ELIOS-OBJ theorem proving in a specification language

Isabelle Gnaedig

UNITÉ DE RECHERCHE
INRIA-LORRAINE

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

1992 25ème anniversaire

N° 1668

Programme 2
Calcul Symbolique, Programmation
et Génie logiciel

ELIOS-OBJ
THEOREM PROVING
IN A SPECIFICATION LANGUAGE

Isabelle GNAEDIG

Mai 1992

RR_1668

ELIOS-OBJ
Theorem proving
in a specification language
**
ELIOS-OBJ
Environnement de preuve
dans un langage de spécification

Isabelle Gnaedig
INRIA Lorraine - CRIN CNRS
Technopôle de Nancy-Brabois - BP 101
54600 Villers-lès-Nancy
France
E-mail: gnaedig@loria.fr

Abstract: In the context of the executable specification language *OBJ3*, an order-sorted completion procedure is implemented, providing automatically convergent specifications from user-given ones. This feature is of first importance to ensure unambiguity and termination of the rewriting execution process. We describe here how we specified a modular completion design in terms of inference rules and control language, using *OBJ3* itself. On another hand, the specific problems encountered to integrate a completion process in an already reduction-oriented environment are pointed out.

Résumé: Cet article décrit la conception et la réalisation d'une implantation de la complétion ordo-sortée dans le contexte du langage de spécification exécutable *OBJ3*. Ce processus, permettant de transformer automatiquement un programme donné par l'utilisateur en une forme convergente, est de première importance, car il assure la non ambiguité et la terminaison des calculs du programme à l'execution. La complétion a été spécifiée de façon modulaire, en termes de règles d'inférence et d'un langage de contrôle, dans le langage *OBJ3* lui-même. D'autre part, ce travail a permis l'étude de problèmes spécifiques à l'intégration d'un mécanisme de preuve, en l'occurence la complétion, dans un environnement de programmation.

ELIOS-OBJ
Theorem proving
in a specification language

I. Gnaedig
INRIA Lorraine - CRIN CNRS
Technopôle de Nancy-Brabois - BP 101
54600 Villers-lès-Nancy
France
E-mail: gnaedig@loria.fr

April 3, 1992

Abstract

In the context of the executable specification language *OBJ3*, an order-sorted completion procedure is implemented, providing automatically convergent specifications from user-given ones. This feature is of first importance to ensure unambiguity and termination of the rewriting execution process. We describe here how we specified a modular completion design in terms of inference rules and control language, using *OBJ3* itself. On another hand, the specific problems encountered to integrate a completion process in an already reduction-oriented environment are pointed out.

1 Introduction

OBJ3 is a programming language based on equational logic: programs are given in terms of abstract data types and their semantics relies on order sorted algebras, which enables inclusion of types. The problem approached here is the correctness of axiom sets, in the following sense. The operational semantics of *OBJ3* is rewriting, which means that when a program is executed on a given value, the set of axioms is interpreted and used as a set of rewrite rules that reduces the value to its normal form. We have to establish whether computations are correct with respect to validity in initial models, whether results are unique and - last but not least - whether computation always terminates.

The completion process of a rewrite rule set is able to ensure the previous requirements. Starting from any axiom set, it provides, when it succeeds (this is a semi-decidable problem), an equivalent set of rules with the same deduction power, confluent (the result of rewriting an expression does not depend of the way the rules are applied: it is unambiguous), and terminating (there is no infinite rewrite chain). Hence, it can be seen as an automatic prover of program correctness. We intend here to design and implement an integrated programming environment, named *ELIOS-OBJ*, allowing programming and proving in the same context.

Our goal here has three aspects: to provide the user with a tool for proving correctness of specifications in the context of *OBJ3*, namely with an order-sorted semantics; to propose an implementation of order-sorted completion described and proved in [6]; and to point out some problems arising in integrating theorem proving aspects (completion here) with programming aspects (the *OBJ3* language).

We describe completion in a high-level formalism allowing modularity and general expressiveness, using inference rules. This approach, first presented for completion in [1], has also be implemented in [13]. A control language is proposed, to combine these inference rules in any kind of strategy. In this formalism, completion can be considered as a particular instance of general deduction mechanisms, described by inference rules and control, as equational proofs, inductive proofs, equation solving by unification, disunification... The specification of our implementation is given in *OBJ3* itself.

The main definitions and the algebraic context are given in Section 2. Section 3 recalls results on order sorted completion established in [5, 6], expressed in terms of inference rules and control. Section 4 describes the control language for expressing and implementing different completion strategies. Section 5 presents the features of the orientation engine, transforming axioms into rules. Section 6 presents an investigation towards the integration of a theorem prover into a programming context. It points out the technical problems encountered in implementing completion in *OBJ3*. They are mainly due to the fact that features for an efficient rewrite engine (as integrated in *OBJ3*) are not necessarily compatible with completion mechanisms. Some examples of completion in *ELIOS-OBJ* are given and illustrated in Section 7.

2 Order-Sorted Algebra

In this section the basic notions about order-sorted algebras are shortly summarized [7].

Given an index set S, an S-sorted set A is just a family of sets, one for each $s \in S$; we will write $\{A_s | s \in S\}$. Similarly, given two S-sorted sets A and B, an S-sorted function $\alpha : A \to B$ is an S-indexed family $\alpha = \{\alpha_s : A_s \to B_s | s \in S\}$. Assume a fixed partially ordered set $(S, \leq)$, called the **sort set**.

An **order-sorted signature** is a triple $(S, \leq, \Sigma)$ where S is a sort set, Σ is an $S^* \times S$ -indexed family $\{\Sigma_{w,s} | w \in S^*, s \in S\}$, and $(S, \leq)$ is a partially ordered set. Elements of Σ are called operators. When the sort set S is clear, we write Σ for (S, Σ). Similarly, when the partially ordered set $(S, \leq)$ is clear, we write Σ for $(S, \leq, \Sigma)$. For operators, we write $f : w \to s$ for $f \in \Sigma_{w,s}$. We say that the rank of f is $w \to s$. An important special case is when w is λ, the empty string; then $f \in \Sigma_{\lambda,s}$ denotes a constant of sort s. Note that the ordering $\leq$ on S extends to strings of the same length in S^* by $s_1 ... s_n \leq s'_1 ... s'_n$ iff $s_i \leq s'_i$ for $i = 1, ..., n$; similarly, $\leq$ extends to pairs $(w, s) \in S^* \times S$ by $(w, s) \leq (w', s')$ iff $w \leq w'$ and $s \leq s'$.

Let $(S, \leq, \Sigma)$ be an order-sorted signature. A $(S, \leq, \Sigma)$-**algebra** A consists of a family $\{A_s | s \in S\}$ of subsets of A, called the **carriers** of A, and a function $f_A : A_w \to A_s$ for each $f \in \Sigma_{w,s}$ where $A_w = A_{s_1} \times ... \times A_{s_n}$ when $w = s_1 ... s_n$ and A_w is a one point set when $w = \lambda$, such that:

1. $s \leq s'$ in S implies $A_s \subseteq A_{s'}$ and
2. $f \in \Sigma_{w,s} \cap \Sigma_{w',s'}$ with $s' \leq s$ and $w' \leq w$ implies $f_A : A_w \to A_s$ equals $f_A : A_{w'} \to A_{s'}$ on $A_{w'}$.

Following [7], we define the order-sorted Σ-term algebra $\mathcal{T}_\Sigma$ as the least family $\{\mathcal{T}_{\Sigma,s} | s \in S\}$ of sets satisfying the following conditions:

- $\Sigma_{\lambda,s} \subseteq \mathcal{T}_{\Sigma,s}$ for $s \in S$;
- $\mathcal{T}_{\Sigma,s'} \subseteq \mathcal{T}_{\Sigma,s}$ if $s' \leq s$;
- if $f \in \Sigma_{w,s}$ with $w = s_1 ... s_n \neq \lambda$ and if $t_i \in \mathcal{T}_{\Sigma,s_i}$ then (the string) $f(t_1 ... t_n) \in \mathcal{T}_{\Sigma,s}$.

- for $f \in \Sigma_{w,s}$, let $\mathcal{T}_f : \mathcal{T}_w \to \mathcal{T}_s$ map $t_1, ..., t_n$ to (the string) $f(t_1...t_n)$.

Following [9], we denote by $\mathcal{D}(t)$ the set of occurrences of t i.e. the domain of the term t viewed as a partial function from $\mathcal{N}^*$ to Σ. We denote by $t_{|\omega}$ the subterm of t at occurrence ω and by $t[\omega \leftarrow t']$ the result of the replacement by t' of $t_{|\omega}$ for $\omega \in \mathcal{D}(t)$. Clearly $\mathcal{T}_\Sigma$ is an order-sorted Σ-algebra.

We restrict to the class of regular signatures. Essentially, regularity asserts that overloaded operations are consistent under restriction to subsorts, so that each well-formed expression on the function symbols has a least sort. An order-sorted signature Σ is **regular** iff for any $w_0 \in S^*$ such that there is a $f \in \Sigma_{w,s}$ with $w_0 \leq w$, then there is a least $(w', s') \in S^* \times S$ such that $f \in \Sigma_{w',s'}$ and $w_0 \leq w'$. Under that condition, $\mathcal{T}_\Sigma$ is an initial order-sorted Σ-algebra [7]. In this case, for any $t \in \mathcal{T}_\Sigma$, there is a least $s \in S$, called **lowest sort** of t and denoted $LS(t)$.

2.1 Equations and rewrite rules

An S-**sorted variable set** is an S-indexed family $X = \{X_s | s \in S\}$ of disjoint sets. A variable x of sort s is also denoted $(x : s)$. Given an order-sorted signature $(S, \leq, \Sigma)$ and a variable set X that is disjoint from Σ, $(S, \leq, \Sigma(X))$ is defined by $\Sigma(X)_{\lambda,s} = \Sigma_{\lambda,s} \cup X_s$ and $\Sigma(X)_{w,s} = \Sigma_{w,s}$ for $w \neq \lambda$.

Note that if Σ is regular, so is $\Sigma(X)$. We can now form $\mathcal{T}_{\Sigma(X)}$ and then view it as a Σ-algebra; let us denote this Σ-algebra by $\mathcal{T}_\Sigma(X)$. It is the **free** Σ-algebra generated by X. $\mathcal{V}(t)$ denotes the set of variables of the term t.

To get an adequate notion of satisfaction (see [11]), an additional hypothesis on the set of sorts S must be satisfied: An order-sorted signature $(S, \leq, \Sigma)$ is **coherent** iff each of the connected components of S for $\leq$ (i.e. each equivalence class under the transitive symmetric closure of $\leq$) has a maximum, and Σ is regular. We will only consider here order-sorted signatures that are coherent.

A Σ-**equation** is a triple (X, t, t') where X is a variable set and $t, t' \in \mathcal{T}_\Sigma(X)$ with $LS(t)$ and $LS(t')$ in the same connected component of $(S, \leq)$. We will use the notation $((\forall X)t = t')$.

2.2 Order-sorted rewriting

For $(S, \leq, \Sigma)$ a coherent order-sorted signature and X, Y two S-sorted variable sets, a **substitution** is an S-sorted function $\sigma : X \to \mathcal{T}_\Sigma(Y)$, extended in a unique way to $\sigma : \mathcal{T}_\Sigma(X) \to \mathcal{T}_\Sigma(Y)$.

Operationally, order-sorted equations are used as rewrite rules. An order-sorted **rewrite rule** is an order-sorted equation $((\forall X)l = r)$ satisfying $\mathcal{V}(r) \subseteq \mathcal{V}(l)$ and denoted $((\forall X)l \to r)$. A **match** from a term $t \in \mathcal{T}_\Sigma(X)$ to a term $t' \in \mathcal{T}_\Sigma(Y)$ is a substitution σ such that $\sigma(t) = t'$.

Let R be a set of rewrite rules. A term $t \in \mathcal{T}_\Sigma(Y)$ rewrites to t' with a rewrite rule $((\forall X)l \to r)$ in R at occurrence ω, which is denoted $t \to_Y^R t' = t[\omega \leftarrow \sigma(r)]$ whenever

1. there is a match $\sigma : X \to \mathcal{T}_\Sigma(Y)$ from l to t at occurrence ω $(\sigma(l) = t_{|\omega})$
2. there is a sort s such that, for x a variable of sort s, $t[\omega \leftarrow x]$ is a well-formed term and $\sigma(l), \sigma(r) \in \mathcal{T}_{\Sigma,s}(Y)$.

The difficulty is that $\sigma(l)$ and $\sigma(r)$ may have different sorts, and the second condition in the previous definition is needed to avoid that replacements produce ill-formed terms.

We define $\xrightarrow{*}{}_Y^R$ to be the reflexive transitive closure of $\to_Y^R$ and $\longleftrightarrow *_Y^R$ to be its symmetric, reflexive and transitive closure. This last equivalence relation is called **order-sorted replacement of equals by equals**. For the notion of *order-sorted* replacement of equals by

equals to be correct and complete with respect to order-sorted deduction, the rewriting relation has to be confluent and sort-decreasing [11].

An order-sorted term rewriting system R is **sort-decreasing** iff $\forall t, t' \in \mathcal{T}_\Sigma(Y)$, $t \rightarrow_Y^R t'$ implies $LS(t) \geq LS(t')$.

In order to give decidable criteria for this property to hold, we need the notion of specialization. A sorted set of variables X can be viewed as a pair $(\bar{X}, \mu)$ where $\bar{X}$ is a set of variable names (i.e. unsorted variables) and μ, the sort assignment, maps the variable names to the set of sorts $\mu : \bar{X} \rightarrow S$. The ordering $\leq$ on S is extended to sort assignments by

$$\mu \leq \mu' \Leftrightarrow \forall x \in \bar{X}, \mu(x) \leq \mu'(x)$$

We then say that μ' **specializes** to μ via the substitution $\rho : (x : \mu'(x)) \rightarrow (x : \mu(x))$ called a **specialization** of $X = (\bar{X}, \mu')$ into $\rho(X) = (\bar{X}, \mu)$.

The notion is then extended to equations and rewrite rules. A specialization of an equation $(\forall X)(l = r)$ is another equation $(\forall \rho(X))(\rho(l) = \rho(r))$ where ρ is a specialization of X. A specialization of a rule $(\forall X)(l \rightarrow r)$ is the rule $(\forall \rho(X))(\rho(l) \rightarrow \rho(r))$ where ρ is a specialization of X.

If the set of sorts is finite, or if each sort has only a finite number of sorts below it, a finite sorted set of variables has a finite number of specializations. This allows deciding the sort-decreasing property. A set of rules R is **sort-decreasing** iff any rule of R is **sort-decreasing**, that is iff for any rule $((\forall X)l \rightarrow r)$ of R, for any specialization ρ of X, the lowest sort of $\rho(l)$ is greater or equal than the lowest sort of $\rho(r)$. An order-sorted term rewriting system R is sort-decreasing if R is a sort-decreasing set of rules.

The definitions R being **confluent** are similar to the unsorted case. Let R be an order-sorted term rewriting system. R is **confluent** iff for any terms $t, t', t'' \in \mathcal{T}_\Sigma(Y)$, $t \xrightarrow{*}_Y^R t'$ and $t \xrightarrow{*}_Y^R t''$ implies there exists t_0 such that $t' \xrightarrow{*}_Y^R t_0$ and $t'' \xrightarrow{*}_Y^R t_0$. R is **Church-Rosser** iff for any terms $t, t' \in \mathcal{T}_\Sigma(Y)$, $t \leftrightarrow *_Y^R t'$ implies there exists t_0 such that $t \xrightarrow{*}_Y^R t_0$ and $t' \xrightarrow{*}_Y^R t_0$.

When the variable set Y can be deduced from the context, we allow it to be omitted and we write $t \longrightarrow^R t'$ for $t \longrightarrow_Y^R t'$.

2.3 Critical pairs

Two reductions applied to a same term can sometimes overlap, yielding critical pairs. Let $\mathcal{G}(t)$ be the set of occurrences ω in t such that the subterm of t at occurrence ω is not a variable. A unifier of two terms t and t' is a substitution σ such that $\sigma t = \sigma t'$.

A non-variable term t' and a term t **overlap** at occurrence ω in $\mathcal{G}(t)$ with a substitution σ iff σ is a unifier of $t_{|\omega}$ and t'.

Given two rules $g \rightarrow d$ and $l \rightarrow r$ such that $\mathcal{V}(g) \cap \mathcal{V}(l) = \emptyset$ and l and g overlap at occurrence ω of $\mathcal{G}(g)$ with the substitution σ, then the pair $(p = \sigma(g[\omega \leftarrow r]), q = \sigma(d))$ is called a **critical pair** of the rule $l \rightarrow r$ on the rule $g \rightarrow d$ at occurrence ω (a trivial one if $\omega = \epsilon, l = g, r = d$).

3 Completion in order sorted algebras

In [6] we describe the completion process in order-sorted algebras by a set of inference rules. Although [6] presents results on equational completion, we implement here completion in the empty theory. This more simple process enables us to focus on interaction and interface between the completion and the language, instead of problems specific to completion.

Recall that an ordering on terms is compatible (with the term structure) if $s \succ t$ implies $f(...s...) \succ f(...t...)$, for all terms s, t and all contexts $f(... ...)$ such that $f(...s...)$ and $f(...t...)$

are well-formed. A reduction ordering is a well-founded and compatible ordering. A reduction ordering warrants termination of a rewriting system R, if $\sigma l \succ \sigma r$ for each rule $l \to r$ of R, and every substitution σ [2].

Let P be a set of equations and $\succ$ a reduction ordering. As pointed out previously, the signature of the algebra, P is defined on, has to be coherent. The *completion procedure* transforms, if possible, P into a confluent and terminating set of rules R, having the same deduction power. This transformation can be described by a derivation chain of the form $(P^0, R^0) \vdash (P^1, R^1) ... \vdash (P^n, R^n) ... \vdash (P^\infty, R^\infty)$ (that may be constant from a given rank n). The completion transformation is based on the well known basic mechanisms: orienting an axiom of P into a terminating and sort-decreasing rule, adding equational consequences named critical pairs, and reducing the left-hand-side and the right-hand-side of axioms and rules. Order-sorted completion can be expressed by the following inference rules:

1. Orienting an equation
$$\frac{P \cup \{s = t\}, R}{P, R \cup \{s \to t\}} \quad \text{if } s \succ t \ \& \ s \to t \text{ sort} - \text{decreasing}$$

2. Adding a critical pair
$$\frac{P, R}{P \cup \{s = t\}, R} \quad \text{if } u \to^R s \ \& \ u \to^R t$$

3. Simplifying an equation
$$\frac{P \cup \{s = t\}, R}{P \cup \{u = t\}, R} \quad \text{if } s \to^R u$$

4. Deleting an equation
$$\frac{P \cup \{s = s\}, R}{P, R}$$

5. Simplifying the right-hand side of a rule
$$\frac{P, R \cup \{s \to t\}}{P, R \cup \{s \to u\}} \quad \text{if } t \to^R u$$

6. Simplifying the left-hand side of a rule
$$\frac{P, R \cup \{s \to t\}}{P \cup \{u = t\}, R} \quad \text{if } s \to^R_{l \to r} u \ \& \ s \rhd l$$

where $\rhd$ is the proper specialization ordering , defined by $s \rhd l$ iff $\exists \sigma, \sigma(l) = s_{|\omega}$ with $\omega \neq \epsilon$, or $\sigma(l) = s$ and σ is not a renaming.

With respect to the completion procedure for unsorted rewriting described in [1], the modifications for the order-sorted completion are localized in the conditions of the first inference rule, where the sort-decreasing test for a new rule must be performed. Remark that the sort-decreasing test in the rule 5 is not needed since $s \to t$ and $t \to u$ sort-decreasing implies $s \to u$ sort-decreasing (by definition of a sort-decreasing rule and by transitivity of the ordering $\leq$ on sorts).

Furthermore, except for the sort-decreasing test, it appears that the specific sort problems are hidden in the definition of rewriting and critical pairs: order-sorted matching, order-sorted unification.

4 Specifying control for completion

The concept of inference rules previously chosen for describing the theoretical aspects of completion, will be completed by a control mechanism on these inference rules, to give particular completion procedures. Control specifies the order in which inference rules are applied. As required in [6], the control has to be fair, which means that all critical pairs of the resulting set of rules R^∞ have to be computed, and the resulting set of pairs P^∞ has to be empty. If the completion is fair and does not fail, then R^∞ is Church-Rosser and sort-decreasing.

Here is developed a simple control language, aimed at expressing any "combination" of inference rules and at providing effective completion procedures. Note that this control language is general enough to be applied on any activity described with inference rules (and on any working universe named UNIVERSE); it is not specific to completion since it doesn't depend on the inference rules themselves.

The language used for describing control is *OBJ3* itself. So, we attempt in the same time to test expressiveness of *OBJ3* for describing an already complex problem. The effective implementation is made in Kyoto Common Lisp, like *OBJ3*.

Let us start from the definition of the specific completion universe. The working domain COMP-UNIVERSE is a pair of sets: the set P of axioms to be oriented and the set R of current rules generated by completion.

Note that the subsort mechanism of $OBJ3$ allows an elegant "error-handling" feature used here when completion fails on a given universe: we just have to define a sort *Comp-universe-with-failure* including the sort *Comp-universe*. Hence, when finishing with success, the completion procedure gives a pair (E, R) of sort *Comp-universe*. When instead failing, it returns an error result of sort *Comp-universe-with-failure*. The modules BOOL, PAIR, PAIRS and RULES used in the following specify respectively the booleans, a pair, a set of pairs and a set of rules. They are not detailed here.

```
obj COMP-UNIVERSE is

protecting  PAIRS, RULES .
sorts Comp-universe Comp-universe-with-failure .
subsorts Comp-universe < Comp-universe-with-failure .
op <_,_> : Pairs Rules -> Comp-universe .
op P : Comp-universe -> Pairs .
op R : Comp-universe -> Rules .
var p : Pairs .
var r : Rules .

eq P(<p,r>) = p .
eq R(<p,r>) = r .

jbo
```

The *protecting* feature is a mechanism for importing modules in *OBJ3* (to become familiar with the *OBJ3* syntax, read [8]).

Let us now define strategies by a control on inference rules. The completion can then be seen as the application of a chosen strategy (available for instance in a strategy library) on a completion universe. This is a simple way to describe concisely a completion strategy, looking it as independent of the data structures (axioms and rules), it is working on. In other words, the

strategy (specified in STRAT-COMP) is a constant with respect to the completion universe U, as specified below:

```
obj COMPLETION is

protecting STRAT-COMP .
protecting COMP-UNIVERSE .
op completion : Comp-universe Strat-comp -> Comp-universe .
var U : Comp-universe .
var s : Strat-comp .

eq : completion(U,s) = apply(s,U) .

jbo
```

We now have to define how the *apply* operation works on a strategy and a universe, introducing here our control language on inference rules. It is expressed in the module STRAT by the classical basic instructions of imperative programming languages: a test *if-then-else*, a loop *while-do*, a sequence operator *concat*, an iterator *iter*. This module is parametrized by the modules TE, expressing the notion of test, IT, expressing the notion of iterator, and INF, defining inference rules written $< U, U', C >$.

```
obj STRAT[TE : TEST, IT : ITERATOR, INF : INFERENCE-RULE] is

protecting UNIVERSE BOOL .
sort Strategy .
op empty-strategy : -> Strategy .
op while_do_ : Test Strategy -> Strategy .
op if-then-else : Bool Strategy Strategy -> Strategy .
op apply : Strategy Universe -> Universe .
op _concat_ : Strategy Strategy -> Strategy .
op iter : Strategy Iterator -> Strategy .
vars C : Bool .
vars S S' : Strategy .
vars U U' : Universe .
var I : Iterator .
var B : Test .

eq apply(empty-strategy, U) = U .

eq apply(<U,U',C>) = if C then U' else U .

eq apply(while B do S,U) = if test-apply(B, U)
                           then apply(while B do S,
                                      apply(S,U))
                           else U .

eq apply((if-then-else(B,S,S'),U) = if test-apply(B, U)
                                    then apply(S,U)
                                    else apply(S',U) .
```

```
eq apply((S concat S'),U) = apply(S',apply(S,U)) .

eq apply(iter(S, I), U) = if iter-apply(I, U) diff error
                          then apply(iter(S, I),
                                     iter-apply(I, apply(S, U)))
                          else U .
jbo .
```

The operation *apply* is itself defined through more specific operations: *test-apply* and *iter-apply*, working respectively on tests and iterators defined in the following two specifications, for the completion case.

```
obj COMP-TEST is

Protecting COMP-UNIVERSE, BOOL, PAIRS, PAIR, RULES .
Sort Comp-Test .
op non-empty-set-of-pairs : -> Comp-Test .
op is-trivial-pair : -> Comp-Test .
op orientable-pair : -> Comp-Test .
op non-empty-critical-pairs : -> Comp-Test .
op test-apply : Comp-Test Comp-Universe -> Bool .
var U : Comp-universe .

eq test-apply(non-empty-set-of-pairs, U) = non-empty(P(U)) .
eq test-apply(is-trivial-pair, U) = is-trivial(current(P(U))) .
eq test-apply(orientable-pair, U) = is-orientable(current(P(U)))
eq test-apply(non-empty-critical-pairs, U) = non-empty(critical-pairs(R(U))) .

jbo .

obj COMP-ITERATOR is

Protecting COMP-UNIVERSE PAIRS RULES .
Sort Comp-Iterator .
op for-each-pair : -> Comp-Iterator .
op for-each-rule : -> Comp-Iterator .
op iter-apply : Comp-Iterator Comp-Universe -> Comp-Universe .
var U : Comp-universe .

eq iter-apply(for-each-pair, U) = <increment-on-set((P(U)), R(U)> .
eq iter-apply(for-each-rule, U) = <P(U), increment-on-set(R(U))> .

jbo .
```

Note that the operations used to define *test-apply* and *iter-apply* are working at lower level than the previous ones, and on data structures, that we will not specify here: *non-empty* is a test of non-emptyness of a set, *current* gives the current element of a set, *is-trivial* and *is-orientable* are working on a pair of terms, *increment-on-set* manages the access to elements of a set, and *critical-pairs* gives the critical pairs of a set of rules.

Then the specification of a strategy for completion can be designed, using the the module STRAT instantiated by the parameters COMP-TEST, COMP-ITERATOR and COMP-INFERENCE-RULE.

The last parameter COMP-INFERENCE-RULE defines inference rules for completion, adapted from those of Section 3: *normalize-lhs-pair*, *normalize-rhs-pair*, *delete-pair*, *orient-pair-l-to-r*, *orient-pair-r-to-l*, *normalize-rhs-rule*, *simplify-lhs-rule*, *failure-inf-rule*, *add-critical-pairs*.

Remark that, in order to be used in an operational way, the first inference rule *orienting an equation* has been splitted in *orient-pair-l-to-r* and *orient-pair-r-to-l*, the third rule *simplifying an equation* has been splitted in *normalize-lhs-pair* and *normalize-rhs-pair*. Note also that the orientation failure case is handled by a new inference rule: *failure-inf-rule*.

```
obj STRAT-COMP

Protecting STRAT[COMP-TEST, COMP-ITERATOR, COMP-INFERENCE-RULE] .
Sort Strat-comp .
op strat-simple : -> Strat-comp .

eq strat-simple = while non-empty-critical-pairs do

                  while non-empty-set-of-pairs do
                   normalize-lhs-pair concat
                   normalize-rhs-pair concat
                  if is-trivial-pair
                  then  delete-pair
                  else if orientable-pair
                       then  orient-pair-l-to-r concat
                             orient-pair-r-to-l concat
                             iter( normalize-rhs-rule concat
                                   simplify-lhs-rule,
                                 for-each-rule)
                       else  failure-inf-rule
                       endif
                  endif
                  end while concat
                  add-critical-pairs

                  end while .

jbo .
```

5 The orientation engine

The test *orientable-pair* includes the complete mechanism for orienting an axiom in order sorted completion, depending on two criteria. First, the axiom has to be oriented in a sort-decreasing way. Second, it has to be oriented according to some reduction ordering, to ensure termination of the computed rewrite system.

A decidable criterion for sort-decreasingness is given through the notion of specialization (see Section 2). For implementing the sort-decreasing test, the rule specialization computing

algorithm, already existing in *OBJ3* for defining the rewrite engine [8], is used.

To handle the termination problem, an usual simplification ordering is chosen: the left-to-right lexicographical path ordering (*LPO* in short) [10]. This ordering is based on a basic ordering on the set F of symbols of the signature: the precedence (denoted by $>_F$). In our system, like in *REVE* [12, 3], the precedence is empty when the completion starts, and incrementally enriched by interaction of the user, as new rules are oriented.

Termination of rewriting in an order sorted algebra can be proved without considering the sort information of operators and terms in the algebra. If a rewriting relation terminates in the homogeneous algebra, then it terminates in the corresponding order sorted algebra. Therefore, the *LPO* is used, where the precedence, namely the ordering on operators, does not take into account the sort information on operators, i.e. their rank. Let us give an example.

```
obj  PRECEDENCE is
sorts Nat Int .
subsorts Nat < Int .
op _+_ : Nat Nat -> Nat .
op _+_ : Int Int -> Int .
op - : Int -> Int .
vars x y : Nat .
eq -(x + y) = (-x) + (-y) .
eq -(-(x + y)) = x + y .
jbo
```

Let us try to orient both axioms into rewrite rules. For ensuring termination, we have to prove $-(x + y) >_{LPO} (-x) + (-y)$. Let us note, the top symbol of the right hand side is "$+ : Int\ Int \rightarrow Int$". The unequality is true if we assume $- >_F +$. We do not precise in the precedence, what "+" of the signature, we are handling with. That means that "$-$" is greater than any "+" of the signature. For the second axiom to be oriented, the previous precedence hypothesis can then be used, although the "+" operator is "$+ : Nat\ Nat \rightarrow Nat$" in both sides of the axiom.

The choice made in our orientation engine is to treat sort-decreasingness, before termination. If an axiom is not sort-decreasing, the user can reverse it or orient it by hand (in this last case, correction of rewriting is not warranted, see Section 2). If however a given axiom is not orientable for the current *LPO*, he can backtrack for chosing another precedence, before trying to reverse it. The structure of our orientation engine, hidden in the test *orientable-pair* (already presented in the STRAT-COMP module) looks like:

```
obj ORIENTATION-ENGINE is

Protecting PAIR .
op orientable-pair : Pair -> Bool .
op orientable-pair-l-to-r : Pair -> Bool .
op orientable-pair-r-to-l : Pair -> Bool .
Var p : Pair .

eq orientable-pair (p) = if orientable-pair-l-to-r (p)
                         then true
                         else if orientable-pair-r-to-l (p)
                              then true
                              else false .
```

```
eq orientable-pair-l-to-r (p) = if sort-decreasing (p)
                                then if is-LPO-oriented (p)
                                     then true
                                     else false
                                else false .

eq orientable-pair-r-to-l (p) = orientable-pair-l-to-r (reverse(p)) .

jbo
```

where the *reverse* operation transforms an equality $g = d$ into its symmetrical equality $d = g$, and *is-LPO-oriented* is the orientation test using the LPO.

This algorithm is used interactively, each time a pair can be oriented into a rule. But the orientation can also be "forced" "by hand", or the pair can be postponed in the current set of axioms, or simply completion can be interrupted and the original set of axioms restored.

6 The technical problems of integrating completion in OBJ3

The goal of our work was to develop an order-sorted completion algorithm interfaced with $OBJ3$. In order to rewrite a minimum amount of code and to have an integrated design of completion in the language, we wanted to reuse already existing tools like the matching algorithm, the rewrite engine, the specialization algorithm. For a given OBJ specification, we have also chosen to perform completion on the internal $OBJ3$ form of the axiom set obtained after compilation of the given specification. Completion works directly on the same structures as the rewrite engine; it modifies them to give directly the compiled OBJ module, corresponding to the completed set of axioms.

Recall that one goal of this work was to discover and solve the problems of integrating a theorem prover in a programming language interpreter. As known, execution of programs and theorem proving have very different requirements and the design of the first is not easily compatible with the requirements of the second.

For instance, an operation like reduction of terms is used in two different ways, first in the reduction process of the OBJ language, where it requires efficiency for applying the rules, second in the simplification mechanism of the completion procedure, where instead it requires efficiency for updating the set of rules. The problems encountered during the integration of a completion procedure in *OBJ3* are now enumerated, and the proposed choices and compromises are explained.

6.1 Adapting *OBJ3* specifications to completion context

OBJ3 provides a mechanism of *order-sorted conditional rewriting*. Until now, however, order-sorted completion as presented in [6] only works on unconditional rules. A filter has been implemented, for transforming conditional specifications into unconditional ones, by discarding conditional axioms, before a completion is started on an OBJ module.

OBJ3 allows *manipulating booleans*, in any user-defined module, adding implicitly an importation of an OBJ predefined module for booleans containing associative-commutative operators like and, or, xor (exclusive or) [8]. Since our completion doesn't handle associative-commutative axioms, and since this predefined module is already complete, we suppress this systematic importation. Instead, booleans are explicitly imported by the user, only if desired.

Let us also take into account the general *importation mechanism*. For the evaluation of a term in an *OBJ3* module, all axioms of all recursively imported modules are used for rewriting. Let us give an *OBJ3* example.

```
obj NAT_S is
sort Nat_s .
op 0 : -> Nat_s .
op s : Nat_s -> Nat_s .
var x, y : Nat_s .
eq  0 + x = x .                          (1)
eq  s(x) + y = s(x + y) .                (2)
jbo

obj FIBO_S is
protecting NAT_S .
op _+_ : Nat_s Nat_s -> Nat_s .
op fib : Nat_s -> Nat_s .
var x, y : Nat_s .
eq  fib(0) = 0 .                         (3)
eq  fib(s(0)) = s(0) .                   (4)
eq  fib(s(s(x))) = (fib(x) + fib(s(x))) .  (5)
jbo
```

In the previous specification, the definition of the Fibonacci function is built on the integers defined in NAT_S. So the rewrite mechanism of *OBJ3* uses the axioms of NAT_S (interpreted as rules) in addition to those of FIBO_S, to reduce terms of FIBO_S. For example, the term $fib(0+0)$ defined in FIBO_S is reduced in $fib(0)$ using (1), before $fib(0)$ is reduced in 0 using (3).

The first possible alternative for completion would be to collect recursively all imported axioms, adding them to those of the current module, and completing the resulting set. This process causes problems with respect to the design of *OBJ3*. Any rule is attached to the module, in which the top operator of left hand side has been declared. Provided new rules may appear during completion, the question is to which module they must be attached. Theorem proving would be involved to prove that the definitions of modules are not changed. Moreover, this mechanism of putting together all the imported axioms for completion would be a contradiction to the modularity of *OBJ3*. Our completion procedure thus applies on axioms of a module in a modular way, without considering the eventually imported rules.

This fact emphasizes the difficulties for integrating a proof mechanism in a modular programming language, and suggests to investigate a theory of modular proofs in the completion context, related to the problem of preserving properties of rewrite systems when considering their union. Results have already been given for confluence and termination of the direct sum of two rewrite systems, that is the union of two rewrite systems having disjoint sets of function symbols. In particular, it is shown in [15] that two term rewriting systems are left-linear, confluent and terminating if and only if the direct sum of these systems is so. Let us also cite a work concerning termination [14], and a work concerning confluence and termination in the case of sorted rewrite systems [4].

6.2 Completion and the static *OBJ3* rewrite engine

The first problem was *to dynamically update the set of rules in the rewrite engine of* OBJ3 during the completion. Two solutions were possible: to write a new rewriting procedure independent

of *OBJ*, for ensuring reduction in completion, or to use the already existing engine of *OBJ*. The second way was chosen for reusability and integration reasons. But the *OBJ* rewrite engine is very close to the set of reducing rules itself. Let us present its mechanism. For efficiency reasons, rules are specialized [11], and then installed in different data structures: for any operator f appearing as top in a left-hand-side of rule is attached:

- a rule-ring for the rules which top of right-hand-side is also f (the "top-respecting" rules)
- a rule set for other rules whose top of left-hand-side is f.

Then the procedures for rewriting or normalizing a term strongly interact with this structure. Installing such a reduction engine is quite expensive. But note that in *OBJ*, since axioms of a module are interpreted as rewrite rules, the rewrite engine is installed once for all, at compiling time. The problem is different with completion since the set of rules changes dynamically. The rewrite engine has then to be redefined each time the set of rules has to be used (by applying inference rules for rewriting), after it has changed. This is the part of the "rule generation" process, inserted between inference rules in the completion strategy, as follows:

```
eq strat-simple = while non-empty-critical-pairs do

                  while non-empty-set-of-pairs do
                  normalize-lhs-pair concat
                  normalize-rhs-pair concat
                  if is-trivial-pair
                  then delete-pair
                  else if orientable-pair
                       then orient-pair-l-to-r concat
                            orient-pair-r-to-l concat
                             rule-generation
                            iter(normalize-rhs-rule concat
                                 simplify-lhs-rule concat
                                  rule-generation,
                                 for-each-rule)
                       else failure-inf-rule
                       endif
                  endif
                  end while concat
                  add-critical-pairs

                  end while .
```

The second problem is also related to the rewriting process. Several *features for improving the reduction* of a term to its normal form were integrated in *OBJ3*. The first one is concerned with the *rewriting strategy assigned to operators*. For example, a strategy *[2 1 0]* given to the operator f means that any term whose top is f will be normalized successively at the occurrences 2, 1 and 0. This strategy is given by the user, in defining the operators. By default, an algorithm computes a strategy for operators during the compilation of modules. It is clear that a term with f as top operator is not reducible at the top f if f is not the top of a left hand side of rule. *OBJ3* then assigns to such operators a rewriting strategy where the occurrence 0 never appears.

Then the matching test for rewriting terms at an f occurrence is avoided. Provided completion can orient axioms in the right hand side direction, terms with f on top can become reducible, and the previous $OBJ3$ feature leads the rewriting strategy to be incomplete. The completeness of computations was restored by adding the *[0]* occurrence to any operator strategy.

The second optimizing feature of rewriting in $OBJ3$ consists in *marking the terms in normal form*, for avoiding to try to reduce them in further computations. Again, since completion dynamically changes the set of rules R, a normal form at a given step of completion can become reducible further. The solution chosen here is to update the "normal form mark" of terms, by deleting it each time R is modified.

All the previous features for improving efficiency of rewriting in *OBJ3* lie on the fact that the set of rules is not considered as evolving. The solution for a further prototype of such an integrated programming environment could be the design of two different rewrite engines: one flexible and dynamic with respect to the rewrite rules (for completion), the other one designed for efficiency of rewriting and used only when executing programs (installed after completion for example).

7 Examples of completion in ELIOS-OBJ

We now give examples of completion in *ELIOS-OBJ*. An $OBJ3$ program specifying the predicate *is-even* on integers is given. Then a completion is performed on this program to give an equivalent one, warranted to be terminating and unambiguous for any data. We also illustrate on this example that an interreduced set of rules can lead to a more efficient execution. For obtaining the same result, the program is executed on the data $is - even(p(p(0)))$ in 5 steps before completion, instead of 1 after.

```
tarsky.loria.fr% cat even.obj

obj EVEN is

 sorts Zero NzNeg Neg NzPos Pos Int Boolean .
 subsorts Zero < Neg < Int .
 subsorts Zero < Pos < Int .
 subsorts NzNeg < Neg .
 subsorts NzPos < Pos .

 op 0 : -> Zero .
 op s : Pos -> NzPos .
 op p : Neg -> NzNeg .
 op true : -> Boolean .
 op false : -> Boolean .
 op is-even : Int -> Boolean .
 op opposite : NzNeg -> NzPos .

 var x : Pos .
 var y : NzNeg .

 eq is-even (0) = true .
 eq is-even (s(0)) = false .
```

```
 eq is-even (s(s(x))) = is-even (x) .
 eq is-even (y) = is-even (opposite(y)) .
 eq opposite (p(0)) = s(0) .
 eq opposite (p(y)) = s (opposite(y)) .

jbo


tarsky.loria.fr% elios-obj

          ********************************************************
           Welcome to ELIOS-OBJ, a sympathetic OBJ with completion
          ********************************************************
                   ************************************
                            ******************

                   Copyright 1988 SRI International
                   Copyright 1991 I. Gnaedig - INRIA Lorraine & CRIN

       To have the list of available commands, do: 'help', 'h' or '?'


ELIOS-OBJ> in ex/even
==========================================
obj EVEN


ELIOS-OBJ> reduce is-even(p(p(0))) .
reduce in EVEN : is-even(p(p(0)))
rewrites: 5
result Boolean: true

ELIOS-OBJ> complete EVEN .

         The starting equations for completion are those
         of the current module:
"EVEN"

Only UNCONDITIONAL equations are retained. They are:

is-even(0) = true
is-even(s(0)) = false
is-even(s(s(x:Pos))) = is-even(x:Pos)
is-even(y:NzNeg) = is-even(opposite(y:NzNeg))
opposite(p(0)) = s(0)
opposite(p(y:NzNeg)) = s(opposite(y:NzNeg))

.../...
```

```
The complete set of rules is:

is-even(0) -> true
is-even(s(0)) -> false
is-even(s(s(v1:Pos))) -> is-even(v1:Pos)
is-even(opposite(v2:NzNeg)) -> is-even(v2:NzNeg)
opposite(p(0)) -> s(0)
opposite(p(v3:NzNeg)) -> s(opposite(v3:NzNeg))
is-even(p(0)) -> false
is-even(s(opposite(v7:NzNeg))) -> is-even(p(v7:NzNeg))
is-even(p(p(0))) -> true
is-even(p(p(v9:NzNeg))) -> is-even(v9:NzNeg)

ELIOS-OBJ> reduce is-even(p(p(0))) .
reduce in EVEN : is-even(p(p(0)))
rewrites: 1
result Boolean: true
ELIOS-OBJ>
```

A second example is given below. It specifies an addition on integers, themselves described in terms of successors of zero and opposites of successors of zero.

```
tarsky.loria.fr% cat addition.obj

obj ADDITION is

 sorts Zero NzNat Nat Int .
 subsorts Zero < Nat .
 subsorts NzNat < Nat .
 subsorts Nat < Int .

 op 0 : -> Zero .
 op s : Nat -> NzNat .
 op _+_ : Nat Nat -> Nat .
 op _+_ : Int Int -> Int .
 op - _ : Int -> Int .

 vars x y : Nat .

 eq 0 + x = x .
 eq x + 0 = x .
 eq x + s(y) = s(x + y) .
 eq s(x) + y = s(x + y) .
 eq - - x = x .
 eq - 0 = 0 .
 eq (- x) + (- y) = - (x + y) .
 eq s(x) + (- s(y)) = x + (- y) .
 eq - s(x) + s(y) = - x + y .

jbo
```

```
ELIOS-OBJ> in ex/addition
=========================================
obj ADDITION


ELIOS-OBJ> complete ADDITION .

...\...

The complete set of rules is:

0 + v1:Nat -> v1:Nat
v2:Nat + 0 -> v2:Nat
v3:Nat + s(v4:Nat) -> s(v3:Nat + v4:Nat)
s(v5:Nat) + v6:Nat -> s(v5:Nat + v6:Nat)
- (- v7:Nat) -> v7:Nat
- 0 -> 0
- v8:Nat + - v9:Nat -> - (v8:Nat + v9:Nat)
s(v10:Nat) + - s(v11:Nat) -> v10:Nat + - v11:Nat
- s(v12:Nat) + s(v13:Nat) -> - v12:Nat + v13:Nat
0 + - v27:Nat -> - v27:Nat
- v28:Nat + 0 -> - v28:Nat
```

Acknowledgments

We would like to thank Claude and Hélène Kirchner for fruitful discussions concerning the design of completion in $OBJ3$, and Pierre Lescanne for carefully reading previous versions of this paper.

References

[1] L. Bachmair and N. Dershowitz. Completion for rewriting modulo a congruence. In *Proceedings 2nd Conference on Rewriting Techniques and Applications, Bordeaux (France)*, volume 256 of *Lecture Notes in Computer Science*, pages 192–203, Bordeaux (France), May 1987. Springer-Verlag.

[2] N. Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3(1 & 2):69–116, 1987.

[3] R. Forgaard and John V. Guttag. Reve: A term rewriting system generator with failure-resistant Knuth-Bendix. Technical report, MIT-LCS, 1984.

[4] H. Ganzinger and R. Giegerich. A note on termination in combinations of heterogeneous term rewriting systems. *Bulletin of European Association for Theoretical Computer Science*, 31, February 1987.

[5] I. Gnaedig, C. Kirchner, and H. Kirchner. Equational completion in order-sorted algebras. In M. Dauchet and M. Nivat, editors, *Proceedings of the 13th Colloquium on Trees in Algebra and Programming*, volume 299 of *Lecture Notes in Computer Science*, pages 165–184. Springer-Verlag, 1988.

[6] I. Gnaedig, C. Kirchner, and H. Kirchner. Equational completion in order-sorted algebras. *Theoretical Computer Science*, 72:169–202, 1990.

[7] J. A. Goguen and J. Meseguer. Order-sorted algebra I: Partial and overloaded operations, errors and inheritance. Technical report, SRI International, Computer Science Lab, 1988. Given as lecture at a Seminar on Types, Carnegie-Mellon University, June 1983.

[8] J. A. Goguen and T. Winkler. Introducing OBJ3. Technical Report SRI-CSL-88-9, SRI International, 333, Ravenswood Ave., Menlo Park, CA 94025, August 1988.

[9] G. Huet and D. Oppen. Equations and rewrite rules: A survey. In R. V. Book, editor, *Formal Language Theory: Perspectives and Open Problems*, pages 349–405. Academic Press, New York, 1980.

[10] S. Kamin and J.-J. Lévy. Attempts for generalizing the recursive path ordering. *Inria, Rocquencourt*, 1982.

[11] C. Kirchner, H. Kirchner, and J. Meseguer. Operational semantics of OBJ-3. In *Proceedings of 15th International Colloquium on Automata, Languages and Programming*, volume 317 of *Lecture Notes in Computer Science*, pages 287–301. Springer-Verlag, 1988.

[12] P. Lescanne. Computer experiments with the REVE term rewriting systems generator. In *Proceedings of 10th ACM Symposium on Principles of Programming Languages*, pages 99–108. Association for Computing Machinery, 1983.

[13] P. Lescanne. Implementation of completion by transition rules + control: ORME. In H. Kirchner and W. Wechler, editors, *Proceedings 2nd International Workshop on Algebraic and Logic Programming, Nancy (France)*, volume 463 of *Lecture Notes in Computer Science*, pages 262–269. Springer-Verlag, 1990.

[14] M. Rusinowitch. On termination of the direct sum of term rewriting systems. *Information Processing Letters*, 26(2):65–70, 1987.

[15] Y. Toyama, J. W. Klop, and H. P. Barendregt. Termination for the direct sum of left-linear term rewriting systems. In N. Dershowitz, editor, *Proceedings 3rd Conference on Rewriting Techniques and Applications, Chapel Hill (North Carolina, USA)*, volume 355 of *Lecture Notes in Computer Science*, pages 477–491. Springer-Verlag, April 1989.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique

ISSN 0249-6399