Compiler Correctness for Parallel Languages

Mitchell Wand*

College of Computer Science
Northeastern University
360 Huntington Avenue, 161CN
Boston, MA 02115, USA
wand@ccs.neu.edu
http://www.ccs.neu.edu/home/wand

Abstract

We present a paradigm for proving the correctness of compilers for languages with parallelism. The source language is given a denotational semantics as a compositional translation to a higher-order process calculus. The target language is also given a denotational semantics as a compositional translation to the same process calculus. We show the compiler is correct in that it preserves denotation up to bisimulation. The target language is also given an operational semantics, and this operational semantics is shown correct in the sense that it is branching-bisimilar to the denotational semantics of the target language. Together, these results show that for any program, the operational semantics of the target code is branching-bisimilar to the semantics of the source code.

1 Introduction

In a series of papers in the early 80's [35, 34, 36, 12] we proposed a paradigm for semantics-based compiler correctness, and over the last several years we have begun putting this paradigm into practice [38, 29]. In this paradigm, the semantics of the source language is given as a compositional translation from the source language to some version of the λ-calculus. The compiler is defined by a compositional translation from the source language to a target language. This target language has a denotational semantics in the same lambda-calculus 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 as the equality of the denotational semantics of any source program and the denotational semantics of its translation. The correctness of the abstract machine takes the form of an adequacy theorem relating the operational and denotational semantics of the target language. Combined, these theorems relate the denotational meaning of the source program to the answer obtained by executing the compiled code on the abstract machine. Figure 1 shows the relationships.

*Work supported by the National Science Foundation under grant numbers CCR-9304144 and CCR-9404646

In this paper, we show how to adapt this paradigm for languages with parallelism. For parallel languages, the story is changed as follows: the semantics of the source language is given as a compositional translation into a higher-order process calculus instead of the λ-calculus. The correctness of the compiler is established by showing that the denotational semantics of any program and its translation are equal up to a bisimulation. The adequacy of the operational semantics of the target language is established by demonstrating a weak bisimulation.

In this paper we will carry out this technique for a small language with parallelism and process-to-process blocking communication. The remainder of this paper is organized as follows: In Section 2 we introduce our process calculus, HOCC. In Section 3 we will give our source language and its denotational semantics in HOCC. In Section 4 we will specify the target language and its denotational semantics, present the compiler, and show that the compiler preserves denotational semantics up to bisimulation (Theorem 2). In Section 5 we will present an abstract machine that executes the target language and prove its adequacy with respect to the denotational semantics of the target language (Theorem 3). In Section 6 we put these results together to get our final version of the correctness of the compiler: that the semantics of any program is branching-bisimilar to the initial state of the machine running the compiled code (Theorem 4). In Section 7 we will consider some of the possible variations on this theme. In Section 8 we briefly consider related work. Last, in Section 9, we summarize the development and our conclusions.

2 HOCC

2.1 Process Calculi in General

We assume that the reader is familiar with the basics of process calculi [23, 22, 6]. Here we summarize the standard definitions we will use.

In general a process calculus is specified by a labelled transition system $\langle Q, A, R\rangle$, where Q is a set of states, A is a set of actions (assumed to be the observables of the system), and $R \subseteq Q \times A \times Q$ is a labelled transition relation; we write $q \xrightarrow{a} q'$ when $(q, a, q') \in R$.

Definition 1 *Given two such calculi $\langle Q, A, R\rangle$ and $\langle Q', A, R'\rangle$, with the same actions, we say that $\simeq$ is a* bisimulation

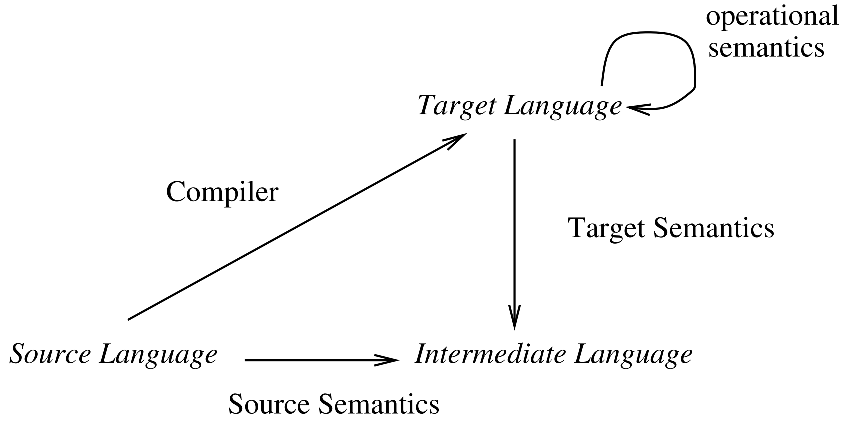

Figure 1: Compiler Correctness Diagram

relation *between them iff:*

1. *If* $q_1 \simeq q_1'$ *and* $q_1 \xrightarrow{a} q_2$*, then there exists a* $q_2' \in Q'$ *such that* $q_1' \xrightarrow{a} q_2'$ *and* $q_2 \simeq q_2'$*.*
2. *If* $q_1 \simeq q_1'$ *and* $q_1' \xrightarrow{a} q_2'$*, then there exists a* $q_2 \in Q$ *such that* $q_1 \xrightarrow{a} q_2$ *and* $q_2 \simeq q_2'$*.*

It is easy to see that bisimulation relations are closed under union, so the union of all bisimulation relations is itself a bisimulation relation, called *(strong) bisimulation*. We say two states are *bisimilar* if they are in this relation.

We can consider the transitive closure of $\rightarrow$: If $w \in A^*$, then $q \overset{w}{\Rightarrow} q'$ iff there exists $q_0, \ldots, q_n$ such that $q = q_0 \xrightarrow{a_1} q_1 \ldots \xrightarrow{a_n} q_n = q'$, and w is the concatenation $a_1 \ldots a_n$.

In some of our systems, A will contain a distinguished element τ, marking a silent transition. In these systems it is useful to consider notions of bisimulation up to silent moves. The notion we will use is:

Definition 2 *We say* $\simeq$ *is a* branching-bisimulation relation *iff for all* $a \in A - \{\tau\}$

1. *If* $q_1 \simeq q_1'$ *and* $q_1 \xrightarrow{a} q_2$*, then either* $a = \tau$ *and* $q_2 \simeq q_1'$ *or there exists a* $q_2' \in Q'$ *such that* $q_1' \overset{\tau^* a}{\Rightarrow} q_2'$ *and* $q_2 \simeq q_2'$*.*
2. *If* $q_1 \simeq q_1'$ *and* $q_1' \xrightarrow{a} q_2'$*, then either* $a = \tau$ *and* $q_1 \simeq q_2'$ *or there exists a* $q_2 \in Q$ *such that* $q_1 \overset{\tau^* a}{\Rightarrow} q_2$ *and* $q_2 \simeq q_2'$*.*

This allows the possibility that two systems may have the same observable behavior, but may differ in their internal (invisible) actions. We allow the possibility that either of the systems does not have a silent transition.

As in the case of strong bisimulation above, branching bisimulation relations are closed under union; two states are said to be *branching-bisimilar* iff they are in the largest branching-bimulation relation, or equivalently, iff there is some branching-bisimulation relation that relates them.

A weaker notion of equivalence is trace equivalence. Define

$$\mathit{Traces}(q) = \{\phi(w) \mid (\exists q')(q \overset{w}{\Rightarrow} q')\}$$

where ϕ is the string transduction that maps τ to the empty string and each other letter to itself. It can be shown that if q and q' are branching-bisimilar, then $\mathit{Traces}(q) = \mathit{Traces}(q')$.

2.2 HOCC

Here we introduce HOCC (Higher-Order Computation and Communication), the process calculus we will use as our intermediate language. The computations of HOCC are based on the λ-calculus, which is contained in HOCC as a sublanguage. The communications of HOCC are point-to-point communications, similar to the kind of communication found in, e.g. PVM [8, 14]. This is simpler than channel-based communication, since we do not have to keep track of a global set of used channels. We define HOCC by first defining its set of states and then defining its actions and its transition relation.

The syntax of the calculus is shown in Figure 2. There are three syntactic categories. The first is the set of process identifiers (also known as process ids or pids) p. The second is the set of λ-terms M, constructed over constants for communication and process identifiers. The last is the set of program terms Π; these are finite partial maps from the set PID of process identifiers to the set of λ-terms. We write $\Pi[p : M]$ for the usual extension of a finite partial map. The program terms Π are the states of the HOCC calculus.

The calculus has only two rules, one for each kind of observable transition. These are shown in Figure 3. The form of the hypotheses is somewhat unusual. For example, if a term M can eventually be β-converted to a term headed by a c_{spawn}, then the process containing M can execute the fork on the current step. Thus computation will effectively use "big-step" semantics and communication will use "small-step" semantics. We sometimes call this a "hybrid" or "instantaneous computation" semantics. This is a characteristic feature of our calculus.

The first rule deals with spawning a new process. The rule starts a new process with id p' and initial process term $(N c_{p'})$. Thus N is applied to its own process identifier. This is how N determines its process id. The original process p continues with term P; it, too, is applied to $c_{p'}$. This is how the original process finds out the pid assigned to the new process. The old process already knows its own pid, since it was started similarly, and the new process can be supplied with a copy of its parent's pid by lexical scoping or by the parent sending it a message. The observable part of this action merely records that process p started a new process p'.

The second rule allows processes to communicate. This

p	$\in PID$	Process Ids
M	$::= v \mid MM \mid \lambda v.M$	Process terms (λ-terms)
	$::= c_{spawn} \mid c_{send} \mid c_{receive}$	(special constants)
	$::= c_p$	(for each $p \in PID$)
Π	$::= [p_1 : M_1] \dots [p_k : M_k]$	Program Terms

Figure 2: Syntax of HOCC terms

$$\frac{M =_\beta (c_{spawn}\, N\, P)}{\Pi[p : M] \xrightarrow{a} \Pi[p' : (N c_{p'})][p : (P c_{p'})]}$$

where p' is a fresh process identifier (i.e., $p' \notin \mathrm{dom}(\Pi) \cup \{p\}$) and $a = \langle spawn\, p\, p' \rangle$.

$$\frac{M =_\beta (c_{send}\, c_{p'}\, V\, M') \quad N =_\beta (c_{receive}\, N')}{\Pi[p : M][p' : N] \xrightarrow{a} \Pi[p : M'][p' : (N'\, V)]}$$

where V is a constant or a closed abstraction, $p' \in PID$, and $a = \langle send\, p\, p'\, c \rangle$, where $c = V$ if V is a constant, and $c = \langle cl \rangle$ if V is an abstraction.

Figure 3: Hybrid transition rules for HOCC terms

rule says that if process p, in state M is willing (after some amount of computation) to send a value V to process p', and process p', in state N is willing (also after some amount of computation) to receive a value, then the communication can happen as the next communication step. Process p continues with M' and process p' continues by applying the receiver N' to the transmitted value V. The observable portion of this interaction is the record that p sent a value to p'; if the value was a constant, then that constant is recorded in the observable action as well, otherwise a token $\langle cl \rangle$, representing a closure, is recorded. Our choice of observable actions for both rules is in keeping with the usual functional-programming convention that the contents of abstractions are not observable.

Note that a term M can be convertible to at most one of the forms $c_{spawn} N P$, $c_{send} c_{p'} V M'$, or $c_{receive} N'$. Furthermore, if the head normal form of M does not match one of these forms, then M cannot communicate, and deadlocks. Thus the term $\lambda x.x$ can serve the role of the empty process. We define $\mathbf{0} = \lambda x.x$.

Clearly, other rules are possible. In general, our approach will work so long as the hypotheses of the rules are β-equivalences.

2.3 β-bisimilarity

In general, strong bisimilarity is difficult to characterize, so instead we need easily-characterized bisimilarity relations. Most of the reasoning we need for our compiler is captured by the relation of *β-bisimilarity*:

Definition 3 (β-bisimilarity) *We say Π and Π' are β-*bisimilar *(and write $\Pi \simeq_\beta \Pi'$) iff* $\mathrm{dom}\,\Pi = \mathrm{dom}\,\Pi'$ *and for all p in their common domain,* $\Pi(p) =_\beta \Pi'(p)$.

So two process states are β-bisimilar if they have the same active process id's and the processes are pointwise β-convertible. Note that this is sensible: Π is a map from process id's to λ-terms, so $\Pi(p)$ and $\Pi'(p)$ are λ-terms, which may be β-convertible.

Theorem 1 *$\simeq_\beta$ is a bisimulation relation.*

Proof: By form of the rules, and the transitivity of $=_\beta$. Consider a single case: Assume $M =_\beta M'$ and $M =_\beta c_{spawn} N P$. Then $M' =_\beta c_{spawn} N P$ as well. So each move open to M is open to M' as well, and vice versa. ∎

β-bisimilarity is a much smaller relation than bisimilarity, but it will suffice for our purposes.

3 The Source Language

We next present a small language that will serve as the source language for our compiler.

3.1 Syntax

The syntax of the language is shown in Figure 4. Here i ranges over identifiers and n ranges over integers. The language consists of identifiers, constants, abstractions, applications, and primitive operations (we have chosen subtraction as a typical primitive), plus three communications operations, called put, get, and spawn. We have made the language untyped, because we do not need the mechanism of types for our compiler.

3.2 Semantics

The semantics of the language is given in Figure 5. It is a compositional (i.e. syntax-directed or denotational) translation from the source language to HOCC. Programs are translated to program terms Π, and expressions are translated to process terms M.

The translation of the source language is mostly a conventional continuation-passing style (CPS) semantics [19, 30]. For each expression e, its translation $\mathcal{E}[\![e]\!]$ has functionality something like

$$Env \to [V \to Comm] \to Comm$$

$$
\begin{array}{lll}
p & ::= e & \text{Programs} \\
e & ::= i \mid n \mid \mathbf{proc}(i)\{e\} \mid (e\ e) \mid e - e & \text{Expressions} \\
 & ::= \mathbf{put}(e, e) \mid \mathbf{get}() \mid \mathbf{spawn}(e) &
\end{array}
$$

Figure 4: Syntax of Source expressions

This indicates that $\mathcal{E}[\![e]\!]$ is intended to be applied to an environment ρ and a continuation κ of type $[V \rightarrow Comm]$, where V is the domain of expressed values and the type $Comm$ of communications (expressed concretely as process terms M) replaces command continuations as the "answer" domain. This type is not to be taken seriously, since our meta-language HOCC is untyped, but it suggests the organization of the semantics. In an ordinary (sequential) CPS semantics, we can regard the semantics of an expression as computing, given an expression ρ and a continuation κ, a command continuation: a transformation of states to answers. Here we take the point of view that the semantics of an expression, given an environment and a continuation, computes a communications event. This is similar to the `com` types in [28] or the type b of behaviors in [1].

The semantics is written using a number of auxiliaries, which we describe briefly here. We will discuss them in more detail in Section 3.3. There are three kinds of expressed values: integers, process id's, and closures. The terms in_j and out_j are injections into and projections from these summands. If the projection fails, the deadlocked term $\mathbf{0}$ results. The auxiliaries *applicate* and *subtract* check to see that their arguments are of the proper summands and then perform the usual operation. $\lceil - \rceil$ denotes the representation of an integer or identifier as a numeral in the λ-calculus. Since our system is untyped, the choice of representation is unimportant. Our source language includes integer constants and a single binary operation, subtraction (chosen rather than addition because it is not commutative); any of the usual operations could be done in a similar style.

When $\mathcal{E}[\![\mathbf{put}(e_1, e_2)]\!]$ is applied to environment ρ and continuation κ, e_1 is evaluated in environment ρ and in a continuation that projects the result into a process id p. The continuation then evaluates e_2 (returning a value v_2), and the result is the term $c_{send} p v_2 (\kappa(in_{int}\lceil 1 \rceil))$ that expresses the desire to send value v_2 to process id p and continue with $\kappa(in_{int}\lceil 1 \rceil)$. So this expression, when executed, performs the communication and returns a value of 1 to its caller κ.

Similarly, the semantics of a **get** clause, applied to an environment ρ and a continuation κ, is the process term $c_{receive}\kappa$. This is a process that receives a message and passes it to continuation κ, so the received message appears as the value of the **get** expression.

A **spawn** expression takes one argument, which should evaluate to a procedure. When executed, it should spawn a new process that applies the procedure to the pid of the new process. The spawn expression should immediately return the pid of the child to its caller. The first argument to c_{spawn} is the term to be executed by the child process. The c_{spawn} rule causes this term to be applied to the child process's pid. Hence the procedure f is applied to the child pid p' (coerced into a value by in_{pid}), in a continuation that ignores the answer and returns the null process $\mathbf{0}$. Thus, if the child procedure ever returns, its process deadlocks. The second argument to c_{spawn} is a continuation that takes the child pid p', coerces it into a value, and sends it to κ, so the current process continues with the child's pid as the value of the **spawn** expression. Note that there need not be an explicit mechanism for passing the parent's pid to the child, since this information can be included inside the procedure f.

Now we can consider the clause for an entire program. A program consists of an expression e. This expression should evaluate to a procedure f, which is applied to its own process id, as if it had been spawned. So the program is translated to the program term $[p_0 : M]$, where p_0 is some initial process identifier and M, shown in Figure 5 above, is a process term that accomplishes this task.

3.3 Auxiliaries

There are three kinds of expressed or denoted values manipulated by our language: integers, process id's, and closures. Since our translation is syntactic rather than model-theoretic, we cannot rely on the construction of a suitable domain. Instead, we use a syntactic encoding inspired by the coding used in [26], which generalizes the Scott integers (reported in [32]):

$$
\begin{aligned}
in_{int} &= \lambda x.\lambda fgh.fx \\
in_{pid} &= \lambda x.\lambda fgh.gx \\
in_{clo} &= \lambda x.\lambda fgh.hx \\
out_{int} &= \lambda v\kappa.v\kappa(\lambda v.\mathbf{0})(\lambda v.\mathbf{0}) \\
out_{pid} &= \lambda v\kappa.v(\lambda v.\mathbf{0})\kappa(\lambda v.\mathbf{0}) \\
out_{clo} &= \lambda v\kappa.v(\lambda v.\mathbf{0})(\lambda v.\mathbf{0})\kappa
\end{aligned}
$$

With these definitions, we have $out_i(in_i\, x)\kappa =_\beta \kappa x$, but for $i \neq j$ we have $out_i(in_j\, x)\kappa =_\beta \mathbf{0}$, the deadlocked term.

Hence we can define *applicate* as

$$
applicate = \lambda v a \kappa.\ out_{clo}\, v(\lambda f.fa\kappa)
$$

This checks whether the value we are trying to apply is a closure; if it is, the closure is applied to an argument and a continuation; otherwise the result is a deadlock.

Similarly, we can define *subtract* as

$$
\begin{aligned}
subtract = \lambda v_1 v_2 \kappa. out_{int}\, v_1 (\lambda n_1. \\
out_{int}\, v_2(\lambda n_2. \\
\kappa(in_{int}(primitive\text{-}sub\; n_1\; n_2))))
\end{aligned}
$$

where *primitive-sub* is a subtraction term for the particular numeral system used (i.e., $primitive\text{-}sub\lceil n \rceil \lceil m \rceil =_\beta \lceil (n - m) \rceil$).

Similarly, we will later code stacks of values via:

$$
\begin{aligned}
(a::x) &= \lambda fg.fax \\
\langle\rangle &= \lambda fg.g
\end{aligned}
$$

$$\mathcal{E}[\![i]\!] = \lambda\rho\kappa.\kappa(\rho\lceil i\rceil)$$
$$\mathcal{E}[\![n]\!] = \lambda\rho\kappa.\kappa(in_{int}\lceil n\rceil)$$
$$\mathcal{E}[\![\mathbf{proc}(i)\{e\}]\!] = \lambda\rho\kappa.\kappa(in_{clo}(\lambda v\kappa.\,\mathcal{E}[\![e]\!](\rho[v/i])\kappa))$$
$$\mathcal{E}[\![(e_1\ e_2)]\!] = \lambda\rho\kappa.\,\mathcal{E}[\![e_1]\!]\rho(\lambda f.\mathcal{E}[\![e_2]\!]\rho(\lambda a.\ applicate\, f\ a\ \kappa))$$
$$\mathcal{E}[\![e_1 - e_2]\!] = \lambda\rho\kappa.\,\mathcal{E}[\![e_1]\!]\rho(\lambda v_1.\,\mathcal{E}[\![e_2]\!]\rho(\lambda v_2.\ subtract\ v_1\ v_2\ \kappa))$$
$$\mathcal{E}[\![\mathbf{put}(e_1, e_2)]\!] = \lambda\rho\kappa.\,\mathcal{E}[\![e_1]\!]\rho(\lambda v_1.\,\mathcal{E}[\![e_2]\!]\rho(\lambda v_2.\ out_{pid}\ v_1\ (\lambda p.c_{send}\ p\ v_2\ (\kappa(in_{int}\lceil 1\rceil)))))$$
$$\mathcal{E}[\![\mathbf{get}()]\!] = \lambda\rho\kappa.c_{receive}\kappa$$
$$\mathcal{E}[\![\mathbf{spawn}(e)]\!] = \lambda\rho\kappa.\,\mathcal{E}[\![e]\!]\rho(\lambda v.\ out_{clo}\ v(\lambda f.c_{spawn}\ (\lambda p'.f\ (in_{pid}\,p')\ (\lambda v.\mathbf{0}))\ (\lambda p'.\kappa(in_{pid}\,p'))))$$
$$\mathcal{P}[\![e]\!] = [p_0 : \mathcal{E}[\![e]\!]\rho_0(\lambda f.\ applicate\, f\ (in_{pid}\ c_{p_0})\ (\lambda v.\mathbf{0}))]$$

Figure 5: Semantics of the Source Language

We can then define

$$pop\, v\kappa = v\kappa\mathbf{0}$$

With these definitions, we have $pop(a{::}x)\kappa =_\beta \kappa a x$, but $pop\langle\rangle\kappa =_\beta \mathbf{0}$.

4 The Compiler and its Correctness

The goal of the compiler is to translate the source language into a target language with both a denotational and operational semantics. In this section we present the target language and its denotational semantics, the compiler, and sketch a proof that the semantics of a program and the denotational semantics of its translation agree up to a bisimulation.

4.1 The Target Language and its Denotational Semantics

Figure 6 shows the syntax of the target language and its associated structures. A system state consists of a set of machines, each associated with a process id. Each machine is either in a start state, a stopped state, or in a state consisting of 4 registers, of which the first contains the machine code it is executing. We express the syntax in terms of code sequences, rather than instructions, because this structure leads to cleaner proofs.

The denotational semantics of the target language, denoted by $\overline{(-)}$, is given in Figure 7. This semantics is a compositional (syntax-directed) translation from the syntactic categories of system states into HOCC. The translation of a system state Q is a program state Π; the translation of a machine state or any of its components is a λ-term M. The semantics is presented in terms of auxiliaries *start*, *fetch*, *pushint*, etc., that constitute the denotational semantics of the individual opcodes of the abstract machine. These are defined in Figure 8. ($dlookup\,u\,n$) looks up a lexical address n on the display u. For historical reasons, we use π as a λ-variable ranging over denotations of code sequences q.

The key to the semantics is the equation

$$\overline{\langle q, u, z, k\rangle} = \overline{q}\,\overline{u}\,\overline{z}\,\overline{k}$$

The right-hand side of this equation forces the denotation $\overline{q}$ of a code sequence to be a λ-term that takes the contents of the three other machine registers and computes the right thing. For example,

$$\begin{aligned}&\overline{\langle\langle \mathtt{pushint}, n, q\rangle, u, z, k\rangle}\\ &= pushint\lceil n\rceil\ \overline{q}\ \overline{u}\ \overline{z}\ \overline{k}\\ &= \overline{q}\,\overline{u}\,((in_{int}\lceil n\rceil){::}\overline{z})\overline{k}\end{aligned}$$

This expresses denotationally the operational intuition that a `pushint` instruction should push an integer value on the stack and then proceed with the code that follows.

4.2 The Compiler

The compiler is a straightforward syntax-directed translation from the source language into postfix-style code and is shown in Figure 9. Our source language has two nonterminals, programs and expressions, so we will have two compilation functions, $\mathcal{CP}$ for programs and $\mathcal{CE}$ for expressions. $\mathcal{CP}$ will take a program and return a system state, and $\mathcal{CE}$ will take an expression, a symbol table and a static continuation (the code sequence to be executed between this expression and the next procedure return), and will return a code sequence. This organization is patterned after [12]. *slookup* looks up an identifier in the static environment (symbol table).

4.3 Correctness of the Compiler

As indicated above, we want to show that the compiler preserves denotational semantics up to bisimulation, that is, for whole programs p, we want the compiler $\mathcal{CP}$ to satisfy

$$\overline{\mathcal{CP}[\![p]\!]} \simeq_\beta \mathcal{P}[\![p]\!] \tag{1}$$

This guarantees that the compiled program and the original behave similarly for all practical purposes. We will prove this by structural induction.

To do the structural induction, we will need an additional induction hypothesis for expressions. For this we use the usual induction hypothesis [12, 38]:

$$\overline{(\mathcal{CE}[\![e]\!]\Gamma q)} =_\beta \lambda u\zeta\kappa.\,\mathcal{E}[\![e]\!](u\circ\Gamma)(\lambda z.\overline{q}\,u(z{::}\zeta)\kappa) \tag{2}$$

This says that if we compile expression e with symbol table Γ and static continuation q, and run the resulting code on display u, local stack ζ, and continuation κ, then the result is the same as evaluating e using the semantics, with 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 q on the resulting register values.

$$
\begin{array}{lll}
Q & ::= [p : s] \ldots & \text{System states} \\
s & ::= \langle \mathtt{start}, q \rangle & \text{Machine states} \\
 & ::= \langle q, u, z, k \rangle & \\
 & ::= \langle \mathtt{stopped} \rangle & \\
q & ::= \langle \mathtt{fetch}, n, q \rangle & \text{Code sequences} \\
 & ::= \langle \mathtt{pushint}, n, q \rangle & \\
 & ::= \langle \mathtt{pushclosure}, q, q \rangle & \\
 & ::= \langle \mathtt{return} \rangle & \\
 & ::= \langle \mathtt{call}, q \rangle & \\
 & ::= \langle \mathtt{sub}, q \rangle & \\
 & ::= \langle \mathtt{put}, q \rangle & \\
 & ::= \langle \mathtt{get}, q \rangle & \\
 & ::= \langle \mathtt{spawn}, q \rangle & \\
 & ::= \langle \mathtt{swap}, q \rangle & \\
u & ::= z & \text{Displays} \\
z & ::= \langle\rangle & \text{Stacks} \\
 & ::= \langle v, z \rangle & \\
k & ::= \langle \mathtt{halt} \rangle & \text{Continuations} \\
 & ::= \langle \mathtt{frame}, q, u, z, k \rangle & \\
v & ::= \langle \mathtt{int}, n \rangle & \text{Values} \\
 & ::= \langle \mathtt{pid}, p \rangle & \\
 & ::= \langle \mathtt{closure}, q, u \rangle &
\end{array}
$$

Figure 6: Syntax of system states

$$
\begin{array}{ll}
\overline{[p : s] \ldots} & = [p : \overline{s}\,] \ldots \\
 & \\
\overline{\langle \mathtt{start}, q \rangle} & = start\ \overline{q} \\
\overline{\langle q, u, z, k \rangle} & = \overline{q}\,\overline{u}\,\overline{z}\,\overline{k} \\
\overline{\langle \mathtt{stopped} \rangle} & = \mathbf{0} \\
 & \\
\overline{\langle \mathtt{fetch}, n, q \rangle} & = fetch\,\lceil n \rceil\,\overline{q} \\
\overline{\langle \mathtt{pushint}, n, q \rangle} & = pushint\,\lceil n \rceil\,\overline{q} \\
\overline{\langle \mathtt{pushclosure}, q', q \rangle} & = pushclosure\ \overline{q'}\ \overline{q} \\
\overline{\langle \mathtt{return} \rangle} & = return \\
\overline{\langle \mathtt{call}, q \rangle} & = call\ \overline{q} \\
\overline{\langle \mathtt{sub}, q \rangle} & = sub\ \overline{q} \\
\overline{\langle \mathtt{put}, q \rangle} & = put\ \overline{q} \\
\overline{\langle \mathtt{get}, q \rangle} & = get\ \overline{q} \\
\overline{\langle \mathtt{spawn}, q \rangle} & = spawn\ \overline{q} \\
\overline{\langle \mathtt{swap}, q \rangle} & = swap\ \overline{q} \\
 & \\
\overline{\langle\rangle} & = \langle\rangle \\
\overline{\langle v, z \rangle} & = (\overline{v}::\overline{z}\,) \\
 & \\
\overline{\langle \mathtt{halt} \rangle} & = (\lambda v.\mathbf{0}) \\
\overline{\langle \mathtt{frame}, q, u, z, k' \rangle} & = \lambda v.\overline{q}\,\overline{u}\,(v::\overline{z}\,)\overline{k'} \\
 & \\
\overline{\langle \mathtt{int}, n \rangle} & = in_{int}\lceil n \rceil \\
\overline{\langle \mathtt{pid}, p \rangle} & = in_{pid}\, c_p \\
\overline{\langle \mathtt{closure}, q, u \rangle} & = in_{clo}(\lambda v\kappa.\overline{q}\,(v::\overline{u}\,)\langle\rangle\kappa)
\end{array}
$$

Figure 7: Denotational Semantics of system states

$$\begin{aligned}
&\mathit{fetch}\; n\; \pi = \lambda u\zeta\kappa.\pi u((\mathit{dlookup}\; u\; n)::\zeta)\kappa\\
&\mathit{pushint}\; v\; \pi = \lambda u\zeta\kappa.\pi u((\mathit{in}_{int}\; v)::\zeta)\kappa\\
&\mathit{pushclosure}\; \pi'\; \pi = \lambda u\zeta\kappa.\pi u((\mathit{in}_{clo}(\lambda v\kappa.\pi'(v::u)\langle\rangle\kappa))::\zeta)\kappa\\
&\mathit{return} = \lambda u\zeta\kappa.\; \mathit{pop}\,\zeta(\lambda v\zeta.\kappa v)\\
&\mathit{call}\; \pi = \lambda u\zeta\kappa.\; \mathit{pop}\,\zeta(\lambda a\zeta.\; \mathit{pop}\,\zeta(\lambda f\zeta.\; \mathit{applicate}\; f\; a\; (\lambda v.\pi u(v::\zeta)\kappa)))\\
&\mathit{sub}\; \pi = \lambda u\zeta\kappa.\; \mathit{pop}\; \zeta\; (\lambda v_2\zeta.\; \mathit{pop}\; \zeta\; (\lambda v_1\zeta.\; \mathit{subtract}\; v_1\; v_2\; (\lambda v.\pi u(v::\zeta)\kappa)))\\
&\mathit{put}\; \pi = \lambda u\zeta\kappa.\; \mathit{pop}\; \zeta\; (\lambda v_2\zeta.\; \mathit{pop}\; \zeta\; (\lambda v_1\zeta.\; \mathit{out}_{pid}\; v_1\; (\lambda p.c_{send}\; p\; v_2\; (\pi u((\mathit{in}_{int}\lceil 1\rceil)::\zeta)\kappa))))\\
&\mathit{get}\; \pi = \lambda u\zeta\kappa.c_{receive}\; (\lambda v.\pi u(v::\zeta)\kappa)\\
&\mathit{spawn}\; \pi = \lambda u\zeta\kappa.\; \mathit{pop}\; \zeta\; (\lambda v\zeta.\; \mathit{out}_{clo}\; v\; (\lambda f.c_{spawn}(\lambda p'.f\; (\mathit{in}_{pid}\, p')\; (\lambda v.\mathbf{0}))\; (\lambda p'.\pi u((\mathit{in}_{pid}\, p')::\zeta)\kappa)))\\
&\mathit{start} = \lambda\pi.\pi\langle\rangle((\mathit{in}_{pid}\, c_{p_0})::\langle\rangle)(\lambda v.\mathbf{0})\\
&\mathit{swap}\; \pi = \lambda u\zeta\kappa.\; \mathit{pop}\; \zeta\; (\lambda v_1\zeta_1.\; \mathit{pop}\; \zeta_1\; (\lambda v_2\zeta_2.\pi u(v_2::(v_1::\zeta_2))\kappa))
\end{aligned}$$

Figure 8: Denotational semantics of instructions

$$\begin{aligned}
&\mathcal{CE}[\![i]\!]\Gamma q = \langle \mathtt{fetch}, (\mathit{slookup}\,\Gamma i), q\rangle\\
&\mathcal{CE}[\![n]\!]\Gamma q = \langle \mathtt{pushint}, n, q\rangle\\
&\mathcal{CE}[\![\mathbf{proc}(i)\{e\}]\!]\Gamma q = \langle \mathtt{pushclosure}, (\mathcal{CE}[\![e]\!](i::\Gamma)\langle\mathtt{return}\rangle), q\rangle\\
&\mathcal{CE}[\![(e_1\; e_2)]\!]\Gamma q = \mathcal{CE}[\![e_1]\!]\Gamma(\mathcal{CE}[\![e_2]\!]\Gamma\langle\mathtt{call}, q\rangle)\\
&\mathcal{CE}[\![e_1 - e_2]\!]\Gamma q = \mathcal{CE}[\![e_1]\!]\Gamma(\mathcal{CE}[\![e_2]\!]\Gamma\langle\mathtt{sub}, q\rangle)\\
&\mathcal{CE}[\![\mathbf{put}(e_1, e_2)]\!]\Gamma q = \mathcal{CE}[\![e_1]\!]\Gamma(\mathcal{CE}[\![e_2]\!]\Gamma\langle\mathtt{put}, q\rangle)\\
&\mathcal{CE}[\![\mathbf{get}()]\!]\Gamma q = \langle\mathtt{get}, q\rangle\\
&\mathcal{CE}[\![\mathbf{spawn}(e)]\!]\Gamma q = \mathcal{CE}[\![e]\!]\Gamma\langle\mathtt{spawn}, q\rangle\\
&\mathcal{CP}[\![e]\!] = [p_0 : \langle\mathtt{start}, (\mathcal{CE}[\![e]\!]\langle\rangle\langle\mathtt{swap}, \langle\mathtt{call}, \langle\mathtt{return}\rangle\rangle\rangle)\rangle]
\end{aligned}$$

Figure 9: The compiler

Theorem 2 (Compiler Correctness)

1. *For any program p,* $\overline{\mathcal{CP}[\![p]\!]} \simeq_\beta \mathcal{P}[\![p]\!]$.
2. *For any expression e, symbol table* Γ*, and code sequence q,*

$$(\overline{\mathcal{CE}[\![e]\!]\Gamma q}\,) =_\beta \lambda u\zeta\kappa.\,\mathcal{E}[\![e]\!](u \circ \Gamma)(\lambda z.\overline{q}\,u(z{::}\zeta)\kappa)$$

Proof: The proof is by induction on the grammatical structure of programs and expressions.

To illustrate the technique, we consider the case of **spawn** expressions. The compiler's output on a **spawn** expression, according to Figure 9, is given by

$$\mathcal{CE}[\![\mathbf{spawn}(e)]\!]\Gamma q = \mathcal{CE}[\![e]\!]\Gamma\langle \mathtt{spawn}, q\rangle \tag{3}$$

The induction hypothesis says that for any symbol table Γ and code sequence q,

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

and we need to show that

$$\begin{aligned}&(\overline{\mathcal{CE}[\![\mathbf{spawn}(e)]\!]\Gamma q}\,)\\ &\quad =_\beta \lambda u\zeta\kappa.\,\mathcal{E}[\![\mathbf{spawn}(e)]\!](u \circ \Gamma)(\lambda z.\overline{q}\,u(z{::}\zeta)\kappa)\end{aligned} \tag{5}$$

Taking the left-hand side of equation 5, we calculate:

$$\begin{aligned}
&(\overline{\mathcal{CE}[\![\mathbf{spawn}(e)]\!]\Gamma q}\,)\\
&= \qquad\qquad \text{(definition of } \mathcal{CE})\\
&(\overline{\mathcal{CE}[\![e]\!]\Gamma\langle \mathtt{spawn}, q\rangle}\,)\\
&= \qquad\qquad \text{(IH at } e)\\
&\lambda u\zeta\kappa.\,\mathcal{E}[\![e]\!](u \circ \Gamma)(\lambda v.\overline{\langle \mathtt{spawn}, q\rangle}\;u\;(v{::}\zeta)\;\kappa)\\
&= \qquad\qquad \text{(definition of } \overline{(-)}\,)\\
&\lambda u\zeta\kappa.\,\mathcal{E}[\![e]\!](u \circ \Gamma)(\lambda v.\;\mathit{spawn}\,\overline{q}\;u\;(v{::}\zeta)\;\kappa)
\end{aligned}$$

From the right-hand-side of equation 5, we calculate:

$$\begin{aligned}
&\lambda u\zeta\kappa.\,\mathcal{E}[\![\mathbf{spawn}(e)]\!](u \circ \Gamma)(\lambda z.\overline{q}\,u(z{::}\zeta)\kappa)\\
&= \lambda u\zeta\kappa.\,\mathcal{E}[\![e]\!](u \circ \Gamma)(\lambda v\;\mathit{out}_{clo}\,v(\lambda f.\\
&\quad c_{spawn}(\lambda p'.f(\mathit{in}_{pid}\,p')(\lambda v.\mathbf{0}))\\
&\qquad (\lambda p'.\overline{q}\,u((\mathit{in}_{pid}\,p'){::}\zeta)\kappa)))
\end{aligned}$$

Comparing these calculations, we see that the induction step will work if *spawn* satisfies

$$\begin{aligned}
\mathit{spawn}\,\pi\;u\;(v{::}\zeta)\;\kappa = &\mathit{out}_{clo}\,v(\lambda f.\\
&c_{spawn}(\lambda p'.f(\mathit{in}_{pid}\,p')(\lambda v.\mathbf{0}))\\
&\quad(\lambda p'.\pi u((\mathit{in}_{pid}\,p'){::}\zeta)\kappa)),
\end{aligned}$$

which we can do by defining

$$\begin{aligned}
\mathit{spawn} = \lambda\pi u\zeta'\kappa.\mathit{pop}\,\zeta'(\lambda v.\;&\mathit{out}_{clo}\,v(\lambda f\zeta.\\
&c_{spawn}\,(\lambda p'.\;\mathit{applicate}\,f(\mathit{in}_{pid}\,p')(\lambda v.\mathbf{0}))\\
&\quad(\lambda p'.\pi u((\mathit{in}_{pid}\,p'){::}\zeta)\kappa))),
\end{aligned}$$

which is the definition of *spawn* in Figure 8. The rest of Figure 8 is derived similarly. For the case of procedures, we also need vary the symbol table Γ, as we varied π here; we then need *slookup* and *dlookup* to have the expected cooperative behavior [12, 37].

We next turn to the case for programs. We want to show that $\overline{\mathcal{CP}[\![p]\!]} \simeq_\beta \mathcal{P}[\![p]\!]$. To derive this, we calculate:

$$\begin{aligned}
&\overline{\mathcal{CP}[\![e]\!]}\\
&= \qquad\qquad \text{(definition of } \mathcal{CP}[\![-]\!])\\
&\overline{[p_0 : \langle \mathtt{start}, (\mathcal{CE}[\![e]\!]\langle\rangle\langle \mathtt{swap}, \langle \mathtt{call}, \langle \mathtt{return}\rangle\rangle\rangle)\rangle]}\\
&= \qquad\qquad \text{(definition of } \overline{(-)}\,)\\
&[p_0 : \mathit{start}\,\overline{(\mathcal{CE}[\![e]\!]\langle\rangle\langle \mathtt{swap}, \langle \mathtt{call}, \langle \mathtt{return}\rangle\rangle\rangle)}\,]\\
&\simeq_\beta \qquad\qquad \text{(IH at } e)\\
&[p_0 : \mathit{start}(\lambda u\zeta\kappa.\,\mathcal{E}[\![e]\!]\emptyset(\lambda f.\\
&\qquad \overline{\langle \mathtt{swap}, \langle \mathtt{call}, \langle \mathtt{return}\rangle\rangle\rangle}\;u(f{::}\zeta)\kappa)]\\
&\simeq_\beta \qquad\qquad \text{(definition of } \mathit{start})\\
&[p_0 : \mathcal{E}[\![e]\!]\emptyset(\lambda f.\\
&\qquad \overline{\langle \mathtt{swap}, \langle \mathtt{call}, \langle \mathtt{return}\rangle\rangle\rangle}\\
&\qquad\quad \langle\rangle(f{::}((\mathit{in}_{pid}\,c_{p_0}){::}\langle\rangle))(\lambda v.\mathbf{0}))]\\
&\simeq_\beta \qquad\qquad \text{(definition of } \overline{(-)}\,)\\
&[p_0 : \mathcal{E}[\![e]\!]\emptyset(\lambda f.\\
&\qquad (\mathit{swap}\,(\mathit{call}\;\mathit{return}))\langle\rangle(f{::}((\mathit{in}_{pid}\,c_{p_0}){::}\langle\rangle))(\lambda v.\mathbf{0}))]\\
&\simeq_\beta \qquad\qquad \text{(definition of } \mathit{swap})\\
&[p_0 : \mathcal{E}[\![e]\!]\emptyset(\lambda f.\\
&\qquad (\mathit{call}\;\mathit{return}))\langle\rangle((\mathit{in}_{pid}\,c_{p_0}){::}(f{::}\langle\rangle))(\lambda v.\mathbf{0}))]\\
&\simeq_\beta \qquad\qquad \text{(definition of } \mathit{call})\\
&[p_0 : \mathcal{E}[\![e]\!]\emptyset(\lambda f.\\
&\qquad \mathit{applicate}\,f\;(\mathit{in}_{pid}\,c_{p_0})(\lambda v.\;\mathit{return}\langle\rangle(v{::}\langle\rangle)(\lambda v.\mathbf{0})))]\\
&\simeq_\beta \qquad\qquad \text{(definition of } \mathit{return})\\
&[p_0 : \mathcal{E}[\![e]\!]\emptyset(\lambda f.\,\mathit{applicate}\,f\;(\mathit{in}_{pid}\,c_{p_0})(\lambda v.(\lambda v.\mathbf{0})v))]\\
&\simeq_\beta \qquad\qquad (\beta\text{-reduction})\\
&[p_0 : \mathcal{E}[\![e]\!]\emptyset(\lambda f.\,\mathit{applicate}\,f\;(\mathit{in}_{pid}\,c_{p_0})(\lambda v.\mathbf{0}))]\\
&= \qquad\qquad \text{(definition of } \mathcal{P}[\![-]\!])\\
&\mathcal{P}[\![e]\!]
\end{aligned}$$

∎

This proof allows us to explain the code generated by $\mathcal{CP}[\![e]\!]$. The *start* instruction initializes the stack with c_{p_0}, the value corresponding to the initial process id. The code for e is run, putting a closure on the stack. The top two elements of the stack are swapped, so they are in the right order for the *call* instruction, which applies the closure to the process id and starts the real execution of the program. When the procedure returns, the value is ignored and the process deadlocks, as desired.

5 Operational Semantics of the Target Language

So far we have given denotational semantics for our source language and for our target language as compositional translations into the process calculus HOCC, and we have shown that the compiler preserves denotation up to bisimulation.

Our next task is to give an operational semantics for the target language and to prove its adequacy relative to the HOCC semantics of the target language. Our strategy for doing this is the one that is implicit in [35, 29]: we want a transition relation $\rightarrow$ on the states $\langle q, u, z, k\rangle$ of our target machine such that $\langle q, u, z, k\rangle \rightarrow \langle q', u', z', k'\rangle$ implies

$$\overline{\langle q, u, z, k\rangle} \rightarrow_\beta \overline{\langle q', u', z', k'\rangle}\,. \tag{6}$$

To do this, however, we will need to handle three important details:

1. Unlike the sequential case, we will have to worry about the labels on the transitions, so that the observable actions of the two systems are equivalent.

2. Unlike HOCC, which uses the instantaneous computation rule, we want the operational semantics of the target machine to be a pure small-step semantics, in which the computational steps of the machine are spelled out.

3. We also need to make sure that the operational semantics has enough transitions. Without this requirement, we could satisfy (6) trivially by giving the machine no transitions at all.

We can satisfy all of these requirements by formulating the operational semantics of the target language as another process calculus, whose states are the system states of Figure 6 and whose actions are those of HOCC augmented by the silent transition τ. The silent transition will correspond to the computational actions of the machine. We will then give a branching bisimulation $\simeq$ between HOCC terms and system states, so that for any system state Q, $\overline{Q} \simeq Q$. This implies that any system state and its denotation will have behavior identical up to silent transitions.

5.1 Transition Rules

A system state consists of a set of 4-register machines, each associated with a process id. At each step, either any one machine can execute a computational instruction, or the system can take a communication step.

We first give the computation rules for individual machines. These specify transitions whenever the current instruction is something other than put, get, or spawn. These rules are given in Figure 10. They are similar to those found in, e.g., [38]. *clookup* performs lookup on the concrete structure u corresponding to *dlookup*. In addition, we have a default rule:

$$\langle q, u, z, k\rangle \longrightarrow_0 \langle \texttt{stopped}\rangle$$

whenever q matches a computational instruction (one of the ones in Figure 10), but $\langle q, u, z, k\rangle$ does not match any of the patterns in that figure. This rule takes care of error-checking. Thus, any finite maximal $\longrightarrow_0$ sequence must terminate either in a stopped state, or in a state in which the current instruction is either put, get, or spawn. This observation will be crucial to the adequacy theorem below.

Now we can give the communication rules for this system. These are shown in Figure 11. There are three rules. The first says that any machine in the system can perform a computational step, yielding a silent transition of the system. The second and third rules correspond to the communication rules in the HOCC system.

5.2 Adequacy of the Operational Semantics

Correspondence is defined as follows:

Definition 4 (correspondence, $\simeq$)
We say an HOCC term Π and a system state Q correspond (and write $\Pi \simeq Q$) iff $\Pi \simeq_\beta \overline{Q}$.

This correspondence cannot be a (strong) bisimulation because the computation, which was implicit (or instantaneous) in HOCC, is now explicit. Our goal is to show that it is a branching bisimulation. To establish this, we will show three things:

1. If $Q \xrightarrow{\tau} Q'$ and $\Pi \simeq_\beta \overline{Q}$, then $\Pi \simeq_\beta \overline{Q}'$.

2. If $Q \xrightarrow{a} Q'$ for $a \neq \tau$ and $\Pi \simeq_\beta \overline{Q}$, then there is a Π' such that $\Pi \xrightarrow{a} \Pi'$ and $\Pi' \simeq_\beta \overline{Q'}$.

3. If $\Pi \xrightarrow{a} \Pi'$ and $\Pi \simeq_\beta \overline{Q}$ then there is a Q' such that $Q \overset{\tau^* a}{\Rightarrow} Q'$ and $\Pi' \simeq_\beta \overline{Q'}$.

The first two conditions constitute a simulation theorem (like the one called "faithfulness" in [29]), which can be established easily by considering each transition of Q individually. The second condition is stronger than that required for Definition 2. The third is an adequacy result, which will require some more global reasoning. We shall do each of these as a separate lemma. We begin by considering the sequential case:

Lemma 1 *If $s \longrightarrow_0 s'$, then $\overline{s} \longrightarrow_\beta^+ \overline{s}'$ via a reduction sequence that includes at least one head reduction.*

Proof: By inspection of each transition $s \longrightarrow_0 s'$, including the default cases. ∎

The head reduction can always be chosen to be the first reduction in the sequence. The fact that the sequence contains at least one head reduction will be crucial in the adequacy lemma below.

Lemma 2 (Simulation for Silent Actions) *If $Q \xrightarrow{\tau} Q'$ and $\Pi \simeq_\beta \overline{Q}$, then $\Pi \simeq_\beta \overline{Q}'$.*

Proof: The silent transitions are given by the rule

$$\frac{s \longrightarrow_0 s'}{Q[p : s] \xrightarrow{\tau} Q[p : s']}$$

By the preceding lemma, we have $\overline{Q[p:s]} = \overline{Q}[p : \overline{s}] \simeq_\beta \overline{Q}[p : \overline{s}'] = \overline{Q[p:s']}$. So if $\Pi \simeq_\beta \overline{Q[p:s]}$, then $\Pi \simeq_\beta \overline{Q[p:s']}$ as well. ∎

Thus computational actions preserve denotation, just as in the sequential case.

Lemma 3 (Simulation for Observable Actions) *If $Q \xrightarrow{a} Q'$ and $\Pi \simeq_\beta \overline{Q}$, then there is a Π' such that $\Pi \xrightarrow{a} \Pi'$ and $\Pi' \simeq_\beta \overline{Q'}$.*

Proof: We have only two rules to consider. We show only the case of the spawn instruction:

$$Q[p : \langle\langle \texttt{spawn}, q\rangle, u, \langle\langle \texttt{closure}, q', u'\rangle, z\rangle, k\rangle]$$
$$\xrightarrow{a}$$
$$Q[p : \langle q, u, \langle\langle \texttt{pid}, p'\rangle, z\rangle, k\rangle]$$
$$[p' : \langle q', \langle\langle \texttt{pid}, p'\rangle, u'\rangle, \langle\rangle, \langle \texttt{halt}\rangle\rangle]$$

where $a = \langle \mathit{spawn}\ p\, p'\rangle$. Let L and R denote the left- and right-hand sides of this rule.

We will show that if $\Pi \simeq_\beta \overline{L}$, then there is a Π' such that $\Pi \xrightarrow{a} \Pi'$ (where $a = \langle \mathit{spawn}\ p\, p'\rangle$) and $\Pi' \simeq_\beta \overline{R}$.

Now,

$$\overline{L} = \overline{Q}[p : M_1]$$

$$
\begin{array}{ll}
\langle \texttt{start}, q\rangle & \longrightarrow_0 \langle q, \langle\rangle, \langle p_0, \langle\rangle\rangle, \langle \texttt{halt}\rangle\rangle \\
\langle\langle \texttt{fetch}, n, q\rangle, u, z, k\rangle & \longrightarrow_0 \langle q, u, \langle (\mathit{clookup}\ u\ n), z\rangle, k\rangle \\
\langle\langle \texttt{pushint}, n, q\rangle, u, z, k\rangle & \longrightarrow_0 \langle q, u, \langle\langle \texttt{int}, n\rangle, z\rangle, k\rangle \\
\langle\langle \texttt{pushclosure}, q', q\rangle, u, z, k\rangle & \longrightarrow_0 \langle q, u, \langle\langle \texttt{closure}, q', u\rangle, z\rangle, k\rangle \\
\langle\langle \texttt{return}\rangle, u, \langle v, z\rangle, \langle \texttt{frame}, q', u', z', k'\rangle\rangle & \longrightarrow_0 \langle q', u', \langle v, z'\rangle, k'\rangle \\
\langle\langle \texttt{call}, q\rangle, u, \langle v, \langle\langle \texttt{closure}, q', u'\rangle, z\rangle\rangle, k\rangle & \longrightarrow_0 \langle q', \langle v, u'\rangle, \langle\rangle, \langle \texttt{frame}, q, u, z, k\rangle\rangle \\
\langle\langle \texttt{sub}, q\rangle, u, \langle\langle \texttt{int}, n\rangle, \langle\langle \texttt{int}, m\rangle, z\rangle\rangle, k\rangle & \longrightarrow_0 \langle q, u, \langle\langle \texttt{int}, (m-n)\rangle, z\rangle, k\rangle \\
\langle\langle \texttt{swap}, q\rangle, u, \langle v, \langle w, z\rangle\rangle, k\rangle & \longrightarrow_0 \langle q, u, \langle w, \langle v, z\rangle\rangle, k\rangle
\end{array}
$$

Figure 10: Computation Rules for Individual Machines

$$
\frac{s \longrightarrow_0 s'}{Q[p:s] \xrightarrow{\tau} Q[p:s']}
$$

$$
\begin{gathered}
Q[p : \langle\langle \texttt{spawn}, q\rangle, u, \langle\langle \texttt{closure}, q', u'\rangle, z\rangle, k\rangle] \\
\xrightarrow{a} \\
Q[p : \langle q, u, \langle\langle \texttt{pid}, p'\rangle, z\rangle, k\rangle][p' : \langle q', \langle\langle \texttt{pid}, p'\rangle, u'\rangle, \langle\rangle, \langle \texttt{halt}\rangle\rangle] \\
\text{where } a = \langle \mathit{spawn}\ p\, p'\rangle \text{ and } p' \notin \mathrm{dom}(Q) \cup \{p\}.
\end{gathered}
$$

$$
\begin{gathered}
Q[p : \langle\langle \texttt{put}, q\rangle, u, \langle v, \langle\langle \texttt{pid}, p'\rangle, z\rangle\rangle, k\rangle][p' : \langle\langle \texttt{get}, q'\rangle, u', z', k'\rangle] \\
\xrightarrow{a} \\
Q[p : \langle q, u, \langle\langle \texttt{int}, 1\rangle, z\rangle, k\rangle][p' : \langle q', u', \langle v, z'\rangle, k'\rangle]
\end{gathered}
$$

where $a = \langle \mathit{send}\ p\, p'\, c\rangle$, where $c = v$ if v is a pid or integer, and $c = \langle \mathit{cl}\rangle$ if v is a closure.

Figure 11: Rules for Small-step semantics

where

$$
\begin{aligned}
& M_1 = \overline{\langle\langle \mathtt{spawn}, q\rangle, u, \langle\langle \mathtt{closure}, q', u'\rangle, z\rangle, k\rangle} \\
& = \qquad\qquad\qquad\qquad (\text{definition of } \overline{(-)}\,) \\
& spawn\,\overline{q}\,\overline{u}\,((in_{clo}(\lambda v\kappa'.\overline{q'}\,(v{::}\overline{u'}\,)\langle\rangle\kappa')){::}\overline{z}\,)\overline{k} \\
& =_\beta \qquad (\text{definition of } spawn,\ \beta\text{-reduction}) \\
& c_{spawn}(\lambda p'.\overline{q'}\,((in_{pid}\,p'){::}\overline{u'}\,)\langle\rangle(\lambda v.\mathbf{0})) \\
& \qquad (\lambda p'.\overline{q}\,\overline{u}\,((in_{pid}\,p'){::}\overline{z}\,)\overline{k}\,)
\end{aligned}
$$

Let us call the last term M_2.

If $\Pi \simeq_\beta \overline{L}$, Π must be of the form $\Pi_0[p : M]$ where $\Pi_0 \simeq_\beta \overline{Q}$ and $M =_\beta M_1 =_\beta M_2$. Therefore, by the rule for c_{spawn}, $\Pi \xrightarrow{a} \Pi'$, where $\Pi' =$

$$
\begin{aligned}
& \Pi_0[p : ((\lambda p'.\overline{q}\,\overline{u}\,((in_{pid}\,p'){::}\overline{z}\,)\overline{k}\,)c_{p'})] \\
& \quad [p' : ((\lambda p'.\overline{q'}\,((in_{pid}\,p'){::}\overline{u'}\,)\langle\rangle(\lambda v.\mathbf{0}))c_{p'})] \\
& \simeq_\beta \qquad\qquad\qquad\qquad (\beta\text{-reduction}) \\
& \Pi_0[p : \overline{q}\,\overline{u}\,((in_{pid}\,c_{p'}){::}\overline{z}\,)\overline{k}\,] \\
& \quad [p' : \overline{q'}\,((in_{pid}\,p'){::}\overline{u'}\,)\langle\rangle(\lambda v.\mathbf{0})] \\
& = \qquad\qquad\qquad\qquad (\text{definition of } \overline{(-)}\,) \\
& \Pi_0[p : \overline{\langle q, u, \langle\langle \mathtt{pid}, p'\rangle, z\rangle, k\rangle}\,] \\
& \quad [p' : \overline{\langle q', \langle\langle \mathtt{pid}, p'\rangle, u'\rangle, \langle\rangle, \langle\rangle\rangle\mathtt{halt}}\,] \\
& \simeq_\beta \overline{Q}\,[p : \overline{\langle q, u, \langle\langle \mathtt{pid}, p'\rangle, z\rangle, k\rangle}\,] \\
& \qquad [p' : \overline{\langle q', \langle\langle \mathtt{pid}, p'\rangle, u'\rangle, \langle\rangle, \langle\rangle\rangle\mathtt{halt}}\,] \\
& = \overline{R}
\end{aligned}
$$

So $\Pi \xrightarrow{a} \Pi'$ and $\Pi' \simeq_\beta \overline{R}$, as desired. This completes the case of the c_{spawn} transition.

The proof for the $c_{send}/c_{receive}$ transition works the same way. ∎

We next attack the third condition. This is a liveness condition: it says that if one has an HOCC term that is the translation of a machine state, and that HOCC term is willing to communicate, then the machine will eventually reach a communicating state. Determining that the HOCC term is willing to communicate may involve an arbitrary amount of β-reduction. So the liveness condition reduces to: if the HOCC term (which is, after all, a λ-term) reduces, via some sequence of β-reductions, to a term headed by one of the communications constants, will the machine, using its fixed reduction strategy, be able to find it? The answer comes from the λ-calculus:

Lemma 4 *Let M be any λ-term. If M has an infinite β-reduction sequence which contains infinitely many head reductions, then M has no head normal form.*

Proof: Let $S = M \to M_1 \to M_2 \to \ldots$ be an infinite β-reduction sequence with infinitely many head reductions. Since every head reduction is a leftmost reduction, we can translate Figure 13.4 of [7] into our notation, we get the diagram of reductions shown in Figure 12. Here the arrows labelled by l^* denote an arbitrary sequence of leftmost reductions; those labelled by l^+ denote a non-empty sequence of leftmost reductions, and those labelled by $\neq l$ denote an arbitrary sequence of non-leftmost reductions.

This shows that there is an infinite leftmost reduction sequence $M \to_l^* L_1 \to_l^+ L_2 \to_l^+ L_3 \ldots$ such that for each i, $L_i \to^* M_i$. Since M_i is not in head normal form, neither can L_i be in head normal form. Hence the leftmost reduction sequence from M contains no head normal form. Hence M has no head normal form. ∎

Now we can prove the third condition:

Lemma 5 (Adequacy) *If $\Pi \xrightarrow{a} \Pi'$ and $\Pi \simeq_\beta \overline{Q}$ then there is a Q' such that $Q \overset{\tau^* a}{\Rightarrow} Q'$ and $\Pi' \simeq_\beta \overline{Q'}$.*

Proof: Consider the case that $a = \langle spawn\ p\, p'\rangle$. It must be that $\Pi = \Pi_0[p : M]$, $M =_\beta c_{spawn}NP$, and $\Pi' = \Pi[p' : (Nc_{p'})][p : (Pc_{p'})]$. Since $\Pi \simeq_\beta \overline{Q}$ we must have $Q = Q_0[p : s]$ where $\Pi_0 \simeq_\beta \overline{Q_0}$ and $M =_\beta \overline{s}$. Hence $\overline{s}$ has a head normal form. Now observe the sequence of states $Q_0[p : s_i]$ obtained by always selecting machine p for execution. By Lemma 1, the sequence of λ-terms $\overline{s_i}$ forms a reduction sequence with at least one head reduction for each i. Since $M =_\beta \overline{s_0}$ has a head normal form, this sequence cannot be infinite, by Lemma 4. Hence it must reach a state s_k that is either a `stopped` state, or in a state in which the current instruction is either `put`, `get`, or `spawn`. Since $\overline{s_k} =_\beta \overline{s_0} =_\beta M =_\beta c_{spawn}NP$, it must be that the first instruction in s_k is a `spawn`. Furthermore, by the definition of *spawn*, the stack of s_k must be non-empty, and its first component must be a closure.

So we have

$$
\begin{aligned}
Q & \overset{\tau}{\Rightarrow} Q_0[p : \langle\langle \mathtt{spawn}, q\rangle, u, \langle\langle \mathtt{closure}, q', u'\rangle, z\rangle, k\rangle] \\
& \xrightarrow{a} Q[p : \langle q, u, \langle\langle \mathtt{pid}, p'\rangle, z\rangle, k\rangle] \\
& \qquad [p' : \langle q', \langle\langle \mathtt{pid}, p'\rangle, u'\rangle, \langle\rangle, \langle \mathtt{halt}\rangle\rangle]
\end{aligned}
$$

An argument like that in Lemma 3 establishes that this is the required Q'.

The case for send/receive is similar, except that we need for both machines p and p' to run to head normal form. ∎

From the two simulation lemmas and the adequacy lemma we conclude:

Theorem 3 (Correctness of the Abstract Machine) *$\simeq$ is a branching bisimulation.*

Proof: It is straightforward to check that Lemmas 2, 3, and 5 imply the conditions for branching bisimulation. ∎

6 Putting it together

Putting these results together we deduce our final version of the correctness of the compiler: that the semantics of any program is branching-bisimilar to the initial state of the machine running the compiled code.

Theorem 4
(Correctness of the Compiler and Machine)
For any program e, $\mathcal{P}[\![e]\!]$ and $\mathcal{CP}[\![e]\!]$ are branching-bisimilar.

Proof: By Theorem 2, $\mathcal{P}[\![e]\!] \simeq_\beta \overline{\mathcal{CP}[\![e]\!]}$. So, by the definition of $\simeq$, $\mathcal{P}[\![e]\!] \simeq \mathcal{CP}[\![e]\!]$. So, by Theorem 3, $\mathcal{P}[\![e]\!]$ and $\mathcal{CP}[\![e]\!]$ are branching-bisimilar. ∎

This implies, for example, the trace equivalence of $\mathcal{P}[\![e]\!]$ and $\mathcal{CP}[\![e]\!]$: by the remark following Definition 2, it follows that $Traces(\mathcal{P}[\![e]\!]) = Traces(\mathcal{CP}[\![e]\!])$.

This also implies the adequacy of the translation in the sense of [1]. Here we provide a sketch of the proof. For expressions e, e', define $e \approx e'$ iff for all program contexts $P[-]$, $P[e]$ and $P[e']$ are bisimilar. Define $\approx$ on system states

$$
\begin{tikzcd}
M \arrow[d, ->>, "l^*"'] & & & \\
L_1 \arrow[r, ->>, "\neq l"] \arrow[d, ->>, "l^+"'] & M_1 \arrow[d, ->>, "l^+"] & & \\
L_2 \arrow[r, ->>, "\neq l"] \arrow[d, ->>, "l^+"'] & \bullet \arrow[r, ->>, "\neq l"] \arrow[d, ->>, "l^+"] & M_2 \arrow[d, ->>, "l^+"] & \\
L_3 \arrow[r, ->>, "\neq l"] & \bullet \arrow[r, ->>, "\neq l"] & \bullet \arrow[r, ->>, "\neq l"] & M_3 \\
\vdots & \vdots & \vdots & \vdots
\end{tikzcd}
$$

Figure 12: Barendregt's Figure 13.4

(Figure 6) similarly. Then $\mathcal{CE}$ is adequate in the sense that if $\mathcal{CE}[\![e]\!] \approx \mathcal{CE}[\![e']\!]$, then $e \approx e'$. To show adequacy of the translation, we consider the contrapositive: if $e \not\approx e'$, then there must be some context $P[-]$ that distinguishes them. But Theorem 2 implies that $\mathcal{CP}[\![P[e]]\!]$ and $\mathcal{CP}[\![P[e']]\!]$ are distinguishable. But it is easy to show that for any program context $P[-]$ there is a system context $Q[-]$ such that for any e, $\mathcal{CP}[\![P[e]]\!] = Q[\mathcal{CE}[\![e]\!]]$. Hence $Q[-]$ is a distinguishing context for $\mathcal{CE}[\![e]\!]$ and $\mathcal{CE}[\![e']\!]$. This notion of "adequacy" only shows that the translation does not confuse any expressions; unlike ours, it does not provide a direct connection between the behavior of the original and translated terms.

7 Variations

It should be clear that the paradigm sketched above can be carried out over a considerable variation of languages. In this section, we will indicate some of the variations that appear possible.

Shared vs. Distributed Memory Since our source language has no state, it does not distinguish between shared-memory and distributed-memory models. In a shared-memory model, our program states would look like

$$[p_1 : M_1] \ldots [p_n : M_n]\Sigma$$

where Σ is a store; operations on the shared memory would be modelled as operations that communicate with Σ with constants like c_{fetch} and c_{store}. Similarly, the system states would have a global memory.

In a distributed-memory model, a local store would be associated with each process, and store operations would be treated as ordinary computation operations. Hence a typical process state would look like $[p : \mathcal{E}[\![e]\!]\rho\kappa\sigma]$, and our machines would have 5 registers instead of 4, adding a store register σ to the existing π, u, z, and k registers. These two models could of course be combined.

Implicit vs. Explicit Process Indexing Our language was based on the implicit assignment of process id's to processes: when a new process was spawned, its process id was assigned non-deterministically. This is reminiscent of the treatment of process identifiers in Unix.

In other models of parallel programming, one might wish to identify these quantities as identifying processors rather than processes. One might wish to spawn new processes on specific processors, and to do computations on processor numbers (for example, in algorithms such as broadcast or reduction on a hypercube). Our paradigm should be easily extendible to treat this.

Channels vs. Process-based Communication We chose process-to-process communication for our language using Occam's Razor: it required the least amount of formal overhead for demonstrating the techniques. Fortuitously, this model is also used in a number of parallel programming systems that appear to be in popular use, e.g. [8, 14]. On the other hand, there seems to be no difficulty adapting our approach to a channel-based system; one would need either a global used-channel resource like the shared store Σ above or an additional "local-available-channels" variable for each machine [1]; see also [5] for a similar algorithm.

Interleaving vs. Concurrency Our machine semantics is an interleaving semantics, in that at most one machine executes an instruction at a time. Gladstein [17] considers the case of concurrent execution, in which several machines may compute or communicate in a given cycle.

Other Communications Primitives Our approach is strongly influenced by [10]: any communication primitive that is needed can be coded as additional rules in HOCC. So long as the general form of the rules is maintained (in particular, the use of $=_\beta$ hypotheses to incorporate the instantaneous computation principle and unambiguity of actions), then the rest of the technique can be carried through as before. In particular, we have considered non-blocking writes, barrier synchronization, distinguished output operations, and restricted forms of dead-process collection, and all seem treatable in this framework. Other primitives, such as choice, seem more complex but not insuperable.

8 Related Work

The work reported in this paper depends upon techniques developed in both the compiler-correctness and concurrency communities. There is a large body of work in each of these communities, so we can at best list a few relevant papers.

The work reported herein is an extension of the techniques introduced in [35, 12]. A number of small to moderate-sized compiler correctness proofs have been accomplished using these techniques: a simple functional language [35, 34], a language with **while**-expressions [36], a language obeying stack-discipline [38], and core Scheme [12]. More recently, this technique has been used for two large proofs: one for a very detailed proof of a complete Scheme system [21], and one for a detailed proof of a compiler for PreScheme, a Scheme-like language designed for systems programming [29].

There has been some work on compiler correctness for languages with concurrency. The Procos project, using techniques of Hoare, Jifeng, and Bowen, has been developing a wide-spectrum approach ranging from requirements analysis to hardware synthesis [11]. Our work corresponds roughly to their translation of the programming language PL into a lower-level software language. Their work uses a "predicative" or relational style of semantics, which allows simpler connections to the specification language in which system requirements are expressed. Our approach is more tailored to the embedding of parallel constructs in existing, more complex programming languages.

The goal of the Silo project [40] is a mechanized verification of a secure distributed system, using a layered approach like the CLI stack [27] and the HOL theorem-prover [20]. They have a used a relational-style semantics to verify a compiler from microSR, a derivative of the distributed programming language SR [3], into an abstract target machine language [39]. Their intention is to concentrate on the lower layers of the stack, corresponding to the translation of high-level communication primitives like ours into lower-level communications. We have not yet considered this problem, but we suspect techniques such as those in [38] will be useful.

Most of the work in the concurrency community can be thought of as investigations of the algebraic and mathematical properties of intermediate languages like our HOCC. We discuss here a few papers that are more closely related.

Bloom [9] introduced the idea of the "instantaneous computation rule". The idea of having the computation treat the communications primitives as uninterpreted constants comes from the Nielsons' formulation of process algebras for CML [28]. They use a small-step semantics for both computation and communication, whereas we use a version of the instantaneous computation rule, which allows us to revert to the λ-calculus when considering computation alone.

Gordon [18] presents four models of input-output in functional language, using an operational semantics. His continuation-passing model uses rules that are almost identical to ours in format.

Facile [15, 16] is intended as a source language; it is specified by a compositional translation into "core Facile," which is more like an intermediate language. Core Facile is based on a typed call-by-value λ-calculus with channel-based communications and a call/cc operator. Amadio [1] gives translations between several versions of this language, including a translation into an environment-based machine similar to ours. He shows that his translations are adequate in the sense that they do not confuse distinguishable programs, as discussed following Theorem 4. The semantics of core Facile is given as a pure small-step semantics, for both computation and communication steps. This complicates the notion of observability, and requires considerable reasoning about commitment, administrative reductions, etc. Our formulation, in which β-equivalence is automatically a bisimulation, seems easier to reason about.

Flanagan and Felleisen [13] consider a sequential functional language with a **future** construct that allows possible speculative computation. The language is more like an intermediate language than a source language, and the goal is to compute a single answer, rather than to realize some behavior. The semantics are specified as labelled transition systems in which the labels are not observable actions but step counts, so $q \stackrel{n,m}{\to} q'$ iff q can go to q' via n steps, of which m are non-speculative. The resulting labelled transition systems are highly non-deterministic but are shown confluent.

Milner has shown how to translate the λ-calculus into the π-calculus in a way that preserves and reflects β-convertibility [24, 25]. Walker [33] shows how two languages similar to POOL [2] can be compositionally translated into the π-calculus, but he gives no relation between this translation and other semantics for the language. Thomsen's original article on CHOCS also illustrates the use of CHOCS as an intermediate language [31]. Thomsen translates the λ-calculus into CHOCS and shows that the translation preserves, but does not reflect, β-convertibility. He also gives a compositional translation from a toy language with procedures into CHOCS.

However, neither the π-calculus nor CHOCS seems so well suited as the λ-calculus for generating reasonable code for the sequential portions of the language, as we have described; by using the λ-calculus as an intermediate language, we get to reuse large amounts of reasoning from sequential-language compiler proofs. Other arguments for the use of λ-calculus as an intermediate language are given by Appel [4].

9 Conclusions

We have shown how the approach to compiler correctness begun in [35, 36, 12, 29] can be extended to languages with parallelism. The semantics of the source language is given as a compositional translation to a higher-order process calculus (instead of to a λ-calculus). The target language is given a denotational semantics as a compositional translation to the same process calculus. The compiler is correct if it preserves denotation up to bisimulation. The target language is also given an operational semantics, and this operational semantics is shown correct in the sense that it is branching-bisimilar to the denotational semantics of the target language.

The analogy between the sequential and parallel cases continues at the next level of detail. Our higher-order process calculus, called HOCC, is formulated to include the λ-calculus syntactically, so that β-convertibility induces a bisimulation; indeed, almost all the required reasoning is done in the λ-calculus. Since both the semantics and the compiler are given compositionally, the proof of congruence takes the form of a simple structural induction on source-language programs. In the sequential case, the adequacy theorem is divided into two parts: a simulation, or soundness, theorem that is shown by an easy induction on the steps of the machine, and a completeness theorem that shows that the machine terminates often enough; the latter is shown by appeal to the completeness of quasi-leftmost reduction sequences (Lemma 4). In the parallel case, the adequacy theorem is expressed as a weak bisimulation; establishing the two halves of the weak bisimulation involves a similar easy induction on machine steps and a similar appeal to the completeness of quasi-leftmost reduction sequences.

The development seems to depend only on fairly weak properties of the systems chosen, so it seems likely that the results will be extendible to a variety of related settings.

Acknowledgements

This work grew out of our conversations with Dave Gladstein, who also created Figure 12. Allyn Dimock gave us excellent presentations on the π-calculus. Thanks to Bryant York, Jens Palsberg, Dino Oliva, Greg Sullivan, Paul Steckler, and Chris Okasaki, who read or listened to the paper in various versions, and gave much useful feedback.

References

[1] Roberto M. Amadio. Translating Core Facile. Technical Report ECRC-1994-3, ECRC, February 1994.

[2] Pierre America, Jaco de Bakker, Joost N. Kok, and Jan 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] G. R. Andrews and R. A. Olsson. *The SR Programming Language: Concurrency in Practice.* Benjamin/Cummings, 1993.

[4] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, Cambridge, 1992.

[5] Lennart Augustsson, Mikael Rittri, and Dan Synek. On Generating Unique Names. *Journal of Functional Programming*, 4:117–123, 1994.

[6] J. C. M. Baeten and W. P. Weijland. *Process Algebra.* Cambridge University Press, 1990.

[7] Henk P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* North-Holland, Amsterdam, 1981.

[8] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, K. Moore, and V. Sunderam. PVM and HeNCE: Tools for Heterogeneous Network Computing. In J. S. Kowalik and L. Grandinetti, editors, *Software for Parallel Computation*, volume 106 of *NATO ASI Series F.* Springer-Verlag, 1993.

[9] Bard 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.

[10] Bard Bloom. Structured Operational Semantics as a Specification Language. In *Conf. Rec. 22nd ACM Symposium on Principles of Programming Languages*, pages 107–117, 1995.

[11] Jonathan Bowen, Martin Franzle, Ernst-Rudiger Olderog, and Anders P. Ravn. Developing Correct Systems. In *Proceedings of the 5th Euromicro Workshop on Real-Time Systems.* IEEE Computer Society Press, 1993.

[12] William Clinger. The Scheme 311 Compiler: An Exercise in Denotational Semantics. In *Proc. 1984 ACM Symposium on Lisp and Functional Programming*, pages 356–364, August 1984.

[13] Cormac Flanagan and Matthias Felleisen. The Semantics of Future. Technical Report COMP TR94-238, Rice University, Department of Computer Science, October 1994.

[14] G. A. Geist and V. S. Sunderam. The PVM System: Supercomputer Level Concurrent Computation on a Heterogeneous Network of Workstations. In *Sixth Annual Distributed-Memory Computer Conference*, pages 258–261, 1991.

[15] Alessandro Giacalone, Prateek Mishra, and Sanjiva Prasad. Facile: A Symmetric Integration of Concurrent and Functional Programming. *International Journal of Parallel Programming*, 18(2):121–160, 1989.

[16] Alessandro Giacalone, Prateek Mishra, and Sanjiva 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.

[17] David Gladstein. *Compiler Correctness for Concurrent Languages.* PhD thesis, Northeastern University, December 1994.

[18] Andrew D. Gordon. *Functional Programming and Input/Output.* Cambridge University Press, Cambridge, 1994.

[19] Michael J. C. Gordon. *The Denotational Description of Programming Languages.* Springer-Verlag, Berlin, Heidelberg, New York, 1979.

[20] Michael J. C. Gordon and T. F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic.* Cambridge University Press, Cambridge, 1993.

[21] J. D. Guttman, V. Swarup, and J. Ramsdell. The VLISP Verified Scheme System. *Lisp and Symbolic Computation*, 8(1/2), 1995. to appear.

[22] Matthew Hennessy and Robin Milner. Algebraic Laws for Nondeterminism and Concurrency. *Journal of the ACM*, 32:137–161, 1985.

[23] Robin Milner. Operational and Algebraic Semantics of Concurrent Processes. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 1201–1242. MIT Press/Elsevier, 1990.

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

[25] Robin Milner, Joachim Parrow, and D. Walker. A calculus of mobile processes (parts I and II). *Information and Computation*, 100:1–77, 1992.

[26] Torben Æ. Mogensen. Efficient Self-Interpretation in Lambda Calculus. *Journal of Functional Programming*, pages 345–364, June 1992.

[27] J. Strother Moore. A Mechanically Verified Language Implementation. *Journal of Automated Reasoning*, 5(4):461–492, December 1989.

[28] Flemming Nielson and Hanne Riis Nielson. From CML to Process Algebras. In *Proceesings of CONCUR '93*, pages 493–508, Berlin, Heidelberg, and New York, 1993. Springer-Verlag.

[29] Dino P. Oliva, John D. Ramsdell, and Mitchell Wand. The VLISP Verified PreScheme Compiler. *Lisp and Symbolic Computation*, 8(1/2), 1995. to appear.

[30] David A. Schmidt. *Denotational Semantics: A Methodology for Language Development.* Allyn and Bacon, Boston, 1986.

[31] Bent Thomsen. A Calculus of Higher Order Communicating Systems. In *Conf. Rec. 16th ACM Symposium on Principles of Programming Languages*, pages 143–154, 1989.

[32] Christopher P. Wadsworth. Some Unusual λ-calculus Numeral Systems. In J. P. Hindley and J. R. Seldin, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism*, pages 215–230. Academic Press, New York and London, 1980.

[33] David 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.

[34] Mitchell Wand. Deriving Target Code as a Representation of Continuation Semantics. *ACM Transactions on Programming Languages and Systems*, 4(3):496–517, July 1982.

[35] Mitchell Wand. Semantics-Directed Machine Architecture. In *Conf. Rec. 9th ACM Symposium on Principles of Programming Languages*, pages 234–241, 1982.

[36] Mitchell Wand. Loops in Combinator-Based Compilers. *Information and Control*, 57(2–3):148–164, May/June 1983.

[37] Mitchell Wand. A Short Proof of the Lexical Addressing Algorithm. *Information Processing Letters*, 35:1–5, 1990.

[38] Mitchell Wand and Dino P. Oliva. Proving the Correctness of Storage Representations. In *Proc. 1992 ACM Symposium on Lisp and Functional Programming*, pages 151–160, 1992.

[39] Cui Zhang, Brian R. Becker, Mark R. Heckman, Karl Legitt, and Ron A. Olsson. A Hierarchical Method for Reasoning about Distributed Programming Languages and its Applications. submitted for publication, 1995.

[40] Cui Zhang, Rob Shaw, Mark R. Heckman, Gregory D. Benson, Myla Archer, Karl Levitt, and Ronald A. Olsson. Towards a Formal Verification of a Secure Distributed System and its Applications. In *Proceedings of the 17th National Computer Security Conference*, pages 103–113, Baltimore, October 1994.