Compiler Correctness for Concurrent Languages

David S. Gladstein* ** and Mitchell Wand*

College of Computer Science, Northeastern University, 360 Huntington Avenue, Boston, MA 02115, USA

Abstract. This paper extends previous work in compiler derivation and verification to languages with true-concurrency semantics. We extend the λ-calculus to model process-centered concurrent computation, and give the semantics of a small language in terms of this calculus. We then define a target abstract machine whose states have denotations in the same calculus. We prove the correctness of a compiler for our language: the denotation of the compiled code is shown to be strongly bisimilar to the denotation of the source program, and the abstract machine running the compiled code is shown to be branching-bisimilar to the source program's denotation.

1 Introduction

Our original goal was to verify a compiler for Linda [8], using that language as a representative of modern concurrent language design. Upon searching the literature, we found a vast amount of work on models of concurrency, but little that was obviously applicable to compiler derivation and verification. Accordingly we decided to tackle first the more primitive "computation + communication" model of systems like PVM [10] or Erlang [4]. Having done this, we expect that our framework will be applicable to the more sophisticated, coordination languages.

1.1 Compiler Derivation Methodology

A recipe [9, 31, 21] for compiling a source language into a target machine language is as follows: The semantics of the source language is given as a compositional translation $\mathcal{E}[\![\cdot]\!]$ from the source language to some metalanguage. The compiler is defined by a compositional translation $\mathcal{C}[\![\cdot]\!]$ from the source language to a target language. This target language has a denotational semantics $\overline{(\,\cdot\,)}$ in the same metalanguage as the source semantics. It also has a plausibly implementable operational semantics, typically given as an abstract machine for which the target code is the machine language. The correctness of the compiler is expressed

* Work supported by the National Science Foundation under grant numbers CCR-901463 and CCR-9304144.

** Author's current address: Invertebrates Department, American Museum of Natural History, Central Park West at 79$^{\text{th}}$ Street, New York, NY 10024, USA.

as the equality of the denotational semantics of any source program p and the denotational semantics of its translation:

$$\mathcal{E}[\![p]\!] = \overline{\mathcal{C}[\![p]\!]} \tag{1}$$

The correctness of the abstract machine is expressed as an adequacy theorem relating the operational and denotational semantics of the target language:

$$\overline{Q} = N \text{ iff } Q \to^{*} F \text{ and } \overline{F} = N \tag{2}$$

for some final configuration F of the abstract machine. Combined, these theorems relate the denotational meaning of the source program to the answer obtained by executing the compiled code on the abstract machine:

$$\mathcal{E}[\![p]\!] = N \text{ iff } \overline{\mathcal{C}[\![p]\!]} = N \text{ iff } \mathcal{C}[\![p]\!] \to^{*} F \text{ and } \overline{F} = N \tag{3}$$

For the case of sequential languages, the metalanguage is usually some version of the λ-calculus, and the notion of equivalence is equality modulo α-renaming and β-reduction, which we write $\stackrel{\beta}{=}$.

In [30] this methodology is adapted to languages with parallelism. The metalanguage is chosen to be the λ-calculus extended with special operators that turn it into a process calculus. It is shown that β-equivalence $\stackrel{\beta}{=}$ extends to a strong bisimulation $\stackrel{\beta}{\simeq}$. This reduces the problem of ensuring bisimilarity to that of ensuring equality in the λ-calculus.

1.2 The Present Work

We want to derive and verify a compiler for a realistic concurrent language, so that running programs will be similar to those written, for example, using PVM [10] or Erlang [4]. Specifically, we shall include asynchronous buffered communication, and we shall allow for execution on multiple computers.

In [30], the calculus has an interleaving semantics, as does the abstract machine. Since we are interested in multicomputer implementations, the present work uses a true-concurrency model of computation. We show how to model concurrency in such a way as to retain the technical developments of [30], while also getting an abstract machine model which is plausibly implementable on a multicomputer.

The rest of the paper is organized as follows: Section 2 describes our conceptual model for true concurrency and gives the details of our metalanguage. Section 3 describes a small source language and gives its denotational semantics in the metalanguage. Section 4 specifies a target machine language and its denotational semantics, presents the compiler, and shows that the compiler preserves denotational semantics up to strong bisimulation (Theorem 3); this corresponds to condition (1). Section 5 presents an abstract machine that executes the target language, and proves its adequacy with respect to the denotational semantics (Theorem 9); this corresponds to condition (2). These theorems are combined in Theorem 10, which corresponds to condition (3). Section 6 briefly discusses a multicomputer Concert/C implementation of the abstract machine. Section 7 briefly considers related work. Section 8 outlines some possibilities for further research.

2 True Concurrency and Our Metalanguage

2.1 The Conceptual Model

Suppose we have a system with two states, and that the system evolves from one state to the other via some time consuming action. We might reasonably model this as a transition system with two states X and Y and a transition $X \stackrel{a}{\to} Y$.

Suppose now that we have two independent copies of that system, with transitions $X_1 \stackrel{a_1}{\to} Y_1$ and $X_2 \stackrel{a_2}{\to} Y_2$. To model their independent, concurrent operation, we might form their product as follows:

$$
\begin{array}{c|ccc}
 & X_1 & \xrightarrow{a_1} & Y_1 \\
\hline
X_2 & (X_1, X_2) & \xrightarrow{a_1} & (Y_1, X_2) \\
\downarrow^{a_2} & \downarrow^{a_2} & & \downarrow^{a_2} \\
Y_2 & (X_1, Y_2) & \xrightarrow{a_1} & (Y_1, Y_2)
\end{array}
$$

This is surely not what we want, since there is no concurrency. Even if we add a diagonal transition $(X_1, X_2) \xrightarrow{a_1 a_2} (Y_1, Y_2)$, there is a problem: the startings and endings of the activities represented by the transitions a_i are synchronized.

Rather than move to another mathematical representation of computation, we can recode our understanding by splitting the transitions. Whenever we have a durational action a represented by a transition $X \stackrel{a}{\to} Y$, we replace it with a new state $\langle a, X \rangle$ and two new new transitions $X \stackrel{a^+}{\to} \langle a, X \rangle$ and $\langle a, X \rangle \stackrel{a^-}{\to} Y$. Here the a^+ and a^- transitions are instantaneous, and the state $\langle a, X \rangle$ is a *durational state*, representing the time during which the system is doing the action a. If we apply this transformation to the previous example and take the interleaving product, we get the following:

$$
\begin{array}{c|ccccc}
 & X_1 & \xrightarrow{a_1^+} & \langle a_1, X_1 \rangle & \xrightarrow{a_1^-} & Y_1 \\
\hline
X_2 & (X_1, X_2) & \xrightarrow{a_1^+} & (\langle a_1, X_1 \rangle, X_2) & \xrightarrow{a_1^-} & (Y_1, X_2) \\
\downarrow^{a_2^+} & \downarrow^{a_2^+} & & \downarrow^{a_2^+} & & \downarrow^{a_2^+} \\
\langle a_2, X_2 \rangle & (X_1, \langle a_2, X_2 \rangle) & \xrightarrow{a_1^+} & (\langle a_1, X_1 \rangle, \langle a_2, X_2 \rangle) & \xrightarrow{a_1^-} & (Y_1, \langle a_2, X_2 \rangle) \\
\downarrow^{a_2^-} & \downarrow^{a_2^-} & & \downarrow^{a_2^-} & & \downarrow^{a_2^-} \\
Y_2 & (X_1, Y_2) & \xrightarrow{a_1^+} & (\langle a_1, X_1 \rangle, Y_2) & \xrightarrow{a_1^-} & (Y_1, Y_2)
\end{array}
$$

Here the two processes may start and stop their activities independently, and furthermore we have explicitly represented the possiblity that the two processes will be active at the same time. This matches our intuition about the behavior of independently executing computers.

The idea of modelling durational events by pairs of instantaneous transitions is certainly not new [16, 17]. Technical difficulties arise when applying this approach to autoconcurrent Petri nets [25, 14], but because the process-centered model of the next section by its very nature lacks autoconcurrency, we are able to use the split-transition representation.

2.2 The Calculus

Syntactic domains

$$
\begin{array}{rcll}
\text{Term} & ::= & c_p & \text{A constant for each } p \in \text{PID} \\
 & | & c_{send} \mid c_{receive} \mid c_{examine} \mid c_{spawn} & \text{Communication operators} \\
 & | & v & \text{Variable} \\
 & | & \lambda v.\text{Term} & \text{Abstraction} \\
 & | & (\text{Term Term}) & \text{Application} \\
\text{Value} & = & \{M \in \text{Term} \mid M \text{ has no free variables}\} & \text{Closed terms} \\
\text{Activity} & ::= & c_{sending} \mid c_{receiving} & \text{Activity markers} \\
 & | & \langle c_{examining} \text{ Value}\rangle \mid \langle c_{spawning}\ c_p\rangle & \\
\text{Active term} & = & \text{Activity} \times \text{Term} & \\
\text{Buffer} & = & \text{Value}^* & \\
\text{Process} & = & (\text{Term} \cup \text{Active term}) \times \text{Buffer} & \\
\text{Program} & = & \text{PID} \overset{\text{fin}}{\to} \text{Process} & \\
\text{Event} & ::= & \langle \textit{send } p\rangle^{\pm} \mid \langle \textit{receive}\rangle^{\pm} & \\
 & | & \langle \textit{examine}\rangle^{\pm} \mid \langle \textit{spawn } p\rangle^{\pm} & \\
\pm & ::= & + \mid - & \\
\text{Action} & = & \text{PID} \times \text{Event} & \text{Transition labels} \\
\text{Transition} & = & \text{Program} \times \text{Action} \times \text{Program} &
\end{array}
$$

Typical variables

$$
\begin{array}{rcll}
p & \in & \text{PID} & \text{Process ids} \\
M, N, P & \in & \text{Term} \cup \text{Active term} & \\
V & \in & \text{Value} & \\
B & \in & \text{Buffer} & \\
X & \in & \text{Process} & \\
\Pi & \in & \text{Program} &
\end{array}
$$

Fig. 1. Syntactic domains and typical variables for the calculus

Figure 1 shows the syntax of our calculus, which is based on an infinite set PID of process identifiers. The usual terms of the λ-calculus are extended with constants for each PID, as well as four communication operators. We also define four activity markers, to annotate processes engaged in durational activity. A process consists of a term (or a term together with an activity marker) and an input buffer. A program is a finite labelled collection of processes. The label of a program transition tells which process made a transition, as well as describing the transition.

Figure 2 gives the transition relation of our calculus; corresponding $^+$ and $^-$ transitions are grouped. The $^+$ transitions are "instantaneous computation" rules [7], in that the terms appearing in the transition may be related by an arbitrary number of β reductions. Within such a reduction sequence, the communication

$$\left[\begin{array}{c}\dfrac{M \stackrel{\beta}{=} (c_{receive}\ N)}{\Pi[p:\langle M,B\rangle] \xrightarrow{p:\langle receive\rangle^{+}} \Pi[p:\langle\langle c_{receiving},(c_{receive}\ N)\rangle,B\rangle]} \\[2ex] \dfrac{X = \langle\langle c_{receiving},(c_{receive}\ N)\rangle,(V{::}B')\rangle}{\Pi[p:X] \xrightarrow{p:\langle receive\rangle^{-}} \Pi[p:\langle (N\ V),B'\rangle]}\end{array}\right]$$

$$\left[\begin{array}{c}\dfrac{M \stackrel{\beta}{=} (c_{examine}\ N) \quad V = \begin{cases}\ulcorner true\urcorner & B = [\,] \\ \ulcorner false\urcorner & B \neq [\,]\end{cases}}{\Pi[p:\langle M,B\rangle] \xrightarrow{p:\langle examine\rangle^{+}} \Pi[p:\langle\langle\langle c_{examining}\ V\rangle,(c_{examine}\ N)\rangle,B\rangle]} \\[2ex] \dfrac{M = \langle\langle c_{examining}\ V\rangle,(c_{examine}\ N)\rangle}{\Pi[p:\langle M,B\rangle] \xrightarrow{p:\langle examine\rangle^{-}} \Pi[p:\langle (N\ V),B\rangle]}\end{array}\right]$$

$$\left[\begin{array}{c}\dfrac{M \stackrel{\beta}{=} (c_{send}\ c_{p'}\ V\ N) \quad V \text{ a closed value}}{\Pi[p:\langle M,B\rangle] \xrightarrow{p:\langle send\ p'\rangle^{+}} \Pi[p:\langle\langle c_{sending},(c_{send}\ c_{p'}\ V\ N)\rangle,B\rangle]} \\[2ex] \dfrac{M = \langle c_{sending},(c_{send}\ c_{p'}\ V\ N)\rangle}{\Pi[p:\langle M,B\rangle][p':\langle M',B'\rangle] \xrightarrow{p:\langle send\ p'\rangle^{-}} \Pi[p:\langle N,B\rangle][p':\langle M',(B'@[V])\rangle]}\end{array}\right]$$

$$\left[\begin{array}{c}\dfrac{M \stackrel{\beta}{=} (c_{spawn}\ N\ P) \quad p' \notin \text{procs}\ \Pi \cup \{p\}}{\Pi[p:\langle M,B\rangle] \xrightarrow{p:\langle spawn\ p'\rangle^{+}} \Pi[p:\langle\langle\langle c_{spawning}\ c_{p'}\rangle,(c_{spawn}\ N\ P)\rangle,B\rangle]} \\[2ex] \dfrac{M = \langle\langle c_{spawning}\ c_{p'}\rangle,(c_{spawn}\ N\ P)\rangle}{\Pi[p:\langle M,B\rangle] \xrightarrow{p:\langle spawn\ p'\rangle^{-}} \Pi[p:\langle (P\ c_{p'}),B\rangle][p':\langle (N\ c_{p'}),[\,]\rangle]}\end{array}\right]$$

where

$$\text{procs}\ \Pi = \text{dom}\ \Pi \cup \{p \mid \exists q{:}\ \Pi(q) = \langle\langle\langle c_{spawning}\ c_p\rangle, M\rangle, B\rangle\}$$

Fig. 2. Communication rules

operators are uninterpreted; this trick of separating computational reductions from communication is due to [20]. We now describe each communication operation in turn.

The $c_{receive}$ operation fetches the head of the input queue, should it be nonempty. A process may try to read its input buffer at any time, but of course the operation cannot terminate until there is input. This is modelled by the $\langle receive\rangle^{+}$ transition always being enabled, and the $\langle receive\rangle^{-}$ transition requiring a nonempty input queue.

The $c_{examine}$ operation examines the input queue, returning $\ulcorner true\urcorner$ for empty and $\ulcorner false\urcorner$ otherwise. (The quasiquotation marks $\ulcorner\cdot\urcorner$ indicate a λ-calculus encoding of the datum, the details of which we leave unspecified.) The $\langle examine\rangle^{+}$ transition takes a snapshot of the state of the buffer, storing it in the activity marker. This snapshot is returned by the $\langle examine\rangle^{-}$ transition, rather than the actual state of the buffer. Thus this test is conservative: a result of $\ulcorner true\urcorner$ may or may not indicate an empty buffer, but a result of $\ulcorner \mathit{false} \urcorner$ always indicates a nonempty buffer.

The c_{send} operation sends a value to a process, or more accurately, puts it into the buffer of a process. Like the $\langle \mathit{receive} \rangle^+$ transition, the $\langle \mathit{send} \rangle^+$ transition is always enabled. The $\langle \mathit{send} \rangle^-$ transition is only enabled when the receiving process actually exists.

The c_{spawn} operation forks another process. The $\langle \mathit{spawn} \rangle^+$ transition is nondeterministic, because the process does not determine the PID of the new process in advance. The process makes a nondeterministic transition into the activity, recording the new, unused PID in the activity marker; a process id is considered to be in use once it is assigned to a fork activity.[1] The $\langle \mathit{spawn} \rangle^-$ activity actually establishes the new process.

A notion of equivalence for process-oriented calculi, called β-bisimilarity and written $\overset{\beta}{\simeq}$, was introduced in [30]. There the processes were λ-terms; two programs Π and Π' were related by $\overset{\beta}{\simeq}$ if $\operatorname{dom} \Pi = \operatorname{dom} \Pi'$, and for each process p in the common domain, $\Pi(p) \overset{\beta}{=} \Pi'(p)$. This equivalence was proved to be a strong bisimulation.

The processes of our calculus are more complicated, being constructed of tuples and sequences of λ-terms. We therefore extend $\overset{\beta}{=}$ to cover tuples and sequences: $\langle M, N \rangle \overset{\beta}{=} \langle M', N' \rangle$ exactly when $M \overset{\beta}{=} M'$ and $N \overset{\beta}{=} N'$, and similarly $[M_1 \ldots M_n] \overset{\beta}{=} [M'_1 \ldots M'_{n'}]$ exactly when $n = n'$ and each $M_i \overset{\beta}{=} M'_i$. With this convention, we may write the definition of equivalence for our calculus:

Definition 1 (β-bisimilarity). Write $\Pi \overset{\beta}{\simeq} \Pi'$ if $\operatorname{dom} \Pi = \operatorname{dom} \Pi'$, and at each p, $\Pi(p) \overset{\beta}{=} \Pi'(p)$.

We can then prove the following for $\overset{\beta}{\simeq}$:

Theorem 2. $\overset{\beta}{\simeq}$ is a strong bisimulation.

Proof. $\Pi \overset{\beta}{\simeq} \Pi'$ be given, and consider each rule for the transitions other than $\langle \mathit{spawn} \rangle^+$. The programs' corresponding processes are $\overset{\beta}{=}$, so by the transitivity of $\overset{\beta}{=}$, any rule which applies to Π applies to Π'.

For $\langle \mathit{spawn} \rangle^+$, the antecedent of the rule has the extra side condition $p' \notin \operatorname{procs} \Pi \cup \{p\}$. Clearly $\operatorname{procs} \Pi = \operatorname{procs} \Pi'$, so any $\langle \mathit{spawn} \rangle^+$ transition open to one program is open to the other. □

3 Source Language Syntax and Semantics

The syntax of our source language is shown in Fig. 3, in which i ranges over identifiers and n over integers. We have the usual pieces of a functional language,

[1] Process ids which occur in a program Π but are not in $\operatorname{procs} \Pi$ are not considered to be in use. Hard-coded PIDs are not expressible in the source language of the next section, so this never happens in the translations of source language programs.

namely identifiers, constants, primitives, abstractions, and applications, as well as four new operators (**put**, **get**, **empty?**, and **spawn**) corresponding to the communication primitives.

$$
\begin{array}{lll}
p ::= a & & \text{Program} \\
a ::= (\mathbf{rec}\ i(i)\ e) & & \text{Abstraction} \\
e ::= n \mid i \mid a \mid (\mathbf{if}\ e\ e\ e) \mid (e\ e) \mid (-\ e\ e) & & \text{Expressions} \\
\quad\mid\ \ (\mathbf{put}\ e\ e) \mid (\mathbf{get}) \mid (\mathbf{empty}?) \mid (\mathbf{spawn}\ e) & &
\end{array}
$$

Fig. 3. Source language syntax

The semantics of the language is given in Fig. 4. Expressions are translated compositionally into terms of our calculus, while programs are translated into entire programs. The environment is a map from identifiers to values; it represents the bindings of variables at the time the expression is evaluated. The continuation is a function mapping the value of the expression to the result of the program; it represents the remainder of the computation, which is waiting for the value of the expression. The translation is essentially a standard continuation-passing semantics; see [9, 22, 23] for detailed expositions. In what follows, when we have a term $(M_1 \ldots M_n)$, we will say that M_1 is in head position, or function position, and that M_n is in continuation position.

$$
\begin{aligned}
\mathcal{E}[\![i]\!] &= \lambda\rho\kappa.\kappa(\rho\, i) \\
\mathcal{E}[\![n]\!] &= \lambda\rho\kappa.\kappa(in_{int} \ulcorner n \urcorner) \\
\mathcal{E}[\![(\mathbf{rec}\ i_1(i_2)\ e)]\!] &= \lambda\rho\kappa.\kappa(in_{clo}(\mathit{fix}\ (\lambda p.\lambda v\kappa.\mathcal{E}[\![e]\!](\rho[(in_{clo}\, p)/i_1][v/i_2])\kappa))) \\
\mathcal{E}[\![(\mathbf{if}\ e_1\ e_2\ e_3)]\!] &= \lambda\rho\kappa.\mathcal{E}[\![e_1]\!]\rho(\lambda b.(\mathit{branch}\ b\ \mathcal{E}[\![e_2]\!]\ \mathcal{E}[\![e_3]\!])\,\rho\,\kappa) \\
\mathcal{E}[\![(e_1\ e_2)]\!] &= \lambda\rho\kappa.\mathcal{E}[\![e_1]\!]\rho(\lambda f.\mathcal{E}[\![e_2]\!]\rho(\lambda a.\mathit{applicate}\ f\ a\ \kappa)) \\
\mathcal{E}[\![(-\ e_1\ e_2)]\!] &= \lambda\rho\kappa.\mathcal{E}[\![e_1]\!]\rho(\lambda v_1.\mathcal{E}[\![e_2]\!]\rho(\lambda v_2.\mathit{subtract}\ v_1\ v_2\ \kappa)) \\
\mathcal{E}[\![(\mathbf{put}\ e_1\ e_2)]\!] &= \lambda\rho\kappa.\mathcal{E}[\![e_1]\!]\rho(\lambda v_1.\mathcal{E}[\![e_2]\!]\rho(\lambda v_2.\mathit{send}\ v_1\ v_2\ \kappa)) \\
\mathcal{E}[\![(\mathbf{get})]\!] &= \lambda\rho\kappa.c_{receive}\kappa \\
\mathcal{E}[\![(\mathbf{empty}?)]\!] &= \lambda\rho\kappa.c_{examine}\kappa \\
\mathcal{E}[\![(\mathbf{spawn}\ e)]\!] &= \lambda\rho\kappa.\mathcal{E}[\![e]\!]\rho(\lambda f.c_{spawn}(\lambda p'.\mathit{boot}\ f\ p')\ (\lambda p'.\kappa\ (in_{pid}\, p'))) \\
\mathcal{P}[\![a]\!] &= [p_0 : \mathcal{E}[\![a]\!]\rho_0(\lambda f.\mathit{boot}\ f\ c_{p_0})]
\end{aligned}
$$

Fig. 4. Source language semantics

The clause for $(-\ e_1\ e_2)$ is typical. Each subexpression is evaluated, the value being bound to the argument variable of an explicit continuation. The values, together with the expression's continuation, are passed to the auxiliary term *subtract*. The auxiliary unboxes the values as necessary for the operation it implements, appeals to another term to do the actual arithmetic, and finally calls the continuation on the appropriately boxed result.

The **put** clause similarly evaluates its arguments and hands them to the term *send*, which proceeds as follows. The first argument should be a process id; it is unboxed, yielding a process term with c_{send} in function position. When the send operation has been completed, computation can continue as specified by the **put** expression's continuation. **put** expressions return *unit*.

The **get** clause is simply the process term with $c_{receive}$ in function position and the expression's continuation in continuation position. Thus upon completion of the receive operation, the expression will return the received value.

The **empty?** clause similarly returns the boolean value generated by the $c_{examine}$ primitive.

The c_{spawn} primitive operates on two arguments: a function to be spawned as a new process, and a continuation. Both arguments are functions, and both are to be called on the process id of the newly created process. The **spawn** clause evaluates its argument, which should yield a closure value representing the function to be spawned. The closure value itself is not of the right form for use by the c_{spawn} primitive, but the **spawn** clause wraps it in an abstraction which is; that abstraction handles the administrative task of boxing the new process id, and also providing the deadlocking continuation †. Likewise, the **spawn** expression's continuation is wrapped in an abstraction which boxes up the new process id before feeding it to the continuation.

The clause for an entire program expects that the program is in the form of a function to be spawned. It specifies that the function should be applied to an initial process id and installed as the one and only process, labelled with that process id.

We now turn to the auxiliaries, which are shown in Fig. 5. In the interest of brevity, we stop short of showing how to encode all of the auxiliary terms into λ-calculus. Rather we state the properties the encodings are expected to have; the details may be found in [13, 30]. The term $\mathbf{0}$ implements deadlock; it cannot communicate. The term † is then a deadlocking continuation.

The language uses four value domains (integers, PIDs, closures, and booleans), and a one-point domain *unit* for representing unspecified values. We omit definitions of the injection and projection terms; encodings as higher-order abstract syntax terms are given in [13]. Taken together, the injection (boxing) terms in_i and the projection (unboxing) terms out_j should have the property

$$out_j(in_i x)\kappa \stackrel{\beta}{=} \begin{cases} \kappa x & \text{if } i = j \\ \mathbf{0} & \text{otherwise} \end{cases}$$

Thus runtime type errors lead to immediate deadlock, while in the absence of type errors unboxing a value simply passes it to the current continuation. We also require that any attempt to unbox *unit* also leads to deadlock.

The term *subtract* makes use of a term *primitive_sub*, which we assume to have the property $(primitive\_sub\ulcorner m\urcorner\ulcorner n\urcorner) \stackrel{\beta}{=} \ulcorner m - n\urcorner$. The term *branch* makes use of a term *primitive_if* which should have the property that $(\mathit{primitive\_if}\ \ulcorner \mathit{true} \urcorner\ M\ N) \stackrel{\beta}{=} M$ and $(\mathit{primitive\_if}\ \ulcorner \mathit{false} \urcorner\ M\ N) \stackrel{\beta}{=} N$.

For later developments, we require an encoding of stacks: an infix constructor ::, an empty stack $\langle\rangle$, and a deconstructor *pop*. We also omit these encodings (which can be found in [13]), but they should have the property

$$\mathit{pop}\, v\, \kappa \stackrel{\beta}{=} \begin{cases} \kappa\, a\, x & \text{if } v \stackrel{\beta}{=} (a{::}x) \\ \mathbf{0} & \text{if } v \stackrel{\beta}{=} \langle\rangle \end{cases}$$

$$\begin{aligned}
\mathbf{0} &= \lambda v.v \\
\dagger &= \lambda v.\mathbf{0} \\
\mathit{applicate} &= \lambda v a \kappa.\mathit{out}_{clo}\, v\, (\lambda f.f\ a\ \kappa) \\
\mathit{branch} &= \lambda b\, e_t e_f.\mathit{out}_{bool}\, b\, (\lambda v.(\mathit{primitive_if}\ v\, e_t\, e_f)) \\
\mathit{subtract} &= \lambda v_1 v_2 \kappa.\mathit{out}_{int}\, v_1 (\lambda n_1.\mathit{out}_{int}\, v_2\, (\lambda n_2.\kappa(\mathit{in}_{int}\, (\mathit{primitive_sub}\, n_1\, n_2)))) \\
\mathit{send} &= \lambda v_1 v_2 \kappa.\mathit{out}_{pid}\, v_1\, (\lambda p.c_{send}\, p\, v_2\, (\kappa\ \mathit{unit})) \\
\mathit{boot} &= \lambda f p.\mathit{applicate}\ f\ (\mathit{in}_{pid}\, p)\, \dagger
\end{aligned}$$

Fig. 5. Auxiliaries

4 Target Language Syntax and Semantics

To the extent that we could execute our calculus via an interpreter, the semantic map $\mathcal{E}$ gives an interpreter for our source language. To move towards a compiler, we separate the information which is known at compilation time from that which is not known until run time. In particular, we split the environment into a compile-time symbol table and a run-time display, and we split continuations into static instruction sequences and dynamic sequences of activation records. Furthermore, we introduce a run-time stack for storing intermediate results, instead of using continuations as, for instance, in the rule for $\mathcal{E}[\![(-\ e_1\ e_2)]\!]$ where two explicit continuations were needed to bind the values of the subexpressions. This leads us to an abstract target machine whose state is a 4-tuple: a current instruction, a display of local variable bindings, a local stack of intermediate results, and a stack of suspended activation records.

The syntax of the target machine language and associated structures is given in Fig. 6. A system state is a finite map from process ids to process states. A process state is either a start state, a stopped state, or a computational state. A computational state consists of a control tuple and an input buffer. The control tuple may stand alone, or may be adjoined to an activity marker; its components were described in the previous paragraph.

The denotations of target machine language programs and states are given in Figs. 6 and 7, as a syntax-directed translation into the metalanguage. The translation follows the pattern established in [9].

The compiler of Fig. 8 translates source terms to target terms. To be correct, the denotation of the compiled code for an expression must match the denotation of the expression itself, after the difference between the compiled and "interpreted" codes' execution contexts are taken into account: the denotation of the compiled code $\mathcal{CE}[\![e]\!]$ for any expression e, symbol table Γ and instruction sequence π, when applied to a display u, local stack ζ, and continuation κ, must be equivalent to the semantic definition $\mathcal{E}[\![e]\!]$ when applied to the environment $u \circ \Gamma$ and a continuation that takes the value returned by e, pushes it on the local stack, and then executes the code sequence π on the resulting register values. This is expressed in the following induction hypothesis:

$$(\overline{\mathcal{CE}[\![e]\!]\Gamma\pi}) \stackrel{\beta}{=} \lambda u\zeta\kappa.\mathcal{E}[\![e]\!](u \circ \Gamma)(\lambda z.\overline{\pi}u(z{::}\zeta)\kappa) \tag{4}$$

This condition is easily verified by structural induction on e, after which the strong bisimilarity of the compiled code's denotation and the semantic definition follows directly:

Theorem 3 (Compiler Correctness). For any program p,

$$[p_0 : start\,\overline{\mathcal{CP}[\![p]\!]}] \stackrel{\beta}{\simeq} \mathcal{P}[\![p]\!]$$

.

5 The Target Machine and Its Correctness

We now present a small-step operational semantics for our abstract machine. We may separate the purely sequential rules from the communicating rules. The sequential rules, which only affect the "computational" part of a process state and not the buffer, may be conveniently expressed as rules for a 4-register machine, and then lifted to a rule expressing the transition of the entire process state. The computation rules and the lifting rule are given in Fig. 9. In addition, we assume that if the instruction is one of the ones in the figure, but the pattern for the rule does not match, we execute the default rule $\langle\langle \pi, u, z, k\rangle, b\rangle \rightarrow_1 \langle \texttt{stopped}\rangle$.

Figure 10 gives the small-step semantics for our abstract machine. Every durational action of each process of the machine is modelled by a pair of transitions, representing the start and end of the action. The pair of rules for τ transitions lifts the computation rules into the instantaneous transition model of computation, making explicit the possibility of the entire machine getting into states where many processes are computing (that is, in $\langle \texttt{computing}, X\rangle$ states) simultaneously. The other eight rules correspond in a straightforward way to the communication rules of Fig. 2.

In the previous section, we showed that the denotation of a compiled program matched the denotation of the program itself. This idea is captured in the following definition:

Syntax	Denotation	
$Q \ = \ [p : s] \ldots$	$[p : \overline{s}] \ldots$	System states
$s ::= \langle \texttt{start}, \pi \rangle$	$\mathit{start}\, \overline{\pi}$	Process states
$\quad \mid \ \langle t, b \rangle$	$\langle \overline{t}, \overline{b} \rangle$	
$\quad \mid \ \langle \texttt{stopped} \rangle$	$\mathbf{0}$	
$t ::= c$	$\overline{c}$	Idle computation
$\quad \mid \ \langle \langle \texttt{computing} \rangle, c \rangle$	$\overline{c}$	Active computation
$\quad \mid \ \langle m, c \rangle$	$\overline{\langle \overline{m}, \overline{c} \rangle}$	Active communication
$c ::= \langle \pi, u, z, k \rangle$	$\overline{\pi}\, \overline{u}\, \overline{z}\, \overline{k}$	Computation state
$\pi ::= \langle \texttt{fetch}, n, \pi \rangle$	$\mathit{fetch}\, \ulcorner n \urcorner\, \overline{\pi}$	Code sequences
$\quad \mid \ \langle \texttt{pushint}, n, \pi \rangle$	$\mathit{pushint}\, \ulcorner n \urcorner\, \overline{\pi}$	
$\quad \mid \ \langle \texttt{pushclosure}, \pi, \pi \rangle$	$\mathit{pushclosure}\, \overline{\pi'}\, \overline{\pi}$	
$\quad \mid \ \langle \texttt{return} \rangle$	return	
$\quad \mid \ \langle \texttt{call}, \pi \rangle$	$\mathit{call}\, \overline{\pi}$	
$\quad \mid \ \langle \texttt{brtrue}, \pi, \pi \rangle$	$\mathit{brtrue}\, \overline{\pi}_t\, \overline{\pi}_f$	
$\quad \mid \ \langle \texttt{sub}, \pi \rangle$	$\mathit{sub}\, \overline{\pi}$	
$\quad \mid \ \langle \texttt{put}, \pi \rangle$	$\mathit{put}\, \overline{\pi}$	
$\quad \mid \ \langle \texttt{get}, \pi \rangle$	$\mathit{get}\, \overline{\pi}$	
$\quad \mid \ \langle \texttt{empty}, \pi \rangle$	$\mathit{empty}\, \overline{\pi}$	
$\quad \mid \ \langle \texttt{spawn}, \pi \rangle$	$\mathit{spawn}\, \overline{\pi}$	
$\quad \mid \ \langle \texttt{swap}, \pi \rangle$	$\mathit{swap}\, \overline{\pi}$	
$u ::= z$	$\overline{z}$	Displays
$z ::= \langle \rangle$	$\langle \rangle$	Stacks
$\quad \mid \ \langle v, z \rangle$	$(\overline{v} {::} \overline{z})$	
$k ::= \langle \texttt{halt} \rangle$	$\dagger$	Continuations
$\quad \mid \ \langle \texttt{frame}, \pi, u, z, k \rangle$	$\lambda v.\overline{\pi}\, \overline{u} (v {::} \overline{z}) \overline{k}$	
$b ::= z$	$\overline{z}$	Buffers
$v ::= \langle \texttt{int}, n \rangle$	$\mathit{in}_{\mathit{int}} \ulcorner n \urcorner$	Values
$\quad \mid \ \langle \texttt{pid}, p \rangle$	$\mathit{in}_{\mathit{pid}}\, c_p$	
$\quad \mid \ \langle \texttt{closure}, \pi, u \rangle$	$\mathit{in}_{\mathit{clo}} (\lambda v \kappa. \overline{\pi} (v {::} \overline{u}) \langle \rangle \kappa)$	
$\quad \mid \ \langle \texttt{bool}, 0 \rangle$	$\mathit{in}_{\mathit{bool}} \ulcorner \mathit{false} \urcorner$	
$\quad \mid \ \langle \texttt{bool}, 1 \rangle$	$\mathit{in}_{\mathit{bool}} \ulcorner \mathit{true} \urcorner$	
$\quad \mid \ \langle \texttt{unit} \rangle$	unit	
$m ::= \langle \texttt{receiving} \rangle$	$\langle c_{\mathit{receiving}} \rangle$	Activity markers
$\quad \mid \ \langle \texttt{emptying}, v \rangle$	$\langle c_{\mathit{examining}}\, \overline{v} \rangle$	
$\quad \mid \ \langle \texttt{sending} \rangle$	$\langle c_{\mathit{sending}} \rangle$	
$\quad \mid \ \langle \texttt{spawning}, v \rangle$	$\langle c_{\mathit{spawning}}\, \overline{v} \rangle$	
$\quad \mid \ \langle \texttt{computing} \rangle$		

Fig. 6. System state syntax and denotations. Note that the rule for $\langle \langle \texttt{computing} \rangle, c \rangle$ takes precedence over the rule for $\langle m, c \rangle$ which follows it.

Definition 4. We say that a program Π in our calculus and a system state Q *correspond* if $\Pi \overset{\beta}{\simeq} \overline{Q}$. We write $\Pi \simeq Q$.

$$
\begin{aligned}
\mathit{fetch}\, n\, \pi &= \lambda u\zeta\kappa.\pi u((\mathit{dlookup}\; u\, n)::\zeta)\kappa\\
\mathit{pushint}\, v\, \pi &= \lambda u\zeta\kappa.\pi u((\mathit{in}_{int}\; v)::\zeta)\kappa\\
\mathit{pushclosure}\; \pi'\pi &= \lambda u\zeta\kappa.\pi u(\mathit{in}_{clo}(\mathit{fix}\;(\lambda p.\lambda v\kappa.\pi'((\mathit{in}_{clo}\; p)::v::u)\langle\rangle\kappa)::\zeta)\kappa)\\
\mathit{return} &= \lambda u\zeta\kappa.\mathit{pop}\;\zeta(\lambda v\zeta.\kappa v)\\
\mathit{call}\,\pi &= \lambda u\zeta\kappa.\mathit{pop}\;\zeta(\lambda a\zeta.\mathit{pop}\;\zeta(\lambda f\zeta.\mathit{applicate}\; f\; a\;(\lambda v.\pi u(v::\zeta)\kappa)))\\
\mathit{brtrue}\;\pi_t\;\pi_f &= \lambda u\zeta\kappa.\mathit{pop}\;\zeta(\lambda v\zeta.(\mathit{out}_{bool}\; v\;(\lambda b.(\mathit{primitive\_if}\; b\;(\pi_t u\zeta\kappa)\;(\pi_f u\zeta\kappa)))))\\
\mathit{sub}\,\pi &= \lambda u\zeta\kappa.\mathit{pop}\;\zeta(\lambda v_1\zeta.\mathit{pop}\;\zeta(\lambda v_2\zeta.\mathit{subtract}\; v_1\; v_2(\lambda v.\pi u(v::\zeta)\kappa)))\\
\mathit{put}\,\pi &= \lambda u\zeta\kappa.\mathit{pop}\;\zeta(\lambda v_1\zeta.\mathit{out}_{pid}\; v_1(\lambda p.c_{send}\; p\; v_2(\pi u(\mathit{unit}::\zeta)\kappa)))\\
\mathit{get}\,\pi &= \lambda u\zeta\kappa.c_{receive}(\lambda v.\pi u(v::\zeta)\kappa)\\
\mathit{empty}\,\pi &= \lambda u\zeta\kappa.c_{empty}(\lambda v.\pi u(v::\zeta)\kappa)\\
\mathit{spawn}\,\pi &= \lambda u\zeta\kappa.\mathit{pop}\;\zeta(\lambda f\zeta.c_{spawn}(\lambda p'.\mathit{boot}\; f\;\dagger)(\lambda p'.\pi u((\mathit{in}_{pid}\; p')::\zeta)\kappa))\\
\mathit{start} &= \lambda\pi.\pi\langle\rangle((\mathit{in}_{pid}\; c_{p_0})::\langle\rangle)\dagger\\
\mathit{swap}\,\pi &= \lambda u\zeta\kappa.\mathit{pop}\;\zeta(\lambda v_1\zeta_1.\mathit{pop}\;\zeta_1(\lambda v_2\zeta_2.\pi u(v_2::(v_1::\zeta_2))\kappa))
\end{aligned}
$$

Fig. 7. Denotational semantics of instructions

$$
\begin{aligned}
\mathcal{CE}[\![i]\!]\Gamma\pi &= \langle\mathtt{fetch}, (\mathit{slookup}\;\Gamma i), \pi\rangle\\
\mathcal{CE}[\![n]\!]\Gamma\pi &= \langle\mathtt{pushint}, n, \pi\rangle\\
\mathcal{CE}[\![(\mathbf{rec}\; i_1(i_2)\; e)]\!]\Gamma\pi &= \langle\mathtt{pushclosure}, (\mathcal{CE}[\![e]\!](i_1::i_2::\Gamma)\langle\mathtt{return}\rangle), \pi\rangle\\
\mathcal{CE}[\![(\mathbf{if}\; e_1\; e_2\; e_3)]\!]\Gamma\pi &= \mathcal{CE}[\![e_1]\!]\Gamma\langle\mathtt{brtrue}, (\mathcal{CE}[\![e_2]\!]\Gamma\pi), (\mathcal{CE}[\![e_3]\!]\Gamma\pi)\rangle\\
\mathcal{CE}[\![(e_1\; e_2)]\!]\Gamma\pi &= \mathcal{CE}[\![e_1]\!]\Gamma(\mathcal{CE}[\![e_2]\!]\Gamma\langle\mathtt{call}, \pi\rangle)\\
\mathcal{CE}[\![(-\; e_1\; e_2)]\!]\Gamma\pi &= \mathcal{CE}[\![e_1]\!]\Gamma(\mathcal{CE}[\![e_2]\!]\Gamma\langle\mathtt{sub}, \pi\rangle)\\
\mathcal{CE}[\![(\mathbf{put}\; e_1\; e_2)]\!]\Gamma\pi &= \mathcal{CE}[\![e_1]\!]\Gamma(\mathcal{CE}[\![e_2]\!]\Gamma\langle\mathtt{put}, \pi\rangle)\\
\mathcal{CE}[\![(\mathbf{get})]\!]\Gamma\pi &= \langle\mathtt{get}, \pi\rangle\\
\mathcal{CE}[\![(\mathbf{empty}?)]\!]\Gamma\pi &= \langle\mathtt{empty}, \pi\rangle\\
\mathcal{CE}[\![(\mathbf{spawn}\; e)]\!]\Gamma\pi &= \mathcal{CE}[\![e]\!]\Gamma\langle\mathtt{spawn}, \pi\rangle\\
\mathcal{CP}[\![e]\!] &= (\mathcal{CE}[\![e]\!]\langle\rangle\langle\mathtt{swap}, \langle\mathtt{call}, \langle\mathtt{return}\rangle\rangle\rangle)
\end{aligned}
$$

Fig. 8. The compiler

The goal now is to show that the successive states of a target machine running the compiled code for a program always correspond to the program's denotation. This amounts to showing that $\simeq$ is a branching bisimulation. We break the proof into several lemmas: we show first that silent transitions of the abstract machine correspond to β-reduction of terms in our calculus, and then that each observable transition of one system is justified by a corresponding transition in the other.

Lemma 5. If $s \rightarrow_1 s'$ then $\overline{s} \rightarrow_\beta^+ \overline{s'}$ via a reduction sequence containing at least one leftmost step.

$$
\begin{array}{ll}
\langle \mathtt{start}, \pi\rangle & \rightarrow_0 \langle \pi, \langle\rangle, \langle\langle \mathtt{pid}, p_0\rangle, \langle\rangle\rangle, \langle \mathtt{halt}\rangle\rangle \\
\langle\langle \mathtt{fetch}, n, \pi\rangle, u, z, k\rangle & \rightarrow_0 \langle \pi, u, \langle (\mathit{clookup}\, u\, n), z\rangle, k\rangle \\
\langle\langle \mathtt{pushint}, n, \pi\rangle, u, z, k\rangle & \rightarrow_0 \langle \pi, u, \langle\langle \mathtt{int}, n\rangle, z\rangle, k\rangle \\
\langle\langle \mathtt{pushclosure}, \pi', \pi\rangle, u, z, k\rangle & \rightarrow_0 \langle \pi, u, \langle\langle \mathtt{closure}, \pi', u\rangle, z\rangle, k\rangle \\
\langle\langle \mathtt{return}\rangle, u, \langle v, z\rangle, \langle \mathtt{frame}, \pi', u', z', k'\rangle\rangle & \rightarrow_0 \langle \pi', u', \langle v, z'\rangle, k'\rangle \\
\langle\langle \mathtt{call}, \pi\rangle, u, \langle v, \langle\langle \mathtt{closure}, \pi', u'\rangle, z\rangle\rangle, k\rangle & \rightarrow_0 \langle \pi', \langle\langle \mathtt{closure}, \pi', u'\rangle, \langle v, u'\rangle\rangle, \langle\rangle, \\
 & \qquad \langle \mathtt{frame}, \pi, u, z, k\rangle\rangle \\
\langle\langle \mathtt{brtrue}, \pi_t, \pi_f\rangle, u, \langle\langle \mathtt{bool}, \mathit{true}\rangle, z\rangle, k\rangle & \rightarrow_0 \langle \pi_t, u, z, k\rangle \\
\langle\langle \mathtt{brtrue}, \pi_t, \pi_f\rangle, u, \langle\langle \mathtt{bool}, \mathit{false}\rangle, z\rangle, k\rangle & \rightarrow_0 \langle \pi_f, u, z, k\rangle \\
\langle\langle \mathtt{sub}, \pi\rangle, u, \langle\langle \mathtt{int}, n\rangle, \langle\langle \mathtt{int}, m\rangle, z\rangle\rangle, k\rangle & \rightarrow_0 \langle \pi, u, \langle\langle \mathtt{int}, (m-n)\rangle, z\rangle, k\rangle \\
\langle\langle \mathtt{swap}, \pi\rangle, u, \langle v, \langle w, z\rangle\rangle, k\rangle & \rightarrow_0 \langle \pi, u, \langle w, \langle v, z\rangle\rangle, k\rangle
\end{array}
$$

$$
\frac{\langle \pi, u, z, k\rangle \rightarrow_0 \langle \pi', u', z', k'\rangle}{\langle\langle \pi, u, z, k\rangle, b\rangle \rightarrow_1 \langle\langle \pi', u', z', k'\rangle, b\rangle}
$$

Fig. 9. Computation rules

Lemma 6 (Sequential Simulation). If $Q \xrightarrow{\tau} Q'$ then $\overline{Q} \overset{\beta}{\simeq} \overline{Q'}$.

Lemma 7 (Communicating Simulation). If $a \neq \tau$, $Q \xrightarrow{a} Q'$ and $\Pi \simeq Q$, then there is a Π' with $\Pi \xrightarrow{a} \Pi'$ and $\Pi' \simeq Q'$.

Lemma 8 (Completeness). If $\Pi \xrightarrow{a} \Pi'$ and $\Pi \simeq Q$ then there is a Q_1 and a Q' with $Q \xrightarrow{\tau} \ldots \xrightarrow{\tau} Q_1 \xrightarrow{a} Q'$, $\Pi \simeq Q_1$, and $\Pi' \simeq Q'$.

The proofs of Lemmas 5–7 are straightforward. The proof of Lemma 8 uses Lemma 5 and the completeness of quasi-leftmost reduction sequences for head normal forms; see [6, 30, 13] for details. Combining the lemmas gives the following:

Theorem 9 (Abstract Machine Correctness). $\simeq$ is a branching bisimulation.

Combining Theorems 3 and 9 we get:

Theorem 10 (Combined Correctness). For any program e, $\mathcal{P}[\![e]\!]$ is branching bisimilar to $[p_0 : \langle \mathtt{start}, \mathcal{CP}[\![e]\!]\rangle]$.

This shows that the behavior given by the semantics is similar to the behavior of the target machine running the compiled program, as desired.

6 Implementation

To demonstrate that the small-step semantics is plausible, we coded an implementation in Concert/C [5], an extension of C which provides facilities for transmitting data structures between processes. Each process of the target machine

$$\left[\begin{array}{c}\dfrac{X \rightarrow_1 X'}{Q[p:X] \xrightarrow{\tau} Q[p:\langle\texttt{computing}, X\rangle]} \\[2ex] \dfrac{X \rightarrow_1 X'}{Q[p:\langle\texttt{computing}, X\rangle] \xrightarrow{\tau} Q[p:X']}\end{array}\right]$$

$$\left[\begin{array}{c}\dfrac{X = \langle\langle\langle\texttt{get}, \pi\rangle, u, z, k\rangle, b\rangle}{Q[p:X] \xrightarrow{p:\langle receive\rangle^+} Q[p:\langle\langle\texttt{receiving}\rangle, X\rangle]} \\[2ex] \dfrac{X = \langle\langle\texttt{receiving}\rangle, \langle\langle\langle\texttt{get}, \pi\rangle, u, z, k\rangle, \langle v, b\rangle\rangle\rangle}{Q[p:X] \xrightarrow{p:\langle receive\rangle^-} Q[p:\langle\langle\pi, u, \langle v, z\rangle, k\rangle, b\rangle]}\end{array}\right]$$

$$\left[\begin{array}{c}\dfrac{X = \langle\langle\langle\texttt{empty}, \pi\rangle, u, z, k\rangle, b\rangle \quad i = \begin{cases}1 & b = \langle\rangle \\ 0 & b \neq \langle\rangle\end{cases}}{Q[p:X] \xrightarrow{p:\langle examine\rangle^+} Q[p:\langle\langle\texttt{emptying}, \langle\texttt{int}, i\rangle\rangle, X\rangle]} \\[2ex] \dfrac{X = \langle\langle\texttt{emptying}, v\rangle, \langle\langle\langle\texttt{empty}, \pi\rangle, u, z, k\rangle, b\rangle\rangle}{Q[p:X] \xrightarrow{p:\langle examine\rangle^-} Q[p:\langle\langle\pi, u, \langle v, z\rangle, k\rangle, b\rangle]}\end{array}\right]$$

$$\left[\begin{array}{c}\dfrac{X = \langle\langle\langle\texttt{put}, \pi\rangle, u, \langle v, \langle\langle\texttt{pid}, p'\rangle, z\rangle\rangle, k\rangle, b\rangle}{Q[p:X] \xrightarrow{p:\langle send\, p'\rangle^+} Q[p:\langle\texttt{sending}, X\rangle]} \\[2ex] \dfrac{X = \langle\texttt{sending}, \langle\langle\langle\texttt{put}, \pi\rangle, u, \langle v, \langle\langle\texttt{pid}, p'\rangle, z\rangle\rangle, k\rangle, b\rangle\rangle}{Q[p:X][p':\langle Y, b'\rangle] \xrightarrow{p:\langle send\ p'\rangle^-} \begin{array}{l}Q[p:\langle\langle\pi, u, \langle\langle\texttt{unit}\rangle, z\rangle, k\rangle, b\rangle] \\ \quad [p':\langle Y, b' @ \langle v, \langle\rangle\rangle\rangle]\end{array}}\end{array}\right]$$

$$\left[\begin{array}{c}\dfrac{X = \langle\langle\langle\texttt{spawn}, \pi\rangle, u, \langle\langle\texttt{closure}, \pi', u'\rangle, z\rangle, k\rangle, b\rangle \quad p' \notin \mathrm{procs}\, Q \cup \{p\}}{Q[p:X] \xrightarrow{p:\langle spawn\, p'\rangle^+} Q[p:\langle\langle\texttt{spawning}, \langle\texttt{pid}, p'\rangle\rangle, X\rangle]} \\[2ex] \dfrac{X = \langle\langle\texttt{spawning}, \langle\texttt{pid}, p'\rangle\rangle, \langle\langle\langle\texttt{spawn}, \pi\rangle, u, \langle\langle\texttt{closure}, \pi', u'\rangle, z\rangle, k\rangle, b\rangle\rangle}{Q[p:X] \xrightarrow{p:\langle spawn\, p'\rangle^-} \begin{array}{l}Q[p:\langle\langle\pi, u, \langle\langle\texttt{pid}, p'\rangle, z\rangle, k\rangle, b\rangle] \\ \quad [p':\langle\langle\langle\texttt{call}, \texttt{return}\rangle, \langle\rangle, \langle\langle\texttt{pid}, p'\rangle, \langle\langle\texttt{closure}, \pi', u'\rangle, \langle\rangle\rangle\rangle, \\ \qquad \langle\texttt{halt}\rangle\rangle, \\ \qquad [\,]\rangle]\end{array}}\end{array}\right]$$

Fig. 10. Small step semantics

program was represented by a separate Concert/C process. The target machine states of Fig. 6 were transliterated into C `structs`. A straightforward interpreter implements those process transitions which do not involve interprocess communication, while message sending and spawning are implemented by the corresponding Concert/C facilities.

Taking the state of a Concert/C process to consist of its program counter together with the values of its program variables, we were able to argue that the collection of Concert/C processes is a simulation of the target machine: each state and transition of the collection of processes corresponds to a legal move of the target machine. Since we could not guarantee anything about the way our processes would be scheduled by the operating system, we were not able to argue that our implementation is complete for the target machine; however, we were able to show that the implementation is safe, in that it cannot deadlock unless the target machine can do so.

Had our machines' semantics been defined in the usual interleaving style, as in [30], the correctness proofs of the preceding sections would still have worked out; the difficulty comes in reasoning about the multicomputer implementation. As in the example of Sect. 2.1, the interleaving semantics specifies that the collection of processes must execute only one step at a time. The nature of a collection of computers, however, is to execute many steps simultaneously and without synchronization between the processors. In the absence of a locking mechanism preventing the computers from making independent progress, which would be silly, it is not clear how to draw a correspondence between the possible behaviors of the collection of computers and the semantic specification of the target machine. By contrast, the interleaved instantaneous transitions of the split-transition semantics captures precisely our understanding of the operation of the collection of computers.

The complete implementation is given in [13].

7 Related Work

The compiler derivation methodology sketched in Sect. 1.1 has been applied to yield correctness proofs for several compilers of modest size: a simple functional language [28, 27], a language with **while**-expressions [29], a language obeying stack-discipline and having nonlocal jumps [31], and core Scheme [9]. Two larger-scale systems have also been proven correct in great detail using these techniques: A complete implementation of Scheme is proven correct in [15], and a compiler for PreScheme, a Scheme-like language designed for systems programming, is proven correct in [21].

As discussed in [30], the literature contains a great deal of work on the mathematical and algebraic properties of models of concurrent computation, but very little about compiler correctness. We briefly discuss a few of the papers that are most closely related to our work; all of them use interleaving models of concurrency.

The Nielsons' formulation of process algebras for Concurrent ML [20] is the basis of the formulation of our calculus; most importantly, they originated the technique of separating computation and communication by having the computation treat the communication primitives as uninterpreted constants. They use a small-step semantics for both computation and communication, whereas we use a version of the instantaneous computation rule of [7]; this allows us to stay within the λ-calculus when considering computation alone.

Facile [11, 12] is a source language whose semantics is given by a compositional translation into the intermediate language "core Facile". Core Facile is based on the typed call-by-value λ-calculus, with channel-based communication and a call/cc operator. In [1], Amadio gives translations between several versions of this language, including a translation into an environment-based machine similar to ours. He proves the adequacy of his translations, in the sense that they do not confuse distinguishable programs. Both the computation and communication semantics of core Facile are given as a pure small-step semantics. This mixing of computation and communication makes the notion of observability more complicated, and requires considerable reasoning about administrative reductions, etc. Our formulation seems easier to reason about, because β-equivalence is automatically a bisimulation.

Milner shows in [18, 19] how λ-calculus can be translated into the π-calculus. Walker [26] translates two languages similar to POOL [2] into the π-calculus, but he gives no relation between his translations and any other semantics for the languages. CHOCS is used as an intermediate language in Thomsen's original article [24]; he translates the λ-calculus into CHOCS and shows that the translation preserves, but does not reflect, β-convertibility. He also gives a semantics for a toy language with procedures as a compositional translation into CHOCS.

Languages like ours which use the "computation + communication" paradigm naturally have substantial sequential components. Neither the π-calculus nor CHOCS is so well suited as the λ-calculus for generating reasonable code for the sequential portions of the language. By using the λ-calculus as an intermediate language, we are able to reuse large amounts of reasoning from sequential-language compiler proofs. Appel [3] gives further arguments for the use of λ-calculus as an intermediate language.

8 Future Work

In the course of this work, several avenues of further research became apparent.

Our account of asynchronous message passing is somewhat inaccurate, in that a process is blocked from the time it sends a message until the time the message is enqueued at the receiver's input buffer. We believe that this can be fixed by giving an account in which buffers and communication paths are reified, and all communication is synchronous; this approach would also apply to more sophisticated coordination mechanisms.

So far the communication and process control operations of our semantics have been equivalent to those provided by the underlying implementation system, in our case Concert/C. We need to give an account of high-level operations in terms of lower-level ones; refinement methods such as those of [21, 7] should be applicable.

References

1. R. M. Amadio. Translating core Facile. Technical Report ECRC-1994-3, ECRC, Feb. 1994.
2. P. America, J. de Bakker, J. N. Kok, and J. Rutten. Operational semantics of a parallel object-oriented language. In *Conf. Rec. 13th ACM Symposium on Principles of Programming Languages*, pages 194–208, 1986.
3. A. W. Appel. *Compiling with Continuations*. Cambridge University Press, Cambridge, 1992.
4. J. Armstrong, R. Virding, and M. Williams. *Concurrent Programming in Erlang*. Prentice Hall, 1993.
5. J. S. Auerbach et al. *Concurrent/C Tutorial and User's Guide: An Introduction to a Language for Distributed C Programming*. IBM T. J. Watson Research Center, June 1994.
6. H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, 1984.
7. B. Bloom. Chocolate: Calculi of higher order communication and lambda terms. In *Conf. Rec. 21st ACM Symposium on Principles of Programming Languages*, pages 339–347, 1994.
8. N. Carriero and D. Gelernter. *How to Write Parallel Programs: A First Course*. MIT Press, 1990.
9. W. Clinger. The Scheme 311 compiler: An exercise in denotational semantics. In *Proc. 1984 ACM Symposium on Lisp and Functional Programming*, pages 356–364, Aug. 1984.
10. A. Geist et al. *PVM 3 User's Guide and Reference Manual*. Oak Ridge National Laboratory, May 1993.
11. A. Giacalone, P. Mishra, and S. Prasad. Facile: A symmetric integration of concurrent and functional programming. *International Journal of Parallel Programming*, 18(2):121–160, 1989.
12. A. Giacalone, P. Mishra, and S. Prasad. Operational and algebraic semantics for facile: A symmetric integration of concurrent and functional programming. In *Proc. ICALP '90*, volume 443 of *Lecture Notes in Computer Science*, pages 765–7780, Berlin, Heidelberg, and New York, 1990. Springer-Verlag.
13. D. Gladstein. *Compiler Correctness for Concurrent Languages*. PhD thesis, Northeastern University, Dec. 1994. Available via anonymous ftp from `ftp.ccs.neu.edu` as `/pub/people/daveg/thesis.ps.Z`.
14. R. Gorrieri and C. Laneve. Split and ST bisimulation semantics. *Information and Computation*, 118:272–288, 1995.
15. J. D. Guttman, V. Swarup, and J. Ramsdell. The VLISP verified scheme system. *Lisp and Symbolic Computation*, 8(1/2), 1995.
16. M. Hennessy. Axiomatising finite concurrent processes. *SIAM Journal of Computation*, 15(5):997–1017, 1995.
17. C. Hoare. *Communicating Sequential Processes*. Prentice Hall International, 1985.
18. R. Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2:119–141, 1992.
19. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes (parts i and ii). *Information and Computation*, 100:1–77, 1992.
20. F. Nielson and H. R. Nielson. From CML to process algebras. In *Proceesings of CONCUR '93*, pages 493–508, Berlin, Heidelberg, and New York, 1993. Springer-Verlag.

21. D. P. Oliva, J. D. Ramsdell, and M. Wand. The VLISP verified PreScheme compiler. *Lisp and Symbolic Computation*, 8(1/2), 1995.
22. J. C. Reynolds. The discoveries of continuations. *Lisp and Symbolic Computation*, 6(3/4):233–248, 1993.
23. G. Springer and D. P. Friedman. *Scheme and the Art of Programming*. MIT Press, 1989.
24. B. Thomsen. A calculus of higher order communicating systems. In *Conf. Rec. 16th ACM Symposium on Principles of Programming Languages*, pages 143–154, 1989.
25. R. van Glabbeek and F. Vaandrager. The difference between splitting in n and $n+1$ (abstract). In E. Best and G. Rozenberg, editors, *Proceedings 3rd Workshop on Concurrency and Compositionality*, Sankt Augustin, Germany, February 1991. GMD-Studien Nr. 191.
26. D. Walker. π-calculus semantics of object-oriented programming languages. In *Proc. Conference on Theoretical Aspects of Computer Software*, Lecture Notes in Computer Science, Berlin, Heidelberg, and New York, 1991. Springer-Verlag.
27. M. Wand. Deriving target code as a representation of continuation semantics. *ACM Transactions on Programming Languages and Systems*, 4(3):496–517, July 1982.
28. M. Wand. Semantics-directed machine architecture. In *Conf. Rec. 9th ACM Symposium on Principles of Programming Languages*, pages 234–241, 1982.
29. M. Wand. Loops in combinator-based compilers. *Information and Control*, 57(2–3):148–164, May/June 1983.
30. M. Wand. Compiler correctness for parallel languages. In *1995 Symposium on Functional Languages and Computer Architecture*, June 1995.
31. M. Wand and D. P. Oliva. Proving the correctness of storage representations. In *Proc. 1992 ACM Symposium on Lisp and Functional Programming*, pages 151–160, 1992.