URL: http://www.elsevier.nl/locate/entcs/volume27.html 7 pages

Towards Verified Lazy Implementation of Concurrent Value-Passing Languages (Abstract)

Anna Ingólfsdóttir (annai@cs.auc.dk)

BRICS, Dept. of Computer Science, Aalborg University[1]

Rosario Pugliese (pugliese@dsi.unifi.it)

Dipartimento di Sistemi e Informatica, Università di Firenze

In recent years it has become a standard to describe the behaviour of value-passing processes by means of labelled transition systems [Mil80,Ing96]. As these semantic descriptions are used to reason about properties of processes, they are often quite abstract and do not provide much information about potential implementation of the mechanisms used to pass a value from one process to another. For instance, in all the calculi we have studied, the result of inputting a value from a channel is described as a substitution of the form $u[v/x]$ where v is a value and x is a value variable that typically occurs free in u. The precise interpretation of the substitution is left to the reader and is considered to be an implementation detail.

However, more and more often operational semantics is used as a stepping stone towards implementations of languages. Thus we believe it could be useful to have semantic rules which provide more concrete information about the implementation strategy for value-passing processes. These rules are a step closer to a final implementation and can therefore be considered as an implementation oriented refinement, or simply an *implementation*, of the original semantics. This could be an important step towards bridging the gap between theory and practice in this field of research. In this study we will describe a semantics that, in some well-defined sense, implements correctly the *late semantics* [Ing96] for the original *CCS* with values, based on the following considerations.

Efficiency in evaluation: We want our implementation to be as effective as possible and therefore we try to avoid unnecessary evaluations and substitutions. We will try to explain this better by examples. We recall here that in the operational semantics of *CCS* with values, an expression is evaluated

[1] Basic Research in Computer Science, Centre of the Danish National Research Foundation.

when a process of the form $\texttt{if}\ be\ \texttt{then}\ p\ \texttt{else}\ q$ is executed and also when an output action is performed.

1. ***Unnecessary substitution:*** Let

$$p = c!25 \,|\, c?x.\ \texttt{if}\ x < 20\ \texttt{then}\ c!(x+1)\ \texttt{else}\ c!(x+2).$$

The standard operational semantics, where we assume that the substitution takes place immediately after the value is received, yields

$$p \xrightarrow{\tau} \texttt{nil} \,|\, \texttt{if}\ 25 < 20\ \texttt{then}\ c!(25+1)\ \texttt{else}\ c!(25+2).$$

Thus the substitution of 25 for x in the expression $x+1$ is performed although its value is not needed for the process to continue.

2. ***Unused expressions evaluated:*** As another example let

$$q = c!(2+5) \,|\, c?x.$$

Then $q \xrightarrow{\tau} \texttt{nil} \,|\, \texttt{nil}$ can be derived from $c!(2+5) \xrightarrow{c!7} \texttt{nil}$ and $c?x \xrightarrow{c?7} \texttt{nil}$. Therefore the expression $2+5$ is evaluated although its value is never inspected.

One way of avoiding the problem described above is to adopt the call-by-name evaluation mechanism and allow processes to exchange unevaluated expressions. But, as well known from the literature, this may introduce another kind of problems as illustrated by the following example.

3. ***The same expression evaluated more than once:*** Let

$$r = c!(2\cdot 5) \,|\, c?x.\ \texttt{if}\ x < 20\ \texttt{then}\ c!x.(\ \texttt{if}\ x = 10\ \texttt{then}\ c!3x).$$

Then

$$\begin{aligned} r &\xrightarrow{\tau} \texttt{nil} \,|\, \texttt{if}\ 2\cdot 5 < 20\ \texttt{then}\ c!(2\cdot 5).(\ \texttt{if}\ 2\cdot 5 = 10\ \texttt{then}\ c!(3\cdot(2\cdot 5))) \\ &\xrightarrow{c!(2\cdot 5)} \texttt{nil} \,|\, \texttt{if}\ 2\cdot 5 = 10\ \texttt{then}\ c!(3\cdot(2\cdot 5)) \\ &\xrightarrow{c!(3\cdot(2\cdot 5))} \texttt{nil} \,|\, \texttt{nil}. \end{aligned}$$

In this case we have to evaluate the expression $2\cdot 5$ twice, both when evaluating $2\cdot 5 < 20$ and $2\cdot 5 = 10$.

The lazy implementation: Our first approach in tackling the problems described above is to apply a *lazy evaluation mechanism* by using a stack, similar to the *heaps* known from the theory of term rewriting [KW96] and implementation of functional languages [Lau93,WF89] (or *implicit substitutions* that appear in the semantics for the lazy lambda calculus [ACC91]). In our setting a heap is a stack of variable bindings $[x_1 = e_1, \cdots, x = e_n, \cdot]$, where x_i is a value variable and e_i a value-expression, $i = 1, 2, \cdots, n$. A configuration in the transition system is given by a pair consisting of a process term with possibly free value-variables together with a heap that binds those. Thus, instead of substituting free occurrences of x in the process term by e when an input action is performed as described above, the element $x = e$ is pushed onto the top of the heap and is only evaluated when and if strictly needed; in *CCS* this should only happen when a process of the form $\texttt{if}\ be\ \texttt{then}\ p\ \texttt{else}\ q$ is to be performed; all the other operations may be considered as data-independent. In this way we completely avoid performing substitutions. Furthermore, once an expression in the heap has been evaluated it is overwritten by its value and does not need to be reevaluated if its value is needed later. In the last example above, we start the run of the process r by providing it with an empty heap, given by $[\cdot]$, and proceed as follows.

3. Using the heap based semantics and allowing uninterpreted expressions to be passed around we get the following run.

$$\begin{aligned} \langle r, [\cdot]\rangle &\xrightarrow{\tau} \langle \texttt{nil} \mid \texttt{ if } x < 20 \texttt{ then } c!x.(\texttt{ if } x = 10 \texttt{ then } c!3x), [x = 2\cdot 5, \cdot]\rangle \\ &\xrightarrow{c!10} \langle \texttt{nil} \mid \texttt{ if } x = 10 \texttt{ then } c!3x, [x = 10, \cdot]\rangle \\ &\xrightarrow{c!(3\cdot 10)} \langle \texttt{nil} \mid \texttt{nil}, [x = 10, \cdot]\rangle. \end{aligned}$$

Here the value of the expression $2\cdot 5$ in the heap is needed to evaluate $x < 20$. At the same time $2 \cdot 5$ is replaced by its value, 10, in the heap and therefore it does not need to be evaluated again when $x = 10$ is inspected.

Passing unevaluated expressions around by recording them directly in the heap as described above is not very efficient. To avoid this, at compile time, we list into an *expression table* all the expressions that occur in the process. Simultaneously we update the syntactic description of the process by replacing each expression with its address in the table. At run time these pointers are passed around together with information about the scope of the variables that occur in the corresponding expression, i.e. a *closure*. A similar conversion is performed on lambda calculus terms in [Lau93] to ensure that function arguments are always variables. However here we go a step further and separate the syntax of the value expressions completely from the heap but replace them by the corresponding pointers to the expression table.

First the syntax for r is replaced by

$$\texttt{r} = c!\underline{1}. \mid c?x.\texttt{ if } \underline{2} \texttt{ then } c!\underline{3}.(\texttt{ if } \underline{4} \texttt{ then } c!\underline{5})$$

and the expression table

1	2	3	4	5
$2 \cdot 5$	$x < 20$	x	$x = 10$	$3x$

Then the run may be described as follows

$$\begin{aligned} \langle \texttt{r}, [\cdot]\rangle &\xrightarrow{\tau} \langle \texttt{nil} \mid \texttt{ if } \underline{2} \texttt{ then } c!\underline{3}(\texttt{ if } \underline{4} \texttt{ then } c!\underline{5}, [x = \underline{1}, \cdot]\rangle \\ &\xrightarrow{c!3} \langle \texttt{nil} \mid \texttt{ if } \underline{4} \texttt{ then } c!\underline{5}, [x = 10, \cdot]\rangle \\ &\xrightarrow{c!\underline{5}} \langle \texttt{nil} \mid \texttt{nil}, [x = 10, \cdot]\rangle, \end{aligned}$$

where $\underline{1}, \underline{2}$ etc. stand for the a pointers to the expression table and 10 for the evaluated value of the expression pointed to by $\underline{1}$.

But our problems are not completely over yet as communication between parallel components of a process yields some further complications. Of course this is not at all surprising as here we are exactly encountering the difference that is between concurrent languages and the sequential ones. To explain this let us have a look at the following examples. For the sake of notational simplicity we keep on recording the expressions directly in the heap instead of replacing them with pointers to the corresponding expression table entry as described above although this is what we do in our actual implementation semantics.

Let $p_1 = a!1 \mid a?x.(d?x.d!(x+1) \mid d!2x \mid b!3x) \mid b?x.b!(x+2)$. Then

$$\begin{aligned}\langle p_1, [\cdot]\rangle &\xrightarrow{\tau} \langle \mathtt{nil} \mid (d?x.d!(x+1) \mid d!2x \mid b!3x)^{(1)} \mid b?x.b!(x+2), [x = 1^{(1)}, \cdot]\rangle\\ &\xrightarrow{\tau} \langle \mathtt{nil} \mid d!(x+1)^{(2)} \mid \mathtt{nil} \mid b!3x^{(1)} \mid b?x.b!(x+2), [x = 2x^{(2)}, x = 1^{(1)}, \cdot]\rangle.\end{aligned}$$

As the same variable can occur again in the heap without overwriting the scope of the previous one, pointers to the right occurrences of the variables in the heap are given by the superscripts (1) and (2). An obvious continuation of this computation would be an exchange of data between the two last parallel components of the process. However pushing $x = 3x$ directly onto the heap and getting the heap $[x = 3x^{(3)}, x = 2x^{(2)}, x = 1^{(1)}, \cdot]$ would not give the expected result; x in $3x$ should be bound by $x = 1$ directly but not by $x = 2x$ and this step in the computation would be better described by

$$\langle \mathtt{nil} \mid d!(x+1)^{(2)} \mid \mathtt{nil} \mid b!3x^{(1)} \mid b?x.b!(x+2), [x = 2x^{(2)} \to x = 1^{(1)} \to \cdot]\rangle \xrightarrow{\tau}$$

$$\begin{tikzcd}
 & x = 2x^{(2)} \arrow[r] & x = 1^{(1)} \arrow[dr] & \\
\langle \mathtt{nil} \mid d!(x+1)^{(2)} \mid \mathtt{nil} \mid \mathtt{nil} \mid b!(x+2)^{(3)}, [& & & \cdot\ ]\rangle \\
 & & x = 3x^{(3)} \arrow[uu] \arrow[ur] &
\end{tikzcd}$$

From the example above we may conclude that variable scopes for concurrent languages are partially ordered but not totally ordered as they are for sequential processes. Therefore it looks more appropriate to use a tree structure, rather than a linearly ordered stack, to describe the scoping rules for such languages. Here we also note the dependency of the variable bindings between the nodes in different branches of the tree due to communication. Thus, in the example above, x in $3x$ gets it value from $x = 1$ without overwriting that binding of x. The binding $x = 2x$ overwrites the binding $x = 1$ but only for an x that might occur in the component that created this new node.

In our approach we implement such a tree structure by an array or a table, which we refer to as a *control table.* Each line of the table represents a node in the tree and contains a pointer to the father node, the name of the variable it binds and either a value or a *closure* which the variable is bound to. Here a closure has basically the usual meaning as we know it from the functional languages although it consists of a reference to an expression in the expression table together with a reference to the node where the scope of this expression starts. This last reference corresponds to the *cross reference in the tree.* The reader familiar with implementations of functional languages may notice the similarity of our approach with the lazy version of Landin's *SECD* machine [Lan66] that is described in for instance [GHT84].

According to our implementation strategy, a new line is added to the control table every time a communication takes place. As lines are never deleted, the computer eventually runs out of space during run-time, even if the process has actually completed its use of most of the lines of the table. To avoid this and reallocate space which is not in use any more we have defined a garbage-collector, based on *marking* (see, e.g., [Coh81] for an overview of garbage-collectors).

In the paper we formalize the approach described above and use the ideas to define an implementation of the original *CCS* with respect to the late semantics.

Correctness: Reasoning about or proving correctness of compilers dates back to [McC63] and [Mor73]. This is another important issue addressed in the present paper. To be able to state precisely and prove correctness of our implementation we have to provide some general framework within which our suggested implementation resides. In the definition we put forward, an implementation is described by a modification of the standard applicative labelled transition system used to model the standard late semantics.

The definition is based on two main ideas. First we realize that in an implementation, unlike for the specifying transition system, some information about the variable bindings may be recorded in the memory of the computer and is not explicitly visible from the outside. Thus in our suggested implementation, at run-time, some variables are not explicitly bound to a value but implicitly, as some of the expressions involved in defining these values may not yet be evaluated. However, all the information needed to evaluate these expressions is recorded in the control table.

Furthermore, in the general framework, if these values are retrieved from the memory the corresponding transition system should be semantically equivalent to the specifying one. Therefore, in our definition we assume that the general labelled transition systems are supplied with an interpretation mapping that, intuitively, can access these implicit values in the memory; by applying it one can turn an implementation into a standard applicative labelled transition system, its interpretation, that can be compared semantically to the specification. What kind of semantic connection there is between these two transition systems is of course a matter of choice; our choice is based on strong bisimulation which is strong enough to capture most of the semantic relations we know from the literature.

Next our intuition tells us that an implementation is more concrete than the specification and should be allowed to have more states; a state in the specification may be implemented by one or more states. This is because the store may contain some superfluous variable bindings and thus two states can be physically different although logically they are the same. For instance, using slightly simplified notation, the two states given by $(c!x.\texttt{nil}, [x = 2])$ and $(c!x.\texttt{nil}, [x = 2, y = 5])$ are physically different but should both implement the state $c!2.\texttt{nil}$. Combining these two views, our correctness criterion can be described as follows:

(i) The implementation of an applicative labelled transition system should have at least as many states as its specification.

(ii) Its interpretation should be strongly bisimilar to its specification.

This in turn can be described by means of *functional bisimulation* [Cas87]. In the paper we formalize these ideas, put forward an implementation refinement of the late version of *CCS* as an instantiation of the general class and prove its correctness with respect to its specification, i.e. the standard late *CCS*. We also prove the correctness of the garbage-collection, i.e. that applying it to the implementation of a system does not change its semantic meaning. As far as we know none of the existing implementations of concurrent or parallel languages is based on lazy evaluation nor are they provided with a correctness proof with respect to some formal semantics.

Related work: In [FKW99] a correctness criterion, similar to ours, and a corresponding proof is presented for a term rewriting system. Like in our case both the abstract semantics and the concrete implementation are described by transition systems but the semantic correspondence is based on a notion of simulation introduced earlier by two of the authors [KW96].

In [Lau93] the author puts forward a transition system that describes a lazy implementation of the functional language Haskell. The correctness of this implementation is proven by showing that this new operational semantics is fully abstract with respect to a standard denotational semantics for the language. 6 Similar operational rules for a concurrent rewrite system, including a garbage-collection, are given in [Jef94]. Like in [Lau93] the author proves the correctness of these rules by showing that the operational semantics they define is equivalent to an existing denotational semantics.

References

[ACC91] M. Abadi, L. Cardelli and P.-L. Curien. Explicit substitution. *Journal of Functional Programming*, 1(4):375-416,October 1991.

[Cas87] I. Castellani. Bisimulations and abstraction homomorphisms. *Journal of Computer and System Sciences*, 34(2/3):210-235, 1987.

[Coh81] J. Cohen. Garbage collection of linked data structures. *ACM Comput. Surv.*, 13(30:341-367,1981.

[FKW99] W. Fokkink, J. Kamperman, P. Walters. Within ARMS's reach: Compilation of left–linear rewrite systems via minimal rewrite systems. To appear in *ACM Transaction on Programming Languages and Systems*, 1999.

[GHT84] H. Glaser, C. Hankin, D. Till *Principles of Functional Programming* Prentice Hall, London, 1984.

[Ing96] A. Ingólfsdóttir. A Semantic theory for value–passing processes, late approach. To appear in *Information and Computation.*

[IP98] A. Ingólfsdóttir and R. Pugliese. Towards verified lazy implementation of concurrent value-passing languages (Available at `http://www.cs.auc.dk/~annai/`).

[Jef94] A. Jeffrey. A fully abstract semantics for concurrent graph reduction. Proc. of LICS, 1994.

[Lan66] P. J. Landin. The next 700 programming languages. *Communications of the ACM*, Vol. 9, pp. 157–166, 1966.

[Lau93] J. Launchbury. A natural semantics for lazy evaluation. *20th Symposium of Principles of Programming Languages.* ACM, New York, 144–154, 1993.

[KW96] J. F. Th. Kamperman and H. R. Walter. Simulating *TRS*s by minimal *TRS*s: a simple, efficient, and correct compilation technique. Tech. Rep. CS-R9605, CWI, Amsterdam. Available at `http://www.cwi.nl/epic`

[McC63] J. McCarthy. Towards a mathematical science of computation. *Proceedings Information Processing '62.* North-Holland, Amsterdam, 21–28, 1962.

[Mil80] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science.* Springer-Verlag, 1980.

[Mor73] F. L. Morris. Advice on structuring compilers and proving them correct. *1st Symposium of Principles of Programming Languages.* ACM, New York, 144-152.

[Pra75] T. Pratt. *Programming languages, design and implementation.* Prentice-Hall International, 1975.

[WF89] S. C. Wray and J. Fairbain. Non-strict languages–programming and implementation. *The Computer Journal*, 32(2):142–151,1989.