Sigma a : A$. We show here two representative cases only.

ValSel Let $a \equiv o.l_j$. We consider the case of an outermost redex first, with $o \equiv [l_i = \varsigma(x_i)\, b_i\ ^{1 \le i \le n}]$, $b \equiv b_j\{x_j \mapsto o\}$, and $A \equiv B_j$. By assumption, $\Gamma \vdash_\Sigma o : [l_i : B_i^{1 \le i \le n}]$ and $1 \le j \le n$. By Determinism, $\Gamma, x_j : [l_i : B_i^{1 \le i \le n}] \vdash_\Sigma b_j : B_j$. By Substitution Lemma 16, $\Gamma \vdash_\Sigma b_j\{x_j \mapsto o\} : B_j$. Otherwise, the reduction is inside o, and the result follows by the induction hypothesis and the rule ValSel.

ValAlg Let $a \equiv f_{B_1, \cdots, B_m}(a_1, \cdots, a_n)$ and $A \equiv P[B_1, \cdots, B_m]$. If there exists a rule $l \to r$ in R and a substitution σ such that $f_{B_1, \cdots, B_m}(a_1, \cdots, a_n) \equiv l\sigma$ and $b \equiv r\sigma$, then because R is type preserving $\Gamma \vdash_\Sigma r\sigma : A$. Otherwise, the reduction is inside some a_i and, by the induction hypothesis and the rule ValAlg, $\Gamma \vdash_\Sigma b : A$.

Proposition 18 Subject Reduction. *Let R be a set of type preserving algebraic rewrite rules. Then if $a \to^* b$, $\Gamma \vdash_\Sigma a : A$ implies $\Gamma \vdash_\Sigma b : A$*

Proof. The proof is by induction on the definition of $\to^*$. The reflexivity case is immediate; the one step reduction case follows from Proposition 17, and the transitivity case follows by the induction hypothesis.

Under the assumption of subject reduction we can also prove the confluence of the typed calculus, provided that R is left-linear and confluent for algebraic objects. The proof is a straightforward adaptation of the one given in Section 3.

4.3 Example: Types for the Editor

The editor presented in Section 2.3 can also be defined in the typed calculus. Assuming that the constant type $Char$ is in Σ, and that we also have a parametric constant $List :: 1$ in Σ, we can define an object type:

$$B \equiv [\mathit{before} : List_{Char}, \mathit{after} : List_{Char}, \mathit{cursor} : Char].$$

The type of the editor, according to the previous rules, and assuming the constant type $String$ is also in Σ, would have the form:

$$\begin{gathered} E \equiv [\mathit{text} : B, \mathit{input} : A_1, \mathit{find} : A_2, \mathit{init} : A_3, \\ \mathit{forward} : A_4, \mathit{backwards} : A_5, \mathit{message} : String] \end{gathered}$$

We can derive the types $A_1, \ldots, A_5$ in E using the typing rules given in the previous section, together with the standard rules for lambda abstraction. We use an environment Σ containing also the constant type $Bool$, the symbol *if then else* with specification $[1](Bool, 1, 1 \triangleright 1)$, and $@ :: [1](List_1, List_1 \triangleright List_1)$, $hd :: [1](List_1 \triangleright 1)$, $tl :: [1](List_1 \triangleright List_1)$, $last :: [1](List_1 \triangleright 1)$, $\mathit{no\text{-}last} :: [1](List_1 \triangleright List_1)$. Γ is a set of assumptions for variables containing $l_1 : List_{Char}$, $l_2 : List_{Char}$.

It is easy to check that the field *text* has type B, and that the types of the other methods are $A_1 \equiv Char \triangleright E$, $A_2 \equiv Char \triangleright E$, $A_3 \equiv E$, $A_4 \equiv E$, $A_5 \equiv E$. Hence we get

$$\begin{aligned} E = [&\mathit{text} : B, \mathit{input} : Char \triangleright E, \mathit{find} : Char \triangleright E, \mathit{init} : E, \\ &\mathit{forward} : E, \mathit{backwards} : E, \mathit{message} : String]. \end{aligned}$$

But this is a recursive equation; to solve it, we need recursive types. This is actually the case each time we want to type objects whose methods return self or an updated self. A *recursive type* $\mu(X)B\{X\}$ where X is a type variable and the notation $B\{X\}$ indicates that X may occur free in B, is the unique solution of the equation $X = B\{X\}$ up to isomorphism. We refer to [2] for a detailed account of recursive types in the ς-calculus. Adding the rules for recursive types to our calculus and the necessary type annotations and folds to the self variables in the editor, we obtain the following type:

$$\begin{aligned} E \equiv \mu(X)\ [&\mathit{text} : B, \mathit{input} : Char \triangleright X, \mathit{find} : Char \triangleright X, \mathit{init} : X, \\ &\mathit{forward} : X, \mathit{backwards} : X, \mathit{message} : String]. \end{aligned}$$

The addition of the fragments Δ_X and Δ_μ, from Abadi and Cardelli's first order object calculus with recursion, and the reduction rules $\mathit{fold}(A, \mathit{unfold}(a)) \to a$ and $\mathit{unfold}(\mathit{fold}(A, b)) \to b$ to our calculus preserves all the properties from Section 4.2.

5 Conclusions

To our knowledge this is the first attempt to combine object calculi with algebraic rewriting. It is well-known that algebraic rewrite rules provide flexibility in the definition of data types. For example, they have been successfully used in combination with the λ-calculus to obtain a uniform system where one can model higher-order functions and data types [3, 7]. Combining the three calculi we inherit the expressiveness of the λ-calculus to manipulate higher-order functions, the versatility of algebraic rewriting to define first-order data types, and the power of the primitives of the ς-calculus to model objects. The λ-calculus can be encoded in the ς-calculus, so we can simply use its encoding; however, the object rewrite rules cannot be encoded and hence the combination presented in this paper is a proper extension of the ς-calculus.

There is another way to add algebraic objects to the calculus, suggested by Martín Abadi in private communication. Instead of adding the set $\mathcal{F}$ of algebraic symbols one could add a set $\mathcal{O}$ of object symbols. An object symbol would just be like a constant added to the grammar, and we would have expressions like $o.l$, where o is an object constant.

In some sense, the two approaches are equally expressive, since we can always translate back and forth:

for f in $\mathcal{F}$, of arity n, invent a symbol o in $\mathcal{O}$, and translate:

$$f(t_1, ..., t_n) = ((...(o.arg_1 := t_1)...).arg_n := t_n).val$$

For example we would need to ensure that if f has arity 4 then $((o.arg_1 := t_1).arg_3 := t_3).val$ translated as $f(t_1 - t_3 -)$ is an illegal expression; also the updating of the val field should be protected. We would also need to add equations or rewrite rules reflecting the behaviour of the constant o as an object, extending $\rightarrow_\varsigma$ so that $(...o.arg_i := t_i...).arg_i = t_i$.

In our approach, we see the algebraic symbols as "constructors of objects", in the same way as the square brackets for instance are constructors of objects in the pure calculus. For example, the natural number 0 is an object, and so is $+(0, 0)$. In this sense, any $f \in \mathcal{F}$ is a constructor of objects, and when the right number of arguments are given, it produces an object. These algebraic objects are like "black boxes", in contrast with the square brackets that build "white boxes" where we can see the contents of the fields. In a language based on our calculus the compiler can make the translation from the algebraic syntax to the one with square brackets and object constants.

There are other possibilities for a type system for our calculus. We can write a type system in Barendregt's cube style where a parametric type is a constant in the context with appropriate arity. For example $LIST : \star \rightarrow \star$. But then we need a rule $(\Box, \Box)$ to be able to show the well-formation of $\star \rightarrow \star$. Although this approach allows us to do what we want, it still allows for more than we intend. For example we can not only form parametric constants, but also abstractions from types to types, in other words the full power of F^ω. Alternatively, we can define the language of kinds with the following grammar: $K ::= \star \mid K \rightarrow K$. But this again allows for more than what we need. For example the kind $(\star \rightarrow \star) \rightarrow \star$ will allow a parametric type to take another parametric type as argument, but we only want a calculus of ground types. Moreover both alternatives allow parametric types and algebraic constants to be partially applied. We then took a third option in which we define the term constructors with specifications in such a way that they can only be typed when applied to all their arguments.

The calculus is extended straightforwardly with the subtyping rules of [2] and the usual subsumption rule for typing. At this stage we do not add subtyping on parametric types, this is a subject of current study and beyond the scope of this paper. We believe that subtyping on constants does not pose any problem provided the set of subtyping axioms on types is transitively closed. Since the subtyping of objects is invariant in its components, the subtyping for parametric types is somewhat orthogonal to the object subtyping. In the example of the lists, if A is a subtype of B one might want to consider $List_A$ a subtype of $List_B$. It remains to be seen if a type inference algorithm in the style of [11] can be defined.

Acknowledgements
We thank Healfdene Goguen for his help in refining earlier versions of the type system, David Aspinall for useful discussions, and Martín Abadi, Ian Mackie, James McKinna, Stefan Kahrs, and three anonymous referees for comments on earlier versions of this paper.

References

1. M. Abadi and L. Cardelli. A theory of primitive objects. Untyped and first-order systems. In *Theoretical Aspects of Computer Software*, pages 296–320. Springer-Verlag, 1994.
2. M. Abadi and L. Cardelli. A theory of primitive objects. Untyped and first-order systems. *Information and Computation*, 125(2):78–102, 1996.
3. Val Breazu-Tannen and Jean Gallier. Polymorphic rewriting conserves algebraic strong normalization. *Theoretical Computer Science*, 83(1), 1991.
4. Val Breazu-Tannen and Jean Gallier. Polymorphic rewriting conserves algebraic confluence. *Information and Computation*, 82:3–28, 1992.
5. A. Gordon and G. Rees. Bisimilarity for a first-order calculus of objects with subtyping. In *Proc. 23rd ACM Symposium on Principles of Programming Languages*, St. Petersburg Beach, Florida, 1996.
6. R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993.
7. J.-P. Jouannaud and M. Okada. Executable higher-order algebraic specification languages. In *Proc. 6th Annual IEEE Symposium on Logic in Computer Science*, pages 350–361, 1991.
8. J. McKinna and R. Pollack. Pure type systems formalized. In J-F Groote and M Bezem, editors, *Proc. of the International Conference on Typed Lambda Calculi and Applications*, volume 664 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.
9. J. Mitchell, F. Honsell, and K. Fisher. A lambda calculus of objects and method specialization. *Nordic Journal of Computing*, 1(1):3–37, 1994.
10. F. Müller. Confluence of the lambda-calculus with left-linear algebraic rewriting. *Information Processing Letters*, 41:293–299, 1992.
11. J. Palsberg. Efficient inference of object types. In *Proc. 9th Annual IEEE Symposium on Logic in Computer Science*, Paris, France, 1994.
12. B. K. Rosen. Tree-manipulating systems and Church-Rosser theorems. *Journal of the ACM*, 20:160–187, 1973.