Structuring Operational Semantics: Simplification and Computation

Eugenio Moggi[1]

DISI
Univ. di Genova
Genova, Italy

Abstract

The paper describes a language consisting of two layers, terms and computation rules, whose operational semantics is given in terms of two relations: simplification and computation. Simplification is induced by *confluent rewriting* on terms. Computation is induced by *chemical reactions*, like those in the Join-calculus. The language can serve as metalanguage for defining the operational semantics of other languages. This is demonstrated by defining encodings of several calculi (representing idealized programming languages).

Keywords: Operational Semantics, Confluent Rewriting, Multiset Rewriting.

Introduction

Monads are a tool for structuring the denotational semantics of programming languages, which identifies the semantics of *computational effects* with the choice of a monad (see [15]). Is there a similar way of structuring operational semantics, namely to separate computational effects from other programming language features?

Plotkin's Structural Operational Semantics (SOS) [19] is widely accepted as a common framework for describing operational semantics. However, its level of generality makes it difficult to identify common patterns and points of variation.

We propose a more disciplined approach to operational semantics, comparable approaches are the reduction semantics [25] and the Chemical Abstract Machine [1]. Our proposal for structuring operational semantics presents some analogies with the *monadic approach* for structuring denotational semantics (indeed it was conceived as a way to provide operational semantics to monadic metalanguages [16]). More specifically, there are two layers:

[1] Email: moggi@disi.unige.it

- the first is called simplification, and it amounts to confluent term rewriting
- the second (which models computational effects) is called computation, and is based mainly on multiset rewriting.

More concretely, our proposal can also be described as a combination of Kahl's Pattern Matching Calculus [13] and the reflexive chemical abstract machine for the Join-calculus of Fournet and Gonthier [7,8]. The paper is organized as follow:

- Section 1 recalls the *essence* of the monadic approach to denotational semantics, mainly to establish some analogies with what we do in an operational setting (see Section 4).
- Section 2 explains how we specify operational semantics in the context of this paper, namely through transition systems, provides some justifications for this choice, and mentions also known limitations.
- Section 3 (the main technical contribution of the paper) describes our approach for structuring operational semantics in two *layers*: simplification and computation.
- Section 4 demonstrates the generality of our approach, by describing encodings of several calculi (representing idealized programming languages).

Acknowledgments. I would like to thank Amr Sabry for discussions on a preliminary version of the proposal, and Barry Jay for comments on a complete draft.

1 The Monadic Approach in a Nutshell

The denotational semantics of a programming language PL consists in an interpretation of the language PL in some suitable mathematical structure. Like in Tarski's semantics the interpretation must be *compositional* (see [23]), i.e. the interpretation must assign meaning to *complete programs* as well as *program fragments*, and the meaning of a complex program fragment must be a function of the meanings of its parts. Without loss of generality we can assume that the mathematical structure used for the interpretation forms a category $\mathcal{C}$ (with suitable properties and structure), and that *program fragments* are interpreted by morphisms in $\mathcal{C}$.

Usually, the same category $\mathcal{C}$ is used for the denotational semantics of different programming languages. Therefore, one can introduce a metalanguage ML with a given interpretation in $\mathcal{C}$, and replace the interpretation $[\![-]\!]_{PL}$ of PL in $\mathcal{C}$ with a translation $(-)_{PL}$ of PL in ML. The metalanguage provides a *language abstraction* for $\mathcal{C}$, which hides irrelevant details, while allowing to recover the interpretation $[\![-]\!]_{PL}$ by composing the translation $(-)_{PL}$ with the interpretation of ML in $\mathcal{C}$.

In the context of denotational semantics [15] proposed monads as a way of modeling *computational types*:

> ... to interpret a programming language in a category $\mathcal{C}$, we distinguish the object A of values (of type A) from the object MA of computations (of type A), and take as denotations of programs (of type A) the *elements* of MA. In particular, we identify the type A with the object of values (of type A) and obtain the object of computations (of type A) by applying an unary type-constructor M to A. We

call M a *notion of computation*, since it abstracts away from the type of values computations may produce.

At the level of metalanguages, the most significant consequence of this observation is the extension of ML with an abstract datatype constructor for a *notion of computation*. The resulting *monadic metalanguage* ML_M has an interpretation in $\mathcal{C}$, which extends the interpretation of ML, and is parameterized in the choice of a monad. Moreover, ML_M can be used as target for translating a programming language, often resulting in simpler and more understandable translations.

2 What Kind of Operational Semantics?

The most widely accepted approach for describing operational semantics is Plotkin's Structural Operational Semantics (SOS) [19]. In this approach and operational semantics is given by inference rules for deriving *operational judgments*. There is no prescribed format for operational judgments (indeed the approach is described through examples), but they should include some syntactic components (corresponding to program fragments), and inference rules should include some pattern matching and transformation of these syntactic components.

However, in most cases one can adopt operational judgments of a more specific format, namely $s \Longrightarrow s'$ or $s \overset{l}{\Longrightarrow} s'$. The resulting operational semantics amounts to the definition of a transition system (TS) or a labeled transition system (LTS). These two formats are widely used in the context of process calculi:

- A transition system $(S, \Longrightarrow)$, where S is a set and $\Longrightarrow \subseteq S \times S$, is suitable for describing the possible evolutions of a *closed system*, i.e. a system does not interact with an *external* environment. An $s \in S$ represents a *state* of the closed system at a given time, while the transition $s \Longrightarrow s'$ says that the system in state s may evolve (in one step) to state s'.
- A labeled transition system $(S, L, \Longrightarrow)$, where S and L are sets and $\Longrightarrow \subseteq S \times L \times S$, is suitable for describing the potential interaction of an open system with its environment. A *label* $l \in L$ specifies the kind of interaction between the open system and its environment, an $s \in S$ represents a *state* of the open system at a given time, and a transition $s \overset{l}{\Longrightarrow} s'$ says that the system in state s may interact with the environment (as specified by l) and evolve to state s' (provided the interaction has occurred).

It is always possible to combine an open system with its environment and obtained a closed system. On the other hand, given a closed system there could be several ways of decomposing it in two parts, or the closed system could be so *entangled* that there is *no way* to decompose it in two parts.

Although we have convincingly argued that a labeled transition system (describing an open system) can be subsumed by a transition system (for a more complex closed system), there are limitations to their expressiveness. For instance, transition systems cannot describe continuous (or hybrid) systems, whose configuration

evolves continuously over time, nor stochastic systems, that have a probability distribution over the set of transitions. Perhaps these systems could be described by a generalization of transition systems based on the functorial operational semantics proposed by [24]. However, for the purpose of this paper we take transition systems as the canonical format for describing operational semantics.

TS versus LTS.

This discussion is an aside on the trade-offs between LTS and TS, and provides further evidence for choosing TS as the preferred format for describing operational semantics. In general an operational (or denotational) semantics should support reasoning about programs. In this respect, a natural question is when a *program fragment* can be replaced by another program fragment without changing the *observable behavior* of the system. In process calculi a lot of work has been devoted in identifying suitable *observational equivalences*. There is a multitude of equivalences that have been proposes, and no clear best choice. However, a general guidelines is that one should seek a *congruence* of open terms (i.e. the counterpart of program fragments) and this congruence should depend on a set of simple *observations*.

The work on process calculi has used both labeled and unlabeled transition systems for defining operational semantics. LTS are suitable for describing the potential interaction of an open system with its environment. Constructs for composing open systems are likely to have a semantic counterpart at the level of LTS. However, work on Higher-Order π-calculus [20] has shown that operational semantics based on TS are preferable for specifying observational equivalences. In fact, observations on a closed system (observations can be defined as semi-decidable predicates on states) can be kept fairly simple, like Milner and Sangiorgi's notion of barb, even when the interactions between components are so complex that cannot be satisfactorily modeled by an LTS. We refer to [8] for the discussion and definition of several equivalences (for the Join calculus) and detailed comparisons among equivalences based on TS and LTS semantics.

3 General Approach

We have taken transition systems as the canonical way of specify an operational semantics. Our approach for structuring operational semantics blends two well-established tools:

- confluent term rewriting and its generalizations, such as combinatory reduction systems [14]
- multiset rewriting, in particular we borrow from the work of Berry and Boudol on the Chemical Abstract Machine [1] and the Join calculus [7,8]

We consider transition systems where a state $s \in S$ is a multiset of *terms* and *computation rules*, and we call these multisets *configurations* (in the chemical analogy computation rules are reaction rules and configurations are chemical solutions):

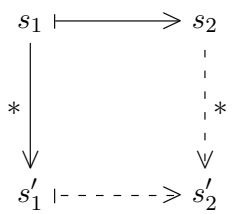
- the multiset of terms corresponds to the state of a closed system at a given time

- computation rules describe the potential evolutions of the system.

Computation rules capture the *computational* features of a programming language. In most cases they involve only multiset rewriting, i.e. replacing a multiset of terms with another multiset of terms, but they could also generate *fresh names* and activate new computation rules.

The *non-computational* features of a programming language are captured by a relation $\longrightarrow$ on terms called *simplification*, which is confluent and compatible, i.e. it can be applied in any order and in any context. Simplification embodies *referential transparency*, and suffices for defining the *operational semantics* of pure functional languages or typed calculi for proof assistants.

Simplification can be extended to configurations (in an obvious way), and we insist that "a computation step $\longmapsto$ is insensitive to further simplification", i.e.



This property is a further instance of *referential transparency*, which allows to ignore when and how simplification is done. In particular, computation rules are *insensitive* to the choice of simplification strategy, thus one can safely exploit techniques commonly used in implementations of pure functional languages [18], like lazy evaluation and graph reduction.

In the rest of this section we describe a specific transition system defined in terms of simplification and computation, while in the following section we exemplify its use for describing the operation semantics of several calculi.

3.1 Terms

We assume three basic syntactic categories (i.e. infinite sets) that are used in the definition of terms (and related notions):

- $a \in \mathsf{A}$ are atoms in the sense of FreshML [22] and FM-set theory [10], i.e. atoms can be permuted (but not substituted)
- $y \in \mathsf{XN}$ are name variables, which can be substituted with atoms (and other name variables)
- $x \in \mathsf{XE}$ are term variables, which can be substituted with arbitrary terms (including term variables).

The definition of terms and the auxiliary notions of names and patterns are given by the following BNF (in the sequel we consider ok and $fail$ as special atoms, and

the clauses $ok\ e$ and $fail$ as instances of $u\ \overline{e}$):

$$
\begin{array}{rl}
\text{Name} & u \in \mathsf{N} ::= a \mid y \\
\text{Pattern} & p \in \mathsf{P} ::= ?x \mid u\ \overline{p} \mid ?y\ \overline{p} \\
\text{Term} & e \in \mathsf{E} ::= x \mid u\ \overline{e} \mid ok\ e \mid fail \mid (p \Rightarrow e_1 | e_2) \mid \\
 & \qquad e_1 @ e_2 \mid e_1 : p \Rightarrow e_2 \mid (e_1; e_2) \mid \text{let } \{x_i = e_i | i \in n\} \text{ in } e
\end{array}
$$

Names u are either atom or name variables. We write $\mathrm{A}(u)$ and $\mathrm{FV}(u)$ for the set of atoms and the set of free (name) variables in u, respectively.

Patterns p are more expressive than those in functional languages and in the Pattern Matching Calculus (PMC) of [13] (e.g. patterns in PMC are given by the BNF $p ::= ?x \mid a\ \overline{p}$). For instance, we can define a term to test equality of atoms. Basically we extend the patterns of PMC with features typical of Linda's templates [11] (and related calculi, such as μKLAIM [2]):

- *Declared variables* are marked with ?. $?x$ matches any term e and binds it to x, $?y$ matches any name u and binds it to y.
- Name expressions u, thus also name variables y, are allowed in patterns. Therefore it is important to distinguish a free occurrence of y the declaration $?y$.

In μKLAIM's templates one can use any expression e, but these e *denote* elements of domains with a decidable equality (such as strings, integers, or names for localities).

There is a linearity constrain for well-formedness of patterns, namely a variable can be declared at most once in a pattern. The sets of atoms $\mathrm{A}(p)$, declared variables $\mathrm{DV}(p)$ and free (name) variables $\mathrm{FV}(p)$ are defined by induction on p

p	$\mathrm{A}(p)$	$\mathrm{DV}(p)$	$\mathrm{FV}(p)$
$?x$	$\emptyset$	$\{x\}$	$\emptyset$
$u\ \overline{p}$	$\mathrm{A}(u, \overline{p})$	$\mathrm{DV}(\overline{p})$	$\mathrm{FV}(u, \overline{p})$
$?y\ \overline{p}$	$\mathrm{A}(\overline{p})$	$\{y\} \cup \mathrm{DV}(\overline{p})$	$\mathrm{FV}(\overline{p}) - \{y\}$
$p\ \overline{p}$	$\mathrm{A}(p, \overline{p})$	$\mathrm{DV}(p, \overline{p})$	$\mathrm{FV}(p) \cup (\mathrm{FV}(\overline{p}) - \mathrm{DV}(p))$

The definition of A, DV and FV is extended to comma separated sequences of syntactic entities by point-wise union, for instance $\mathrm{A}(u, p) = \mathrm{A}(u) \cup \mathrm{A}(p)$.

The definition of $\mathrm{FV}(p)$ says that occurrences of y on the right of $?y$ are bound. We don't have compelling examples that exploit this feature. If one wish to forbid this binding, then FV should be defined as $\mathrm{FV}(?y\ \overline{p}) = \mathrm{FV}(\overline{p})$ and $\mathrm{FV}(p\ \overline{p}) = \mathrm{FV}(p, \overline{p})$.

Terms e are basically borrowed from the PMC of [13] and have the following informal semantics:

- $u\ \overline{e}$ is a constructor (name) applied to a sequence of terms
- $ok\ e$ denotes a successful term

- $fail$ denotes failure (e.g. of pattern matching)
- $(p{\Rightarrow}e_1|e_2)$ denotes a function which tries to match the argument against p, if that fails it applies e_2 to the argument
- $e_1@e_2$ is function application
- $e_1 : p{\Rightarrow}e_2$ tries to match e_1 against p, if successful it applies the matching substitution to e_2
- $(e_1; e_2)$ allows failure recovery, namely if e_1 fails, then it returns e_2
- let $\{x_i = e_i | i \in n\}$ in e allows mutually recursive definitions (the declared variables x_i must be distinct).

The sets of atoms $\mathrm{A}(e)$ and free variables $\mathrm{FV}(e)$ are defined by induction on e. The clauses for $\mathrm{A}(e)$ are straightforward, thus we give only the clauses for $\mathrm{FV}(e)$

e	$\mathrm{FV}(e)$
x	$\{x\}$
$u\ \overline{e}$	$\mathrm{FV}(u, \overline{e})$
$(p{\Rightarrow}e_1\|e_2)$	$\mathrm{FV}(p, e_2) \cup (\mathrm{FV}(e_1) - \mathrm{DV}(p))$
$e_1@e_2$	$\mathrm{FV}(e_1, e_2)$
$(e_1; e_2)$	$\mathrm{FV}(e_1, e_2)$
let $\{x_i = e_i \| i \in n\}$ in e_n	$(\cup\{\mathrm{FV}(e_i) \| i \in n+1\}) - \{x_i \| i \in n\}$
$e_1 : p{\Rightarrow}e_2$	$\mathrm{FV}(e_1, p) \cup (\mathrm{FV}(e_2) - \mathrm{DV}(p))$
$e\ \overline{e}$	$\mathrm{FV}(e, \overline{e})$

Examples. We give some examples of terms, the informal claims about their operational behavior rest upon the definition of simplification (see Section 3.2).

- Test for atom equality is defined as $eq \equiv (?y_1{\Rightarrow}(y_1{\Rightarrow}true|?y_2{\Rightarrow}fals e|fail)|fail)$, where $true$ and $fals e$ are some given atoms. The test enjoys the expected properties, namely for any atom a_1 and a_2 the term $eq@a_1@a_2$ simplifies to $true$ if $a_1 = a_2$, otherwise it simplifies to $fals e$.

  More precisely, $eq@a_1$ simplifies to $eq_1 \equiv (a_1{\Rightarrow}true|?y_2{\Rightarrow}fals e|fail)$, because a_1 matches $?y_1$. The simplification of $eq_1@a_2$ first tries to match a_2 against a_1, and if that fails then it matches a_2 against $?y_2$ (which does always succeed).

  In general eq could be applied to any pair of terms e_1 and e_2, and simplification of $eq@e_1@e_2$ could have two other outcomes:
  · it simplifies to $fail$ if a pattern matching fails, e.g. $a\ e$ fails to match $?y_1$
  · It gets stuck, because we cannot decide whether a term matches a pattern.
  The second possibility happens because simplification is defined on open terms (and patterns). For instance, we cannot decide whether variable x matches pattern $?y_1$, in fact different substitution instances of x yields different outcomes. Similarly, we cannot decide whether atom a matches pattern y, in fact different

substitution instances of y yields different outcomes.

- Lambda-abstraction $\lambda x.e$ can be defined as $(?x{\Rightarrow}e|fail)$, in this way β-reduction is decomposed in (two) simplification steps.
- Encoding of datatypes. In an untyped language types can be represented either as subsets of terms or as partial equivalence relations (PER) on terms (modulo simplification). We use atoms as constructors to form terms of the given datatype, while destructors can be defined by pattern matching and recursion.

  For instance, for the datatype N of natural numbers we use two atoms: one for zero $z : N$ and the other for successor $s : N \to N$. Moreover, the destructor $it : X \to (X \to X) \to N \to X$ can be defined as follows

$$\text{let } it = (\lambda x.\lambda x_f.(z{\Rightarrow}x \mid s\ ?x_n{\Rightarrow}it@x@x_f@x_n \mid fail)) \text{ in } \ldots$$

  One can easily check that it enjoys the *equational properties*
  · $it@e@e_f@z$ simplifies to e
  · $it@e@e_f@(s\ e_n)$ simplifies to $e_f@(it@e@e_f@e_n)$
  implied by the characterization of natural number as initial algebra.

3.2 Simplification

We define a relation $e \longrightarrow e'$ on terms (modulo α-conversion), called simplification, which is the analogue of β-reduction. Simplification is defined as the *compatible closure* of the *left-linear and non-overlapping* rewrite rules given in Figure 1, and it is directly inspired by reduction for the PMC of [13]), which decomposes pattern matching in a sequence of elementary steps.

Proposition 3.1 *Simplification enjoys the following properties:*

- *Preservation of atoms and free variables, i.e.*
  $e \longrightarrow e'$ *implies* $\mathrm{A}(e') \subseteq \mathrm{A}(e)$ *and* $\mathrm{FV}(e') \subseteq \mathrm{FV}(e)$
- Equivariance $\dfrac{e \longrightarrow e'}{e[\pi] \longrightarrow e'[\pi]}$
  where π is a permutation of atoms and $e[\pi]$ is the term obtained by permuting the atoms in e as specified by π
- Substitutivity $\dfrac{e \longrightarrow e'}{e[\rho] \longrightarrow e'[\rho]}$
  where ρ maps name/term variables to names/terms respectively, and $e[\rho]$ is parallel substitution with renaming of bound variables to avoid variable capture (therefore it is convenient to work with terms modulo α-conversion)
- Compatibility $\dfrac{e \longrightarrow e'}{C[e] \longrightarrow C[e']}$ *$C[-]$ term context with one hole*

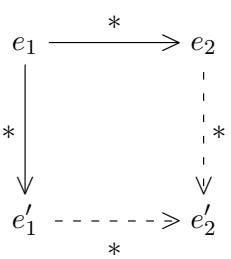
$$
\begin{array}{rcll}
(p{\Rightarrow}e_1|e_2)@e & \longrightarrow & (e : p{\Rightarrow}ok\ e_1; e_2@e) & \\
(ok\ e; e') & \longrightarrow & e & \\
(fail; e') & \longrightarrow & e' & \\
e :?x{\Rightarrow}e' & \longrightarrow & e'[x : e] & \\
u\ \overline{e} :?y\ \overline{p}{\Rightarrow}e' & \longrightarrow & \overline{e} : \overline{p}[y : u]{\Rightarrow}e'[y : u] & \text{when } |\overline{e}| = |\overline{p}| \\
a\ \overline{e} : a\ \overline{p}{\Rightarrow}e' & \longrightarrow & \overline{e} : \overline{p}{\Rightarrow}e' & \text{when } |\overline{e}| = |\overline{p}| \\
\text{let } \{x_i = e_i | i \in n\} \text{ in } e & \longrightarrow & e[x_i : \text{let } \{x_i = e_i | i \in n\} \text{ in } e_i | i \in n] & \\
v@e & \longrightarrow & fail & \text{when } v \not\equiv (p{\Rightarrow}e_1|e_2) \\
(v; e') & \longrightarrow & fail & \text{when } v \not\equiv ok\ e \mid fail \\
v :?y\ \overline{p}{\Rightarrow}e' & \longrightarrow & fail & \text{when } v \not\equiv u\ \overline{e} \text{ with } |\overline{e}| = |\overline{p}| \\
v : u_2\ \overline{p}{\Rightarrow}e' & \longrightarrow & fail & \text{when } v \not\equiv u_1\ \overline{e} \text{ with } |\overline{e}| = |\overline{p}| \\
a_1\ \overline{e} : a_2\ \overline{p}{\Rightarrow}e' & \longrightarrow & fail & \text{when } a_1 \neq a_2 \text{ and } |\overline{e}| = |\overline{p}|
\end{array}
$$

where $v ::= u\ \overline{e} \mid (p{\Rightarrow}e_1|e_2)$ ranges on terms that cannot change top-level structure by simplification or instantiation, and $\overline{e} : \overline{p}{\Rightarrow}e'$ is defined by induction on $|\overline{p}| = |\overline{e}|$

- ${\Rightarrow}e'$ is e'
- $e\ \overline{e} : p\ \overline{p}{\Rightarrow}e'$ is $e : p{\Rightarrow}(\overline{e} : \overline{p}{\Rightarrow}e')$

Fig. 1. Simplification Rules

- *Confluence, i.e.*



Simplification suffices to model most features of pure functional languages, including records and variants, but it fails to model generativity of datatypes (which requires *generation of fresh names*).

3.3 Computation Rules

Before defining the computation relation on configurations we introduce another layer of syntax for join patterns and computation rules, which builds on top of patterns and terms.

Join pattern	$J ::= \{(u_i\ \overline{p}_i \mid i \in n)\}$	a multiset of patterns $u\ \overline{p}$
Computation rule	$r ::= J > \nu\overline{y}.R \uplus E$	with $R \uplus E$ multiset of rules and terms

Join patterns specify trigger conditions, namely a computation rules with join pattern J can be activated only by a multiset E that is an *instance of* J.

The Join-calculus [7] has simpler join patterns, consisting of patterns of the form $u\ ?y_1 \ldots ?y_n$ (or $u\ ?\overline{y}$ for short) with certain linearity constrains. Comparable level of complexity for triggers can be found in the Kell calculus [3,21] and in KLAIM [6,2].

The sets of atoms $\mathrm{A}(J)$, declared variables $\mathrm{DV}(J)$ and free (name) variables $\mathrm{FV}(J)$ of J are defined by point-wise union, e.g. $\mathrm{DV}(\{(u_i\ \overline{p}_i \mid i \in n)\}) = \cup_{i \in n}\mathrm{DV}(u_i\ \overline{p}_i)$. We impose the following linearity constrain for well-formedness of join patterns: a term variable x can be declared at most once in J. In general, a name variable y can be declared in more than one pattern $u\ \overline{p}$ of J, but a multiset E of terms is an instance of J only if all occurrences of y in J are replaced by the same atom. For instance, $\{(\mathit{SendTo}\ a\ e_1, \mathit{GetFrom}\ a\ e_2)\}$ is an instance of $\{(\mathit{SendTo}\ ?y\ ?x_1, \mathit{GetFrom}\ ?y\ ?x_2)\}$, but $\{(\mathit{SendTo}\ a_1\ e_1, \mathit{GetFrom}\ a_2\ e_2)\}$ is not.

Computation rule $J > \nu\overline{y}.R \uplus E$ specifies the following potential evolution of the system: an instance of J is consumed, the name variables $\overline{y}$ are replaced by fresh atoms, and a suitable substitution instance of $R \uplus E$ is released.

The reaction rules $J > e$ of the Join-calculus appear simpler, but this is deceptive. In fact, the terms of the Join-calculus represent multisets of terms with parallel composition, and allow local declarations of reaction rules and fresh names. Therefore, computation rules can be recast as Join-calculus reaction rules (except for our join patterns, that are inherently more expressive). We consider a strength to have a stratification, where terms are defined independently from computation rules, and likewise simplification is defined independently from computation.

The sets of atoms $\mathrm{A}(r)$ and free variables $\mathrm{FV}(r)$ of r are defined as follows (where $\mathrm{A}(-)$ and $\mathrm{FV}(-)$ are extended to multisets by point-wise union)

- $\mathrm{A}(J > \nu\overline{y}.R|E) = \mathrm{A}(J, R, E)$
- $\mathrm{FV}(J > \nu\overline{y}.R|E) = \mathrm{FV}(J) \cup (\mathrm{FV}(R, E) - \overline{y} - \mathrm{DV}(J))$, i.e. the variables declared in J and the fresh names $\overline{y}$ are bound in $R \uplus E$.

Remark 3.2 The Join-calculus enjoys a *local property* convenient for a distributed implementation of the calculus. Namely, one can partition the set of atoms in such a way that: each reaction rule is located at a given element of the partition, and messages (i.e. terms of the form $a\ \overline{e}$) must move to a specific element of the partition (uniquely determined by a) in order to get involved in a reaction. We could impose constrains for well-formedness of computation rules, that ensure a similar local property. For the sake of simplicity, we refrain from doing this. However, all the examples of computation rules we consider satisfy these constrains.

The definition of simplification extends in the obvious way to computation rules and multisets $R \uplus E$. Moreover, Proposition 3.1 continues to hold if one consistently replaces terms with multisets, e.g. compatibility becomes

$$\text{Compatibility}\ \frac{R \uplus E \longrightarrow R' \uplus E'}{C[R \uplus E] \longrightarrow C[R' \uplus E']}\ C[-] \text{ multiset context with one hole}$$

Examples. We show how computation rules could be used to define an interpreter for operations on references. First, we fix some atoms (type annotations are used as informal explanation), e.g.

- $prg : MY$ labels programs (of computational type MY), i.e. $prg\ e$ means that e is a program
- $str : RX, X$ labels store cells, i.e. $str\ y\ v$ means that the reference y (of type RX) stores a value v (of type X)
- $new : X \to (RX \to MY) \to MY$ is the operation that takes a value v, creates a new reference y initialized with v, and then passes y to the rest of the program
- $get : RX \to (X \to MY) \to MY$ is the operation that takes a reference y, fetches the value v stored in y, and then passes v to the rest of the program

The following computational rules specify the interpretation of *new* and *get*

- $prg\ (new\ ?x\ ?k) > \nu y.prg\ (k@y),\ str\ y\ x$ (where , is multiset union)
- $prg\ (get\ ?y\ ?k),\ str\ ?y\ ?x > prg\ (k@x),\ str\ y\ x$

To show that the rules induce the expected behavior we need to set up a suitable configuration (see Section 3.4 and 4.1) including the rules, several programs and a shared store (i.e. a multiset of terms of the form $prg\ e$ and $str\ a\ e$).

3.4 Computation

We take as configurations closed multisets $R \uplus E$, i.e. $\mathrm{FV}(R, E) = \emptyset$. The computation relation $s \longmapsto s'$ on configurations is defined by the following rewrite rule

$$s \uplus J\rho \uplus J > \nu\overline{y}.R \uplus E \longmapsto s \uplus (R \uplus E)[\overline{y} : \overline{a}][\rho] \uplus J > \nu\overline{y}.R \uplus E \quad \text{where}$$

- ρ is a closed substitution, i.e. it maps name/term variables to atoms/closed terms
- $\overline{a}$ is a sequence of fresh atoms, i.e. not in $\mathrm{A}(s \uplus J\rho \uplus J > \nu\overline{y}.R \uplus E)$
- $J\rho$ is the *instantiation* of J with ρ, where $-\rho$ is defined on patterns and join

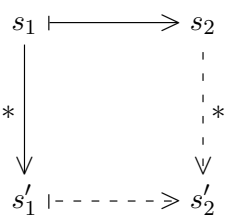
patterns as follows ($-\rho$ is always closed, when the substitution ρ is closed)

$-$	$-\rho$
$?x$	$e \quad$ where $e \equiv \rho(x)$
$u\ \overline{p}$	$u[\rho]\ \overline{p}\rho$
$?y\ \overline{p}$	$a\ \overline{p}\rho \quad$ where $a \equiv \rho(y)$
$p\ \overline{p}$	$p\rho\ \overline{p}\rho$
$\{(u_i\ \overline{p}_i \mid i \in n)\}$	$\{((u_i\ \overline{p}_i)\rho \mid i \in n)\}$

Remark 3.3 Computation on configurations makes use only of instantiation $-\rho$ and substitution $-[\rho]$ with closed ρ, therefore they do not require renaming of bound variables. This means that computation does not have to consider configurations modulo α-conversion. On the contrary, simplification makes use of substitution $-[\rho]$ with arbitrary ρ, therefore it requires renaming of bound variables. This means that simplification has to consider configurations modulo α-conversion, but some simplification strategies could make a more restrictive use of substitution.

Proposition 3.4 *Computation enjoys the following properties:*

- Equivariance $\dfrac{s \longrightarrow s'}{s[\pi] \longrightarrow s'[\pi]}$
- Extension $\dfrac{s_1 \longrightarrow s_2}{s \uplus s_1 \longmapsto s \uplus s_2}\ \mathrm{A}(s)$ *disjoint from* $\mathrm{A}(s_2) - \mathrm{A}(s_1)$
- *computation is preserved by simplification (Simulation), i.e.*



In general the computation relation is not confluent. Moreover a computation step cannot model a collective operation, such as a broadcast, that involve an arbitrarily large multiset of terms.

Transition System

At this point we have introduced all the ingredients to defines a transition system on the set S_c of configurations.

Definition 3.5 $TS_c = (S_c,\ =_c\Longrightarrow)$ is the transition system where the relation $=_c \Rightarrow$ is given by $\xrightarrow{*} \longmapsto$, i.e. $s =_c \Rightarrow s'$ provided one can go from s to s' by a finite sequence of simplification steps followed by one computation step.

Some simplification is essential to enable a computation step. In fact, through simplification a multiset E of terms could become an instance of the join pattern J that triggers a computation rule. Notice that there is no need to simply under the scope of a binder to make E become an instance of J.

Proposition 3.6 *The relation* $\stackrel{*}{\longrightarrow}$ *is a bisimulation relation on* TS_c*, i.e.*

(i) *if* $s_1 \stackrel{*}{\longrightarrow} s_2$ *and* $s_1 =_c\Longrightarrow s_1'$*, then exists* s_2' *s.t.* $s_2 =_c\Longrightarrow s_2'$ *and* $s_1' \stackrel{*}{\longrightarrow} s_{\cdot 2}$

(ii) *if* $s_1 \stackrel{*}{\longrightarrow} s_2$ *and* $s_2 =_c\Longrightarrow s_2'$*, then exists* s_1' *s.t.* $s_1 =_c\Longrightarrow s_1'$ *and* $s_1' \stackrel{*}{\longrightarrow} s_{\cdot 2}$

Proof. Easy consequence of confluence for $\longrightarrow$ and the simulation property. □

Therefore we can replace TS_c with an equivalent transition system, where states are equivalence classes of configurations modulo simplification and $S_1 =_C\Longrightarrow S_2 \stackrel{\Delta}{\Longleftrightarrow} s_1 \longmapsto s_2$ for some $s_1 \in S_1$ and $s_2 \in S_2$.

Proposition 3.7 *The relation* $s_1 \mathbf{R}_\pi s_2 \stackrel{\Delta}{\Longleftrightarrow} s_1[\pi] = s_2$ *is a bisimulation relation on* TS_c*, for any permutation* π *on atoms*

Proof. Easy consequence of equivariance for $\longrightarrow$ and $\longmapsto$. □

Therefore we can replace TS_c with an equivalent transition system, where states are equivalence classes of configurations modulo modulo permutations of atoms, i.e. $[s] = \{s[\pi] \mid \pi \text{ permutation}\}$. Often there is a set of atoms with a special meaning, e.g. for defining basic observations on configurations. In this case, one should consider only permutations that are the identity on this set of atoms.

4 Encodings

We show the expressiveness of the transition system TS_c by encoding in it abstract machines (i.e. small-step operational semantics) for existing calculi representing idealized programming languages. In each example of encoding first we specify

- the syntax PL of the calculus
- a transition system $TS_{PL} = (S_{PL}, =_{PL}\Longrightarrow)$, describing an abstract machine for PL, i.e. the states in S_{PL} involve programs in PL and usually some other stuff.

Then we define

- a *compositional* translation $(-)^*$ from PL to the set E of terms (in the monadic approach this corresponds to a translation from PL to a metalanguage ML_M)
- a translation $[\![-]\!]$ from S_{PL} to the set S_c of configurations, which extends the compositional translation $(-)^*$ (in the monadic approach this corresponds to choosing a monad for interpreting ML_M, more precisely the choice of computation rules corresponds to choosing a monad).

Finally, we shown that the translation $[\![-]\!]$ is a *good encoding*. In the ideal case one can define a lock-step bisimulation $\mathbf{R}$ between TS_{PL} and TS_c, i.e.

(i) if $s_1 \mathbf{R} s_2$ and $s_1 =_{PL}\Longrightarrow s_1'$, then exists s_2' s.t. $s_2 =_c\Longrightarrow s_2'$ and $s_1' \mathbf{R} s_2'$

(ii) if $s_1 \mathbf{R} s_2$ and $s_2 =_c\!\Longrightarrow s_2'$, then exists s_1' s.t. $s_1 =_{PL}\!\Longrightarrow s_1'$ and $s_1' \mathbf{R} s_2'$

and show that $s \mathbf{R} [\![s]\!]$ for every $s \in S_{PL}$. In other cases, when a transition in TS_{PL} is simulated by several transitions in TS_c, one has to use relations with weaker properties (e.g. weak bisimulations).

4.1 Monadic Metalanguage with References

We present an encoding for a monadic metalanguage with references, first described in [16]. The encoding is particularly simple, because the operational semantics of monadic metalanguages is defined in terms of simplification and computation. Other monadic metalanguages with different computational effects are described in [17], and should have similar encoding.

- The terms $M \in PL$ of the monadic metalanguage are given by the following BNF (where a ranges over atoms representing references)
  $M ::= x \mid \lambda x.M \mid M_1 @ M_2 \mid ret\ M \mid do\ M_1\ M_2 \mid new\ M \mid get\ M \mid set\ M_1\ M_2 \mid a$
- The states of the transition system TS_{PL} are triples $(\mu | M, S)$ where
  · $M \in PL_0$ is a closed term
  · $\mu : \mathsf{A} \overset{fin}{\rightarrow} PL_0$ is a *store*, i.e. a finite maps from references to closed terms
  · $S ::= none \mid push\ M\ S$ is a *control stack*, i.e. a stack of closed terms $M \in PL_0$.

  The transition relation $=_{PL}\!\Longrightarrow$ is $\overset{*}{\longrightarrow} \longmapsto$, where $\longrightarrow$ (simplification) is β-reduction and $\longrightarrow$ (computation) is defined by the following rules
  · $(\mu | do\ M_1\ M_2, S) \longmapsto (\mu | M_1, push\ M_2\ S)$
  · $(\mu | ret\ M_1, push\ M_2\ S) \longmapsto (\mu | M_2 @ M_1, S)$
  · $(\mu | new\ M, S) \longmapsto (\mu, a : M | ret\ a, S)$ with $a \in \mathsf{A}$ fresh
  · $(\mu | get\ a, S) \longmapsto (\mu | ret\ M, S)$ if $\mu(a) = M$
  · $(\mu, a : M' | set\ a\ M, S) \longmapsto (\mu, a : M | ret\ a, S)$

The compositional translation $(-)^*$ (which extends in the obvious way to control stacks) is basically the identity, if we identify $\lambda x.e$ with $(?x \Rightarrow e | fail)$ and consider the term constructors *ret*, *do*, ...to be atoms. $[\![(\mu | M, S)]\!]$ is the configuration $\{(str\ a\ M^* | \mu(a) = M)\} \uplus prg\ M^*\ S^* \uplus R$, where *str* and *prg* are atoms (for representing stores and program threads) and R is the set containing the following computation rules (which are in one-one correspondence with the rules above)

- $prg\ (do\ ?x_1\ ?x_2)\ ?x_S > prg\ x_1\ (push\ x_2\ x_S)$
- $prg\ (ret\ ?x_1)\ (push\ ?x_2\ ?x_S) > prg\ (x_2 @ x_1)\ (push\ x_S)$
- $prg\ (new\ ?x)\ ?x_S > \nu y.prg\ (ret\ y)\ x_S,\ str\ y\ x$ (where , is multiset union)
- $prg\ (get\ ?y)\ ?x_S,\ str\ ?y\ ?x > prg\ (ret\ x)\ x_S,\ str\ y\ x$
- $prg\ (set\ ?y\ ?x)\ ?x_S,\ str\ ?y\ ?x' > prg\ (ret\ y)\ x_S,\ str\ y\ x$

Finally, we take as lock-step bisimulation the relation $(\mu | M, S) \mathbf{R} s \overset{\Delta}{\Longleftrightarrow} s$ is equivalent to $[\![(\mu | M, S)]\!]$ modulo simplification

4.2 Join-calculus

We give an encoding for the Join-calculus, which relates TS_c to the reflexive chemical abstract machine for the Join-calculus. The key feature of this encoding is that it does not use simplification (and the terms involved are very simple, namely those of the form $u\ \overline{u}$). We recall the syntax of the (asynchronous core of the) Join-calculus, and refer to [8] for further details on the Join-calculus and the reflexive chemical abstract machine. There is a basic syntactic category of variables, that we can identify with our name variables y, the other syntactic categories are

$$\begin{array}{rl} \text{Join Pattern} & J ::= y\ ?\overline{y} \mid (J_1|J_2) \\ \text{Process} & P ::= y\ \overline{y} \mid \text{def } D \text{ in } P \mid (P_1|P_2) \mid 0 \\ \text{Definition} & D ::= J > P \mid (D_1 \wedge D_2) \mid \top \end{array}$$

The reflexive chemical abstract machine is defined by a transition system TS_{JC} whose states (called chemical solutions) are multisets of definitions D and processes P, and the transition relation is defined in terms of heating, cooling and reaction.

We define a compositional translation $(-)^*$ with the following properties

- J^* is a join pattern ($|$ corresponds to multiset union)
- D^* is a multiset of rules ($\wedge$ corresponds to multiset union, $\top$ to the empty set)
- P^*_y is a multiset of terms and rules, in this case the translation depends on and extra parameter, i.e. a name variable y which should not occur in P

The interesting clauses of the translation are:

- $(\text{def } D \text{ in } P)^*_y = y \uplus (y > \nu\overline{y}, y'.D^* \uplus P^*_{y'})$, where $\overline{y}$ is the set of *declared variables* $\mathrm{DV}(D)$ and $y' \notin \mathrm{FV}(D, P, y)$
- $(P_1|P_2)^*_y = y \uplus (y > \nu y_1, y_2.(P_1)^*_{y_1} \uplus (P_2)^*_{y_2})$, where $y_1, y_2 \notin \mathrm{FV}(P_1, P_2, y)$
- $(J > P)^* = J^* > \nu y.P^*_y$, where $y \notin \mathrm{FV}(J, P)$

The first clause makes essential use of the extra parameter y. If fact, we cannot take $(\text{def } D \text{ in } P)^*_y = \emptyset > \nu\overline{y}.D^* \uplus P^*_y)$, otherwise the rule will be always active (due to the empty join pattern), and we would have multiple copies of D and P. On the other hand, the translation behaves correctly only if there is exactly one occurrence of y, thus the two clauses are designed to preserve this property.

We now define a relation $\mathbf{R} \subseteq S_{JC} \times S_c$ between chemical solutions (i.e. the states of TS_{JC}) and configurations as follows $\mathcal{D} \vdash \mathcal{P}$ is related to $s \overset{\Delta}{\Longleftrightarrow}$

- there is a choice of distinct name variables y_i, one for each element P_i in the multiset $\mathcal{P}$, s.t. y_i is fresh for $\mathcal{D}$ and $\mathcal{P}$
- $s = \mathcal{D}^*[\rho] \ \uplus\ (\uplus_i\ P^*_{iy_i}[\rho])$ for some injective map ρ from name variables to atoms

Clearly, for every chemical solution there is at least one configuration related to it. Therefore, by making a choice we can define a translation $[\![-]\!]$ from S_{JC} to S_c.

The relation $\mathbf{R}$ is a weak bisimulation between TS_{JC} and TS_c, more precisely $(\mathcal{D} \vdash \mathcal{P})\mathbf{R}s$ implies

(i) s cannot be simplified, and any computation step of s is simulated by either one reaction or one heating step of $\mathcal{D} \vdash \mathcal{P}$

(ii) any reaction step of $\mathcal{D} \vdash \mathcal{P}$ is simulated by one computation step of s, and any heating step is simulated by at most one computation step of s.

We ignore cooling steps, since their only purpose is to tie back a chemical solution to a process P (more precisely a chemical solution of the form $\vdash P$).

4.3 Mobile Ambients: Centralized Implementation

We give an encoding of Mobile Ambients (MA) [5,4] corresponding to a *centralized* implementation of MA, in the sense that the computation rules for interpreting MA are *located* in one place (see Remark 3.2). We have not investigate alternative encodings corresponding to a *distributed* implementation of MA along the line of [9]. However, it is unlikely that one can avoid the technical complications (like the used of coupled weak simulations) for proving the correctness of the distributed implementation of MA in the Join-calculus.

In comparison to the original definition of the syntax and operational semantics for MA, we make some adjustments, in order to make the properties of the encoding simpler to formulate. For the syntax we take as basic syntactic categories atoms a and name variables y (and use also names $u ::= a \mid y$). For the operational semantics we use a transition system TS_{MA} similar to the reflexive chemical abstract machine (to avoid the use of structural congruence).

- The processes $P \in PL$ of MA are given by the following BNF
  $P ::= 0 \mid (P_1|P_2) \mid !P \mid \nu y.P \mid u[P] \mid M.P$, where M ranges over capabilities
  $M ::= in\ u \mid out\ u \mid open\ u$
- The states of TS_{MA} are *nested multisets* $\mathcal{P}$ of closed processes, i.e. S_{MA} is the least solution to the domain equation $S_{MA} = \mu(MA_0 + \mathsf{A} \times S_{MA})$, where $\mu(X)$ is the set of finite multisets with elements in X.
- The transition relation $=_{MA}\Longrightarrow$ is by the compatible closure (for nested multisets) of the following rewrite rules
  · $0 \longrightarrow \emptyset \qquad P_1|P_2 \longrightarrow P_1, P_2 \qquad !P \longrightarrow P, !P$
  · $n[P] \longrightarrow n[\{(P)\}]$ (the lhs is an element in MA_0, i.e. process $n[P]$, while the rhs is an element of $\mathsf{A} \times S_{MA}$, i.e. n with the singleton multiset $\{(P)\}$)
  · $\nu y.P \xrightarrow{new} P[y : n]$ with n *fresh* atom (strictly speaking this not a rewrite rule, as the side-condition is global and n is chosen non-deterministically)
  · $n[in\ m.P,\ \mathcal{P}_1],\ m[\mathcal{P}_2] \xrightarrow{in} m[n[P,\ \mathcal{P}_1],\ \mathcal{P}_2]$
  · $m[n[out\ m.P,\ \mathcal{P}_1],\ \mathcal{P}_2] \xrightarrow{out} n[P,\ \mathcal{P}_1],\ m[\mathcal{P}_2]$
  · $open\ m.P,\ m[\mathcal{P}] \xrightarrow{open} P,\ \mathcal{P}$
  The heating rules (indicated with $\longrightarrow$) are related to structural congruence, and the reaction rules (indicated with $\longrightarrow$) correspond to reduction rules.

The compositional translation $(-)^*$ of MA is straightforward, for each clause in the BNF of processes (and capabilities) we have a corresponding atom

- $0^* = nil \qquad (P_1|P_2)^* = par\ P_1^*\ P_2^* \qquad (!P)^* = rep\ P^*$
- $(\nu y.P)^* = new\ (?y \Rightarrow P^* | fail)$
- $(u[P])^* = box\ u\ P^*$
- $(in\ u.P)^* = in\ u\ P^*$, and similarly for the other capabilities

The translation of $\mathcal{P} \in S_{MA}$ into configurations is $[\![\mathcal{P}]\!] = R_{MA} \uplus [\![\mathcal{P}]\!]_{a_r}$, where R_{MA} is a set of computation rules, the atom a_r is the *unique identifier* (UId) for the root ambient, and $[\![-]\!]_a$ is a multiset of terms defined by induction on nested multisets (in particular it commutes with multiset union). The terms in $[\![\mathcal{P}]\!]_a$ are of the form

- $prg\ a\ e$ is a thread executing process e in ambient a (more precisely with UId a), in particular $[\![P]\!]_a = prg\ a\ P^*$
- $amb\ a\ n\ a_p$ says that ambient a has name n and parent ambient a_p

  These terms encode the tree structure of a nested multiset, therefore for each UId a (except the root UId a_r) there should be exactly one of these terms, in particular $[\![n[\mathcal{P}]]\!]_a = (amb\ a'\ n\ a) \uplus [\![\mathcal{P}]\!]_{a'}$ where a' is a fresh UId.

Some rules in R_{MA} introduce terms of the form $opn\ a\ a_p$, which says that ambient a has been opened by parent ambient a_p. When this happens $amb\ a\ n\ a_p$ is removed, but threads and sub-ambients in a must migrate to a_p. The rules in R_{MA} are:

- computation rules for heating
  - $prg\ ?y\ nil > \emptyset$
    $prg\ ?y\ (par\ ?x_1\ ?x_2) > prg\ y\ x_1,\ prg\ y\ x_2$
    $prg\ ?y\ (rep\ ?x) > prg\ y\ x,\ prg\ y\ (rep\ x)$
  - $prg\ ?y\ (new\ ?x) > \nu n.\ prg\ y\ (x@n)$
  - $prg\ ?y\ (box\ ?n\ ?x) > \nu y'.amb\ y'\ n\ y,\ prg\ y'\ x$ (y' is the UId for a new ambient with name n)
- computation rules for capabilities
  - $\underline{amb\ ?y'\ ?m\ ?y''},\ prg\ ?y\ (in\ ?m\ ?x),\ amb\ ?y\ ?n\ ?y'' > prg\ y\ x,\ amb\ y\ n\ y'$
    $\underline{u\ \overline{p}}$ means that the matching term is read, but not removed. In other words, $(\underline{u\ \overline{p}},\ J > \ldots)$ is a shorthand for the computation rule $(u\ \overline{p},\ J > \ldots,\ |u\ \overline{p}|)$, where $|p|$ is the term obtained by erasing the ? in pattern p.
  - $\underline{amb\ ?y'\ ?m\ ?y''},\ prg\ ?y\ (out\ ?m\ ?x),\ amb\ ?y\ ?n\ ?y' > prg\ y\ x,\ amb\ y\ n\ y''$
  - $prg\ ?y\ (open\ ?m\ ?x),\ amb\ ?y'\ ?m\ ?y > prg\ y\ x,\ opn\ y'\ y$
- auxiliary computation rules for open
  - $\underline{opn\ ?y'\ ?y},\ prg\ ?y'\ ?x > prg\ y\ x$
  - $\underline{opn\ ?y'\ ?y},\ amb\ ?y''\ ?n\ ?y' > amb\ y''\ n\ y$

Remark 4.1 One could consider an extension of MA with HO communication, i.e.

- $P ::= \ldots \mid x \mid \langle P \rangle \mid (x)P$ – extended BNF for processes
- $\langle P_1 \rangle \mid (x)P_2 \xrightarrow{comm} P_2[x : P_1]$ – additional reaction rule (for nested multisets).

It is quite easy to extend the encoding to this calculus, in fact

- the compositional translation $(-)^*$ for the three new clauses is
  $x^* = x \qquad \langle P \rangle^* put\ P^* \qquad ((x)P)^* = get\ (?x \Rightarrow P^* | fail)$
- the set R_{MA} has one extra computation rule (for communication)
  $$prg\ ?y\ (get\ ?x_1) \mid prg\ ?y\ (put\ ?x_2) > prg\ y\ (x_1 @ x_2)$$
  while the definition of $[\![\mathcal{P}]\!]_a$ requires no changes.

Finally, we take as weak bisimulation between TS_{MA} and TS_c the relation $\mathcal{P}\mathbf{R}s \overset{\Delta}{\Longleftrightarrow} s$ (modulo permutation of UId) reduces by simplification and the auxiliary computation rules for open to $[\![\mathcal{P}]\!] \uplus G$, where G is *garbage* for $[\![\mathcal{P}]\!]$, i.e. a set of terms $\{(opn\ a'_i\ a_i | i \in n)\}$ where the a'_i are distinct UId not occurring in $[\![\mathcal{P}]\!]$. Note that $[\![\mathcal{P}]\!] \uplus G$ cannot be reduced further by simplification or auxiliary computation rules for open.

Conclusions and Issues

We have proposed a general approach for structuring operational semantics, which distinguishes between simplification and computation.

- Simplification describes things that one does not care to control/program, because they are *simple* and *semantics preserving* (referential transparency), thus it embodies the spirit of pure functional languages, where the user should not be concerned about evaluation strategies adopted by an implementation.
- Computation describes things that can have observable computational effects.

Our proposal builds on top of the Pattern Matching Calculus [13] and the reflexive chemical abstract machine for the Join-calculus [7]. However, one could make different choices without jeopardizing the distinction between simplification and computation. for instance:

- allow *first-class* patterns, along the line of the pure pattern calculus [12]
- have more refined computation rules, that can model stochastic systems
- allow more complex configurations, to describe parts of a closed system that cannot be modeled by a multiset of terms, e.g. a loosely specified environment.

References

[1] Berry, G. and G. Boudol, *The chemical abstract machine*, Theoretical Computer Science **96** (1992), pp. 217–248.

[2] Bettini, L., V. Bono, R. D. Nicola, G. Ferrari, D. Gorla, M. Loreti, E. Moggi, R. Pugliese, E. Tuosto and B. Venneri, *The Klaim Project: Theory and practice*, in: *Global Computing - Programming Environments, Languages, Security and Analysis of Systems*, LNCS **2874** (2003).

[3] Bidinger, P. and J.-B. Stefani, *The kell calculus: Operational semantics and type system*, in: E. Najm, U. Nestmann and P. Stevens, editors, *FMOODS*, Lecture Notes in Computer Science **2884** (2003), pp. 109–123.

[4] Cardelli, L. and A. Gordon, *Mobile ambients*, TCS **240** (2000).

[5] Cardelli, L. and A. D. Gordon, *Mobile ambients*, in: M. Nivat, editor, *FoSSaCS'98*, LNCS **1378** (1998), pp. 140–155.

[6] DeNicola, R., G. Ferrari and R. Pugliese, *Klaim: a kernel language for agents interaction and mobility*, IEEE Transactions on Software Engineering **24** (1998), pp. 315–330.

[7] Fournet, C. and G. Gonthier, *The reflexive chemical abstract machine and the join-calculus*, in: *Conf. Rec. POPL '96: 23rd ACM Symp. Princ. of Prog. Langs.*, 1996.

[8] Fournet, C. and G. Gonthier, *The join calculus: A language for distributed mobile programming*, in: G. Barthe, P. Dybjer, L. Pinto and J. Saraiva, editors, *Advanced Lectures from Int. Summer School on Applied Semantics, APPSEM 2000 (Caminha, Portugal, 9–15 Sept. 2000)*, Lecture Notes in Computer Science **2395**, Springer-Verlag, Berlin, 2002 pp. 268–332.

[9] Fournet, C., J.-J. Lévy and A. Schmitt, *An asynchronous distributed implementation fo mobile ambients*, in: J. van Leeuwen, O. Watanabe, M. Hagiya, P. D. Mosses and T. Ito, editors, *Theoretical Computer Science: Exploring New Frontiers of Theoretical Informatics, Proceedings of the International IFIP Conference TCS 2000 (Sendai, Japan)*, LNCS **1872**, IFIP (2000), pp. 348–364.

[10] Gabbay, M. J. and A. M. Pitts, *A new approach to abstract syntax involving binders*, in: *Proc. 14th Ann. IEEE Symp. Logic in Comput. Sci.*, 1999, pp. 214–224.

[11] Gelernter, D., *Generative communication in Linda*, ACM Transactions on Programming Languages and Systems **7** (1985), pp. 80–112.

[12] Jay, B. and D. Kesner, *Pure pattern calculus*, in: P. Sestoft, editor, *ESOP*, Lecture Notes in Computer Science **3924** (2006), pp. 100–114.

[13] Kahl, W., *Basic pattern matching calculi: a fresh view on matching failure*, in: Y. Kameyama and P. J. Stuckey, editors, *FLOPS*, Lecture Notes in Computer Science **2998** (2004), pp. 276–290.

[14] Klop, J. W., "Combinatory Reduction Systems," Mathematisch Centrum, Amsterdam, 1980, ph.D. Thesis.

[15] Moggi, E., *Notions of computation and monads*, Information and Computation **93** (1991).

[16] Moggi, E. and S. Fagorzi, *A monadic multi-stage metalanguage*, in: *Proc. FoSSaCS '03*, Lecture Notes in Computer Science **2620** (2003).

[17] Moggi, E. and A. Sabry, *An abstract monadic semantics for value recursion*, Theoretical Informatics and Applications **38** (2004).

[18] Peyton Jones, S. L., "The Implementation of Functional Programming Languages," Prentice-Hall, 1987.

[19] Plotkin, G. D., *A structural approach to operational semantics*, Technical Report DAIMI FN-19, Aarhus University Computer Science Department (1981).

[20] Sangiorgi, D., "Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms," PhD thesis CST–99–93, Department of Computer Science, University of Edinburgh (1992).

[21] Schmitt, A. and J.-B. Stefani, *The kell calculus: A family of higher-order distributed process calculi*, in: C. Priami and P. Quaglia, editors, *Global Computing*, Lecture Notes in Computer Science **3267** (2004), pp. 146–178.

[22] Shinwell, M. R., A. M. Pitts and M. J. Gabbay, *FreshML: Programming with binders made simple*, in: *Proc. 8th Int'l Conf. Functional Programming* (2003).

[23] Tennent, R. D., "Semantics of Programming Languages," Prentice Hall, New York, 1991, 7 pp.

[24] Turi, D., "Functorial Operational Semantics and Its Denotational Dual," Ph.D. thesis, Free University, Amsterdam (1996).

[25] Wright, A. K. and M. Felleisen, *A syntactic approach to type soundness*, Information and Computation **115** (1994), pp. 38–94.