Interner Bericht

A Term-Rewriting Perspective of Object Oriented Program Specifications

Carlos Loría-Sáenz

311/01

FACHBEREICH
INFORMATIK

UNIVERSITÄT
KAISERSLAUTERN

Postfach 3049 · D-67653 Kaiserslautern

A Term-Rewriting Perspective of Object Oriented Program Specifications

Carlos Loría-Sáenz [*][†]
Universität Kaiserslautern
Fachbereich Informatik
AG Formale Methoden und Deduktion
(Prof. Dr. J. Avenhaus)
Postfach 3049
67663 Kaiserslautern (Germany)
E-Mail: loria@informatik.uni-kl.de

311/01

[*]This work resumes some of the main results of a short visit of the author to Kaiserslautern supported by the *Deutschen Akademischen Austauschdienst* (DAAD), the *Instituto Tecnológico de Costa Rica* (ITCR) and the AG-Avenhaus of the *Universität Kaiserslautern.*

[†]Author's Home Address: *Instituto Tecnológico de Costa Rica, Depto. Computación, Apto. 159-7050, Cartago COSTA RICA*

Abstract

In this work we propose a set of term-rewriting techniques for modelling object-oriented computation. Based on symbolic variants of *explicit substitutions calculi*, we show how to deal with imperative statements like assignment and sequence in specifications in a pure declarative style. Under our model, computation with classes and objects becomes simply normal form calculation, exactly as it is the case in term-rewriting based languages (for instance the functional languages). We believe this kind of unification between functions and objects is important because it provides plausible alternatives for using the term-rewriting theory as an engine for supporting the formal and mechanical reasoning about object-oriented specifications.

1 Introduction

Object-oriented specifications help to capture the structure of a software system based on the description of its data relationships (for example: inheritance and aggregation) where the concept of *object state* plays a central role for modelling (the reactive usually time dependant) systems behavior.
On the other side, functional based specifications are more concerned with the transactional behavior (input-output relationship) of a system where the notion of *referential transparency* (and therefore state independence) is considered as fundamental for promoting a smooth platform for mechanical deductive reasoning.
Although these are different perspectives of system modelling, they are not opposite but complementary and therefore both are necessary to cope with the complexity of software modelling from both angles. So it is very natural and important to promote any effort in unifying them as a way to increase our ability of reasoning about complex software systems in a common and well sustained formal theory.
With this objective in mind, in this report we aim at describing a deductively based framework which we believe gives us a better understanding of the combination of the object-oriented specifications with the functional ones, represented in this case by the term-rewriting systems.
By using procedural extensions of rewriting calculi with explicit substitutions ([2, 19]) we achieve very intuitive representations of objects and gain evidence that both perspectives naturally meet in a simple computational model. Moreover, given that the combination can be achieved without forcing nor weakening the fundamental principles of both models, a fair possibility of employing the term-rewriting machinery for formally and mechanically reasoning about object-oriented software specifications is thereby opened. We are convinced that in this form a solid software verification theory and practice can be employed for increasing the confidence of software abstract models.

2 Scope and Related Work

This material corresponds to a first attempt of the author to model imperative features occurring in object-oriented languages by means of special classes of rewrite systems. We have tried to incorporate in our approach some for us new fields which are wide and deeply technical. Essentially, we have considered the theory of explicit substitutions ([2, 19]) and the π-calculus ([52, 59]) as our experimental candidates. For the characteristics of this work, we just covered a selected part of these fields, it is therefore quite possible that some important references are not included [1]. For the same reasons, we have restricted ourselves to present the material in a preliminary and intuitive form, only. Our goal was essentially to gain some experience and evaluate alternatives for answering our questions under narrow time constraints. We could not always go into the technical details in the way and depth that we had wanted, however we believe that our proposal is robust and useful enough and it points to directions where more development is possible. This includes, fundamentally, the formalization and integration with some previous work on rewrite systems with extra-variables ([8, 77] and the λ-calculus with explicit substitution, as we explain in this work. We see very promising possibilities for such a development that we may call **`rewrite systems with mutable variables`**.

Unfortunately, a very important part of our research could not be incorporated into this document for the mentioned limitations of time. This includes some ideas for combining the π-calculus with the mutable rewrite system profiled in this report (i.e. incorporating the concept of `process` in our calculus). These ideas permit considering concurrent object-oriented programming from a symbolic point of view, they are based on [5, 29]. We hope to be able to extend this material with those ideas in a forthcoming paper. However, we had to leave the main references in the bibliography for the interested reader only.

Without claiming it is an exhaustive list, let us briefly review some closely related work and sources: We can find several important attempts for introducing the imperative style in the lambda calculus, for the same reasons motivating this work. For instance, the λs calculus of [25] that extended the seminal work of [63]. These works are not based on explicit substitution, directly.
Another important previous approach to imperative calculus is [56] which uses a different presentation compared to ours but is grounded on the same principles. It shows applications to mutable ADTs but the encoding is not as explicit as it is in this work. Inheritance is not covered.
A more recent work is [46] which presents a general call-by-need calculus, that can be used for the imperative purposes, too. It is comparable with the explicit substitutions calculi but according to the authors it permits avoiding the problems of copying inherent to the formers

[1]For instance the very interesting work [23] contains a lot of references we have not considered for the lack of time, only.

(and our model, too). Modelling higher-order term-rewriting using de Bruijn encoding can be found in [58, 13].
A more elaborated work than ours is [18] with its ρ-calculus. The presentation of this calculus is rather particular, the lambda abstractions are replaced by arrow abstractions, leading to a computation style based on explicit non-deterministic pattern-matching. This calculus is used for denoting the operational semantics of the ELAN language.
Another well-known alternative to model state based computation without leaving the pure declarative style is, of course, the `Monad` pattern of [75]. In some general sense, we use this encoding technique in this work.
A calculus for modelling a subset of the JAVA semantics oriented to the formal verification of type soundness is presented in [38] following the lines of [74].

3 Structure of the Work

We have organized this work mainly in 4 sections. In sections 4 and 5 we present, in a very intuitive form, the concepts of objects and rewrite systems that we want to join in a symbolic framework. Here we use simple examples and a JAVA notation to enhance readability and comprehension. We discuss and expose the main requirements and advantages for such a combination. In section 6, we describe a calculus that simulates computation in the presence of imperative statements. This is necessary for the selected object-oriented style. In section 7, for reaching our imperative oriented purposes, we extend the standard $\lambda\sigma_{\Uparrow}$ calculus of [19](a symbolic version) with assignments and sequences (we do not include pattern-matching, however). Based on this extension, we represent classes and objects in the same direction as in [1, 23] but being more detailed with the handling of imperative features and maintaining the functional facet of the language and including explicit classes (however without considering any typing system nor sharing). As explained, we just study sequential computation. In section 8, we conclude by describing some open questions and solutions based on our approach.

4 Basic Object-Oriented Notions

Objects can be understood as symbolic representations of some *entities* occurring in a given domain of knowledge to be abstracted and modeled for software development purposes. They are characterized by means of *classes* which specify the attributes of each one of its members as well as relevant dependency relationships (structural and functional) with other classes of objects (see [1] for details on object foundations). To establish a point of reference, we want to illustrate some of the *facilities* commonly appearing in object-oriented languages. Our intention is to represent them by means of a special kind of term-rewriting systems, as we will explain later. For making the examples more readable we use a JAVA-like notation and some toy examples.

So let us first consider a quite simple specification representing a bank account as an object. As usual, we may want to make deposits and withdrawals of an account, so we specify methods allowing these operations.

```
class SimpleAccount {
    Money currentAmount = new Money(0);
    public Money consult() { return currentAmount; }
    public void deposit(Money m) {
        this.currentAmount = this.currentAmount.add(m);
    }
    public void withdraw(Money m) {
        this.currentAmount = this.currentAmount.sub(m);
    }
}
```

We are assuming the existence of the class `Money`, which is aggregated in `SimpleAccount`, having the required interface (`Money add(Money)`, `Money sub(Money)`, `boolean equals(Money)`. etc).
As observed, initially we do not worry about the consistency of the transactions. Among other things we employ well-known object-oriented patterns like internal states (represented here by the variable named `currentAmount`) which can be changed by assignment. We also note the explicit **self-reference** represented by the identifier `this`. In this case its use is unnecessary because any undefined reference in a method is resolved lexically with respect to the class, but we want to remark that self-reference is possible. Message sending is denoted by the dot operator, which is also used for variable accessing [2].

To complement this basic functionality, we may want to extend such a class, by means of inheritance, adding some more specific features like the validation of those transactions leading to a negative

[2] Additionally, the specification of the external access (visibility) for the class members is denoted by the **public** keyword or by its absence roughly meaning *horizontally private* according to the JAVA semantics; however we are not concerned about visibility at this moment.

account balance. For the sake of simplicity we just ignore any (wrong) transaction, in such a case we do not state any exception or similar abnormal condition. We may have the following inherited class of `SimpleAccount`:

```
class ValidatedAccount extends SimpleAccount {
    public void deposit(Money m) {
        if (m.isPositive()) super.deposit(m);
    }
    public void  withdraw(Money m) {
        if (m.isPositive() && m.lessEqThan(currentAmount))
        super.withdraw(m);
    }
}
```

Here we can observe a typical pattern of delegation and a method of overriding. Messages to objects of this class are filtered, validated and passed over to the parent object if they are acceptable. Important to us with respect to this facility is that message sending to the parent object is possible by referencing the variable **super**.
In some other place in our specification, finally, we may want to create and use some objects belonging to those classes. For instance in the following way:

```
class UserClass {...
    public void  testAccountClasses(){
        SimpleAccount sa = new SimpleAccount();
        sa.deposit(new Money(100));
        sa.withdraw(new Money(200));                    (*)
 It holds: sa.consult().equals( new Money(-100))
        ValidatedAccount va = new ValidatedAccount();
        va.deposit(new Money(100));
        va.withdraw(new Money(200));                    (**)
 It holds: va.consult().equals( new Money(100))
    } ...
}
```

We have marked two lines in the code by `(*)` and `(**)`. Here some assertions should statically hold when the method `testAccountClasses` reaches these points. Of course assertions about systems based on objects can be more complicated: for instance, we see that any object of class `SimpleAccount` (or of `ValidatedAccount`) may concurrently coexist (in an independent thread) with other objects interchanging messages with them. Hence, it is absolutely possible that some transactions incorrectly overlap in time leading to unsound states for that account. That such a possibility can never happen at run time of the system, represents something that we want to assure. Liveness conditions are also properties to be guaranteed. In general, we need methods of very high level for verification of specifications from different system perspectives and layers. However, somewhere in a lower-level we certainly need symbolic engines allowing us to manipulate terms representing programs and assertions about them. So in our simple example, we want to `simplify` some sequence of statements and formulae reducing them to **true** if they are valid with respect to some logical framework. See for instance the following (where '$\supset$' stands for *implies*).

$$\begin{aligned}(ValidatedAccount\, va = \mathbf{new}\; ValidatedAccount();\\ va.deposit(\mathbf{new} Money(100));\\ withdraw(\mathbf{new} Money(200)))\\ \supset\\ va.consult().equals(\mathbf{new} Money(100))\\ \rightarrow^{*} \mathbf{true}\end{aligned}$$

The symbol $\rightarrow^{*}$ denotes some simplification relation - that we will define later - for describing the computation (in some undetermined number of steps) of the Boolean expression to its normal form **true**, meaning that the assertion is valid. Our goal is to provide term-rewriting based alternatives for describing such a kind of operational simplication relation reflecting the object-oriented and procedural symbolic computation. That will require, among others, that a subexpression like **new** `Money(100)` or even `Money` itself has a symbolic representation and consequently a computable normal form.

Now we are arriving at the main question dealt with in this work: How can we operationalize object-oriented computation as a term-rewriting process? In the following we will try to answer that question.

Although we have presented an easy formulation with the help of this simple example, we do not want to give the, obviously naïve, impression that we have to deal with a simple problem in a more realistic scenario.

We already mentioned that object-oriented programs usually are carried out concurrently, therefore, an object-oriented reasoning framework must be able to handle temporal inference or something similar,

among many other conditions [3]. We also know that, in addition to consider dealing with assertions regarding the *local* behavior of objects as above, we have to require the capability to formally reason about the dynamic interaction between objects in the system under events and transitions, in order to guarantee any *global* property of the system as a whole. Specific frameworks like temporal logics can be used for this purpose as it is well known (see for instance [36]).
However, at the present time, we do not know about any approach integrating term-rewriting based inference with temporal reasoning, we do not aim in this work in postulating one [4]. However, we do believe that it is possible to formulate a basic kernel of rules for symbolic and mechanical manipulation of expressions containing objects and equational formulae. Such an *engine* should give operationality when we couple it to other forms of deductive verification for reactive systems in a way we are not yet able to make precise (for instance, considering approaches like [37, 20]). That remains an interesting question to be studied in a future work.

5 Basic Term-Rewriting Notions

Term-rewriting systems (TRS) are models of computation with a strong theoretical foundation which offers many useful practical methods and tools for supporting equational based reasoning (covering both theorem-proving and computation). We do not intend to introduce the formalism and its capabilities, we simply give an insight into the main ideas; we refer to [7, 22, 10] for more details.
TRSs give efficient operational support to *Abstract Data Types* (ADT) based methodologies (see [24] for details about the foundations of ADTs) which can be used for modelling and prototyping software systems from a declarative but also computable (executable) perspective.

Under such a paradigm, we specify our system requirements by means of sets of `rules` which describe the way that components of the system must produce and transform data. Rule systems can be organized in a modular form, facilitating its understanding and maintenance.

Symbolic terms are used for representing data and function application in a uniform manner. Computation under TRSs means re-

[3]Error-handling is of course another fundamental issue to take into consideration as well as very nasty things like 'pointers'.

[4]Pointers to such a kind of combination using conditional rewrite systems are presented in [48, 49].

duction of data terms to their normal (irreducible) forms of data. Theorem-proving is, under adequate circumstances, reduced to syntactic identity between normal forms, which permits considering computation and theorem-proving under the same mechanism. Many techniques are available for symbolically checking the existence of normal forms (guaranteed by the `termination` property) and uniqueness of normal forms, also called `confluence` of TRSs (the satisfaction of termination and confluence together is named `convergence`).
We finally remark that TRSs also allow us to apply equational techniques for reasoning about induction (and therefore recursion) as well as symbolic equation solving, what fundamentally serves to cover the logic programming style of computation.
As done in the former section, let us give some examples illustrating many of the theoretical concepts mentioned above.
Typically, we may specify operators describing algebraic properties of some data structure. For example the existence of a neuter element (id) and the associativity of the composition of functions ($\_ \circ \_$) over some domain. We may describe such a structure by the following set of rules:

$$\begin{aligned} f \circ id &\to f \\ id \circ f &\to f \\ (f \circ g) \circ h &\to f \circ (g \circ h) \end{aligned}$$

This corresponds to a first-order specification; as an example of deduction, one can easily prove that $(f \circ (id \circ g)) = id \circ (f \circ g)$ by normal form building (in each side of the equation we get the term $f \circ g$ as normal form). In such a case we say both sides are `joinable`.The validity of this decission procedure is guaranteed because the rule system can be shown to be convergent [5].

We can extend this system with a higher-order set of rules by specifying a `map` function as well as one fundamental property of it (we use the ML notation for lists).

$$\begin{aligned} map\, f\, [] &\to [] && (1) \\ map\, f (x :: L) &\to (f\, x) :: (map\, f\, L) && (2) \\ map\, id\, L &\to L && (3) \\ map\, (f \circ g)\, L &\to (map\, g\, (map\, f\, L)) && (4) \end{aligned}$$

[5]The termination property can be verified by applying the technique of *reduction orderings*. By checking that all the critical pairs (overlappings between left-hand sides of rules) are joinable we get a strong form of confluence named local confluence. This and termination together assure full confluence and therefore convergence.

As a way to establish some points of comparison and contrast with respect to objects, we can also reformulated our previous example of an account using a more formal ADT style (we employ an abstract notation with the obvious meaning):

```
ADT SimpleAccount
using Money
exporting newAccount, consult, deposit,withdraw
SIG
     newAccount :  Money → Account;
     deposit:  Money, Account → Account;
     withdraw:  Money, Account → Account;
     consult:  Account → Money;
RULES
     consult(newAccount(m))→ m;
     deposit(m, newAccount(n)) → newAccount(Money#add(m,n));
     withdraw (m, newAccount(n)) → newAccount(Money#sub(m,n));
END
```

In contrast to the object-oriented specification, we can see that there is no internal state in the `Account` ADT. We can define a computation by using this ADT and simulate the object-oriented specification that we had in the previous section, but we need to reformulate our `UserClass`. We define it in the following way:

```
class UserClass ...{
    public void  testAccountClasses(){
        SimpleAccount sa = new SimpleAccount();
        sa.deposit(new Money(100));
        sa.withdraw(new Money(200));
        ...
    } ...
}
```

The first two lines of the method `testAccountClasses` represent, roughly speaking, the following expression (where we assume a constructor `newMoney` in some given ADT `Money`).

$$
\begin{aligned}
&withdraw\ (newMoney(200),\\
&\qquad deposit\ (newMoney(100),\\
&\qquad\qquad newSimpleAccount(newMoney(0))
\end{aligned}
$$

We compute (a part of [6]) its normal form by proceeding step by step:

[6]We do not reduce with the `Money` ADT, for simplicity.

$$
\begin{array}{l}
\mathit{withdraw}(\mathit{newMoney}(200), \\
\qquad\qquad \mathbf{deposit}\ (\mathit{newMoney}(100), \\
\qquad\qquad\qquad\qquad \mathit{newSimpleAccount}(\mathit{newMoney}(0)))) \\
\rightarrow \\
\mathbf{withdraw}(\mathit{newMoney}(200) \\
\qquad\qquad \mathit{newSimpleAccount}(\mathit{add}(\mathit{newMoney}(100), \mathit{newMoney}(0))) \\
\rightarrow \\
\mathit{newSimpleAccount}\ (\mathit{sub}\ (\mathit{newMoney}(200), \\
\qquad\qquad\qquad\qquad (\mathit{add}(\mathit{newMoney}(100), \mathit{newMoney}(0)))
\end{array}
$$

In the general case, when computing with (unconditional) term-rewriting systems, we try to find rewritable positions (called redexes) in expressions by matching them against the left-hand side of some rule, let's say $l \rightarrow r$. A successful match yields a `substitution`, which is a mapping binding variables to expressions. Let us assume that the matching process computes a substitution σ [7]. Once having this σ, we rewrite the redex with the right-hand r of the rule, after having replaced its variables x by $\sigma(x)$. After that, the substitution disappears from the reduction context.
When interpreters of programming languages are developed however, substitutions are actually implemented as `environments` and, for rather obvious reasons of efficiency, we do not physically apply the substitution to r, but just remember the environment and look-up for the value of any bound variable only when it is needed.
Putting this in symbolic terms, instead of making a rewrite step of the form $\sigma(l) \rightarrow \sigma(r)$ in some context, we perform the step $l[\sigma] \rightarrow r[\sigma]$, where, in general $e[\sigma]$ denotes the expression e closed or surrounded by the environment σ (without applying it). Exactly, this idea of remembering the rewrite substitution as a part of the rewriting context leads us to employ explicit substitutions techniques, as we explain in the following. As we will see, this makes it possible to rewrite expressions which may mutate their environment, which is important for imperative computations.

Before going that step, we first want to illustrate a very plausible technique for relating state-based computation with term-rewriting systems which can be used as a start point in 'visualizing' object-oriented computation as a form of rewrite computation. For such a purpose, let us again recur to one of our previous examples with a minor modification:

[7]When the left-hand side of the rule has the form $f(x_1, \ldots, x_n)$ the substitutions simply binds the formal parameters with the actuals for any function call of f, that is, pattern-matching reduces to parameter passing.

```
class SomeClass ...{
    public Money anotherTestMethod(){
        SimpleAccount sa = new SimpleAccount();
        sa.deposit(new Money(100));
        sa.withdraw(new Money(200));
        return sa.consult(new Money(200));
    } ...
}
```

Taking a look at this class, we rapidly become seduced to express the operational behavior of **anotherTestMethod** by means of a **conditional rule**. In fact, ignoring the type decorations and considering *return* just a variable, we may represent this specification as follows [8] [9]:

$$
\begin{aligned}
anotherTestMethod() \to return \Leftarrow \quad & SimpleAccount() \to sa, \\
& sa.deposit(newMoney(100)) \to sa, \\
& sa.withdraw(newMoney(200)) \to sa \\
& sa.consult() \to return.
\end{aligned}
$$

This kind of rules is special for two reasons:

- Firstly, they have **conditions** which need to be tested (reduced) before the rewrite step can take place. This feature of allowing rule conditions is manageable, because there is a well-understood theory and practice of conditional systems as a natural extension of unconditional systems.
- Secondly, some variables appearing in the conditions do not occur in the function header (they are called **extra-variables** henceforth). Once again there is a way out: for special classes of rule systems, which are very similar to those exemplified above, we have a sound and effective machinery for reasoning about them (see [8, 77, 57] for instance).

However, looking more deeply at this specific case we face a new problem: some extra-variables bound by the rewrite substitution can mutate after a rule condition becomes evaluated (in the example: the variable *sa* will be updated several times during the reduction).
To deal with this problem, we propose that the 'comma' should be considered a binary left associative operator denoted by ($\_;\_$) that exactly corresponds to sequences in imperative programming languages.

[8]The symbol $\Leftarrow$ is the analogous to the symbol :-- of PROLOG

[9]Note that we permit that classes rewrite to terms, we will elaborate more on that soon.

Under this condition, the evaluation of a rule $l \to r \Leftarrow (c_1; c_2; \ldots; c_n)$ can be performed according to the following general **rewrite procedure with mutable variables**:

1. A matching substitution for the variables of l is computed as already explained above for unconditional rules.
2. Then the conditions of the rule must be reduced one by one from left to right; they are of the form

$$actualCondition; otherConditions$$

3. The condition *actualCondition* may create new variables or update old ones in the current rewrite substitution during its reduction. Its final result should be passed over to the *otherConditions* continuing the reduction in the same fashion. At any point, results must be pairs of the form

$$(returnValue, substitution)$$

   where *result* is the computed value and *substitution* is the updated environment [10] [11]. In this form the next condition in the sequence is able to receive the actual environment.
4. When we are done with the last condition, we passes over its result to the right-hand side of the rule yielding (probably after some further reduction) the definitive rewrite result.

Once again we clearly state the requirement that substitutions become a part of the rewrite context and need a corresponding mechanism for handling those rewrite contexts as symbolic expressions. Our proposal for such a procedure will be formulated appealing to explicit substitution calculi. However, such a procedure is actually independent of our way to describe the effect of methods in object-oriented programs by TRSs with mutable extra-variables. It also permits, more generally seen, an integration of classes and rewrite rules. In other words, we just represent a class definition as a rule, which may contain internal rules (with mutable variables) representing its variables and methods. For illustrating this idea, let us modify our last class in the following way:

```
class LastClass ...{
    SimpleAccount sa = new SimpleAccount();
```

[10] It can be observed that the encoding $(returnValue, substitution)$ resembles a programming style using `Monads`! ([75]).

[11] In section 6, we will denote this pair by $returnValue \ll substitution$

```
    public Money anotherTestMethod(){
        sa.deposit(new Money(100));
        sa.withdraw(new Money(200));
        return sa.consult(new Money(200));
    } ...
}
```

Our representation of this class as a rule system with internal rules looks like (remember that ; is an operator)

$LastClass() \to ($
$\quad SimpleAccount() \to sa;$
$\quad anotherTestMethod() \to return \Leftarrow$
$\qquad\qquad (sa.deposit(newMoney(100)) \to sa;$
$\qquad\qquad sa.withdraw(newMoney(200)) \to sa;$
$\qquad\qquad sa.consult() \to return);$
$)$

The variable *sa* is now global in the scope of the class scope. Now it is clear that a class rewrites to a term representing a rule system with mutable variables.In fact, we can simplify it a little bit by eliminating the variable **return**. We get

$LastClass() \to ($
$\quad SimpleAccount() \to sa;$
$\quad anotherTestMethod() \to sa.consult() \Leftarrow$
$\qquad\qquad sa.deposit(newMoney(100)) \to sa;$
$\qquad\qquad sa.withdraw(newMoney(200)) \to sa;$
$)$

Moreover, we can actually eliminate the conditions at all by treating them as a term:

$LastClass() \to ($
$\quad SimpleAccount() \to sa;$
$\quad anotherTestMethod() \to$
$\qquad\qquad (sa.deposit(newMoney(100)) \to sa;$
$\qquad\qquad sa.withdraw(newMoney(200)) \to sa;$
$\qquad\qquad sa.consult());$
$)$

Observe that, under this representation, we need to manipulate rule systems as symbolic terms in order to model objects and classes. In any case, it should now be clear that term-rewriting systems constitutes a plausible model for object-oriented computation, if we have a symbolic mechanism to cope with mutation and sequences. In the next section we focus our attention on this issue, we do not deepen further on term-rewriting with mutation of variables.

Before leaving this section however, let us finally remark that at the time we write this work, we do not know about other approaches for handling mutation by term-rewriting in the way we have just lined out. We believe it is a new idea for the operational semantics of `rules with mutable variables`. Obviously, large parts of the formalism still need to be formulated out in detail. Because of the lack of time, that task is beyond the scope of this report [12].

6 Explicit Substitution with Mutation

Reviewing the literature, we find that in the past decade, beginning with [2] a lot of work was developed for incorporating the metaconcept of substitution as a term in the language of the λ-calculus ([11]) leading to many variants of the original model (we refer to [9] for more pointers on this calculus and its applications). In general, such calculi are based on the `de Bruijn` encoding, where the variables are replaced by indices representing their scoping depth in expressions [13]. One of the main purposes for such an effort is to avoid the problems of the α-conversion (variable renaming) necessary for assuring lexical scoping during symbolic computations. Another important problem is to study time and complexity of the parameter-passing process, that is the β-reduction, as it is also mentioned in [9].
Explicit substitution calculi are actually term-rewriting systems which model composition of substitutions, index handling and β-reduction for an extended syntax of the λ-terms that includes the (already mentioned) notation $e[s]$ and the substitutions in the language. Because the variables become numbers, those rewrite systems implicitly use arithmetic over natural numbers in a built-in form. The references show many applications of this technique for transforming higher-order specifications and problems to first-order ones (for instance higher-order unification or unified computation under higher-order term-rewriting systems among others). A central problem thereby is to obtain well-behaved calculi, that is, those satisfying properties like confluence, termination and completeness and efficient simulation

[12]In this sense, a very special question is to find a declarative meaning for the concept of `state` from an pure equational point of view, when state is no longer a 'metaterm' is, but a valid term according to the signature of the equational specification.

[13]That is, the number of λs occurring in the path going from the root of the term to the variable (which exactly coincides with the index in the evaluation stack of the expression for the any binding for that variable).

of the λ-calculus ([42]).
Since we evidently perceive a theory very adequate for our intentions, we apply the explicit substitution approach to our purposes. However, despite the computational problems associated with the α-conversion, we do not use the de Bruijn index notation. The reason is that rule systems working over this encoding contain rules of a lower-level than that we need for explaining symbolic computation. For the sake of clarity, we do not want to go to such a level of abstraction (at least not in this work), we stay on the pure symbolic level and pay the cost of α-conversion.
For our formulation we will essentially follow [19], adapting it to our needs [14]. Our term rewrite system for assignment and sequence can be shown to be confluent modulo an equivalence (which, among others, depends on α-conversion).
In some aspects, we have to make some simplifications to cope with the time limitations and complexity constraining this work. Probably the most strong one will be that we do not care about typing in an explicit form. However we think that our constructions can be extended in this sense without big problems because they represent standard and well understood features in typed programming languages that we simply code symbolically. Our concept of assignment has always the meaning of update (set), because the look-up function never fails. It is quite possible to modify the calculus for handling definition and updating separately, in an natural form. Another uncontemplated element is the incorporation of pattern-matching.

Before going into more technical details, we want to present the intensional meaning of our symbolism, which we think is not very complicated to infer. Essentially, our problem is to reduce (in the sense of the λ-calculus) some expression e which may have free variables. These are however recorded in an environment and e must have access to this environment. The proposal of the explicit substitution field consists in encoding the terms in the form $e[s]$ where $\_[\_]$ is a binary operator and s is the substitution representing the enclosing environment. All the calculi stem from this principle, that basically works for us too.
However, we want to permit assignment and sequence in our language, hence we need to represent something more. When we see a term like $e[s]$, we do not know whether it is evaluated or not. Neither whether s is the substitution before or after the evaluation of e. For these obvious

[14]We will not give it an explicit name because we think there are too many already (see for instance [44]). We think it is just an imperative extension of the $\lambda\sigma_{\Uparrow}$ of Curien et al with names.

reasons we introduce an additional notation $e \ll s$ [15] for representing the second case (i.e. after evaluation) and leave $e[s]$ for the first case (i.e. before evaluation). So, the reduction relation $\rightarrow$ embedded in our calculus, takes expressions of the form $e_0[s_0]$ and reduce them to expressions of the form $e_n \ll s_n$ where e_n is irreducible and free of $\ll$-operators. Symbolically

$$e_0[s_0] \rightarrow^* e_n \ll s_n \tag{5}$$

We also require to formulate a reduction strategy in our calculus to simulate a left to right ordered reduction with propagation of results. So we have a rule like the following:

$$(M; N)[s] \rightarrow (M[s]); N \tag{6}$$

In simple words, for evaluating $(M; N)[s]$ we first need need to evaluate $M[s]$. As expected there will be counterparts for expressing the propagation from left to right. We also have another rule of the form

$$((X \ll s); N) \rightarrow N[s] \tag{7}$$

It represents the case, where the leftmost expression in a sequence was reduced to $(X \ll s)$ where X is a variable and s is the resulting environment. In such a case, we ignore X (according to the standard meaning of `sequence`) and continue reducing N in context of s.
The calculus, as usual, contains a basic kernel of rules for representing the look-up in environments as well as the necessary operations for handling substitutions (composition and updating). Our syntax is the following

Definition 6.1 *(TERM+SUBST)*

```
TERM     ::=  VAR | (TERM) | TERM TERM |
              λ VAR.TERM |
              TERM[SUBST] |
              TERM ≪ SUBST |
              VAR ≫ SUBST |
              VAR = TERM |
VAR      ::=  X | Y | Z | ...
SUBST    ::=  id | (SUBST) | ⇑_VAR^TERM(SUBST) |
              SUBST ∘ SUBST |
              SUBST || SUBST |
```

[15]$(\_ \ll \_ : TERM, SUBST \rightarrow TERM))$

We usually use the letters M, N for denoting generic terms, s for substitutions, X, Y, Z (upper and lowercased) for variables, we write parenthesis only to solve ambiguities. Sometimes we use non-curried notations. Symbols like $+$ and numbers are used, it is intended that they are some special variables (we will never bind numbers, however). We denote an assignment by $X = M$ (the other forms was already introduced). The symbol id, as we already have above, is the identity substitution, and $\circ$ stands for composition. The expression $\Uparrow_X^N (s)$ denotes the substitution obtained by inserting the binding of variable X with N at the top of the substitution s. We write $s_1 \parallel s_2$ for expressing the substitution resulting from inserting/overriding all the bindings in s_1 into s_2. We write $\Uparrow_x^N$ as short hand for $\Uparrow_x^N (id)$. It represents a single binding. We also can eliminate the parenthesis in substitutions and write them as a sequence of bindings. For instance, instead of $\Uparrow_x^1 (\Uparrow_y^2 (id))$ we may simply write $\Uparrow_x^1 \Uparrow_y^2$.
The expression $X \gg s$ is used for denoting the term which results from looking-up X in s. The standard $X[s]$ returns the bound value as well as its environment, as we see below. We assume that this operations never fails, for simplicity [16].

Now we can add a formal semantics to this notation. The following set of rules expresses the basic meaning of variable look-up and the manipulation of environments; we denote this semantics in a rewrite relation that we denote by $\rightarrow_\sigma$.

Definition 6.2 *($\rightarrow_\sigma$ rewrite system)*

$$\begin{array}{rcll}
id \circ s & \rightarrow_\sigma & s & (8) \\
(s_1 \circ s_2) \circ s_3 & \rightarrow_\sigma & s_1 \circ (s_2 \circ s_3) & (9) \\
\Uparrow_x^N (s_1) \circ s_2 & \rightarrow_\sigma & \Uparrow_x^{N[s_2]} (s_1 \circ s_2) & (10) \\
s \parallel id & \rightarrow_\sigma & s & (11) \\
id \parallel s & \rightarrow_\sigma & s & (12) \\
\Uparrow_X^N (s_1) \parallel s_2 & \rightarrow_\sigma & \Uparrow_X^N (s_1 \parallel s_2) & (13) \\
(s_1 \parallel s_2) \circ s_3 & \rightarrow_\sigma & (s_1 \circ s_3) \parallel (s_2 \circ s_3) & (14) \\
X \gg id & \rightarrow_\sigma & X & (15) \\
X \gg \Uparrow_X^N (s) & \rightarrow_\sigma & N & (16) \\
X \gg \Uparrow_Y^N (s) & \rightarrow_\sigma & X \gg s \;\; when\, Y \neq X & (17) \\
X[s] & \rightarrow_\sigma & (X \gg s) \ll s \;\; X\, variable. & (18) \\
(X \gg s_1)[s_2] & \rightarrow_\sigma & X[s_1 \circ s_2] & (19)
\end{array}$$

[16]As a consequence the variable assignment will be only considered as an updating operation.

$$(M[s_1])[s_2] \quad \rightarrow_\sigma \quad M[s_1 \circ s_2] \tag{20}$$
$$(MN)[s] \quad \rightarrow_\sigma \quad (M[s])N \tag{21}$$
$$(M \ll s_1)[s_2] \quad \rightarrow_\sigma \quad M[s_2] \ll (s_1 \circ s_2) \tag{22}$$
$$(M \ll s_1) \ll s_2 \quad \rightarrow_\sigma \quad M \ll s_2 \tag{23}$$
$$(X \ll s)N \quad \rightarrow_\sigma \quad X(N[s]) \tag{24}$$
$$M \ll \Uparrow_x^{(N \ll s_1)}(s_2) \quad \rightarrow_\sigma \quad M \ll \Uparrow_x^N(s_2) \tag{25}$$
$$M[\Uparrow_x^{(N \ll s_1)}(s_2)] \quad \rightarrow_\sigma \quad M[\Uparrow_x^N(s_2)] \tag{26}$$
$$(\lambda x.M)[s] \quad \rightarrow_\sigma \quad \lambda y.M[\Uparrow_x^y(s)] \ll s \;\; y\,fresh \tag{27}$$
$$((\lambda x.M) \ll s)N \quad \rightarrow_\sigma \quad (\lambda x.M)(N[s]) \tag{28}$$
$$X(N \ll s) \quad \rightarrow_\sigma \quad (XN) \ll s \tag{29}$$

As we mentioned above, the system $\rightarrow_\sigma$ actually represents a left-most computation; we see that its normal forms have the form $M \ll s$ where M is a variable or a lambda abstraction. We assume that every computation starts with an expression of the form $N[s]$ where N is a standard lambda term and s is any substitution. When the left-most normal form is calculated, a propagation to the right is performed in order to evaluate the rest of the expression. The system $\rightarrow_\sigma$ does not reduce beta-redexes; for such a purpose we define:

Definition 6.3 *($\rightarrow_\beta$)*

$$(\lambda x.M)N \quad \rightarrow_\beta \quad M[\Uparrow_x^N] \tag{30}$$

The rule 30 represents the beta-reduction without applying the substitution as it is standard by explicit substitution calculi. We see that according to the strategy which controls the $\rightarrow_\sigma$ reductions, it is always applied to a term reduced by the rule 27 [17]. System $\rightarrow_\sigma \cup \rightarrow_\beta$ represents an explicit substitution based calculus which does not alter its environments, we need to extend it with imperative statements for our purposes with objects.

We proceed by introducing a rule system for computing with assignment and sequence based on the σ-computation:

Definition 6.4 *($\rightarrow_\Delta$)*

$$(X = M)[s] \quad \rightarrow_\Delta \quad M[s] \ll (\Uparrow_X^{M[s]} \| \, s) \tag{31}$$
$$(M;N)[s] \quad \rightarrow_\Delta \quad M[s];N \tag{32}$$
$$(X \ll s);N \quad \rightarrow_\Delta \quad N[s] \tag{33}$$
$$((\lambda x.M) \ll s);N \quad \rightarrow_\Delta \quad N[s] \tag{34}$$

[17]Under this assumption, we avoid another variable renaming that in other case will be necessary.

A calculus with mutable variables is now simply defined by the following relation:

Definition 6.5 *($\rightarrow_{\sigma,\beta,\Delta}$)*

$$\rightarrow_{\sigma,\beta,\Delta} \quad = \quad \rightarrow_{\sigma} \cup \rightarrow_{\beta} \cup \rightarrow_{\Delta} \tag{35}$$

We usually drop the subindexes of $\rightarrow_{\sigma,\beta,\Delta}$ and write $\rightarrow$, simply. We now can compute some examples illustrating these rules. For instance, by proceeding with an innermost strategy we yield:

$$\begin{aligned}
(y = 3; (x = y)[\Uparrow_x^2])[\Uparrow_y^1] &\quad \rightarrow_{\Delta} && (36)\\
(y = 3; (y \ll (\Uparrow_x^y \| \Uparrow_x^2)))[\Uparrow_y^1] &\quad \rightarrow_{\sigma} && (37)\\
(y = 3; (y \ll \Uparrow_x^y))[\Uparrow_y^1] &\quad \rightarrow_{\sigma}^{+} && (38)\\
((y = 3)[\Uparrow_y^1]); (y \ll \Uparrow_x^y) &\quad \rightarrow^{+} && (39)\\
(3 \ll (\Uparrow_y^3 \| \Uparrow_y^1)); (y \ll \Uparrow_x^y) &\quad \rightarrow_{\sigma} && (40)\\
(3 \ll \Uparrow_y^3); (y \ll \Uparrow_x^y) &\quad \rightarrow_{\sigma} && (41)\\
(y \ll \Uparrow_x^y)[\Uparrow_y^3] &\quad \rightarrow_{\sigma} && (42)\\
(y[\Uparrow_y^3]) \ll ((\Uparrow_x^y) \circ (\Uparrow_y^3)) &\quad \rightarrow_{\sigma}^{+} && (43)\\
(3 \ll \Uparrow_y^3) \ll (\Uparrow_x^{3 \ll \Uparrow_y^3} \Uparrow_y^3) &\quad \rightarrow_{\sigma} && (44)\\
(3 \ll (\Uparrow_x^{3 \ll \Uparrow_y^3} \Uparrow_y^3) &\quad \rightarrow_{\sigma} && (45)\\
3 \ll (\Uparrow_x^3 \Uparrow_y^3) &\quad && (46)
\end{aligned}$$

Outermost, we get:

$$\begin{aligned}
(y = 3; (x = y)[\Uparrow_x^2])[\Uparrow_y^1] &\quad \rightarrow_{\sigma} && (47)\\
((y = 3)[\Uparrow_y^1]; (x = y)[\Uparrow_x^2]) &\quad \rightarrow_{\Delta,\sigma}^{+} && (48)\\
(3 \ll \Uparrow_y^3); (x = y)[\Uparrow_x^2] &\quad \rightarrow_{\Delta} && (49)\\
(x = y)[\Uparrow_x^2][\Uparrow_y^3] &\quad \rightarrow_{\sigma} && (50)\\
(x = y)[(\Uparrow_x^2) \circ (\Uparrow_y^3)] &\quad \rightarrow_{\sigma}^{+} && (51)\\
(x = y)[\Uparrow_x^{2 \ll \Uparrow_y^3} \Uparrow_y^3] &\quad \rightarrow^{+} && (52)\\
3 \ll \Uparrow_x^3 (\Uparrow_x^{2 \ll \Uparrow_y^3} \Uparrow_y^3) &\quad \rightarrow_{\sigma}^{+} && (53)\\
3 \ll (\Uparrow_x^3 \Uparrow_y^3) &\quad && (54)
\end{aligned}$$

As we can notice in this specific case, we get syntactic joinability. In general however, the fresh variables introduced by rule 30 and the copying generated by some rules like 10 forces us to introduce a definition of congruence (of a very technical nature) which takes into account substitutions and some explicit form of α-conversion when we compare terms. In the case of substitutions, we interpret them as mappings and compare them extensionally up to a set of variables W (which always contains those variables introduced by renaming during any reduction). In the case of variable renaming, we explicitly introduce the α-conversion as a part of the congruence, as we see in the following definition. In addition to that, we also permit equivalence according to the rules of look-up in the congruence ($\gg$-rules).

Definition 6.6 *($\leftrightarrow_{W,\alpha,\gg}$) Let W be a set of variables, $\leftrightarrow_{\gg}$ the congruence*[18] *induced by the rules 15, 16 and 17. We define $\leftrightarrow_{W,\alpha,\gg}$ as the congruence induced by the following inference system:*

$$\frac{y \in W}{\lambda x.M \leftrightarrow_{W,\alpha,\gg} \lambda y.M[\Uparrow^y_x]} \tag{55}$$

$$\frac{M \leftrightarrow_{\gg} N}{M \leftrightarrow_{W,\alpha,\gg} N} \tag{56}$$

$$\frac{X \in W \wedge (s_1 \leftrightarrow_{W,\alpha,\gg} s_2)}{X \gg s_1 \ll s_1 \leftrightarrow_{W,\alpha,\gg} X \gg s_2 \ll s_2} \tag{57}$$

$$\frac{\forall X(X \notin W) : X \gg s_1 \ll s_1 \leftrightarrow_{W,\alpha,\gg} X \gg s_2 \ll s_2}{s_1 \leftrightarrow_{W,\alpha,\gg} s_2} \tag{58}$$

$$\frac{M_1 \leftrightarrow_{W,\alpha,\gg} M_2,\ N_1 \leftrightarrow_{W,\alpha,\gg} N_2}{(M_1N_1) \leftrightarrow_{W,\alpha,\gg} (M_2N_2)} \tag{59}$$

$$\frac{M \leftrightarrow_{W,\alpha,\gg} N,\ s_1 \leftrightarrow_{W,\alpha,\gg} s_2}{(M \ll s_1) \leftrightarrow_{W,\alpha,\gg} (N \ll s_2)} \tag{60}$$

And so on for every possible term of our syntax.

Using this congruence and assuming that W contains all the fresh variables introduced for every reduction, we are able to state the following result:

Claim 6.1 *(Basic Confluence Properties)*

1. $\rightarrow_\sigma \cup \rightarrow_\Delta$ *is convergent modulo* $\leftrightarrow_{W,\alpha,\gg}$
2. $\rightarrow_\sigma \cup \rightarrow_\Delta \cup \rightarrow_\beta$ *is confluent modulo* $\leftrightarrow_{W,\alpha,\gg}$

[18]In every case, to be understood as the minimal congruence inductively defined over the term structure.

Proof. (Sketch) Part 1 can be verified by reduction orderings and critical pair joinability.We must notice that the λ-terms are to be considered simple algebraic terms, where λ is just another operator. Part 2 can be proven by essentially repeating the proof of Theorem 4.3.3 of [19]. The Lemma 4.3.1 must be adapted to consider $\leftrightarrow_{W,\alpha,\gg}$. In each case, we have to prove that $\leftrightarrow_{W,\alpha,\gg}$ is coherent ([7], Definition 4.1.4) with the other reductions, that is

$$((M \leftrightarrow_{W,\alpha,\gg} N) \wedge (M \rightarrow_{\sigma,\beta,\Delta} N')) \supset (N \downarrow_{\leftrightarrow_{W,\alpha,\gg}} N') \quad (61)$$

This can be done by induction on the term structure. []

With respect to termination, we find good reasons to conjecture the following.

Conjecture 6.1 *$\rightarrow_{\sigma,\beta,\Delta}$ terminates over every set of terms where $\rightarrow_\beta$ terminates.*

The relation $\rightarrow_\beta$ should terminate over every simply typed set of terms (for an instance), where we need to explicitly define typing in such a case for our language, however, as already mentioned, this was not performed in this work for the lack of time.

A very important observation to be made at this point is that reductions using $\rightarrow_{\sigma,\beta,\Delta}$ (independently of confluence) can produce dynamic bindings of free variables when these simultaneosly occur bound by a binder in the same expression. Handling this problem requires an adjustment of the rule system at the level of $\rightarrow_\sigma$ and $\rightarrow_\beta$ rules for protecting such a kind of variables before we evaluate the body of a λ-expression. The corresponding system is presented in the Appendix. As a side effect of this change, assignments to variables inside of the body of any lambda-abstraction are local to this body if they are not in its enclosing environment, as expected.

7 Modeling Classes and Objects

In a section 5 we intuitively showed how classes and objects can be represented by means of rule systems which may contain internal rules, being these their local data and methods. By using our rewrite machinery of explicit substitution we are now able to incorporate objects and classes to our language and formulate an operational semantics for such a symbolization. We will model objects and classes trying essentially to follow the description of section 4, that is as a mimic of

JAVA, in a very simplified form, of course. Our goal is just to state the basic principles.
We require an internal representation for classes, we have considered in this work two options: the JAVA-like notation and the term-rewriting notation. For instance, if we specify a class describing `cells` storing integer values, we could specify such a class as follows:
In the JAVA-style:

```
class Cell {
      Cell(int x){val=x;}
      public int val=0;
      public int set(int y){val=y;}
}
```

As a term-rewriting system with mutable variables we put:

```
Cell(x) → (x → val;
           set(y) → (y → val);
          )
```

These are very concrete syntax forms, we want a more independent and abstract representation that makes more evident that both actually meet in a common and more general symbolization. Under this representation, we will write the declaration of the class `Cell` as follows. This is the so-called `external` class representation:

```
Cell = class x. (
                val =x;
                set = λ y. (this.val = y);
                )
```

where **class** is a variable binder [19]. Because classes are terms like others, they have to be to evaluated; in such a case we encode the internal assignments of a class declaration by means of two substitutions,

- One for the locally declared variables: the `this` environment
- And the other for the inherited variables, that is the `super` environment (which is *id* when there is no parent class).

Under this encoding and once evaluated, the class `Cell` looks like follows, which corresponds to the so-called `internal` class representation [20]:

$$\varsigma x.\langle \mathbf{this} \ll (\Uparrow^{x}_{val}\Uparrow^{\lambda y.(\mathbf{this}.val=y;)}_{set}), \mathbf{super} \ll id\rangle \qquad (62)$$

[19]For simplicity we consider monadic classes only, the extension to several arguments should be straightforward.

[20]We can use a simpler (less verbose) representation ignoring the 'this' and 'super' tags. However, its use simplifies our class evaluation mechanism.

where ς is a binder like λ is and $\langle \_, \_ \rangle$ is a new operator that we will use for representing internally classes as well as objects. Classes become objects when they are juxtaposed with other terms. For example, from class **`Cell`**, we produce an object (a Cell which stores a 10), simply by writing:

$$(\varsigma x.\langle \mathbf{this} \ll (\Uparrow^{x}_{val} \Uparrow^{\lambda y.(\mathbf{this}.val=y;)}_{set}), \mathbf{super} \ll id\rangle)10 \tag{63}$$

In such a case, a so-called ς-conversion takes place as we will explain below.The result will be equivalent to:

$$\langle \mathbf{this} \ll (\Uparrow^{x[\Uparrow^{10}_{x}]}_{val} \Uparrow^{(\lambda y.(\mathbf{this}.val=y;))[\Uparrow^{10}_{x}]}_{set} \Uparrow^{10}_{x}), \mathbf{super} \ll id\rangle \tag{64}$$

Class inheritance is handled by introducing the (overloaded) $\_ \parallel \_$ operation, which takes two classes and merges its environments in the way we specify below. We permit the extension of a class that already was extended, that is some kind of multiple inheritance. However, conflicts are solved in favor of the 'oldest' parent as we will see.
We need to handle qualified reference, like the *this.X* and the *super.X* forms, for accessing object members. For this purpose, we consider these as a special term construction, we extend the $\rightarrow_{\sigma}$ system so that it can cope with this format in a very simple way, which results compatible with our object representation.Essentially, we will consider that $\langle \_, \_ \rangle$ is a special kind of multiple substitution. The $\cdot$ operator is the accessor to this substitution, we will introduce specific rewrite rules for $\cdot$ and $\langle \_, \_ \rangle$ expressions.
We first extend our syntax in the following way:

Definition 7.1 *(TERM+SUBST+CLASS)*

$$\begin{array}{lll}
\texttt{TERM} & ::= & \texttt{VAR} \mid \texttt{(TERM)} \mid \texttt{TERM TERM} \mid \\
 & & \lambda\texttt{VAR.TERM} \mid \\
 & & \texttt{TERM[SUBST]} \mid \\
 & & \texttt{TERM} \ll \texttt{SUBST} \mid \\
 & & \texttt{VAR} \gg \texttt{SUBST} \mid \\
 & & \texttt{VAR} = \texttt{TERM} \mid \\
 & & \texttt{TERM} \cdot \texttt{TERM} \mid \\
 & & \texttt{CLASS} \\
\texttt{CLASS} & ::= & \mathbf{class}\texttt{VAR.TERM} \mid \\
 & & \varsigma\texttt{VAR.SUBST} \mid \\
 & & \texttt{CLASS} \parallel \texttt{CLASS} \mid \\
\texttt{VAR} & ::= & \texttt{X} \mid \texttt{Y} \mid \texttt{Z} \mid \mathbf{this} \mid \mathbf{super} \mid \ldots \\
\texttt{SUBST} & ::= & \texttt{id} \mid \texttt{(SUBST)} \mid \Uparrow^{TERM}_{VAR}\texttt{(SUBST)} \mid \\
 & & \texttt{SUBST} \circ \texttt{SUBST} \mid \\
 & & \texttt{SUBST} \parallel \texttt{SUBST} \mid \\
 & & \langle\ \texttt{VAR} \ll \texttt{SUBST}\ ,\texttt{VAR} \ll \texttt{SUBST}\ \rangle
\end{array}$$

The following set of rules extends the definition 6.5. We denote it by $\rightarrow_{\sigma\varsigma}$ It does look long and complex, but it follows essentially the same (repetitive) pattern for handling qualified variables for every operator related with substitutions.

Definition 7.2 *($\rightarrow_{\sigma\varsigma}$: Extension to $\rightarrow_\sigma$)*

$$this \gg \langle this \ll s_1, super \ll s_2\rangle \rightarrow_\sigma \langle this \ll s_1, super \ll s_2\rangle \tag{65}$$

$$X \gg \langle this \ll s_1, super \ll s_2\rangle \rightarrow_\sigma X \gg s_1 \tag{66}$$

$$this.X \gg \langle this \ll s_1, super \ll s_2\rangle \rightarrow_\sigma X \gg s_1 \tag{67}$$

$$super.X \gg \langle this \ll s_1, super \ll s_2\rangle \rightarrow_\sigma X \gg s_2 \tag{68}$$

$$\langle this \ll s_1, super \ll s_2\rangle \circ s_3 \rightarrow_\sigma \langle this \ll (s_1 \circ s_3), super \ll (s_2 \circ s_3)\rangle \tag{69}$$

$$\Uparrow_X^M (s_0) \circ \langle this \ll s_1, super \ll s_2\rangle \rightarrow_\sigma \langle this \ll \Uparrow_X^{M[s_1]} (s_0 \circ s_1), super \ll s_2\rangle \tag{70}$$

$$\Uparrow_{this.X}^M (s_0) \circ \langle this \ll s_1, super \ll s_2\rangle \rightarrow_\sigma \langle this \ll \Uparrow_X^{M[s_1]} (s_0 \circ s_1), super \ll s_2\rangle \tag{71}$$

$$\Uparrow_{super.X}^M (s_0) \circ \langle this \ll s_1, super \ll s_2\rangle \rightarrow_\sigma \langle this \ll s_1, super \ll \Uparrow_X^{M[s_2]} (s_0 \circ s_2)\rangle \tag{72}$$

$$\langle this \ll s_1, super \ll s_2\rangle \parallel s_3 \rightarrow_\sigma \langle this \ll (s_1 \parallel s_3), super \ll (s_2 \parallel s_3)\rangle \tag{73}$$

$$\Uparrow_X^M (s_0) \parallel \langle this \ll s_1, super \ll s_2\rangle \rightarrow_\sigma \langle this \ll \Uparrow_X^M (s_0) \parallel s_1), super \ll s_2\rangle \tag{74}$$

$$\Uparrow_{this.X}^M (s_0) \circ \langle this \ll s_1, super \ll s_2\rangle \rightarrow_\sigma \langle this \ll \Uparrow_X^{M[s_1]} (s_0 \parallel s_1), super \ll s_2\rangle \tag{75}$$

$$\Uparrow_{super.X}^M (s_0) \parallel \langle this \ll s_1, super \ll s_2\rangle \rightarrow_\sigma \langle this \ll s_1, super \ll \Uparrow_X^{M[s_2]} (s_0 \parallel s_2)\rangle \tag{76}$$

Our next step is to extend the definition 6.4 for computing with classes and objects. The induced relation is denoted by $\rightarrow_{\Delta\varsigma}$. Just for the sake of simplicity we will assume that neither the evaluation of a class nor the combination of classes can produce any change in the environment enclosing the corresponding operation. The modification for dealing with that case is quite simple.

Let us first give an example, showing the way we combine classes which allow us to achieve inheritance. So, suppose that we want to extend the class `Cell` defined above, for adding another method `inc` which increments the internal `val`.In our language, we would write the following:

$$\begin{aligned}(\mathbf{class}\, x.(inc = \lambda y.\mathbf{this}.set(y + \mathbf{this}.val))) \parallel \\ (\mathbf{class}\, y.(val = y; set = \lambda x.\mathbf{this}.val = y))\end{aligned} \tag{77}$$

The class of the left hand side extends the class at its right. When a term like this one is going to be evaluated every component class is first evaluated independently. Then, the rules of $\rightarrow_{\Delta\varsigma}$ and $\rightarrow_{\beta\varsigma}$ defined below will appropriately combine the corresponding `this` and `super` substitutions for producing the new class.
Class and object creation, represented here by ς-conversion, is indeed a special form of β-conversion, that we have separated for simplicity. Consequently, we consider rules related with that task as an extension of the definition 6.3.

Definition 7.3 *($\rightarrow_{\beta\varsigma}$: Rules for class and object creation)*

$$\begin{aligned}&\varsigma x.\langle \mathbf{this} \ll s_1, \mathbf{super} \ll s_2\rangle)M \rightarrow_{\beta\varsigma} \\ &\qquad \langle \mathbf{this} \ll (s_1 \circ \Uparrow_x^M), \mathbf{super} \ll s_2\rangle\end{aligned} \tag{78}$$

$$\begin{aligned}&(\varsigma x.\langle \mathbf{this} \ll s_1, \mathbf{super} \ll s_2\rangle \parallel \varsigma y.\langle \mathbf{this} \ll s_3, \mathbf{super} \ll s_4\rangle)M \\ &\qquad\qquad \rightarrow_{\beta\varsigma} \\ &(\varsigma x.\langle \mathbf{this} \ll (s_1 \parallel (s_3 \circ \Uparrow_y^M), \mathbf{super} \ll s_2 \parallel (s_3 \circ \Uparrow_y^M))\rangle\end{aligned} \tag{79}$$

Rule 78 corresponds to the `new` operation of JAVA, it creates a new class instance: the class parameter x becomes instantiated with M in the **this** environment and an object is returned. On the other side, rule 79 yields a new class from the extension of two classes: we notice that an extended class can only produce objects when all its parameters are bound. The parameter of the `super` class will be instantiated as first [21]. In this rule we see that s_1 corresponds to the local (i.e. the `this`) state of the extended class. This class has previously inherited s_2 from some other class. The new class receives the s_3 state in its `this` state. Its `super` is formed from the old inherited state s_2 and the new s_3. The new class has indirectly access to s_4 because this one is contained in s_3. In each case, the class parameter y of the class is

[21] Corresponding roughly to a `super(M)` call in the constructor code in the JAVA style, for creating the parent instance.

initialized with the M value and stored in the corresponding environment.

Now, we are ready to introduce the rules for expressing the computation with objects. These include the transformation of external represented classes into internal represented classes, the assignment of object variables and object method calling.

Definition 7.4 *($\rightarrow_{\Delta_\varsigma}$: The extension of $\rightarrow_{\Delta}$)*

$$(\mathbf{class}\, x.M)[s] \rightarrow_{\Delta_\varsigma} \varsigma y.\langle (M;\mathbf{this})[\Uparrow_x^y (s)], \mathbf{super} \ll s\rangle \ll s \tag{80}$$

$$(\mathbf{class}\, x.M \parallel \mathbf{class}\, y.N)[s] \rightarrow_{\Delta_\varsigma} ((\mathbf{class}\, x.M)[s] \parallel (\mathbf{class}\, y.N)[s]) \tag{81}$$

$$(((\varsigma x.M) \ll s) \parallel ((\varsigma y.N) \ll s)) \rightarrow_{\Delta_\varsigma} ((\varsigma x.M) \parallel (\varsigma y.N)) \ll s \tag{82}$$

$$(O.X = M)[s] \rightarrow_{\Delta_\varsigma} O[s].(X = M) \tag{83}$$
$$(\langle s_1, s_2\rangle \ll s_3).(X = M) \rightarrow_{\Delta_\varsigma} \langle s_1, s_2\rangle.(X = M[s_3]) \tag{84}$$
$$(\langle s_1, s_2\rangle.(X = (M \ll s_3)) \rightarrow_{\Delta_\varsigma} \mathbf{this}[\Uparrow_X^M \parallel (\langle s_1, s_2\rangle) \circ s_3)] \tag{85}$$

$$((O.X)M)[s] \rightarrow_{\Delta_\varsigma} O[s].(XM) \tag{86}$$
$$(\langle s_1, s_2\rangle \ll s_3).(XM) \rightarrow_{\Delta_\varsigma} \langle s_1, s_2\rangle.(X(M[s_3])) \tag{87}$$

$$\begin{gathered}(\langle s_1, s_2\rangle.(X(M \ll s_3))\\ \rightarrow_{\Delta_\varsigma}\\ ((X \gg \langle s_1, s_2\rangle)M); (\mathbf{this}[(\langle s_1, s_2\rangle \circ s_3)])\end{gathered} \tag{88}$$

$$\begin{gathered}\varsigma\, x.\langle \mathbf{this} \ll s_1, \mathbf{super} \ll s_2\rangle[s_3]\\ \rightarrow_{\Delta_\varsigma}\\ \varsigma\, x.\langle \mathbf{this} \ll (s_1 \circ s_3), \mathbf{super} \ll (s_2 \circ s_3)\rangle \ll s_3\end{gathered} \tag{89}$$

$$(\varsigma x.M \parallel \varsigma y.N)[s] \quad \rightarrow_{\beta_\varsigma} \quad (\varsigma x.M)[s] \parallel (\varsigma y.N)[s]$$

$$\begin{gathered}\langle \mathbf{this} \ll s_1, \mathbf{super} \ll s_2\rangle[s_3]\\ \rightarrow_{\Delta_\varsigma}\\ \langle \mathbf{this} \ll (s_1 \circ s_3), \mathbf{super} \ll (s_2 \circ s_3)\rangle \ll s_3\end{gathered} \tag{90}$$

For an explanation, let us take a look at this definition above. The rule 80 starts the evaluation of a class in an environment s. Similarly, the rule 81 simply distributes the environment s to each class for its independent evaluation. Rules 83-85 represent the assignment to variables in objects. The mechanics consists of three cycles: First, O must be evaluated to get the referenced object; here X is the variable to modify, M is the value to assign. Once we get the object (rule 84) we continue evaluating M (rule 84). Finally in the last cycle, after having computed the value for M, we perform the assignment, using the fact that the objects are coded as special kinds of substitutions (rule 85). The rules for method calling (message sending) proceed analogously, they work in a three cycles phase, too. (rules 86-88).

Rule 89-90, are used to 'close' internal represented classes and objects with respect to some enclosing environment.

We have to remark that an explicit object update is required after invoking object operations because our language is based on copying. For instance, if we consider again our class `Cell` and we would want to create an object and update it we should have to proceed as follows:

$$\begin{array}{ll}
(& \mathit{Cell} = \mathbf{class}\, x.(\mathit{val} = x; \mathit{set} = \lambda x.(\mathbf{this}.\mathit{val} = x;)); \\
 & \mathit{aCell} = \mathit{Cell}\, 10; \\
 & \mathit{aCell} = \mathit{aCell}.(\mathit{set}\, 20); \\
 & \mathit{aCell}.\mathit{val}; \\
) &
\end{array}$$

To finish this section, we will reformulate our claims for the new rewrite relation with classes and objects.

Claim 7.1 *(Confluence Properties)*

1. $\rightarrow_{\sigma\varsigma} \cup \rightarrow_{\Delta\varsigma}$ *is convergent modulo* $\leftrightarrow_{W,\alpha,\gg}$
2. $\rightarrow_{\sigma\varsigma} \cup \rightarrow_{\Delta\varsigma} \cup \rightarrow_{\beta\varsigma} \cup \rightarrow_{\beta}$ *is confluent modulo* $\leftrightarrow_{W,\alpha,\gg}$

Proof. (Sketch) It is possible to reconstruct the proof of claim 6.1 because the extensions $\rightarrow_{\sigma\varsigma}$ do not create any divergence with respect to $\rightarrow_{\sigma}$. Furthermore, the $\rightarrow_{\beta}$ reduction (rule 30) is extended without producing overlappings, hence $\rightarrow_{\beta\varsigma} \cup \rightarrow_{\beta}$ remains confluent. Lemma 4.3.1 of [19] is used with this extended relation (parallelized). The definition of $\leftrightarrow_{W,\alpha,\gg}$ needs to be extended and requires some technical adjustments, because classes have parameter like the λ-abstractions. The α-conversion must be extended for considering the **class** and ς binders. []

8 Conclusions and Future Work

The techniques of explicit substitution and the term-rewriting systems are adequate and suitable for the symbolic representation and manipulation of object-oriented and functional specifications, as we have learned in this work. Based on the preliminary experience gained by this work, we detect a very promising field around this kind of models with applications of theoretical and practical nature, especiallly to the formal verification of systems. There are many possibilities of further development, we want to mention some of the most relevant, briefly. These are apart from the technical questions we have postulated at different places at this document.
In the first place, we need to make precise the way a symbolic mechanism (like ours) can support symbolic temporal reasoning. The author believes that some interesting ideas can be taken from [55, 37] for example.
In the second place, we need to consider specifications with concurrent objects. We have developed extensions of the ideas presented in this report for incorporating channel-based process intercommunication in the spirit of the π-calculus; for the reasons already explained, they could not be included in this report. Essentially, the idea we have explored is to include in our language the concept of process with its corresponding symbolic representation. This is a term with explicit environment and a channel for receiving messages from other processes (corresponding to the principles of the language ERLANG, [5]). The approach of [28, 29] is also very attractive due to the pattern-matching on multiple channels which makes the modelling of formal specifications in languages like SDL ([73, 30]) easier [22].
A third area of interest comprehends the incorporation of pointers and sharing in the specification in order to avoid the need to copy objects, which is implicit in our language ([23]). Our idea in this sense is to consider the address as a primitive object which can be assigned to variables like other terms. A last area worth to mention is the pure declarative facet, in terms of equational reasoning, of all this computational mechanism.

9 Acknowledgments

We thank Prof. Dr. Jürgen Avenhaus for accepting me as guest in Kaiserslautern during this research, for the support and the discus-

[22]Our approach however does not handle inheritance yet. It is recognized that inheritance and concurrency are problematic([27]).

sions about some of these ideas. We appreciate the support of Bernd Löchner, Martin Kronenburg and Dirk Zeckzer, too. We finally thank the DAAD-CONARE and the ITCR for the financial support.

10 Appendix: Dynamic Binding

The following changes must be applied to the rules of $\rightarrow_\sigma$ and $\rightarrow_\beta$ [23] for avoiding dynamic binding of quantified variables also occurring free outside their scope. The idea bases on a (new) update operation:

$$\oplus : SUBST, SUBST \rightarrow SUBST$$

which provides a mechanism for variable updating when leaving scoped expressions. In addition to that, quantified variables must be protected when a lambda-abstraction is closed, i.e. when it is converted into a closure (rule 92 below). We mark bound variables in the form $_X$ and adapt the reductions appropiately.
To accomplish these requirements we begin modifying the definition of $\rightarrow_\sigma$ as follows.

Definition 10.1 *In Definition 6.2, replace rules 23 and 27 by rules 91 and 92, respectively. Further, include the following new rules (94-102):*

$$
\begin{array}{rcll}
(M \ll s_1) \ll s_2 & \rightarrow_\sigma & M \ll (s_1 \oplus s_2) & (91)\\
(\lambda\, x.M)[s] & \rightarrow_\sigma & \lambda\, y.M[\Uparrow^{y}_{\_x}] \ll (\Uparrow^{x}_{x} \circ s)\; y\, new & (92)\\
x \gg \Uparrow^{M}_{\_x} (s) & \rightarrow_\sigma & M & (93)\\
s \oplus id & \rightarrow_\sigma & id & (94)\\
id \oplus s & \rightarrow_\sigma & s & (95)\\
\Uparrow^{M}_{X} (s_1) \oplus \Uparrow^{N}_{X} (s_2) & \rightarrow_\sigma & \Uparrow^{M}_{X} (s_1 \oplus s_2) & (96)\\
\Uparrow^{M}_{X} (s_1) \oplus \Uparrow^{N}_{Y} (s_2) & \rightarrow_\sigma & \Uparrow^{N}_{Y} (\Uparrow^{M}_{X} (s_1) \oplus s_2)\;\; X \neq Y & (97)\\
(s_1 \oplus s_2) \circ s_3 & \rightarrow_\sigma & (s_1 \circ s_3) \oplus (s_2 \circ s_3) & (98)\\
(s_1 \oplus s_2) \oplus s_3 & \rightarrow_\sigma & s_1 \oplus (s_2 \oplus s_3) & (99)\\
s_1 \oplus (s_2 \circ s_1) & \rightarrow_\sigma & s_2 \circ s_1 & (100)\\
s \oplus \Uparrow^{N}_{X} (s) & \rightarrow_\sigma & \Uparrow^{N}_{X} (s) & (101)\\
M \ll (\Uparrow^{N \ll s_1}_{X} (s_1) \oplus s_2) & \rightarrow_\sigma & M \ll (\Uparrow^{N}_{X} (s_1) \oplus s_2) & (102)
\end{array}
$$

Now, the $\rightarrow_\beta$ relation is simply redefined by the following way:

Definition 10.2 *(Scoped $\rightarrow_\beta$)*

$$(\lambda\, x.M)(N \ll s) \;\rightarrow_\beta\; M[\Uparrow^{N}_{\_x}] \ll s \qquad (103)$$

[23]We notice that similar adjustments are needed for the other binders *class* and ς.

The rules specifying $\oplus$, in combination with the definition above, avoid that locally defined variables (inside $\lambda x.M$) propagates beyond the λ-expression. Because a β-reduction can only proceed after the application of rule 92, we know that the original parameter was already marked and must be stored in s bound to its external term.

References

[1] M. Abadi and L. Cardelli. *A Theory of Objects.* Springer Verlag, 1996.

[2] M. Abadi, L. Cardelly, P-L. Curien, and J.-J. Lévy. Explicit substitutions. *Journal of Functional Programming,* 1(4):375–416, October 1991.

[3] Jim Alves-Foss, editor. *Formal Syntax and Semantics of JAVA,* volume 1523 of *LNCS.* Springer Verlag, 1999.

[4] Gregory R. Andrews. *Concurrent programming: principles and practice.* The Benjamin/Cummings Publishing Co., Redwood, CA USA, 1991.

[5] Joe Armstrong. Erlang- a survey of the language and its industrial applications. Technical report, Ericsson Telecommunications System Laboratory, September 1996. http://www.erlang.org.

[6] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in ERLANG.* Prentice Hall, 2nd edition, 1996.

[7] Jürgen Avenhaus. *Reduktionssysteme.* Springer Verlag, 1995.

[8] Jürgen Avenhaus and Carlos Loría-Sáenz. On conditional rewrite systems with extra-variables and deterministic logic programs. In *Proceedings of the 5th International Conference LPAR,* volume 822 of *LNAI.* Springer, 1994.

[9] Mauricio Ayala-Rincón and César Muñoz. Explicit substitutions and All That. Technical report, Group ULTRA, University of Edinburgh, July 2000.

[10] Franz Baader and Tobias Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

[11] H. P. Barendregt. *The Lambda Calculus: Its syntax and semantics.* North-Holland, 1984.

[12] G. Berry and Boudol. G. The Chemical Abstract Machine. *Theoretical Computer Science,* 96:217–248, 1992.

[13] Eduardo Bonelli, Delia Kesner, and Alejandro Ríos. A de Bruijn notation for higher-order rewriting. In L. Bachmaier, editor, *Proceedings of the 11th International Conference RTA 2000,* volume 1833 of *LNCS,* pages 62–79. Springer Verlag, July 2000.

[14] E. Börger, A. Cavarra, and E. Riccobene. An ASM semantics for UML Activity Diagrams. In Rus [68], pages 293–308.

[15] Kim B. Bruce, Luca Cardelli, and Benjamin C. Pierce. Comparing object encodings. *Information and Computation*, 155(1-2):108–133, 1999.

[16] Luca Cardelli and John C. Mitchell. *Operations on Records.* In Mitchell and Gunter [54], 1994.

[17] Horatiu Cirstea. Specifying authentication protocols using ELAN. In *Workshop on Modeling and Verification*, Besacon, France, December 1999.

[18] Horatiu Cirstea and Claude Kirchner. Introduction to the rewriting calculus. Technical Report 3818, INRIA, December 1999.

[19] Pierre-Luis Curien, Thérèse Hardin, and Jean-Jacques Lévy. Confluence properties of Weak and Strong Calculi of Explicit Substitutions. *Journal of the ACM*, 43(2):362–397, March 1996.

[20] Mads Dam and Lars åke Fredlund. On the verification of open distributed systems. In *Proceedings of the ACM Symposium on Applied Computing*, volume 28, pages 532–540. ACM Press, June 1998.

[21] Werner Damm and David Harel. LSC's: Breathing life into Messages Sequence Charts. In P. Ciancarini et al., editor, *3rd IFIP FMOODS*, pages 293–312. Kluwer-AP, 1999.

[22] N. Dershowitz and J.P. Jouannaud. *Rewrite Systems*, volume B, chapter 6, pages 243–320. Elsevier, 1990.

[23] Daniel Dougherty, Fréderic Lang, Pierre Lescanne, Luigi Liquori, and Kristoffer Rose. A generic object-calculus based on addressed term rewriting systems. Technical Report 1999-54, INRIA, December 1999.

[24] H. Ehrig and B. Mahr. *Fundamentals of algebraic specifications I*, volume 6 of *EATCS*. Springer-Verlag, 1985.

[25] Matthias Felleisen and Daniel P. Friedman. A calculus for assignment in higher-order languages. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages POPL 87*, pages 314–325. ACM Press, January 1987.

[26] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. *A programmer's reduction semantics for classes and mixins*, pages 241–269. Volume 1523 of Alves-Foss [3], 1999.

[27] Cédric Fournet and Georges Gonthier. A calculus of mobile agents. In *7th Conference on Concurrency Theory(CONCUR)*, volume 1119 of *LNCS*. Springer Verlag, 1996.

[28] Cédric Fournet and Georges Gonthier. The reflexive cham and the join-calculus. In *ACM Symposium POPL*, January 1996.

[29] Cédric Fournet and Luc Maranget. *The JOIN-Calculus language: Release 1.04*. INRIA, 1999.

[30] Magnus Fröberg. Automatic code generation from SDL to a declarative programming language. [69].

[31] David Gray. *Introduction to the formal design of real-time systems*. Applied Computing. Springer Verlag, 1999.

[32] Kevin Hammond and Greg Michaelson. *Research Directions in Parallel Functional Programming*. Springer Verlag, 1999.

[33] Michael Hanus and Christian Prehofer. Higher-order narrowing with definitional trees. *Journal of Functional Programming*, 9(1):33–75, 1999.

[34] David Harel. From play-in scenarios to code: an achievable dream. In Maibaum [45], pages 22–34.

[35] Robert Harper and Benjamin Pierce. A record calculus based on symmetric concatenation. Technical Report CMU-CS-90-157R, School of Computer Science CMU, July 1991.

[36] Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5), May 1997.

[37] Frank Huch. Verification of Erlang programs using abstract interpretation and Model-Checking. *ICFP 99*, pages 261–272, September 1999.

[38] Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *ACM OOPSLA*, Denver, Colorado, November 1999.

[39] T. Itu and A. Yonezawa, editors. *Theory and practice of parallel programming (TPPP)*, volume 907 of *LNCS*. Springer Verlag, November 1994.

[40] Thomas Kühne. *A Functional Pattern System for Object-Oriented Design*. Verlag Dr. Kovac, 1999.

[41] Frédéric Lang, Daniel Dougherty, Pierre Lescanne, and Kristoffer Rose. Addressed term rewriting systems. Technical Report 1999-30, INRIA, June 1999.

[42] Frédéric Lang and Pierre Lescanne. On Strong Normalization of Explicit Substitution Calculi. Technical Report RR-1999-37, INRIA, August 1999.

[43] Doug Lea. *Concurrent Programming in Java*. Java Series. Addison-Wesley, 1997.

[44] Pierre Lescanne. From $\lambda\sigma$ to $\lambda\upsilon$ a journey through calculi of explicit substitutions. In *Proceedings of the 21st ACM Symposium on Principles of Programming Languages POPL 94*, pages 60–69. ACM Press, January 1994.

[45] T. S. E. Maibaum, editor. *3rd International Conference Fundamentals approaches to software engineering FASE*, volume 1783 of *LNCS*. Springer Verlag, 2000.

[46] John Maraist, Martin Odersky, and Philip Wadler. The call-by-need Lambda calculus. *Journal of Functional Programming*, 8(3):275–317, May 1998.

[47] Stephen J. Mellor, Steve Tockey, Rodolphe Arthaud, and Philippe Leblanc. Software-platform-independent, precise action specifications for UML. In J. Bézivin and P.A. Muller, editors, *The Unified Model Language- Beyond the notation*, June 1998.

[48] José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1996.

[49] José Meseguer. Rewriting logic and Maude: Concepts and Applications. In L. Bachmaier, editor, *Proceedings of the 11th International Conference RTA 2000*, volume 1833 of *LNCS*, pages 2–26. Springer Verlag, July 2000.

[50] Robin Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice-Hall, New York, 1989.

[51] Robin Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2:119–141, 1992.

[52] Robin Milner. The polyadic π-calculus: A tutorial. In F. L. Bauer, W. Brauer, and H. Schichtenberg, editors, *Logic and Algebra of Specification*, volume 94 of *F NATO ASI*. Springer Verlag, 1993.

[53] Robin Milner. *Communicating and Mobile Systems: the π-calculus*. Cambridge Press, 1999.

[54] J. C. Mitchell and C Gunter, editors. *Theoretical Aspects of Object-Oriented Programming*. MIT Press, 1994.

[55] Ben Moszkowski. *Executing temporal logic programs*. Cambridge University Press, 1986.

[56] Martin Odersky, Dan Rabin, and Paul Hudak. Call-by-name, assignment and the Lambda Calculus. In *Proceedings of the 20th ACM Symposium on Principles of Programming Languages POPL 93*, pages 43–56. ACM Press, January 1993.

[57] Enno Ohlebusch. Transforming conditional rewrite systems with extra-variables into unconditional systems. In *Proceedings of the 6th LPAR*. Springer, 1999.

[58] Bruno Pagano. X.R.S : explicit reduction systems -a first-order calculus for higher-order calculi. In C. Kirchner and H. Kirchner, editors, *Automated deduction, CADE-15*, volume 1421 of *LNAI*, pages 72–87. Springer-Verlag, 1998.

[59] Joachim Parrow. An introduction to the π-calculus. Technical report, Depto. Teleinformatics, Royal Institute of Technology Sweeden, 2000. to appear in Handbook of Process Algebra, ed. Bergstra, Ponse and Smolka Elsevier 2000.

[60] Benjamin C. Pierce. Foundational calculi for programming languages. To appear in CRCR Handbook of Computer Science and Engineering, December 1995.

[61] Benjamin C. Pierce and David N. Turner. *Concurrent objects in a process calculus*, pages 187–215. Volume 907 of Itu and Yonezawa [39], November 1994. http://www.cis.upenn.edu/ bpierce/papers/index.html.

[62] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the π-calculus. CSCI 476, Indiana University, 1997.

[63] Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[64] Christian Prehofer. *Solving Higher-order equations: from logic to programming.* Progress in theoretical computer science. Birkhäuser, 1998.

[65] G. Reggio, E. Astesiano, and H. Hussmann. Analyzing UML active class and associated state machines- a lightweight formal approach. [45], pages 127–146.

[66] Didier Rémy and Jérome Vouillon. On the virtual (un)reality of virtual types (in preparation). Technical report, INRIA, March 2000. http://pauillac.inria.fr/ remy/work/virtual.html.

[67] John H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.

[68] T. Rus, editor. *8th International Conference Algebraic Methodology and Software Technology AMAST*, volume 1816 of *LNCS*. Springer, May 2000.

[69] A. Sarma and O. Faergemand, editors. *Proceeding of the 6th SDL-Forum*, Darmstadt, Germany, October 1993.

[70] S. Doaitse Swierstra, Pedro R. Henriquez, and Jose N. Oliveira, editors. *Advanced Functional Programming*, volume 1608 of *LNCS Tutorial.* Springer Verlag, 1998.

[71] Thommy Thorn. Programming languages for mobile code. Technical Report 1083, INRIA, France, March 1997.

[72] D. N. Turner. *The π-calculus: Types, polymorphism and implementation.* Lfcs, University of Edinburgh, 1995.

[73] Kenneth J. Turner, editor. *Using Formal Description Description Techniques.* John Wiley and Sons, 1993.

[74] David von Oheimb and Tobias Nipkow. *Machine-checking the JAVA specification. Proving type safety*, pages 117–156. Volume 1523 of Alves-Foss [3], 1999.

[75] Philip Wadler. The essence of functional programming. In *Talk at 19th POPL*, 1992.

[76] David Walker. Objects in the π-calculus. *Information and Computation*, 116:253–271, 1995.

[77] T. Yamada, A. Middeldorp, J. Avenhaus, and C. Loría-Sáenz. Logicality of conditional rewrite systems. *Theoretical Computer Science*, 236(1-2):209–232, March 2000.