PROGRAMMING WITH EQUALITIES, SUBSORTS, OVERLOADING, AND PARAMETRIZATION IN OBJ

JEAN-PIERRE JOUANNAUD,* CLAUDE KIRCHNER,† HÉLÈNE KIRCHNER,‡ AND ARISTIDE MÉGRELIS†‡§

▷ OBJ is a declarative language, with mathematical semantics given by order-sorted equational logic and an operational semantics based on order-sorted term rewriting. OBJ also has user-definable abstract data types with mixfix syntax and a flexible type system that supports overloading and subtypes. In addition, OBJ has a powerful generic module mechanism, including nonexecutable "theories" as well as executable "objects", plus "module expressions" that construct whole subsystems. Design and implementation choices for the OBJ interpreter are described here in detail. ◁

1. INTRODUCTION

The OBJ programming language originates from J. Goguen's pioneering work at UCLA [20] in the late seventies. It was then developed at SRI International [5, 21, 13] with the collaboration of several visitors. Different versions have also been worked out at several places [1, 47, 3], but we focus here on the released version of OBJ-3. This paper is an overview of interesting features of OBJ in which we have been involved in collaboration with K. Futatsugi, J. Goguen, J. Meseguer, and T. Winkler.

OBJ is a logical programming language whose latest versions (OBJ-2 [5] and OBJ-3 [13, 21]) are based on order-sorted equational logic. Programs are order-sorted equational specifications, and computation is a form of equational deduction performed by rewriting.

Type structure supports conceptual clarity and detection of many errors at program

Address correspondence to Jean-Pierre Jouannaud, Université de Paris Sud, Centre d'Orsay, Bat 490, 91405 Orsay, France.
Accepted 9 November 1990.
*LRI. LRI-CNRS.
†INRIA Lorraine and CRIN-CNRS, BP 101, 54600 Villers-lès-Nancy Cedex, France.
‡CRIN-CNRS and INRIA Lorraine, BP 101, 54600 Villers-lès-Nancy Cedex, France.

655 Avenue of the Americas, New York, NY 10010 0743-1066/92/$3.50

entry time. However, implementing strong typing with many-sorted logic is quite rigid and lacks the expressive power needed for handling errors and partiality. Many of these problems are overcome by using an order-sorted type structure that brings inheritance, operator overloading, and error handling within the realm of equational logic, and makes many seemingly partial or problematic functions total and well defined on the right subsort [17, 18]. This order-sorted type structure is provided by a partial ordering on the set of sorts that is interpreted as set inclusion in order-sorted algebras. For instance, a subsort relation `Nat < Int` is interpreted as the inclusion $\mathbb{N} \subseteq \mathbb{Z}$ of the naturals into the integers in the standard model. In addition, operator symbols such as `_+_` may have overloaded declarations, for instance `_+_: Nat Nat -> Nat`, `_+_: Int Int -> Int`, that are required to yield the same result when restricted to arguments of the same subsorts. Overloading is a syntactical facility for handling operators defined on different subsets that may intersect, for achieving a kind of polymorphism. Moreover, such operators are in general partial functions, which is expressed thanks to the subsort relationship.

Although all basic results of equational logic generalize to the many-sorted case, order-sorted deduction is more subtle [17]. For example, replacement of equals by equals and term rewriting require a careful analysis that was initiated in [22] and is further developed in [27, 33].

We briefly summarize in Section 2 current features of the OBJ language, and in Section 3 discuss some specifics of the implementations. Section 4 presents the behavior of OBJ-3 by means of a few classic examples, used in the literature to illustrate delicate questions about order-sorted computations. Finally, Section 5 suggests some future developments. In the whole paper, we assume familiarity with the OBJ-3 syntax described in [21] and illustrated in the annex by an example.

2. OBJ FEATURES

Modularity and strong typing are two important features of a programming language, supported by OBJ in a powerful way.

2.1. Modules and Generics

OBJ has three basic kinds of building blocks:

Objects declare new sorts of data, and define new operators by equational (or conditional) axioms, which become executable code when interpreted as rewrite rules. To an object is associated one particular model, namely its initial order-sorted algebra.

Theories also declare sorts, operators, and axioms, but the latter are used to define properties of modules, including requirements for generic module interfaces; thus, these axioms are not considered executable. To a theory is associated a class of models containing all the order-sorted algebras that satisfy its requirements. Objects and theories are both *modules*.

Views express bindings of actual modules to requirements, and are used to combine modules into larger program units, through *module expressions*.

OBJ modules can import other modules directly and then use their capabilities; three kinds of importations are available, qualified by the keywords *using*, *extending*, or *protecting* according to the properties (no junk, no confusion) of the initial algebras involved in the enrichment. This leads to a module hierarchy.

OBJ modules can also be parametrized (or generic). Interface declarations for parametrized modules are not purely syntactic, as for example in Ada, but are given by theories that may contain semantic requirements. All these features allow for a more reusable code, since it can be tuned up for different applications by modifying module expressions and parameter values. Moreover, debugging, maintenance, readibility, and portability are all improved.

2.2. Subsorts and Polymorphism

Since the language can model the inclusion of one data type in another, the OBJ subsort mechanism provides, within the framework of the initial algebra semantics, the following capabilities:

Subsort polymorphism based on overloaded operator symbols, as in the number hierarchy of naturals, integers, and rationals, expressed by stating that the sort `Nat` is a subsort of `Int` and `Int` is a subsort of `Rat`. Then an operator like + can be defined at several levels of the sort hierarchy. Parametric polymorphism is provided through parametrized modules, since their operators are available to all their instances.

Multiple inheritance in the sense of object-oriented programming, allowing a sort to be a subsort of two or more others. Then an operator defined on a supersort is automatically inherited by the subsort.

Common difficulties in algebraic data types with partial operations disappear on viewing the operations as total on the right subsorts. For instance, the `tail` operator on lists, undefined for an empty list, has the set of nonempty lists as its definition domain.

Exceptions can be treated in several styles, without the need for any special exception-handling mechanism.

3. OBJ SPECIFICS

Some important choices have been made in the design and the implementation of the OBJ interpreter. They are reviewed and explained in this section.

3.1. Lowest Sort

The OBJ user declares sorts, subsorts, and operators, with the sole restriction that type checking is possible: an OBJ term is expected to be of exactly one type, although it may be, in general, of several sorts. Let us be more precise about the definition of types.

Let OBJSPEC be an OBJ specification given by a partially ordered set of sort symbols S, a set of operators Σ, and a set of axioms E. To every term t, we associate a set of sorts, written $sort(t)$, as follows:

(1) If $(x : s) \in$ OBJSPEC then $s \in sort(x)$.

(2) If $f : s_1 \cdots s_n \to s \in$ OBJSPEC and $s_1 \in sort(t_1), \cdots, s_n \in sort(t_n)$, then $s \in sort(f(t_1 \cdots t_n))$.

(3) If $s < s' \in$ OBJSPEC and $s \in sort(t)$ then $s' \in sort(t)$

The least element in $sort(t)$, whenever it exists, is called the *lowest sort* of t or the *type* of t. It will be denoted hereafter by LS(t).

Signatures which provide a lowest sort for every term whose set of sorts is not empty are called *regular*. Only regular signatures are allowed in OBJ. Regularity can easily be checked on the signature. Here is an example of a nonregular signature, written in the OBJ syntax:

```
obj DUMMY is
   sorts s s' s''.
   subsorts s  < s', s < s''.
   op f : s' -> s'.
   op f : s'' -> s''.
   op a : -> s.
endo
```

The previous signature is not regular, because `f(a)` has the two sorts `s'` and `s''`, but neither is smaller than the other. We can see here that nonregularity arises in the context of overloading. In this case, the signature can be made regular by adding the following new operator declaration: `op f : s -> s.`

So we can clearly check at parsing time whether an OBJ term is *well formed*, that is, whether it has a lowest sort. Terms that are not well formed at parsing time are said to be *ill formed*.

3.2. *The Mixfix Parser*

Most programming languages have a fixed syntax. In OBJ, the user gives the syntactic rules related to the *operators* he declares. There are therefore two parsing processes:

A top-level parsing. This is the normal parsing process associated to the fixed part of OBJ syntax. The top-level parser looks for keywords such as 'obj', 'th', 'sorts', 'op', 'eq', 'reduce', etc.

A term parsing. This is the parsing process associated to the user's defined operators. The term parser looks for terms given as right- or left-hand sides of axioms, or for terms to be reduced. For example, in

```
eq x + s ( y ) = s ( x + y ) .
```

there are two expressions to parse as terms: 'x + s (y)', and 's (x + y)'. In

```
reduce s ( 0 ) + s ( s ( 0 ) ) .
```

there is one expression to parse as a term: 's (0) + s (s (0))'.

As the most interesting part is the term parser, we shall not describe the top-level parser.

A *term* is a well-formed string of operator symbols, variables, parentheses, and commas, such as '`f␣(␣pop␣(␣x␣)␣,␣y␣)`' (9 tokens) or '`i␣+␣j`' (3 tokens). In these examples, successive symbols are separated by spaces (explicitly marked '`␣`'). In general, these spaces are not necessary.

The notation used in programming languages for terms (and other constructs) can be cumbersome, since the user is limited by a syntax fixed by the language designer. OBJ avoids this difficulty by allowing user-definable syntax. The notation for operators may be functional, prefix, infix, postfix, and even "mixfix" (where the operator is split in several parts around the operands). In this way, most parentheses can be avoided. For example, it is possible to declare "modulus" and "factorial" operators in such a way that the following expressions are parsed:

'`| z |`',

'`| z | ! !`', or

'`[ 1 / | z | ] ! + 3 * 2 i`'

('`[ r ]`' designates the truncated part of a real number `r`).

The term parsing mechanism implemented in OBJ takes two arguments into account. First, it uses information about sorts. Second, it uses precedence and associativity rules, which are given for each operator. To illustrate this point, let us look at an example of an operator declaration whose parsing attributes are explicit:

```
op _ + _ : Int Int -> Int [ prec 10 gather ( e E ) ].
```

The attribute `prec` gives the precedence of the operator '`+`'; `gather` gives its gathering rule, also called an associativity rule. Both attributes are given by the user and specify how nonparenthesized expressions will be parsed: the precedence of an operator is related to its binding power (the less the precedence, the greater the binding power); the associativity rule helps to resolve conflicts between two or more nonparenthesized subexpressions.

Sort Information. Some parses are unacceptable simply because the arguments of some operator are of the wrong sort. This is the ordinary rule, which applies when the parser looks for a term *in an axiom*: either the sorts fit (the actual sort is the expected sort, or a subsort of the expected sort), or the parser detects a sort conflict and rejects the string.

In case the parser looks for a term given *for reduction*, we relax this simple rule; the arguments of an operator may be *of any sort*. If the actual sort of an argument does not fit, the parser inserts a *retract* between the argument and the operator: the retract is a pseudooperator that signals a pending problem that may be resolved when reductions proceed. For example, if `s` and `/` are operators decleared as `s: Nat -> Nat` and `/ : Rat NzRat -> Rat`, the expression `s ( 1 / 1 )` will be parsed with a retract (`1 / 1` is given in chance to be a natural).

The rewriting engine of OBJ is instructed to handle correctly such "ill formed" terms (see Section 3.4). In the previous example, the computation would actually prove that `1 / 1` is indeed a natural, in the sense that it will be eventually type-checked as a natural number.

There is an advantage in taking care of the sort information. Some mistakes are detected earlier, hence saving unnecessary computations. Also in some cases, the parser

discriminates the correct parse, thanks to sort information. For example, consider the string

```
length 1 !
```

It must be parsed as the factorial of the length of the list `1`, and not as the length of the factorial of `1` (the sort of `1` does not fit the factorial operator).

Precedence and Gathering Rule. These two purely syntactic attributes work hand in hand. They are inspired by a PROLOG implementation [2].

Precedence information is used to compute, for each substring likely to be a subterm, a quantity called *weight*. Then comes the question whether that substring is acceptable as an argument to some higher-level operator. To decide, we match the weight of the substring against the gathering rule of the higher-level operator.

Let us look at an example, with the following declarations:

```
op _ ! : Int -> Int [ prec 2 gather ( E ) ] .
op _ ^ _ : Int Int -> Int [ prec 5 gather ( e E ) ] .
op _ * _ : Int Int -> Int [ prec 10 gather ( e E ) ] .
op _ + _ : Int Int -> Int [ prec 20 gather ( e E ) ] .
```

and we intend to parse '`5 ^ 5 ^ 3 ! + 3 * 7 ^ 4`'.

How to compute the weights? Substrings made of a constant, a variable, or a function plus arguments written functionally, and parenthesized substrings, all have weight 0. In the example above, '`5`', as a substring, weighs 0. Other examples of substrings weighing 0 are (in another context) '`f ( x )`', '`( x + y )`'.

How to combine subterms to build a larger subterm? If not made of a constant, a variable, or a function plus arguments written functionally, and if not parenthesized, strings are made of operators written in a *mixfix* notation. The rule in this case is that such an operator accepts, as arguments marked '`E`' (*Expression*), substrings whose weights are less than or equal to its precedence, and accepts, as arguments marked '`e`' (*expression*), substrings whose weights are (strictly) less than its precedence. The weight of the total string is then the precedence of its "top" operator.

For instance, the substring '`3 !`' is made of the postfix operator '`!`' and a left argument which is the substring '`3`'; '`3`' of weight 0 is acceptable to '`!`', for $0 \leq 2$; therefore '`3 !`' can be parsed as a *subterm*. The weight of '`3 !`' is then the precedence of its "top" operator, in this case 2.

In the same way, '`7 ^ 4`' can be parsed as a subterm and weighs 5. So '`3 * 7 ^ 4`' can be parsed as a subterm of weight 10. The same rule applies recursively until the string is completely parsed.

It is easy to check that the string given above parses as a term, as expected:

```
(5 ^ (5 ^ (3 !) ) )+(3* (7 ^ 4) )
```

As it is, when given an ill-formed string, the parser rejects it without asking the user about a possible correction; this absence of parsing-error recovery is a possible place for improvement.

3.3. Operational Semantics

The OBJ-2 and OBJ-3 systems implement the same mathematical semantics based on order-sorted equational logic, but differ in their operational semantics. For OBJ-2, the

operational semantics [22] used a translation of order-sorted algebra into many-sorted algebra that reduced computation to standard term rewriting but could generate many rules, with a resulting loss in efficiency. By contrast, OBJ-3 achieves a simpler and more efficient solution through the direct performance of *order-sorted rewriting*. The following review of the main features of the operational semantics of OBJ-3 is based on work with J. Meseguer [33].

Several choices have been made in the implementation of order-sorted rewriting. In order to introduce them, let us consider simple examples of integer and rational arithmetic that may illustrate some of the peculiarities involved in the operational semantics. The OBJ-3 program we are referring to in this section is given in the Appendix.

The first important choice is the definition of *order-sorted matching*. Consider for instance the axiom

```
x+sy=s(x+y),                                    (1)
```

where `x` and `y` are variables of sort `Nat` and `s : Nat -> Nat` is the successor function. If we apply this axiom, oriented from left to right, as a rewrite rule to the integer expression

```
(-4)+(s0),                                      (2)
```

where `(-4)` stands for `-(s s s s 0)`, we would obtain the term `s ((-4) + 0)`, which is ill formed if we assume that the successor function only exists for the naturals and not for the integers. Therefore, it becomes crucial to check the sorts of the variables when matching a left-hand side. An *order-sorted match* is a substitution σ that satisfies

for all x such that $\sigma(x) \neq x$, $\quad \text{LS}(\sigma(x)) \leq \text{LS}(x)$.

Finding out the sort of a subterm would in principle require parsing. However, such sort information can be precomputed at parse time and kept stored in the term to be reduced, in order to be used when needed or to be entirely disregarded otherwise. Our expression then would look as follows:

```
                    +(Int Int -> Int)
                   /  \
(NzInt -> NzInt)-     s(Nat -> NzNat)
                |     |
      (-> NzNat)4     0(->Nat)
```

where `NzNat` (`NzInt`) is the subset of nonzero natural numbers (nonzero integers), and we assume that `-_` is overloaded as `-_: Int -> Int` and `-_: NzInt -> NzInt`. Such a form is actually the *lowest parsed form* of the expression. An operator with its rank is called a disambiguated operator, and we have placed the rank information in parenthesis to suggest that it can be forgotten; the tree can then be considered identical to the original expression: this means that we can "turn" that information "on and off" depending on the actual use. For instance, we can match our axiom in the standard way, forgetting rank information, but when reading a variable, we then inspect the sort of the term matched to it, in

order to ensure correctness. In this case, the variable y matched with a term of sort `NzInt` results in a failure.

Whenever a large set of rules has to be scanned, efficiency is greatly improved by giving direct access to the subset of rules whose left-hand side has the same top operator as the term to be reduced. In our example, we may associate the axiom (1) with the disambiguated operator `+ : Nat Nat -> Nat` obtained by lowest-parsing its left-hand side. In that case, we would not even attempt matching (1) to (2), since only terms with disambiguated top operator `+ : Nat Nat -> Nat` would be candidates. So *localization of rules* is provided through disambiguated operator symbols.

Also, in that context, an additional gain in efficiency can be realized by avoiding checking the sorts of *general variables*. For instance, in the axiom (1), there is no need whatsoever to check the sorts of variables `x` and `y`, since they are as general as possible for `+` of rank `Nat Nat -> Nat`. By contrast, the sort of integer variable `i` in the axiom

```
i * (r / r')=(i * r)/r'
```

(where `r` has sort `Rat`, `r'` has sort `NzRat`, and `_/_` denotes rational division) is not as general as possible and has to be checked, even after associating (3) to the operator `_*_ : Rat Rat -> Rat`.

The choice of associating rules to disambiguated top operators induces the need for *specialization of rules*. Note that, assuming that we also had the operator declarations

```
op _*_  : NzRat NzRat -> NzRat
op _/_  : NzRat NzRat -> NzRat,
```

we should also associate (3) to the disambiguated operator `_*_ : NzRat NzRat -> NzRat` in order to rewrite the expression `7*(3/4)`, i.e., we have to "specialize" axiom (3) to smaller top operators.

After matching succeeds, the instance of the left-hand side has to be replaced by the corresponding instance of the right-hand side. In general, after instantiation, the resulting term is no longer in lowest parsed form. A last improvement is *sort compilation of right-hand sides* of rules. The appropriate lowest parse of the right-hand side is computed in advance for each possible sort for variables. A right-hand-side instance in lowest-parsed form is then obtained in a straightforward way.

In summary, the following approach is taken:

At module installation time:

1. localize rules by associating them to disambiguated top operator symbols,
2. specialize rules to smaller overloaded top operator symbols,
3. precompute lowest parses of right-hand sides for different cases of left-hand-side instances.

At run time:

1. look at the rank of the top operator symbol and try only the rules associated to that operator,

2. match left-hand sides with complete disregard of sort information, except for checking the sort of those variables that are not fully general,
3. replace the instance of the left-hand side by the precomputed instance of the right-hand side in lowest parsed form,
4. if necessary, update rank information above the position where the replacement has taken place whenever the sort has decreased at that position.

The efficiency of such a process is close to that of standard term rewriting; the only differences are the possible need for checking sorts (this can be done in constant time), and the occasional performance of step 4, which is done by need only. The gain in expressive power is considerable. For example, arithmetic computation can take place in the entire number hierarchy, with a syntax that entirely agrees with that of standard mathematical practice.

The necessary foundations for the operational semantics are developed in [33] and involve the following points:

(1) *Relating validity in all models to order-sorted rewriting*, denoted $\rightarrow^R$, giving conditions for the soundness and completeness of OBJ's operational semantics. To get these results, additional hypotheses on the set of rules are needed: An order-sorted term rewriting system R is *sort-decreasing* if and only if for any terms t, t', $t \rightarrow^R t'$ implies $\text{LS}(t) \geq \text{LS}(t')$. An OBJ specification defined by (S, Σ, R) is *fair* if R is confluent and sort-decreasing. Decidable sufficient conditions can be given to check sort-decreasingness [22, 33]. Then an equational theorem $(t = t')$ is valid in all models of a fair order-sorted specification (S, Σ, R) if and only if $t \overset{*}{\leftrightarrow}^R t'$, where $\overset{*}{\leftrightarrow}^R$ denotes the reflexive symmetric transitive closure of the order-sorted rewriting relation. It is shown in [9] how one can complete an order-sorted specification into a fair one.

(2) *Generalizing ordinary rewriting to order-sorted rewriting modulo axioms*, since OBJ supports rewriting modulo axioms such as associativity and commutativity. The definition of fair specification extends to equational term rewriting systems (R, A): the set of nonorientable axioms A is assumed to be *sort-preserving*, that is, for any terms, t, t', $t \leftrightarrow^A t'$ implies $\text{LS}(t) = \text{LS}(t')$, where $\leftrightarrow^A$ denotes one application of an axiom in A.

(3) *Studying the process of specialization of rules*, providing criteria and algorithms for such a process and for the associated aspects of generality of variables and compilation of right-hand sides.

(4) *Developing correctness criteria for the operational semantics of modules that are structured in a hierarchical fashion* with different importation relations between modules, such as `using`, `extending`, or `protecting`. In particular, developing correctness criteria for considering modules independently at compile time.

3.4. *Type Checking in* OBJ

Even in fair specifications, interaction between subsorts and axioms may produce ill-formed terms which are equivalent, by using the axioms, to well-formed terms. Let us illustrate this point by a now classical example:

```
obj STACK [X :: TRIV] is
   sorts stack non-empty-stack.
   subsorts non-empty-stack < stack.

   op nil  : -> stack.
   op push : elem stack -> non-empty-stack.
   op pop_ : non-empty-stack -> stack.
   op top_ : non-empty-stack -> elem.

   vars e : elem, s : stack.
   eq top push(e, s)=e.
   eq pop push(e, s)=s.
endo
```

Now, instantiating X by INT yields a stack of integers:

```
obj STACK-INT is STACK[INT] endo.
```

We may now compute STACK-INT expressions, for example, top push (1, nil), which gives the expected result 1. But here is a more surprising example: the expression

```
pop pop push (1, push (2, push (3, nil)))
```

produces a parsing failure.

The problem is that pop push (1, push (2, push (3, nil))) is of type stack; hence pop cannot be applied, since it requires an argument of sort non-empty-stack, which is strictly smaller than stack. Of course, subsorts would be almost useless if such expressions could not be computed; hence such terms must be parsed and computed.

Remark that the first subterm of the whole term, i.e. pop push (1, push (2, push (3, nil))), is well formed and results in push (2, push (3, nil)). Applying pop to the result is now possible and yields the expected stack: push (3, nil). So we may well accept ill-formed terms, provided computations happen always on well-formed subterms, just as previously.

This can be formalized by constructing a conservative extension of stacks as follows [46]:

```
obj STACK2 [X :: TRIV] is
   sorts stack non-empty-stack error-stack error-elem.
   subsorts non-empty-stack < stack < error-stack.
   subsorts elem < error-elem.

   op nil : -> stack.
   op push : elem stack -> non-empty-stack.
   op pop_ : non-empty-stack -> stack.
   op top_ : non-empty-stack -> elem.
   op push : error-elem error-stack -> error-stack.
   op pop_ : error-stack -> error-stack.
   op top_ : error-stack -> error-elem.
```

```
vars e: elem, s : stack.
   eq top push(e, s)=e.
   eq pop push(e, s)=s.
endo
```

Now, pop pop push (1, push (2, push (3, nil))) has type error-stack, and its subterm pop push (1, push (2, push (3, nil))) is still of type stack. Remark that the last axiom in STACK does not apply to the whole term, since the sorts do not agree, but it of course applies to the subterm, yielding pop push (2, push (3, nil)). The type of the whole term is now stack; it has been lowered, since the type of the first subterm has itself been lowered from stack to non-empty-stack. Hence the same axiom applies now on top, yielding the expected result.

This shows that type information cannot be computed only at parsing time, since types may change at runtime, due to overloading and subsorts. However, type checking can be made static by computing in a extended signature, as shown before. This signature is a conservative extension of the previous one; hence it behaves as expected.

A drawback of the previous extension is that it adds quite a lot of new sorts, and it forces the computations to be bottom-up. In some cases, for example computations on streams, bottom-up evaluation is not feasible, and another mechanism must be provided. Such a mechanism was implemented in OBJ-2, and consisted of another conservative extension of signatures [22]:

```
obj STACK3 [X :: TRIV] is
   sorts stack non-empty-stack.
   subsorts non-empty-stack < stack.

   op nil  : -> stack.
   op push : elem stack -> non-empty-stack.
   op pop_ : non-empty-stack -> stack.
   op top_ : non-empty-stack -> elem.
   op c_   : non-empty-stack -> stack.
   op r_   : stack -> non-empty-stack.

   vars e : elem, s : stack, ns : non-empty-stack.
   eq top push(e, s)=e.
   eq pop push(e, s)=s.
   eq r c ns=ns.
endo
```

Injections from a subsort into a supersort are called *coercions*, while operators going the other way around are called *retracts*. The added axiom says that the retract of a coerced expression yields that very same expression unchanged. But the converse is not true; coercing a retracted expression does not have any meaning in general.

Now, the previous expression is parsed with retracts and coercions and can be evaluated in the extended signature: pop r pop push (1, c push (2, c push (3, nil))) evaluates first to pop r c push (2, c push (3, nil)), then to pop push (2, c push (3, nil)), finally to c push (3, nil), and the result is a nonempty stack. Coercions and retracts act as type decorations, and the axiom r c ns=ns allows checking types at runtime.

Note that the first conservative extension does not imply any overhead, while the second implies some (very reasonable) overhead. On the other hand, the second is not

restricted to bottom-up strategies, and hence is more general. Some work should be done on combining the two disciplines in order to have the advantages of both. Note also that OBJ-3 implements the retract notion without the help of coercions.

Sort constraints are another OBJ construct (not implemented in OBJ-3) which requires systematic runtime type checking. Since not all subsorts can be defined by constructors, OBJ allows defining a subsort of some sort s as the set of all elements t of sort s that satisfy some property. Type checking then requires of course evaluation, or, as before, the definition of a conservative extension. Here is an example:

```
obj BOUNDED-STACK [X :: TRIV] is
   extending INT.

   sorts stack bounded-stack non-empty-stack.
   subsorts non-empty-stack < bounded-stack < stack.

   op nil     : -> bounded-stack.
   op bound   : -> int.
   op push    : elem bounded-stack -> stack.
   op pop_    : non-empty-stack -> bounded stack.
   op top_    : non-empty-stack -> elem.
   op length_ : stack -> int.

   vars x : elem, s : stack, bs : bounded-stack.
   as non-empty-stack : push(x, s) if length s < bound.
   eq top push(e, bs)=e.
   eq pop push(e, bs)=s.
   eq length nil=0.
   eq length push(x, s)=length(s)+1.
endo
```

where the line

```
as non-empty-stack : push(x, s) if length s < bound.
```

must be understood as follows: `push(x, s)` is of type `non-empty-stack` if `length s < bound`.

The conservative extension is defined as before, and the sort constraint is transformed into a conditional axiom:

```
obj BOUNDED-STACK2 [X :: TRIV] is
   extending INT.

   sorts stack bounded-stack non-empty-stack.
   subsorts non-empty-stack < bounded-stack < stack.

   op nil     : -> bounded-stack.
   op bound   : > int.
   op push    : elem bounded-stack -> stack.
   op pushas   : elem bounded-stack -> non-empty-stack.
   op pop_    : non-empty-stack -> bounded-stack.
   op top_    : non-empty-stack -> elem.
   op length_ : stack -> int.

   vars x : elem, s : stack, bs : bounded-stack.
```

```
   eq push (x, s)=pushas(x, s) if length s < bound.
   eq top pushas(e, bs)=e.
   eq pop pushas(e, bs)=s.
   eq length nil=0.
   eq length push(x, s)=length(s)+1.
endo
```

As we can see, the push operator either remains a push when it can be parsed or becomes a pushas operator otherwise. Of course, there is still the need for a conservative extension as previously, in order to compute with terms which cannot be parsed at parsing time. This is not done here.

3.5. Module-Expression Evaluation

Another feature of OBJ is its support for *parametrized programming* [6, 19, 12], which allows modifying and combining modules to form larger systems, with the interfaces between program units described by theories. These commands appear in *module expressions*, which describe and create complex combinations of modules. Module expressions are built from given modules by instantiating, summing, and renaming.

The simplest module expressions are previously defined nonparametrized modules. Some of these, such as the integers and booleans, are built in.

To *instantiate* a parametrized object means to provide actual objects satisfying each of its requirement theories. Actual objects are designated by *views*, which tell how to *bind* required sorts and operators to those actually provided; i.e., a view maps the sort and operator symbols in the formal requirement theory to those in the actual object, in such a way that all axioms of the requirement theory are satisfied. More generally, a view can map an operator in the formal theory to an expression in the actual object. The result of such an instantiation replaces each requirement theory by its corresponding actual module, using the views to bind actual names to formal names, being careful that multiple copies of shared submodules are not produced.

Note that there can be more than one view from a theory to an actual parameter. Often there is a *default view* that "does the obvious thing," for example, in instantiating a sorting module by binding the ordering predicate, defined in the theory POSET, to the usual order on the naturals [12], as follows:

```
th POSET is
   protecting BOOL.
   sort Elt.
   op _<_ : Elt Elt -> Bool.
   vars e1 e2 e3 : Elt.
   eq e1 < e1=false.
   cq e1 < e3=true if e1 < e2 and e2<e3.
endth

view NATG from POSET to NAT is
   sort Elt to Nat.
   op _<_ to _<_.
endv
```

Another view, which is not a default view, is given by the association of the ordering predicate to the division operator assumed defined on natural numbers:

```
view NATD from POSET to NAT is
   sort Elt to Nat.
   vars e e' : Elt.
   op e < e' to e divides e' and e=/=e'.
endv
```

Renaming uses a sort mapping and an operator mapping to create a new module from an old one where the names of sorts and the syntax of operators have been changed.

A *sum* is a new module that *adds*, or *combines*, all the information in its summands. Several examples of the use of these possibilities are given in [12].

Let us be more specific about the implementation choices. In the interpreter, *canonicalized module expressions* are used internally as names of modules. The goal of the canonicalization is to identify module expressions that are trivial variations or easily seen to be equivalent. Exactly one instance of a module is created for each canonicalized name: it is *the* module referred to by the name. An important issue is that modules that are used more than once should be *shared*, i.e., only one copy should be made.

The module-expression canonicalization process proceeds by first canonicalizing subexpressions and then checking if each expression has already been created. If so, the canonicalization process uses this previously created value. Otherwise, renames are pushed inward into instantiations and views, and composed with other renames; default views are created; and sums are pushed into views. This canonicalization process may require creating new subobjects for the resulting module.

The module-expression evaluation process proceeds by canonicalizing the expression, checking to see if the desired module has already been created, in which case the previously created module is returned as the value of the expression, and otherwise building the module described by the canonicalized expression. In instantiating an object, some of its subobjects may be instantiated, in which case their names are created and evaluated as just described.

The kind of module composition supported by module expressions has several advantages:

(1) This composition is more powerful than the purely functional composition of functional programming, in that a single module instantiation can perform many different function compositions at once. For example, a complex-arithmetic module `CPXA` that takes real-arithmetic modules as actual parameters could be instantiated with single-precision reals, `CPXA[SP-REAL]`, or double-precision reals, `CPXA[DP-REAL]`, or multiple-precision reals, `CPXA[MP-REAL]`. Each instantiation involves the substitution of dozens of functions into dozens of other functions; this would be *much* more effort with just functional composition available. Thus a part of the expressiveness of higher-order programming is available in a structured and flexible, but still rigorous, way. (See [11] for more examples, including some hardware verification.)

(2) The logic remains first-order, so that understanding and verifying code is much simpler than with higher-order logic.

(3) Semantic declarations are allowed at module interfaces given by requirement theories.

(4) Besides instantiation, module expressions also allow renaming module parts and "summing" modules, i.e., making all their contents available at once.

In the framework of many-sorted algebras, correctness criteria for parameter passing have been studied for instance in [4]. The extension to order-sorted parametrized specifications of these results is explored in [43, 42].

4. SOME OUT-OF-SCOPE TESTS

Order-sorted logic has already been extensively studied, and variations about deduction and models have been proposed: let us mention [45, 10, 22, 26, 27, 46, 37]. From the beginning, the study of order-sorted computations has revealed several examples illustrating subtle difficulties. We review some of these examples in this section and give the behavior of OBJ-3 on these examples.

It should be emphasized that in order to have correct behavior of the OBJ-3 interpreter, the hypotheses made in its design and recalled above must be satisfied: this section shows what happens when they are violated. For now, it is the responsibility of OBJ-3 user to check that its specifications satisfy the hypotheses of nonempty sort and fairness. We are currently working on the design and implementation of an integrated safe programming environment to support such verifications [9, 8].

4.1. Empty Sorts

As in many-sorted deduction [38], variables quantified on empty sorts may cause unsound deductions. OBJ-3 does not check whether sorts are empty or not. Thus, in the following example, the nonempty sort hypothesis of OBJ-3 is violated, and the deduction `b=a` is correct only for models whose domain of sort `s` is nonempty. It is not valid in the initial model of the specification:

```
obj EMPTY is
   sort s s'.
   op a : -> s'.
   op b : -> s'.
   op f : s-> s'.
   var x : s.
   eq f(x)=a.
   eq b=f(x).
endo
reduce b.
==========================================
obj EMPTY
Warning: variables in RHS not subset of those in LHS
    eq b=f(x)
```

```
==========================================
reduce in EMPTY : b
rewrites: 2
result s': a
```

4.2. *Congruence Closure*

There exist different definitions of order-sorted congruence. In [26] the following example is given to illustrate the fact that two terms a and b may be equivalent but not f(a) and f(b):

```
obj CONGR is
   sort s s' s'' s1 s2 s3.
   subsorts s < s''.
   subsorts s' < s''.
   subsorts s1 < s3.
   subsorts s2 < s3.

   op a : -> s.
   op b : -> s'.
   op f : s -> s1.
   op f : s' -> s2.

   eq a=b.
endo

reduce f(a).

==========================================
reduce in CONGR : f(a)
rewrites: 1
result s1: f(r:s'>s(b))
```

The specification violates the fairness hypothesis of OBJ-3, since the axiom a = b oriented from left to right is not sort-decreasing. obj-3 rewrites f(a) to f (r:s'> s (b)), which contains a retract from s' to s that cannot be eliminated.

4.3. *Transitivity Using Ill Formed Terms*

Another famous example [46] illustrates the fact that equational replacement (or rewriting) can produce ill formed terms and use them as intermediate steps in equational deduction:

```
obj TRANS is
   sort s s'.
   subsorts s < s'.

   op a : -> s.
   op b : -> s.
   op c : -> s'.
   op f : s-> s'.
```

```
    eq a = c.
    eq c = b.
endo

reduce f(a).

=========================================
reduce in TRANS : f(a)
rewrites: 2
result s': f(b)
```

This example violates the OBJ-3 hypothesis of fairness. But this specific example is correctly handled thanks to the retract mechanism. OBJ-3 performs the following reduction:

```
f(a) -> f ( r:s'> s (c) ) -> f ( r:s'> s (b) ) -> f(b).
```

This reduction thus handles the ill-formed term `f(c)`.

4.4. Sort-Decreasingness

Non-sort-decreasing rewriting systems are tricky because the critical pair check for confluence is no longer valid [46].

In the following example, due to G. Smolka, which violates the sort-decreasing hypothesis of OBJ-3, the term `f(a)` rewrites ambiguously to `f(b)` and to a. However, `f(b)` is irreducible, and the rewriting system is thus nonconfluent. Nevertheless there is no critical pair between the two rules:

```
obj PC is
    sort s s'.
    subsorts s < s'.

    op a : -> s.
    op b : -> s'.
    op c : -> s'.
    op f : s -> s.
    op f : s' -> s'.

    var x : s.
    eq f(x)=x.
    eq a = b.
endo

reduce f(a).
reduce f(b).

=========================================
reduce in PC : f(a)
rewrites: 2
result s: r:s'>s(b)

=========================================
reduce in PC : f(b)
rewrites: 0
result s': f(b)
```

Without further specification given by the user, OBJ-3 performs here the following bottom-up reduction:

```
f(a) -> f ( r:s'> s (b) ) -> r:s'> s (b)
```

using first the rule `a -> b` and second the rule `f(x) -> x`.

Note that the introduction of retracts gives to the user a precise information about what is wrong in the specification.

5. FUTURE

There are many possible extensions for OBJ or similar languages based on order-sorted equational logic. Let us mention a few of them:

Efficiency. Rewrite-rule compilers have been designed in the non-order-sorted framework in [25, 24, 30, 40, 28, 35, 31]. A compiler for OBJ can be based on ideas proposed in [23]. This should provide a gain in efficiency of at least an order of magnitude. However, compiling equalities like associativity and commutativity is still open, as well as compiling subsort information.

Type system. Currently, OBJ handles overloading, subsorts, and polymorphic operators defined in parametrized modules. But true polymorphic operators are reduced to the `if_then_else_` operator and to the built-in equality operator. An attractive framework for handling full polymorphism in algebraic specifications has been proposed by P. Moses [41], but has not been implemented yet.

Handling predicates. Enhancing the power of OBJ by providing a true logic-programming language is surely desirable [15]. This goal is contingent on the discovery of complete linear strategies for first-order Horn-clause calculus with equality. Promising work in this area should lead to interesting applications [34], carrying over the semantically clean features of OBJ (sorts, subsorts, parametrized modules) to the logic-programming world.

Object-oriented programming. The ground theory for OBJ has been extended to cope with states as they appear in object-oriented programming [16]. This work should lead to a solid basis for integrating the object-oriented programming approach within the logic programming approach.

Verification tools. OBJ-3 implements order-sorting rewriting rather than many-sorted rewriting via a functorial transformation as in OBJ-2. As a consequence, new tools will be developed for checking properties in order-sorted theories, for example order-sorted completion [9, 8, 48, 7], order-sorted unification [32, 39], and order-sorted inductive completion. This work is currently under way.

Architecture. A last area of research is hardware design for executing OBJ programs. Among such attempts, let us point out the concurrent term-rewriting model of computation [14] for the Rewrite Rule Machine developed at SRI [49, 36], and the pattern-matching hardware at Stony Brook [44].

APPENDIX. A SPECIFICATION OF NATURALS, INTEGERS, AND RATIONALS

The example of the number hierarchy illustrates many features involved in the operational semantics. We give here a part of a specification developed in OBJ-3 by J.

Meseguer. The complete example, including complex numbers and quaternions, can be found in [21]. In this example, == and =/= denote built-in equality and disequality boolean functions available for each sort.

```
obj NAT is
   sorts Zero NzNat Nat.
   subsorts Zero NzNat < Nat.
   op 0 : -> Zero.
   op s_ : Nat -> NzNat.
   op p_ : NzNat -> Nat.
   op _+_ : Nat Nat -> Nat [assoc comm].
   op _*_ : Nat Nat -> Nat.
   op _*_ : NzNat NzNat -> NzNat.
   op _>_ : Nat Nat -> Bool.
   op d : Nat Nat -> Nat [comm].
   op quot : Nat NzNat -> Nat [comm].
   op gcd : Nat NzNat -> NzNat [comm].

   var n m : Nat.
   var n' m' : NzNat.
   eq p s n = n.
   eq n + 0 =n.
   eq (s n) + (s m) = s s (n + m).
   eq n * 0 = 0.
   eq 0 * n = 0.
   eq (s n) * (s m) = s (n + (m + (n * m))).
   eq 0 > n = false.
   eq n' > 0 = true.
   eq (s n) > (s m) = n > m.
   eq d(0, n) = n.
   eq d(s n, s m) = d(n, m).
   eq quot(n, m') = if ((n > m') or (n == m'))
                    then s quot(d(n,m'),m')
                    else 0 fi.
   eq gcd(n', m') = if (n' == m')
                    then n'
                    else (if n' > m'
                          then gcd(d(n',m'),m')
                          else gcd(n',d(n',m')) fi) fi.
endo

obj INT is
   protecting NAT.
   sorts NzInt Int.
   subsorts Nat < Int.
   subsorts NzNat < NzInt < Int.

   op -_ : Int -> Int.
   op -_ : NzInt -> NzInt.
   op _+_ : Int Int -> Int [assoc comm].
```

```
    op -*- : Int Int -> Int.
    op _*_ : NzInt NzInt -> NzInt.
    op quot : Int NzInt -> Int.
    op gcd : NzInt NzInt -> NzNat [comm].
    vars i j : Int.
    vars i' j' : NzInt.
    vars n' m' : NzNat.
    eq - - i = i.
    eq - 0 = 0.
    eq i + 0 = i.
    eq m' + (- n') = if n' == m'
                     then 0
                     else (if n' > m'
                           then - d(n',m')
                           else d(n',m') fi) fi.
    eq (- i) + (- j) = - (i + j).
    eq i * 0 = 0.
    eq 0 * i = 0.
    eq i * (- j) = - (i * j).
    eq (- j) * i = - (i * j).
    eq quot(0,i') = 0.
    eq quot(- i',j') = - quot(i',j').
    eq quot(i',-j') = - quot(i',j').
    eq gcd(- i',j') = gcd (i',j').
jbo
obj RAT is
    protecting INT.
    sorts NzRat Rat.
    subsorts Int < Rat.
    subsorts NzInt < NzRat < Rat.
    op _/_ : Rat NzRat -> Rat.
    op _/_ : NzRat NzRat -> NzRat.
    op -_ : Rat -> Rat.
    op -_ : NzRat -> NzRat.
    op _+_ : Rat Rat -> Rat [assoc comm].
    op _*_ : Rat Rat -> Rat.
    op _*_ : NzRat NzRat -> NzRat.
    vars i' j' : NzInt.
    vars r q : Rat.
    vars r' q' : NzRat.
    eq r / (r' / q') = (r * q') / r'.
    eq (r / r') / q' = r / (r' * q').
    cq j' / i' = quot (j', gcd(j', i')) / quot(i',gcd(j',i'))
                    if gcd(j',i') =/= s 0.
    eq r / s 0 = r.
    eq 0 / r' = 0.
    eq r / (- r') = (- r) / r'.
```

```
    eq - (r / r') = (- r) / r'.
    eq r + (q / r') = ((r * r') + q) / r'.
    eq r * (q / r') = (r * q) / r'.
    eq (q / r') * r = (r * q) / r'.
jbo
```

This is the place to express our appreciation to J. Goguen, who invited us to SRI in 1983-84 and 1985-86. We would also like to acknowledge our collaboration with J. Meseguer and T. Winkler. Parts of this paper are strongly inspired by joint papers [33, 13]. It is an extended version of [29].

REFERENCES

1. Cavenaghi, C., De Zanet, M., and Mauri, G., MC-OBJ: A C Interpreter for OBJ, Technical Report, Dip. Scienze dell'Informazione, Univ. di Milano, 1987.
2. Clocksin, W. and Mellish, C., *Programming in Prolog*, Springer-Verlag, 1981.
3. Coleman, D., Gallimore, R., and Stavridou, V., The Design of a Rewrite Rule Interpreter from Algebraic Specifications, *IEEE Trans. Software Eng.*, July 1987, pp. 95-104.
4. Ehrig, H. and Mahr, B., *Fundamentals of Algebraic Specification* 1. *Equations and Initial Semantics*, EATCS Monogr. Theoret. Comput. Sci. 6, Springer-Verlag, 1985.
5. Futatsugi, K., Goguen, J., Jouannaud, J.-P., and Meseguer, J., Principles of OBJ-2, in:B. Reid (ed.), *Proceedings 12th ACM Symposium on Principles of Programming Languages*, Assoc. for Computing Machinery, 1985, pp. 52-66.
6. Futatsugi, K., Goguen, J., Meseguer, J., and Okada, K., Parameterized programming in OBJ-2, in: Robert Balzer (ed.), *Proceedings of Ninth International Conference on Software Engineering*, IEEE Computer Soc. Press, Mar. 1987, pp. 51-60.
7. Ganzinger, H., Order-Sorted Completion: The Many-Sorted Way, in: *Proceedings International Joint Conference on Theory and Practice of Software Development: Colloquium on Software Engineering*, Springer-Verlag, Lecture Notes in Comput. Sci. 351, 1989 pp. 244-258.
8. Gnaedig, I., Kirchner, C., and Kirchner, H., Equational Completion in Order-Sorted Algebras, *Theoret. Comput. Sci.* 72, pp. 169-202, (1990).
9. Gnaedig, I., Kirchner, C., and Kirchner, H., Equational Completion in Order-Sorted Algebras, in: M. Dauchet and M. Nivat (ed.), *Proceedings of the 13th Colloquium on Trees in Algebra and Programming*, Lecture Notes in Comput. Sci. 299, Springer-Verlag, 1988, pp. 165-184.
10. Gogolla, M., Algebraic Specifications with Partially Ordered Sorts and Declarations, Interner Bericht FB-169, Abteilung Informatik, Univ. of Dortmund, 1983.
11. Goguen, J., Higher-Order Functions Considered Unnecessary for Higher-Order Programming, in: *University of Texas, Year of Programming, Institute on Declarative Programming*, Addison-Wesley, 1988.
12. Goguen, J., Parameterized programming, *IEEE Trans. Software Eng.* SE-10(5): 528-543 (Sept. 1984).
13. Goguen, J., Kirchner, C., Kirchner, H., Megrelis, A., Meseguer, J., and Winkler, T., An Introduction to OBJ-3, in: J.-P. Jouannaud and S. Kaplan (eds.) *Proceedings 1st International Workshop on Conditional Term Rewriting Systems*, Lecture Notes in Comput. Sci. 308, Springer-Verlag, June 1988, pp. 258-263: also as internal report CRIN: 88-R-001.
14. Goguen, J., Kirchner, C., and Meseguer, J., Concurrent Term Rewriting as a Model of Computation, in: R. Keller and J. Fasel (editors.), *Proceedings of Graph Reduction Workshop*, Springer-Verlag, 1987, pp. 53-93.
15. Goguen, J. and Meseguer, J., EQLOG: Equality, Types, and Generic Modules for Logic Programming, in: Douglas DeGroot and Gary Lindstrom (eds.), *Functional and Logic*

Programming, Prentice-Hall, 1986, pp. 295–363. An earlier version appears in *J. Logic Programming* (2): 179–210 (Sept. 1984).

16. Goguen, J. and Meseguer, J., Extensions and Foundations for Object-Oriented Programming, in: Bruce Shriver and Peter Wegner (eds.), *Research Directions in Object-Oriented Programming*, MIT Press, 1987, pp. 417–477. Preliminary version in *SIGPLAN Notices* 21 (10): 153–162 (Oct. 1986); Technical Report CSLI-87-93, Center for the Study of Language and Information, Stanford Univ., Mar. 1987.
17. Goguen, J. and Meseguer, J., Order-Sorted Algebra I: Partial and Overloaded Operations, Errors and Inheritance, Technical Report, Computer Science Lab., SRI International, 1988; presented at Seminar on Types, Carnegie-Mellon Univ., June 1983.
18. Goguen, J. and Meseguer, J., Order-Sorted Algebra Solves the Constructor-Selector, Multiple Representation and Coercion Probglem, in: *Proceeding of the Second Symposium on Logic in Computer Science*, IEEE Computer Soc. Press, 1987, pp. 18–29.
19. Goguen, J., Meseguer, J., and Plaisted, D., Programming with Parameterized Abstract Objects in OBJ, in: *Theory and Practice of Software Technology*, North-Holland, 1983, pp. 163–193.
20. Goguen, J. and Tardo, J., An Introduction to OBJ: A Language for Writing and Testing Software Specifications, in: M.K. Zelkowitz (ed.), *Specification of Reliable Software*, IEEE Press, 1979, pp. 170–189, reprinted in: N. Gehani and A. McGettrick (eds.), *Software Specification Techniques*, Addison-Wesley, 1985, pp. 391–420.
21. Goguen, J. and Winkler, T., Introducing OBJ 3, Technical Report SRI-CSL-88-9, SRI International, 333 Ravenswood Ave., Menlo Park, CA 94025, Aug. 1988.
22. Goguen, J. A., Jouannaud, J.-P., and Meseguer, J., Operational Semantics for Order-Sorted Algebra, in: W. Brauer (ed.), *Proceeding of the 12th International Colloquium on Automata, Languages and Programming*, Lecture Notes in Comput. Sci., 194, Springer-Verlag, 1985, pp. 221–231.
23. Heuillard, T., Compiling Condition Rewriting Systems, in: S. Kaplan and J.-P. Jouannaud (eds.), *Proceedings of 1st International Workshop on Conditional Term Rewriting Systems*, Lecture Notes in Comput. Sci. 308, Springer-Verlag, 1987, pp. 111–128.
24. Hoffmann, C. M. and O'Donnell, M. J., Programming with Equations, *Trans. Programming Languages and Systems* 1(4): 83–112 (1982).
25. Huet, G. and Levy, J.-J., Computations in Non-ambiguous Linear Term Rewriting Systems, Technical Report, INRIA Laboria, 1979.
26. Isakowitz, T. and Gallier, J., Congruence Closure in Order-Sorted Algebra, Technical Report, Univ. of Pennsylvania, June 1988.
27. Isakowitz, T. and Gallier, J., Rewriting in Order-Sorted Equational Logic, in: R. Kowalski and K. Bowen (eds.), *Proceedings of Logic Programming Conference*, MIT Press, 1988.
28. Johnsson, T., Target Code generation from G-Machine Code, in: J. H. Fasel and R. M. Keller (eds.), *Graph Reduction*, Lecture Notes in Comput. Sci. 279, Springer-Verlag, 1987, pp. 119–159.
29. Jouannaud, J.-P., Kirchner, C., Kirchner, H., and Megrelis, A., OBJ: Programming with Equalities, Subsorts, Overloading and Parameterization, in: *Proceedings of the 1st International Workshop on Algebraic and Logic Programming*, Akademie-Verlag, Nov. 1988, pp. 41–52; Report CRIN 88-R-148.
30. Kaplan, S., A Compiler for Conditional Term Rewriting Systems, in: P. Lescanne (ed.), *Proceedings Second Conference on Rewriting Techniques and Applications*, Lecture Notes in Comput. Sci., 256, Springer-Verlag, 1987, pp. 25–41.
31. Kieburtz, R. B., The G-Machine: A Fast Graph-Reduction Evaluator, in: J.-P. Jouannaud (ed.), *Proceedings of Conference on Functional Programming Languages and Computer Architecture*, Lecture Notes in Comput. Sci. 201, Springer-Verlag, 1985.
32. Kirchner, C., Order-Sorted Equational Unification, presented at Fifth International Conference on Logic Programming, Seattle, Aug. 1988; also as Rapport de Recherche INRIA 954, Dec. 1988.

33. Kirchner, C., Kirchner, H., and Meseguer, J., Operational Semantics of OBJ-3, in: *Proceedings of 15th International Colloquium on Automata, Languages and Programming*, Springer-Verlag, 1988, pp. 287–301.
34. Kounalis, E. and Rusinowitch, M., On Word Problems in Horn Theories, in: E. Lust and R. Overbeek (eds.), *Proceedings 9th International Conference on Automated Deduction*, Springer-Verlag, 1988, pp. 527–537.
35. Laville, A., Lazy Pattern Matching in the ML Language, in: *Proceedings 7th Conference on Foundations of Software Technology and Theoretical Computer Science*, Lecture Notes in Comput. Sci., Springer-Verlag, Dec. 1987, pp. 400–419.
36. Leinwand, S. and Goguen, J., Architectural Options for the Rewrite Rule Machine, in: S. Kartashev and S. Kartashev (eds.), *Proceedings of Second International Supercomputing Conference*, Vol. I, International Supercomputing Inst., 1987, pp. 63–70.
37. Megrelis, A., A Logic of Semi-functions, Inclusion and Equality. The Setting, Technical Report 89-R-58, Centre de Recherche en Informatique de Nancy, 1989.
38. Meseguer, J. and Goguen, J. A., Initiality, Induction and Computability, in: M. Nivat and J. Reynolds (eds.), *Algebraic Methods in Semantics*, Cambridge U.P., 1985.
39. Meseguer, J., Goguen, J. A., and Smolka, G., Order-Sorted Unification, in: Unification, C. Kirchner (ed.) Academic Press, 1990, pp. 457–488.
40. Montanari, U. and Goguen, J., An Abstract Machine for Fast Parallel Matching of Linear Patterns, Technical Report SRI-CSL-87-3, Computer Science Lab., SRI International, May 1987.
41. Moses, P., Abstract Data Types as Lattices, Technical Report DAIMI IR-78, Computer Science Department, Aarhus Univ., 1988.
42. Poigné, A., Parametrization for Order-Sorted Algebraic Specification, *J. Comput. System Sci.*, to appear.
43. Poigné, A., Partial Algebras, Subsorting and Dependent Types. Prerequisites of Error Handling in Algebraic Specifications, in *Proceedings of Workshop on Abstract Data Types*, Lecture Notes in Comput. Sci. 332, Springer-Verlag, 1988, pp. 208–234.
44. Ramakrishnan, I., R2M: A Reconfigurable Rewrite Machine, in: *Proceedings of 2nd Workshop on Unification*, 1988.
45. Schmidt-Schauß, M., Computational Aspects of an Order-Sorted Logic with Term Declarations, Ph.D. Thesis, Univ. Kaiserslautern, Germany, 1987.
46. Smolka, G., Nutt, W., Goguen, J. A., and Meseguer, J., Order-Sorted Equational Computation, in: *Resolution of Equations in Algebraic Structures*, Vol. 2, pp. 297–367, 1989, H. Aït-Kaci and M. Nivant (eds.), Academic Press.
47. Sridhar, S., An Implementation of OBJ-2: An Object-Oriented Language for Abstract Program Specification, in: *Proceedings of Sixth Conference on Foundations of Software Technology and Theoretical Computer Science*, Lecture Notes in Comput. Sci. 241, Springer-Verlag, 1986, pp. 81–95.
48. Waldmann, U., Semantics of Order-Sorted Specifications, Technical Report 297, Univ. Dortmund, Germany, 1989.
49. Winkler, T., Leinwand, S., and Goguen, J., Simulation of concurrent term rewriting, in: S. Kartashev and S. Kartashev (eds.), *Proceedings of Second International Supercomputing Conference*, Vol. I, International Supercomputing Inst., 1987, pp. 199–208.