COMPILER CORRECTNESS FOR CONCURRENT LANGUAGES

A Thesis
Presented to the Faculty of the Graduate School
of the College of Computer Science
of Northeastern University
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy

by
David S. Gladstein
December 1994

sāṁkhya-yogau pṛthag bālāḥ pravadanti na paṇḍitāḥ
ekam apy āsthitaḥ samyag ubhayor vindate phalam.

'There must be a difference between theory and practice', so say the simple-minded, not the wise. Apply yourself to only one whole-heartedly and win the fruit of both.

yat sāṁkhyaiḥ prāpyate sthānaṁ tad yogair api gamyate;
ekaṁ sāṁkhyaṁ ca yogaṁ ca: yaḥ paśyati sa paśyati.

[True,] the men of [contemplative] theory attain a [high] estate, but that [same estate] achieve the men of practice too; for theory and practice are all one: who sees [that this is true], he sees [indeed].

Bhagavad-Gītā 5. 4–5

This thesis is dedicated to my family and friends, who kept me going.

Abstract

The goal of my thesis is to extend current methods for compiler derivation and verification to languages which exhibit true concurrency. The work has proceeded through the following steps.

A process calculus, called CLAM, was defined, based on the λ-calculus and modelling true concurrency. CLAM was used to define the semantics of a simple concurrent functional language, called Urlang, which was inspired by the programming language Erlang, which is in industrial use. A bytecode compiler for Urlang was then derived and proven correct, and a multiprocessor bytecode interpreter was implemented in Concert/C and proven correct. Finally it is argued that those features of Erlang which are not present in Urlang are easily added into our framework, either via source-to-source transformations into core Urlang or by means of straightforward additions to the operational semantics of CLAM.

Acknowledgments

This thesis would not have been possible without my advisor, Mitchell Wand. I thank him for his unflagging support, sage advice (technical and otherwise), and friendship throughout the genesis of this work.

I wish to express my gratitude to the other members of my committee: Karl Lieberherr, John Ramsdell, and Bryant York. They gave me valuable comments and advice, at the end working under extreme time pressure.

I would like to thank the various members of the weekly Semantics Seminar for many stimulating presentations and discussions.

Germán Goldszmit and the Concert/C group at IBM provided amazingly timely technical support during my initial implementation efforts.

Last, but not least, I wish to thank Rémy Evard, the Systems Group, and the Crew. They have built a computing environment the like of which I may never see again.

Contents

List of Figures

Chapter 1

Introduction

Sometimes the light's all shining on me;
Other times I can barely see.
Lately it occurs to me
What a long, strange trip it's been.

—*Grateful Dead*

1.1 Motivation and Summary

Because compilers are a trusted part of almost every computer system, it is important to have trustworthy proofs of their correctness. Because it is expedient to build concurrent applications by adding communication libraries to existing procedural languages, people do so.

This thesis extends the current work in compiler verification to languages for concurrent computation, in particular to languages which mirror current practice. This requires a formal calculus for modelling concurrent computation, as well as methods of reasoning about that calculus. The calculus is used to define the semantics of a simple language which is typical of the systems in use on today's machines for concurrent programming. Extensions of current methods are used to derive a compiler for the language and to verify its correctness, and to derive a sequence of progressively less abstract machines which preserve the language's operational semantics.

The style of concurrent computing we are espousing is inspired by the language Erlang [AVW93], the general style of which is the unifying theme of this work. While Erlang plays a central role in the genesis of this work, it is by no means the only programming system with process-based concurrency and asynchronous message passing: that paradigm is also present in the PVM and Concert/C systems, which are discussed in section 2.1. It is the prevalence of this paradigm in multicomputer programming that makes it a worthwhile area of study.

We consider the following facets of this work to be our major original contributions:

1. We develop a value-passing calculus, CLAM, for concurrent computation which models truly concurrent behavior despite having an operational semantics described in terms of interleaved transitions (Chapter 4).

2. We give a semantics for a small functional language, Urlang, in terms of CLAM, and show how how the interleaving description of CLAM allows us to build on previous work in compiler derivation to derive a bytecode compiler for Urlang and prove it correct (Chapter 5).

3. We demonstrate the plausibility of CLAM by implementing the Urlang bytecode interpreter in Concert/C and proving the implementation correct (Chapter 6).

4. We argue for the significance of the foregoing accomplishments by showing how our framework is sufficient to implement those features of Erlang not already present in our language (Chapter 7).

1.2 Overview

This section provides an overview of the rest of this thesis.

Chapter 2 reviews some of the previous work which has influenced or contributed directly to the present study: existing concurrent programming systems, the Wand-Clinger compiler correctness programme, and some formal calculi for concurrency, including HOCC.

As this work is an attempt to marry compiler correctness to concurrent computation via the formalisms of process calculus, it is appropriate for Chapter 3 to provide some introductory material on compiler correctness and process calculi in general.

The foundation of the current work is a concurrent extension of the untyped λ-calculus, called CLAM. It is designed specifically to provide a true concurrency model of computation, as well as to model asynchronous message passing. Chapter 4 presents the philosophical and technical considerations which led to the CLAM formalism, as well as the calculus itself.

The raw λ-calculus is not a programming language for most people, but it can serve as the basis of formalisms for defining programming language semantics and deriving compilers and abstract machines for those languages. CLAM plays the same role in the present work, in the setting of concurrent computation. Chapter 5 defines the syntax and semantics of a small, concurrent, functional language, called Urlang. As is typical, the semantics is a compositional translation of source (Urlang) phrases into terms of a calculus (CLAM) in an environment-continuation style. Using extensions of standard techniques, an abstract machine is derived and proven correct.

It is no trick to define an abstract machine and prove it equivalent to some calculus, since the criteria for distinguishing an abstract machine from a formal calculus are very subjective. However, we will want our abstract machines to be *plausibly implementable,* and that requires that we be able to implement the states and transitions of the machine efficiently on a real computing system. For this reason we did an actual Concert/C implementation of our abstract machine, presented in Chapter 6. Using an actual, implemented concurrent programming language as the basis for our bytecode interpreter helps insure against objections of implausibility, as long as the implementation of each bytecode corresponds to a small amount of straight line code in the implementation language.

One might argue that it's too easy to make a language easy to implement, especially in the context of an academic research project. To forestall this objection to the realism of Urlang, we show how a stylized version of Erlang, called FINE, can be defined in terms of the constructs of Urlang. While the translation given in Chapter 7 does not account for every feature of Erlang, it would be straightforward, if unenlightening, to extend the present work to do so.

The semantics of CLAM as presented in our thesis proposal suffered from two defects. The first was that the processes of a program proceed in lockstep: although each had the option of not taking a step, such steps as were taken were synchronized, rather like the supersteps of BSP [Val90]. This was unintentional, and it is not clear whether a bisimulation could be constructed between CLAM with that semantics and the Concert/C implementation. The second, and more serious defect was that the use of RPC function calls in the bytecode interpreter allowed the possibility of deadly embrace between processes in cases where the semantics did not. (Obviously this means that the first problem was insurmountable; the question remains as to whether the original semantics can be implemented, given that there is a simple software workaround to the deadly embrace problem.) Chapter 8 gives the details, and points out how the present methodology overcomes these obstacles.

Chapter 2

Previous Work

This section discusses the state of various areas related to the present work.

2.1 Current Concurrent Programming Practice

Process calculi and languages can be categorized along several dimensions. For each, one can consider the impact of various design decisions on the life of the programmer.

- Is the process structure fixed, or determined dynamically? If fixed, the programmer writes down a set of processes. If dynamic, processes spawn other processes.

  Obviously, dynamic structure is more flexible, allowing the program to adapt the process structure to the contingencies of the problem at hand.

- Is communication synchronous, requiring two ready partners for a communication event, or is it asynchronous, involving a buffer?

  This seems to me to be a matter of some neutrality, as the two facilities are interdefinable. It may be that asynchronous message passing fits better the conceptual model of workflow which occurs in many distributed computations.

- Is communication between processes, or along a channel?

  Again, the two facilities are easily interdefinable. I believe that process-centered communication leads to more readable programs, simply because the semantics of a message must reside entirely in the message itself, rather than perhaps being represented partially by the channel on which the message is sent.

- What, if anything, gets communicated? Is it an arbitrary value, a value from a restricted domain, or perhaps a dataless signal?

  It has been argued that value passing is unnecessary, since it is definable in terms of dataless signals: simply associate one signal with each possible data value [Mil89]. This is not as outrageous as it first sounds, since a bundle of 3 signals allows one to communicate variable length bit strings.

Obviously, systems which permit more kinds of values to be communicated are better for practical use than those which permit fewer, all things being equal. The programmer will do what he has to to encode data in a message; the less encoding needing to be done by the application, the more concise the program.

- What computational features are there, if any?

  The trend in implementations seems to be to take (or make up) a reasonable algorithmic language, and add sufficent features to allow programs to spawn and communicate. The purpose of scientific computation is, after all, computation; concurrency and communication are necessary evils which we tolerate in the search for a linear speedup.

In support of the foregoing raft of opinions, we discuss three concurrent programming systems, Erlang [AVW93], PVM [G+93], and Concert/C [A+94]. Erlang is a full-fledged programming language, while PVM is just a set of UNIX communication and process control routines. Erlang was developed for prototyping purposes by people interested mostly in comfort, while PVM was created by computational scientists interested mostly in speed. Concert/C falls in between the two. It is an extension to ANSI C, and the communication facilities are manifested as new statements and datatypes in the extended language, not just library routines.

What is striking is that in all three cases, the underlying model of concurrency is that of processes communicating via asynchronous messages, and that the various camps of users are quite satisfied with their respective systems. This argues for the usefulness of systems based on this particular model of concurrency.

2.1.1 Erlang

Erlang is a language developed by Ericsson laboratories, initially intended for prototyping in telephone systems. It eventually turned out to be efficient enough for production use. Erlang's features were selected for ease of use, and not for efficiency of implementation. For pragmatic reasons, features were kept to a minimum, consistent with making it easy to write concise programs. Erlang's major features are as follows:

- Only functional style is supported, the intent being that functional programs are intrinsically easier to understand than imperative programs. All variables are single-assignment, and data structures cannot be modified.

- The language provides Scheme-like data (symbols, strings, lists, tuples), easy-to-use "brackets and commas" constructors, and automatic garbage collection. Since programs may need to construct and destructure a lot of different kinds of complex messages, it is important that the operations involved in dealing with messages be easy to use. The virtues of automatic storage management are well known.

- Concurrency is centered around processes. Processes are used for conceptual modeling (users, phone calls, etc.), as well as programming.

- Pattern-matching is used for function invocation and receiving messages. This leads to very succinct programs. Since messages are tagged, the programmer concentrates on what the message says, rather than (say) imputing meaning based on what channel it came in on.
- I/O ports and X widgets are modelled as processes, so they send and receive messages.
- Erlang's implementation makes distributed computing almost transparent: there's a variant of spawn which names a host, and after that you forget about where the process is.
- Data is copied out of a process' address space upon message send, so processes' data spaces stay small enough that gc doesn't interfere with "soft real-time" applications.
- There are no internal functions, closures, or first-class functions, but the language is properly tail recursive.
- Functions can be replaced while a program is running, by replacing top-level function definitions.
- A process may trap its own runtime errors, and may arrange to be notified about (or killed by) other processes' errors.

2.1.2 PVM

PVM (Parallel Virtual Machine) is a popular system for doing concurrent programming in C or Fortran. It was designed with scientific computing in mind, so its features are the minimum to get the job done, with a minimum of overhead. The main characteristics of a PVM program are:

- A PVM process is essentially a separately compiled, standalone UNIX program which can communicate with other such processes. PVM's spawn operation is analogous to a UNIX fork/exec, rather than, say, looking like a function call.
- Communication consists of sending asynchronous messages to a process. Messages are buffered upon arrival.
- Messages must be marshalled and unmarshalled by the user program. PVM provides packing and unpacking routines only for primitive data types, contiguous homogeneous arrays of primitive types, and null-terminated strings.
- Messages are tagged with the sender's id and a user-specified code number. Message receipt may be conditioned on a particular sender and/or message code.
- The user may supply a predicate function for deciding which buffered message, if any, is to be received next. This enables one to examine message content, analogous to Erlang's pattern-directed receive.

- Processes may send signals, and may kill each other.
- PVM's concept of the Parallel Virtual Machine allows the user to specify which actual computers are to be available to run processes. There is a simple user interface program available; alternately, the PVM's configuration can be constructed entirely under program control.
- Processes may be spawned on a specific host, or one can let the system use a load-balancing heuristic to select the processor.

From the C programmer's point of view, the worst part of PVM is having to write code to marshall and unmarshall data structures. Of course, the severity of the problem depends critically on the data structures used, and hence the problem domain. PVM's routines can pack or unpack an entire homogeneous array of data in a single call; if you're only dealing with matrices, that corresponds to one call per data structure, which is about as good as it gets. At the opposite extreme, if you make use of heterogeneous structures, you might need a line of code for every slot of every structure.

2.1.3 Concert/C

Concert/C is a strict superset of ANSI C, with additional types and statements for distributed computing. It does not seem to have been intended strictly for scientific computing; rather the emphasis is on making the distribution of data and computation easy and safe. Like PVM, Concert/C processes are standalone programs, and communication is asynchronous. The differences are:

- Messages are sent to ports, not processes. A process may declare one of its ports to be 'initial', causing the port to be communicated to the parent process.
- All marshalling and unmarshalling is automatic. Annotations may be provided by the user to specify various details of the marshalling procedure.
- Ports are typed, and the type can be any data type. The user's program sends and receives data structures.
- There is a Remote Procedure Call facility, which allows the user to declare that a C function is to be called remotely. The function's name then corresponds to a port, but is still typed and used like a function pointer; however, unlike an actual function pointer, it can be passed to other processes. In simple cases, the RPC facility completely hides the details of communication. In more complex cases, like when function arguments are modified, some user annotations may be necessary to get the desired behavior.
- Selection and polling operations permit the reception of multiple message (or remote function call) types while maintaining type safety.

To a first approximation, Concert/C's automatic communication features remedy the main defect of PVM, but at the cost of greatly increasing the complexity of the language. Experience with the system will be needed to determine whether the tradeoff is worth it.

2.2 Compiler Correctness

The current programme of compiler derivation and correctness proof [Cli84, WO92, Oli93] has been successful in the realm of sequential languages. The general idea is as follows.

One starts with a denotational definition of a language's semantics, in which source language phrases are mapped to λ-calculus terms, representing functions. This definition is an interpreter, in that one can write a λ-calculus reducer which would execute a program's definition. One then defines a sequence of machines which are progressively less abstract, and writes a compiler for each machine, mapping source code into sequences of machine instructions. One then shows that a program's compiled code for each machine is in some appropriate sense equivalent to that for the machine's less-concrete predecessor. A more detailed sketch of this approach is given in section 3.1, with examples. In the case of the recent ν-lisp project [Oli93], the verification of a subset of Scheme started from the denotational definition and actually reached down to a RISC machine model, even dealing with issues like register allocation.

Until our work and that of [Wan94a], this verfication paradigm had not been extended into the realm of concurrency.

2.3 Erlang

Erlang [AVW93] has proven to be useful both for prototyping and for real-world implementation of concurrent systems. Section 2.1.1 discusses some features that make this so. This section criticizes some of the less felicitous aspects of the language.

Despite its success, Erlang is open to many criticisms; like C++ [Str93], its features were determined largely by the needs of a group of users who simply wanted a better way to get their work done. The following features of the language are particularly at variance with the state of the art in functional language design:

- Functions may only be defined at top level, so there can be no internal functions.
- Functions are not first class values. There are no anonymous functions, and there are no closures.
- There is no syntactic construct to delimit the scope of a variable; the scope runs from the point of introduction to the end of the enclosing function. Variables cannot be rebound.
- The behavior of an assignment statement depends on the enclosing binding context: the statement `(x,y)=(1,2)` may or may not create new bindings for `x` and `y`. The target variables `x` and `y` are interpreted as holes in the pattern to be filled in, as long

as they are unbound; if they already have values, the interpretation is that their values should be checked against the corresponding parts of the source data.

- In all pattern matching constructs, patterns may be augmented by predicate expressions, for example to check the sign of a numeric argument. However, the predicate expressions are not normal expressions, but expressions in a restricted sublanguage which, in particular, forbids calls to user-defined functions.

These perceived defects are easily remedied; the remedy for most of them consists of removing restrictions which would take considerable work to express and implement formally.

2.4 Formal Calculi

The majority of formal calculi and formalized programming systems ([BB90, BMT92, PGM90, Mil89, Mil92, Rep91]) differ from our model of concurrent computation in two ways: they specify synchronous communication, and they have an interleaving model of concurrency. In addition, not all of these systems allow the communication of arbitrary values, if they have values at all. While SDL [CCI87] specifies buffered communication and true concurrency, it uses a finite-state machine model of computation instead of the λ-calculus or an equivalently powerful system.

It is certainly true that most of the formal systems proposed for modelling concurrency are interdefinable, whether the communication be synchronous or asynchronous, whether the concurrency be interleaving or distributed, and whether there be communicable values or not. One must keep in mind that we are not engaged in an exercise in the theory of computation: we are trying to model concurrent computation as it actually occurs on collections of computers. For this reason, we insist on a using a model which accurately reflects our understanding of the true state of affairs, and that means a model of computation with true concurrency, asynchronous buffered communication, and value passing.

2.5 HOCC

HOCC [Wan94a] is an extension of the λ-calculus with interleaving concurrency and synchronous communication. This work introduced the notion of β-bisimulation, in which two programs (which are collections of λ-terms) are bisimilar exactly when their corresponding λ-terms are $\stackrel{\beta}{=}$, that is, equal up to α-renaming and β-conversion. This has the happy effect of reducing the problem of proving the operational equivalence of concurrent programs to the well-understood problem of proving that they are equal in the theory λ.

We find that β-bisimilarity has a natural counterpart in our calculus, given the correct notion of equality on its terms.

Chapter 3

Preliminaries

We present some technical background which will be needed for later developments. Section 3.1 gives an overview of the Wand-Clinger methodology of compiler derivation for sequential languages. Section 3.2 discusses the process calculus formalism.

3.1 Compiler Correctness

For purposes of discussion, we will consider a variant of the simple language for which a compiler was derived in [WO92]. The story presented here is intended to give the reader a taste of the Wand-Clinger methodology of compiler derivation. The sequential portion of the compiler development in Chapter 5 proceeds along these lines.

The source language L's grammar is

$$
\begin{array}{rcll}
E & ::= & I & \text{variable} \\
 & | & n & \text{integer constant} \\
 & | & (op_1\ E) & \text{unary primitive application} \\
 & | & (op_2\ E\ E) & \text{binary primitive application} \\
 & | & (rec\ I(I)\ E) & \text{recursive function definition} \\
 & | & (E\ E) & \text{application} \\
 & | & (if\ E\ E\ E) & \text{conditional}
\end{array}
$$

A language is given a semantics by a map $\mathcal{E}$, which maps source language phrases to terms in the λ-calculus, or more accurately, the λ-calculus extended with extra constants and reduction rules [Bar84, §15.3]. We refer to this calculus as IL, since we are using it as an intermediate language. The constants of L correspond to constants of IL, and the primitive operations of L correspond to the extra reduction rules of IL, which are often called δ rules. Of course the main job of a semantics is to define the control and data flow of L's "combinators", or language constructs which in some way combine subsidiary phrases.

Let us take the following names and definitions for some semantic domains:

Exp	Expressions of L
Id	Identifiers
Val	Values
$Env = Id \to Val$	Environments
Ans	Answers[1]
$Cont = Val \to Ans$	Continuations

Source language phrases are mapped to terms with a particular structure. They take an environment and a continuation, compute a value, and (usually) call the continuation on the value, thus producing an answer. Thus the type of our semantic function is

$$\mathcal{E} : Exp \to Env \to Cont \to Ans$$

For concreteness, let us consider some examples. The semantics of an identifier is that it is a reference to a program variable, which is supposed to be bound to some value. If this is the case, the binding is represented in the environment. The result of evaluating a variable is its bound value, so that is what is passed to the continuation:

$$\mathcal{E}[\![I]\!] = \lambda\rho\kappa.(\kappa\ (\rho\ [\![I]\!])) \tag{1}$$

L's primitive combinators provide richer examples. Corresponding to each integer n in L there is a constant $\ulcorner n \urcorner$ in IL. Corresponding to the each operator $\oplus$ in L there is a constant $\oplus_{\mathsf{cps}}$ in IL, whose δ rule is

$$(\oplus_{\mathsf{cps}}\ \ulcorner m \urcorner\ \ulcorner n \urcorner\ \kappa) \to (\kappa\ \ulcorner m \oplus n \urcorner)$$

The semantics of L's $\oplus$ combinator might then be

$$\mathcal{E}[\![e_1 \oplus e_2]\!] = \lambda\rho\kappa.\mathcal{E}[\![e_1]\!]\rho(\lambda v_1.\mathcal{E}[\![e_2]\!]\rho(\lambda v_2.(\oplus_{\mathsf{cps}}\ v_1\ v_2\ \kappa))) \tag{2}$$

To the extent that we could execute IL, say by writing an interpreter for it, the function $\mathcal{E}$ gives an interpreter for L. To move towards a compiler, we separate the information which is known at compile time from that which is not: the environment is split into a compile-time symbol table and a run-time display, and continuations are split into static 'instruction' lists and dynamic continuations. Furthermore, we introduce a run-time stack for storing

[1] Why do we need a separate domain of answers? A common design choice is for computations to 'return' answers which consist either of a success value or failure error message. In ML we might write this datatype of answers as

```
datatype Ans =
  success of Val
| failure of string;
```

intermediate results, instead of using variable bindings as in the foregoing example. Our new semantic domains are

$$\begin{array}{rl} Lex & \text{Lexical addresses} \\ CEnv = Id \rightarrow Lex & \text{Symbol tables} \\ REnv = Lex \rightarrow Value & \text{Run-time displays} \\ Stack = Value^* & \text{Run-time stacks} \\ Ins = REnv \rightarrow Stack \rightarrow Cont \rightarrow Ans & \text{Instructions} \end{array}$$

The job of the compiler $\mathcal{C}$ is to produce code for a given program phrase, given the symbol table and the code for the rest of the basic block in which the phrase occurs.[2]

The job of the compiled code should be to do the 'same thing' as the interpreted code; however, the compiled code and the interpreted code expect somewhat different arguments, because their execution contexts are different. The correctness condition is

$$(\mathcal{C}[\![e]\!]\Gamma\pi)u\zeta\kappa \stackrel{\beta}{=} \mathcal{E}[\![e]\!](u \circ \Gamma)(\lambda v.(\pi u(v :: \zeta)\kappa))$$

This says that the compiled code for a given phrase is equivalent to the interpreted code, if the interpreted code is run in the environment which is the composition of the run-time and compile-time environments, and with a continuation which pushes its argument onto the runtime stack and then runs the rest of the block.

This condition is used to solve for the definition of instructions. In the interest of brevity, we will present the results without explaining the derivation. A variable compiles into a **fetch** instruction:

$$\mathcal{C}[\![I]\!]\Gamma\pi = (\mathsf{fetch}\ \Gamma[\![I]\!]\pi)$$

if we define **fetch** appropriately so that

$$(\mathsf{fetch}\ l\pi)u\zeta\kappa \stackrel{\beta}{=} (\pi u((u\ l) :: \zeta)\kappa)$$

we can show with a little work that the correctness condition is satisfied.

A binary primitive application compiles into the code for the arguments followed by an instruction for the primitive operation:

$$\mathcal{C}[\![e_1 \oplus e_2]\!]\Gamma\pi = \mathcal{C}[\![e_1]\!]\Gamma(\mathcal{C}[\![e_2]\!]\Gamma(\oplus_{\mathsf{inst}}\ \pi))$$

To make this work, we define

$$(\oplus_{\mathsf{inst}}\ \pi)u(v_2 :: v_1 :: \zeta)\kappa \stackrel{\beta}{=} \oplus_{\mathsf{cps}}\ v_1 v_2(\lambda v.(\pi u(v :: \zeta)\kappa))$$

and again the correctness condition is satisfied; this time, however, we need to use the condition as an induction hypothesis, that is to say we must assume that the compilation of the subexpressions is correct.

[2] We said 'code', but wrote Ins. For convenience, we assume that the instructions are linked, so that an instruction is the head of a block of code.

The fetch and $\oplus_{\mathsf{inst}}$ instructions have an important property: their definitions are of the form $\pi u \zeta \kappa = \pi' u' \zeta' \kappa'$, so they may be thought of as the instructions of a 4-register abstract machine. The next step in the compiler derivation process [WO92, Oli93] is to write down the definition of the instruction set and transitions of this abstract machine, and prove that it preserves the language's operational semantics. From there, one can work out a flat storage representation for the abstract machine's data and instructions, and (given a reasonable set of storage machine instructions) derive storage machine instructions which preserve the reduction sequence of the abstract machine.

In the interest of brevity we omit the description of the 4-register machine's data domains and its transition relation; the interested reader is referred to [Oli93] for the details. The development of Chapter 5 is actually somewhat different from the story presented here, and will therefore be presented in complete detail in that chapter.

3.2 Process Calculi in General

One of the most fundamental mathematical structures in computer science is the transition system: an abstract set of states, and a relation describing the transitions between them. If we interpret the states as the nodes and the transition relation as a the edges of a graph, we have something akin to a picture of an automaton. If we wanted to draw a picture of, say, a finite-state recognizer, we would want to label the arcs with tokens. This construct is captured in the following definition:

Definition 1 (LTS) A Labelled Transition System is a tuple $\langle S, L, \rightarrow \rangle$ consisting of a set S of states, a set L of labels, and a transition relation $\rightarrow \subseteq S \times L \times S$. Instead of $(s_1, l, s_2) \in \rightarrow$ we usually write $s_1 \xrightarrow{l} s_2$.

Note that we can have multiple transitions from one state to another, as long as they are labelled differently. This is different from having a two-place transition relation together with a labelling on transitions.

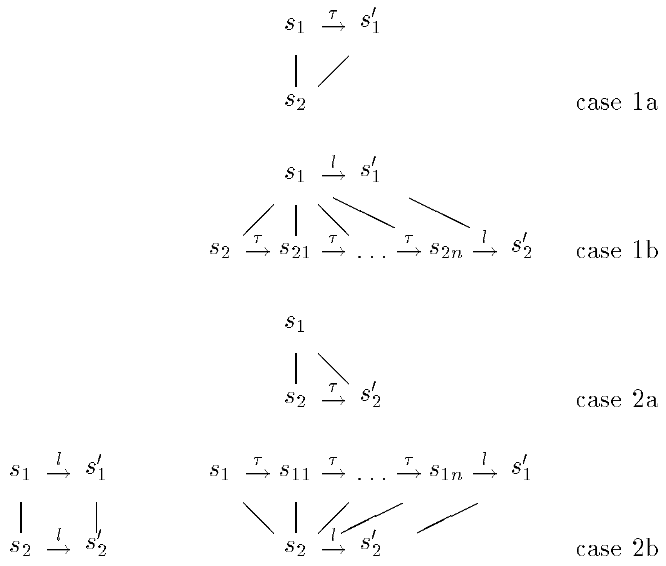
While this definition might seem biased towards representing automata, in fact it also includes term rewriting systems and other formal calculi. The λ-calculus provides an example of an LTS:

$$\langle \Lambda, \{\beta\}, \{(M, \beta, N) \mid M \rightarrow_\beta N\} \rangle$$

Here the states are terms, and transitions represent β-reductions.

Henceforward we shall switch at will between referring to LTSs as machines, calculi, or what have you.

When should two LTSs be considered equivalent? An obvious answer is that they should be isomorphic when considered as edge-labelled graphs; it is equally obvious that this is too strong—adding an unreachable state to a finite-state machine precludes isomorphism, but has no impact on the operation of the machine. Instead, we ask only that each machine be able to do what the other can do. Action resides in the transition relation of each machine, and we consider actions (transitions) to correspond if their labels are the same.



(a) Bisimulation (b) Branching bisimulation

Figure 1: Bisimulation and branching bisimulation. Solid lines indicate related states. The case numbers refer to Definition 3.

The machines are equivalent if, having said how the states of one correspond to the states of other, corresponding states have corresponding transitions into corresponding states. This is formalized in the definition of bisimulation [Par81], which is also known as strong bisimulation:

Definition 2 (Bisimulation) Let $M_1 = \langle S_1, L, \rightarrow_1 \rangle$ and $M_2 = \langle S_2, L, \rightarrow_2 \rangle$ be given LTSs. We say that a relation $R \subseteq S_1 \times S_2$ is a bisimulation between M_1 and M_2 if, whenever $s_1 R\, s_2$,

1. If $s_1 \xrightarrow{l} s_1'$, then there is some s_2' with $s_2 \xrightarrow{l} s_2'$ and $s_1' R\, s_2'$.
2. If $s_2 \xrightarrow{l} s_2'$, then there is some s_1' with $s_1 \xrightarrow{l} s_1'$ and $s_1' R\, s_2'$.

LTSs are said to be bisimilar if there is a nonempty bisimulation between them.

Bisimulation is depicted in Figure 1(a).

Consider the computation of a machine M, meaning a sequence of state transitions $s_1 \xrightarrow{l_1} s_2 \xrightarrow{l_2} \ldots$ and suppose there is a machine M', with R a bisimulation relation between the two. Assuming that M' has a state which is related to s_1, say s_1', then M' has a computation $s_1' \xrightarrow{l_1} s_2' \xrightarrow{l_2} \ldots$ where each $s_i R\, s_i'$; however, in general it will have other computations as well. Thus we should not think of the machines as running in lockstep, but rather that for each computation of M, there is some computation of M' which follows it in lockstep.

The practical need for equivalences between LTSs is this: one often finds it convenient to express the specification of a system as an LTS, and then wants to implement it as a different LTS and prove that the implemented system is equivalent to the specification. Bisimulation works nicely here, guaranteeing two important properties:

1. any computation of the implementation corresponds to some computation of the specification, i.e., the implementation is *faithful* to the specification

2. any computation allowed by the the specification corresponds to some computation of the implementation, i.e., the implementation is *complete*

One should avoid the temptation to think of faithfulness and completeness of implementations as things that would be nice to have, but can't really be expected in real life. We will have more to say about this at the end of this section.

In real life, systems will often be implemented by a process of transition refinement, so the implementation machine might go through several transitions to accomplish what the specification could do in a single transition. To account for this, we need to introduce the idea of an internal, or unobservable transition. Such transitions will have a special label, typically τ. Then to mimic an observable move of one machine, another machine will be able to make an arbitrarily long sequence of internal moves, followed at last by a corresponding observable move. This situation is captured in the notion of branching bisimulation [Vog92]:

Definition 3 (Branching Bisimulation) Let $M_1 = \langle S_1, L, \rightarrow_1 \rangle$ and $M_2 = \langle S_2, L, \rightarrow_2 \rangle$ be given LTSs, and let τ be a distinguished element of L, called the silent action. We say that a relation $R \subseteq S_1 \times S_2$ is a branching bisimulation between M_1 and M_2 if, whenever $s_1 R\, s_2$,

1. If $s_1 \xrightarrow{l} s_1'$, then either
   (a) $l = \tau$ and $s_1' R\, s_2$, or
   (b) there is some sequence $s_2 \xrightarrow{\tau} s_{21} \xrightarrow{\tau} \ldots s_{2n} \xrightarrow{l} s_2'$ with $s_1' R\, s_2'$, and for each each $1 \leq i \leq n$, $s_1 R\, s_{2i}$.

2. If $s_2 \xrightarrow{l} s_2'$, then either
   (a) $l = \tau$ and $s_1 R\, s_2'$, or
   (b) there is some sequence $s_1 \xrightarrow{\tau} s_{11} \xrightarrow{\tau} \ldots s_{1n} \xrightarrow{l} s_1'$ with $s_1' R\, s_2'$, and for each each $1 \leq i \leq n$, $s_{1i} R\, s_2$.

The definition of branching bisimulation is depicted in Figure 1(b). Case 1b of that figure shows an important detail which is easily overlooked. For purposes of discussion, call M_1 the specification and M_2 the implementation. Until the implementation squeezes off the observable transition, its internal states s_{2i} must be related to s_1. In particular, the

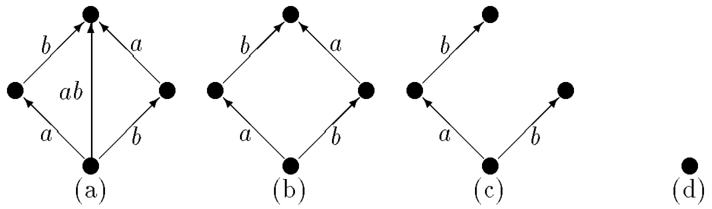


Figure 2: A specification and three simulations

internal transitions may not foreclose the possibility that the transition eventually taken by the implementation might correspond to any of the other transitions possible from s_1. Another way of looking at it is that the states $s_2 \ldots s_{2n}$ might as well be considered to be one big state, at least as far as the branching structure of the process is concerned. Thus it seems that branching bisimulation is more appropriate for describing state refinement than transition refinement. We will have more to say about this in Section 4.2.

Earlier we mentioned the temptation to implement systems which were faithful to their specifications, but not complete; this might be the case, say, when writing a uniprocessor simulator for a concurrent system. Consider the specification of Figure 2(a), which specifies that actions a and b can occur in either order, or simultaneously, all three alternatives leading to the same state. Were we to write a simulator for the system, we might very well implement it in such a way that it did not simulate every possible behavior of the specification, although all of the simulator's runs were 'justified' by a possible run of the specification; this is depicted in Figure 2(b). This kind of relationship between a specification and a simulator is formalized as follows:

Definition 4 (Simulation) Let $M_1 = \langle S_1, L, \rightarrow_1 \rangle$ and $M_2 = \langle S_2, L, \rightarrow_2 \rangle$ be given LTSs, and let $R \subseteq S_1 \times S_2$ be a given relation. We say that M_2 simulates M_1 via R if, whenever $s_1 R\, s_2$ and $s_2 \xrightarrow{l} s_2'$, there is some s_1' with $s_1 \xrightarrow{l} s_1'$ and $s_1' R\, s_2'$.

The relation between simulation and bisimulation should then be clear:

Theorem 1 If M_2 simulates M_1 via R and M_1 simulates M_2 via R^{-1}, then R is a bisimulation between M_1 and M_2.

The reason for the popularity of bisimulation as a notion of equivalence, rather than simulation, is that it guarantees that related states are equipotent, thus ruling out the cases depicted in Figure 2(c,d). The simulation of Figure 2(b) is also ruled out; whether it is useful or not depends on the application.

One possible scenario is that we write a program in a concurrent language, and the concurrency leads to nondeterminism. We write our programs in such a way that any any maximal computation produces the correct answer. While we would prefer execution sequences which exhibit lots of concurrent computation, we will not consider 'thin' executions to be incorrect. We will, however, insist on maximal executions: a computation does us no good if it doesn't reach an answer. We capture this notion in the following definition:

Definition 5 (Deadlock-preserving Simulation) Suppose that M_2 simulates M_1 via R. We say that R is deadlock-preserving if whenever $s_1 R\, s_2$ and $s_2 \not\rightarrow$, then $s_1 \not\rightarrow$.

Thus the simulation of Figure 2(b) is deadlock-preserving, but those of Figure 2(c,d) are not. Note that the name "deadlock-preserving" is correct despite the one-sidedness of the definition: that the implementation does not have a transition, when the reference system does not, is guaranteed by the definition of simulation.

In Chapter 6, we will justify the implementation of a concurrent programming language by showing that it simulates its semantic specification via a deadlock-preserving simulation.

3.3 Summary

We have given a brief overview of the Wand-Clinger methodology of compiler derivation, and presented the fundamentals of process calculi. We next present our own process calculus in Chapter 4, and use it to derive a compiler in Chapter 5.

Chapter 4

CLAM: Concurrent λ-Calculus

> Abstractions are indispensable in mathematics; still, they should not be pursued for their own sake and carry one to a point whence there is no descent to earth. We should always think of these transitions from abstract thought to practice as a necessary phase of man's apprehending of objective reality. In cases where the possibility of such a transition turns out to be very doubtful, we have to revise the abstractions employed and seek to alter them.
>
> —*A. A. Markov*

Fundamental to the present work is the observation that the execution of systems which exhibit true concurrency can be described by linear structures. This idea is introduced via a real-world analogy in Section 4.1. Section 4.2 gives an example which motivates the technique of splitting durational events into starting and ending transitions, and shows how this solves several problems raised in Section 3.2. Section 4.3 mentions some of the history of the notion of event splitting. Section 4.4 gives an intuitive picture of the operation of CLAM, a concurrent extension of the λ-calculus based on these ideas, and Section 4.5 presents the technical details.

4.1 The Television Analogy

Imagine watching several televisions simultaneously and writing a log of when programs start and end. You might get something like this:

Time	Channel	Event	Abbreviation
6:00	64	Deep Space Nine starts	$DS9^+$
6:00	HBO	Ghostbusters starts	G^+
7:00	64	Deep Space Nine ends	$DS9^-$
7:01	64	Star Trek: The Next Generation starts	$STNG^+$
8:00	64	Star Trek: The Next Generation ends	$STNG^-$
8:00	HBO	Ghostbusters ends	G^-

Taken together, the channels of interest are a truly concurrent system, being independent of each other. The log, although linear in nature, is an accurate representation of the activity of that system. For any instant of time covered by the log, we can recover the state of all the channels. The log imposes a total order on events which is stricter than the real-world precedence relation: if A precedes B in real time, then A precedes B in the log. However, the log, being linear, must put some order on simultaneous events.

Now consider the logical precedence relation on these events, which we write $\prec$, representing our understanding of which events must logically precede which others. Our understanding of television programming is that we have two independent chains of events, one per channel:

$$\begin{gathered}\mathrm{DS9}^{+} \prec \mathrm{DS9}^{-} \prec \mathrm{STNG}^{+} \prec \mathrm{STNG}^{-} \\ \mathrm{G}^{+} \prec \mathrm{G}^{-}\end{gathered}$$

This ordering is certainly consistent with our original observation, but also with some potential ordering observations $<$ which seem unreasonable, for instance

$$\mathrm{DS9}^{+} < \mathrm{G}^{+} < \mathrm{G}^{-} < \mathrm{DS9}^{-} < \mathrm{STNG}^{+} < \mathrm{STNG}^{-}$$

Given what we know about the lengths of the programs, it seems like this shouldn't be possible. How could a 90 minute (without commercials) movie start and finish during an hour long show? The answer lies in the facts that the running times are minimum durations, and that programs can be interrupted and resumed. DS9 might be interrupted for well over an hour, say for the announcement of a war, and then resumed where it left off.

Concurrent computation on a collection of workstations can be viewed similarly: the channels are processors, and the television programs are computer programs. Our knowledge about running times must be taken as a minimum, since programs run in multiprogramming environments in which no rate of progress is guaranteed. Following the television analogy, our concurrency semantics will model causal relationships and ignore real time, and the evolution of concurrent computations will be expressed in terms of an interleaving of starting and ending events.

4.2 Event Splitting and Refinement

Figure 3 shows a specification and two possible implementations. The specification intends to allow one of three actions nondeterministically.

The first implementation rolls a die to decide which action do to. (Rolling and reading the die is a pair of operations which we take to be private to the implementation, and therefore not observable.) When the die has been rolled and read, the choice has been made, but the action has not been executed; thus the three internal states of the implementation do not correspond to any state of the specification. This situation could, however, be described by an ST bisimulation [Vog92], where both the states and transitions of systems are put in correspondence. In this case, the respective initial and final states of the specification and implementation correspond, and along each path through the implementation, the internal state and the two transitions correspond to a transition in the specification.

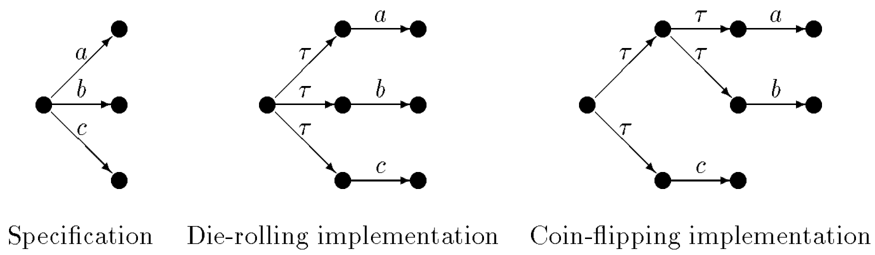


Figure 3: A specification and some problematic implementations

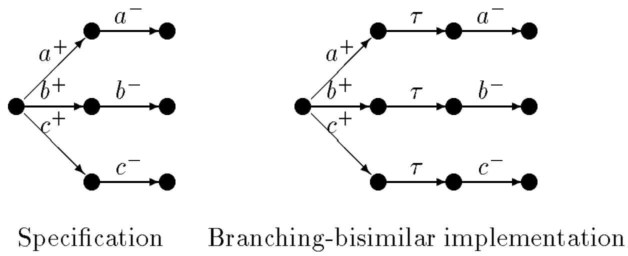


Figure 4: The split-transition version of the specification from Figure 3, with a branching-bisimilar implementation

The second implementation has only a two-sided coin instead of a die, so it might have to flip the coin twice to decide amongst three alternatives. The uppermost initial transition in the diagram depicts a very problematic internal state: choice c has been eliminated, but neither choice a or b has been decided upon. Clearly, to identify this implementation with its specification, an equivalence would have to abstract completely away from internal actions; even ST bisimulation will not do.

Suppose now that we follow the analogy of the previous section, and split each transition into a start transition, an internal durational state, and an end transition. The specification in Figure 4 shows the result, which is of course the same shape as the first implementation of Figure 3. Thus the die-rolling implementation can be made strongly bisimilar to the new specification, if we change each τ transition to the appropriate $^+$ transition, and each non-τ transition to the appropriate $^-$ transition. Furthermore, the implementation of 4 is easily seen to be in a branching bisimulation with the specification. In Section 3.2, we remarked that branching bisimulation seemed problematic when faced with the refinement of transitions, but was well suited to dealing with the refinement of states. Representing durational events as a start event, a durational state, and an end event means that there is always an internal state to be refined, so branching bisimulation is viable.

4.3 Some History

The idea of splitting durational events is not new. The following dictum comes from [Hoa85, page 24]:

> The actual occurrence of each event in the life of an object should be regarded as an instantaneous or an atomic action without duration. Extended or time-consuming actions should be represented by a pair of events, the first denoting its start and the second denoting its finish.

It is by no means universally accepted that splitting durational events is a cure-all. As it happens, the above quotation was cited in a paper [vGV91] which established that

> durational and structural aspects of actions can in general not be modeled in interleaving semantics, even when time-consuming actions are represented by pairs of instantaneous actions denoting their start and finish.

This is not an artifact of any particular model; rather the authors claim "that our counterexamples can be recreated on the language level in all major process algebras." Rather than belabor the apparent contradiction between the impossibility claim of the paper and the observed success of splitting in all known practical situations, we divulge the punch line of the work cited:

> All counterexamples necessarily involve *autoconcurrency*, the possibility that two instances of the same action occur simultaneously. If there is no autoconcurrency, then there can be no confusion about how finish actions match up with start actions, and splitting becomes a consistent way of dealing with durational actions.

As we shall see in the following sections, our model of concurrent computation does not exhibit autoconcurrency.

4.4 The Conceptual Model

Our desire is to formulate a calculus which represents multiple processes executing in a truly concurrent fashion, and to describe the evolution of terms of the calculus by a sequence of starting and ending events. Before delving into the mathematical details of our calculus we shall give a top-down overview of our conceptual model.

A concurrent program consists of a collection of processes, each with a unique process identifier (PID). Many processes may execute at the same time. A process is a computer program together with an input buffer, or mailbox, to which messages may be delivered asynchronously, that is, without action on the part of the receiving program. At any instant of time, a process may be in an idle state, or in an active state. If we focus only on observable events, then an idle process never does anything, and all observable activity comes from active processes. Active states are always entered from idle states, and active states always lead to idle states. One can think of the idle-to-active transitions as representing "starting to do something", the active states as "doing something", and the active-to-idle transitions as representing "being finished doing something". Intuitively the transitions don't take time, but time is spent in the active states. When we say that this model exhibits true concurrency, we mean that many processes may simultaneously be in active states.

The evolution of the state of a concurrent program over time is described as a linear sequence of transitions of individual processes. Since each transition represents an activity starting or ending, and the transitions do not take time, such a linear description can describe a program run which exhibits any degree of true concurrency.

4.5 CLAM

CLAM is a Concurrent LAMbda calculus. The program part of each idle process is a λ-term, and the buffer part is a sequence of λ-terms. If an idle process embarks upon an activity, the ensuing active process is represented as an activity marker together with the idle process state. At such time as the activity ends, the process will again be in an idle state.

CLAM is based on the λ-calculus with constants. Starting with an infinite set PID of process identifiers, we define a set of constants which augment basic λ-terms. Figure 5 shows the syntax of our terms and defines the rest of the syntactic domains which make up our calculus.

The transition relation on programs requires some additional domains, since we want the transitions to be labelled. As shown in Figure 6, the program transition relation has labels which consist of a PID and an event descriptor, which describes the communication start or end event corresponding to the transition.

The rules for $\rightarrow$ are given in Figure 7, in which the starting and ending transitions are grouped for clarity. We have written $X \stackrel{p:E}{\longrightarrow} Y$ instead of $(X,(p,E),Y) \in\rightarrow$. Active states are entered by a $^+$ transition and left by a $^-$ transition. Because the $^+$ transitions are conceptualized as merely beginning an activity, these transitions are kept as simple as possible. We now consider the details of each activity.

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_{empty} operation examines the input queue, returning $\ulcorner 1 \urcorner$ for empty and $\ulcorner 0 \urcorner$ otherwise. Since this operation is conceptualized as taking finite time, what happens if a message is sent during the examination? The $\langle empty\rangle^+$ transition takes a snapshot of the state of the buffer, storing it in the activity marker. This snapshot is returned by the $\langle empty\rangle^-$ transition, rather than the actual state of the buffer. Thus this test is conservative: a result of $\ulcorner 1 \urcorner$ may or may not indicate an empty buffer, but a result of $\ulcorner 0 \urcorner$ always indicates a nonempty buffer.

$$
\begin{array}{rcll}
p & \in & \text{PID} & \text{Process ids} \\
\text{Term} & ::= & c_p & \text{A constant for each } p \in \text{PID} \\
 & | & c_{spawn} & \text{Communication operators} \\
 & | & c_{send} & \\
 & | & c_{receive} & \\
 & | & c_{empty} & \\
 & | & c_{spawning} & \text{Activity marker constants} \\
 & | & c_{sending} & \\
 & | & c_{receiving} & \\
 & | & c_{emptying} & \\
 & | & v & \text{Variable} \\
 & | & \lambda v.\text{Term} & \text{Abstraction} \\
 & | & (\text{Term Term}) & \text{Application} \\
M & \in & \text{Term} & \\
\text{Value} & \subset & \text{Term} & \text{Closed terms} \\
V & \in & \text{Value} & \\
\text{Buffer} & = & \text{Value}^* & \\
B & \in & \text{Buffer} & \\
\text{Idle state} & = & \text{Term} \times \text{Buffer} & \\
M, B & \in & \text{Idle state} & \\
\text{Marker} & \subset & \text{Term} & \\
\text{Active state} & = & \text{Marker} \times \text{Idle state} & \\
X & \in & \text{Active state} & \\
\text{Process} & = & \text{Idle state} \cup \text{Active state} & \\
\text{Program} & = & \text{PID} \stackrel{\text{fin}}{\rightarrow} \text{Process} & \\
\Pi & \in & \text{Program} &
\end{array}
$$

Figure 5: CLAM syntactic domains

$$
\begin{array}{rcl}
\text{Ev} & ::= & \langle spawn\ p \rangle^{\pm} \\
 & | & \langle send\ p \rangle^{\pm} \\
 & | & \langle receive \rangle^{\pm} \\
 & | & \langle empty \rangle^{\pm} \\
\pm & ::= & + \\
 & | & - \\
\text{Transition relation} & \subseteq & \text{Program} \times (\text{PID} \times \text{Ev}) \times \text{Program} \\
\rightarrow & \in & \text{Transition relation}
\end{array}
$$

Figure 6: Transition domains

$$\left[\begin{array}{c}
\dfrac{M \overset{\beta}{=} (c_{receive}\ N)}{\Pi[p : M, B] \xrightarrow{p:\langle receive\rangle^{+}} \Pi[p : \langle\langle c_{receiving}\rangle, (c_{receive}\ N), B\rangle]} \\[3ex]
\dfrac{X = \langle\langle c_{receiving}\rangle, (c_{receive}\ N), (V{::}B')\rangle}{\Pi[p : X] \xrightarrow{p:\langle receive\rangle^{-}} \Pi[p : (N\ V), B']}
\end{array}\right]$$

$$\left[\begin{array}{c}
\dfrac{M \overset{\beta}{=} (c_{empty}\ N) \quad V = \begin{cases} \ulcorner 1 \urcorner & B = [\,] \\ \ulcorner 0 \urcorner & B \neq [\,] \end{cases}}{\Pi[p : M, B] \xrightarrow{p:\langle empty\rangle^{+}} \Pi[p : \langle\langle c_{emptying}\ V\rangle, (c_{empty}\ N), B\rangle]} \\[3ex]
\dfrac{X = \langle\langle c_{emptying}\ V\rangle, (c_{empty}\ N), B\rangle}{\Pi[p : X] \xrightarrow{p:\langle empty\rangle^{-}} \Pi[p : (N\ V), B]}
\end{array}\right]$$

$$\left[\begin{array}{c}
\dfrac{M \overset{\beta}{=} (c_{send}\ c_{p'}\ V\ N) \quad V \text{ a closed value}}{\Pi[p : M, B] \xrightarrow{p:\langle send\ p'\rangle^{+}} \Pi[p : \langle\langle c_{sending}\rangle, (c_{send}\ c_{p'}\ V\ N), B\rangle]} \\[3ex]
\dfrac{X = \langle\langle c_{sending}\rangle, (c_{send}\ c_{p'}\ V\ N), B\rangle \quad Y = M', B'}{\Pi[p : X][p' : Y] \xrightarrow{p:\langle send\ p'\rangle^{-}} \Pi[p : N, B][p' : M', (B'@[V])]} \\[3ex]
\dfrac{X = \langle\langle c_{sending}\rangle, (c_{send}\ c_{p'}\ V\ N), B\rangle \quad Y = \langle N', M', B'\rangle}{\Pi[p : X][p' : Y] \xrightarrow{p:\langle send\ p'\rangle^{-}} \Pi[p : N, B][p' : \langle N', M', (B'@[V])\rangle]}
\end{array}\right]$$

$$\left[\begin{array}{c}
\dfrac{M \overset{\beta}{=} (c_{spawn}\ N\ P) \quad p' \notin \text{procs } \Pi}{\Pi[p : M, B] \xrightarrow{p:\langle spawn\ p'\rangle^{+}} \Pi[p : \langle\langle c_{spawning}\ c_{p'}\rangle, (c_{spawn}\ N\ P), B\rangle]} \\[3ex]
\dfrac{X = \langle\langle c_{spawning}\ c_{p'}\rangle, (c_{spawn}\ N\ P), B\rangle}{\Pi[p : X] \xrightarrow{p:\langle spawn\ p'\rangle^{-}} \Pi[p : (P\ c_{p'}), B][p' : (N\ c_{p'}), [\,]]}
\end{array}\right]$$

where

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

Figure 7: Communication rules

The c_{send} operation sends a value to a process, or more accurately, puts it into the buffer of a process. Similar to the $c_{receive}$ operation, the $\langle send\rangle^+$ transition is always enabled. The $\langle send\rangle^-$ transition is only enabled when the receiving process actually exists. There are two $^-$ transitions, to handle the receiving process being in either an active or idle state.

The c_{spawn} operation forks another process. The $\langle spawn\rangle^+$ 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. Note that a process id is considered to be in use once it is assigned to a fork activity. The $\langle spawn\rangle^-$ activity actually establishes the new process.

4.6 Equivalence

The fundamental notion of equivalence for λ-calculus based concurrent calculi is due to Wand [Wan94b]:

Definition 6 (β-bisimilarity) Write $\Pi \overset{\beta}{\simeq} \Pi'$ if Π and Π' are structurally identical up to $\overset{\beta}{=}$, which is to say that $\mathrm{dom}\ \Pi = \mathrm{dom}\ \Pi'$, and at each p, $\Pi(p) \overset{\beta}{=} \Pi'(p)$.

In the work cited, processes are simply λ-terms, while in CLAM they are slightly more complex structures. We can co-opt the definition for $\overset{\beta}{=}$ as it stands, if we institute the convention of "piecewise equivalence":

Convention 1 We shall feel free to use $\overset{\beta}{=}$ on simple structures built up from λ-terms, for example pairs, n-tuples, and sequences.

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

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

Proof: Let $\Pi \overset{\beta}{\simeq} \Pi'$ be given, and consider each rule for the transitions other than $\langle spawn\rangle^+$. The programs' corresponding processes are $\overset{\beta}{=}$, so by the transitivity of $\overset{\beta}{=}$, any rule which applies to Π applies to Π'. Inspection shows that the respective results of applying the rule to the two programs are still $\overset{\beta}{=}$.

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

4.7 Summary

We have presented a model which expresses true concurrency by means of iterleaved instantaneous transitions, and embodied the model in the value-passing concurrent calculus CLAM. We have given a definition of equivalence for CLAM terms, which we shall use in proving the correctness of a compiler in Chapter 5.

Chapter 5

Compiler Correctness

Let $X = X$.
—*Laurie Anderson*

5.1 Overview

The outline of the Wand-Clinger compiler derivation methodology given in Section 3.1, along with various recent works [WO92, Oli93], suggests a general recipe for designing an environment-continuation semantics for a functional language. One starts with standard language constructs, namely variables, abstractions, and applications. Next one decides on some data domains, posits a way of encoding the data values in the λ-calculus (or, alternatively, extends the calculus with appropriate constants), and posits primitive function-like operators to implement primitive operations on the data. The semantic definitions of variable reference, abstraction, and application are standard. The semantics of a constant looks like that of a variable, except that there is no environment lookup. The semantics of a primitive operation invocation looks like a function call, except that there is no function expression to be evaluated, and the actual function application is replaced by a δ-rule. Once the semantics is complete, the process of designing a high-level abstract machine to implement it is again standard.[1]

Wand [Wan94b] has showed that the recipe keeps working even when the primitives are communication operators; his notion of β-bisimilarity is engineered specifically to ensure that this is so. The meaning of the communication primitives is given by the operational semantics for the extended calculus; the job of the compiler is essentially administrative. The real work comes in the design of an abstract machine, or at least, it does if the machine

[1]This process is sufficiently stylized that one can easily imagine casting it into a categorical sort of framework: one starts with a structure representing the data algebra, and a functor maps it into a call-by-value functional language in which the constants and functions of the algebra are represented. On a more concrete level, one could implement this in ML, and have the functor produce a bytecode compiler and runtime system.

is to be plausibly implementable. Unlike the extensions induced by the typical function-like primitive operators, there is no limit to the requirements which might arise from the communication rules. The best advice to the language designer might be to keep eventual implementation in mind when designing the communication portion of a concurrent language.

The communication primitives in [Wan94b] have an interleaving semantics, while we are interested in true concurrency. We have already seen, however, that there is no problem in describing the operation of a truly concurrent system in an interleaving framework. Therefore much of the compiler derivation of that work will apply to our language; the two part company at the point where one has to consider the operational semantics of the communication operators.

This chapter is intended to be self-contained within the context of the thesis. Therefore we present all of the details of the compiler derivation, even though, for the reasons cited, there is a lot of overlap between the present work and [Wan94b].[2]

In this chapter, all of the figures except Figure 17 share content with that work. Places in which there are material differences are marked with the symbol $\star$; as promised, there will be few. An exception to the marking convention is the material involving recursive procedures and conditionals, the treatment of which comes from [WO92].

Section 5.2 presents the syntax and semantics of our language Urlang. Section 5.3 presents the compiler derivation. In Section 5.4 the abstract machine is presented and proved correct.

We follow common practice in suppressing the details associated with implementing primitive operations We show how to "wire up" subtraction; wiring up other operations is straightforward, as shown in Chapter 6.

Since an overview of compiler derivation was already given in Section 3.1, we shall adopt a fairly brisk pace when presenting the details of the sequential portion of our language. The interested reader is referred to [WO92] and especially [Oli93] for a more detailed exposition.

5.2 Urlang: The Source Language

The syntax of our source language is shown in Figure 8, in which i ranges over identifiers and n over integers.

5.2.1 Syntax

We have the usual pieces of a functional language, namely constants, variables, primitives, abstractions, and applications, as well as four new operators (**put**, **get**, **empty**, and **spawn**) corresponding to CLAM primitives.

[2] In fact, our development is based on an early draft of [Wan94b]. In the later versions of that work, what we call the compiler ($\mathcal{CE}$ in Figure 12) and the translator ($\mathcal{TE}$ in Figure 18) have been merged. The difference between the two versions is mainly one of presentation.

	p	$::=$	a	Program
	a	$::=$	(**rec** $i(i)$ e)	Abstraction
	e	$::=$	i	Expressions
		$\mid$	n	
		$\mid$	a	
		$\mid$	(**if** e e e)	
		$\mid$	(e e)	
		$\mid$	($-$ e e)	
		$\mid$	(**put** e e)	
		$\mid$	(**get**)	
$\star$		$\mid$	(**empty**)	
		$\mid$	(**spawn** e)	

Figure 8: Source language syntax

5.2.2 Examples

We give some simple examples of Urlang programs, simply to show how the communication features of the language work.

The simplest possible program is as follows:

```
(rec top-level (root-pid)
     42)
```

Recall that a complete Urlang program is not an expression, but rather an abstraction. When spawned, `top-level` is called on its process id, after which it can return 42.

Next we have a program which spawns another process. Here the child process receives a message (which should be an integer), adds 42 to it, and sends the result back to the parent. Note that this program uses the macros `let` and `progn`, which are defined in Appendix A.3.

```
(rec top-level (root-pid)
     (let ((server-function (rec server-function (child-pid)
                                (let ((argument (get))
                                      (result (+ argument 42)))
                                  (put root-pid result))))
           (server (spawn server-function)))
       (progn
         (put server 17)
         (result_is (get)))))
```

The forms of the main program and the spawned function are the same, namely a function which expects its process id as its argument. The `server-function` is defined in a lexical environment containing `root-pid`, so that value need not be sent to it.

Finally, to illustrate the `empty` construct, we exhibit a function which constructs a list of all of the currently available messages:

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

Figure 9: Source language semantics

```
(rec loop (ignore)
     (if (empty)
         nil
         (cons (get) (loop nil)))))
```

A larger example is given in Appendix C.

5.2.3 Semantics

The semantics of the language is given in Figure 9. Expressions are translated compositionally into CLAM terms, while programs are translated into CLAM programs. For the sequential portion of the language, the translation is a typical environment-continuation semantics. The language uses four value domains (integers, PIDs, closures, and booleans) which are encoded as higher-order abstract syntax; in addition, we have a one-point domain *unit* for representing unspecified values. The definitions of the injection and projection terms and the other auxiliaries are given in Figure 10, and will be discussed subsequently. For now it suffices to know that the in_j terms box up values and the out_j terms unbox values, and that a run-time type error causes the offending process to deadlock.

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.

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 process id to send to is unboxed from the first argument,

$$
\begin{aligned}
in_{int} &= \lambda x.\lambda fghi.f\,x \\
in_{pid} &= \lambda x.\lambda fghi.g\,x \\
in_{clo} &= \lambda x.\lambda fghi.h\,x \\
in_{bool} &= \lambda x.\lambda fghi.i\,x \\
unit &= \lambda fghi.\mathbf{0} \\
\mathbf{0} &= \lambda v.v \\
\dagger &= \lambda v.\mathbf{0} \\
out_{int} &= \lambda v\kappa.v\,\kappa\,\dagger\,\dagger\,\dagger \\
out_{pid} &= \lambda v\kappa.v\,\dagger\,\kappa\,\dagger\,\dagger \\
out_{clo} &= \lambda v\kappa.v\,\dagger\,\dagger\,\kappa\,\dagger \\
out_{bool} &= \lambda v\kappa.v\,\dagger\,\dagger\,\dagger\,\kappa \\
applicate &= \lambda va\kappa.out_{clo}\,v\,(\lambda f.f\ a\ \kappa) \\
branch &= \lambda b\,e_t e_f\,\rho\kappa.out_{bool}\,b\,(\lambda v.((primitive\_if\ v\ e_t\ e_f)\,\rho\,\kappa)) \\
subtract &= \lambda v_1 v_2\kappa.out_{int}\,v_1(\lambda n_1.out_{int}\,v_2\,(\lambda n_2.\kappa(in_{int}\,(primitive\_sub\,n_1\,n_2)))) \\
send &= \lambda v_1 v_2\kappa.out_{pid}\,v_1\,(\lambda p.c_{send}\,p\,v_2\,(\kappa\ unit)) \\
boot &= \lambda fp.applicate\ f\,(in_{pid}\,p)\,\dagger
\end{aligned}
$$

Figure 10: Auxiliaries

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. (Of course the expression $(\kappa unit)$ might reduce prior to to the communication, but it cannot communicate because it is not in head position.) **put** expressions return an unspecified value, which of course should not be used.

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 value generated by the c_{empty} 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. (Note that if the argument of the **spawn** expression is not a closure, the spawning process does not deadlock, although the spawned process will; this should probably be changed.)

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.

$$
\begin{aligned}
(a{::}x) &= \lambda fg.f\ a\ x \\
\langle\rangle &= \lambda fg.g \\
pop &= \lambda v\kappa.v\kappa\mathbf{0}
\end{aligned}
$$

Figure 11: Compiler auxiliaries

The auxiliary terms are given in Figure 10. The term $\mathbf{0}$ implements deadlock; it cannot communicate. The term $\dagger$ is then a deadlocking continuation. Taken together, the injection (boxing) terms in_i and the projection (unboxing) terms out_j 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.

The rest of the auxiliary terms are straightforward, being mostly concerned with boxing and unboxing values. The term *subtract* makes use of an undefined 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 has the property that $(primitive\_if\ulcorner true\urcorner\ M\ N) \stackrel{\beta}{=} M$ and $(primitive\_if\ulcorner false\urcorner\ M\ N) \stackrel{\beta}{=} N$.

For later use, Figure 11 gives an encoding of stacks, which deadlocks upon stack underflow. Note the use of $\mathbf{0}$ instead of $\dagger$; since $\langle\rangle$ is a nullary constructor, the corresponding projector does not take a continuation argument.

5.3 The Compiler

Following the programme laid out in Section 3.1, we now re-express the semantics of our language in terms of a compile-time symbol table Γ, a compile-time code sequel π, a run-time display u, an intermediate value stack ζ, and a dynamic continuation κ. The compiler is given in Figure 12, and the instruction terms are given in Figure 13. Notice that while the semantics of an entire program $\mathcal{P}[\![e]\!]$ is a program term, $\mathcal{CP}[\![e]\!]$ is a process term. Intuitively, the semantics of a source program is a CLAM program, while the compiler produces code for the program's initial process.

The compiler correctness condition is simply that the compiled program, when launched, should be β-bisimilar to the semantic definition of the program:

$$[p_0 : start\,\mathcal{CP}[\![a]\!]] \stackrel{\beta}{\simeq} \mathcal{P}[\![a]\!] \tag{3}$$

This amounts to showing that the respective process terms are β-equivalent:

$$start\,\mathcal{CP}[\![a]\!] \stackrel{\beta}{=} \mathcal{E}[\![a]\!]\rho_0(\lambda f.boot\ f\ c_{p_0}) \tag{4}$$

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

Figure 12: The compiler

To show this, we will need to show that the usual compilation correctness condition holds on the expressions of the language, namely that for all expressions e,

$$(\mathcal{C}[\![e]\!]\Gamma\pi)u\zeta\kappa \overset{\beta}{=} \mathcal{E}[\![e]\!](u\circ\Gamma)(\lambda v.(\pi u(v{::}\zeta)\kappa)) \tag{5}$$

Of course, the instructions were designed to make this condition hold. Using this as an induction hypothesis, it is easy to prove the following:

Theorem 3 For all programs e, the condition (3) holds.
Proof: It suffices to prove that equation (5) holds for each clause of the syntax. That it holds for the sequential portion of the language is reported in [WO92]. The **put**, **get**, and **spawn** clauses are reported to hold in [Wan94b, Theorem 2]; they were checked mechanically by the theorem prover described in [WS94], along with most of the sequential portion of the language.[3] For purposes of this proof, our **empty** clause is simply an alphabetic variant of **get**, so the proof for **get** applies here as well: one simply substitutes **empty**, *empty*, and c_{empty} for **get**, *get*, and $c_{receive}$, respectively. ∎

5.4 The Abstract Machine

We now give an abstract machine for our language. The process terms of CLAM produced by the compiler are represented by abstract data structures, and program terms are finite functions from process ids to these structures. The syntax of system states is given in Figure 14. The usual 4-tuple machine is extended to a 5-tuple machine with the addition of a

[3] The automatically checked language differed from ours in that it had anonymous procedures, as opposed to recursive procedures, and lacked a conditional.

$$
\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}\; (\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) \\
\star \qquad \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}
$$

Figure 13: The instructions

buffer, here represented for brevity as a stack. There are two kinds of process state, representing the idle and transitional states. In the latter there needs to be extra information, namely an activity marker.

The intended correspondence between system states and terms is given by the function $\bar{\cdot}$, which is written out in Figure 15.

The operational semantics of the machine is conveniently partitioned into a sequential part and a communicating part. Since the sequential rules are, by construction, not involved with communication, we find it convenient to write them as rules for a 4-register machine and then lift them to the 5-register machine. The computation rules and the lifting rule are given in Figure 16. 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 $\langle \pi, u, z, k, b \rangle \to_1 \langle \texttt{stopped} \rangle$.

Figure 17 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 two-stroke 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 Figure 7.

This correspondence is formalized as follows:

Definition 7 We say that a CLAM term Π and an abstract machine term Q correspond if $\Pi \overset{\beta}{\simeq} \overline{Q}$. We write $\Pi \simeq Q$.

$$
\begin{array}{cccll}
 & Q & = & [p:s]\ldots & \text{System states} \\
 & s & ::= & \langle \texttt{start}, \pi\rangle & \text{Process states} \\
\star & & | & t & \\
\star & & | & \langle m, t\rangle & \\
 & & | & \langle \texttt{stopped}\rangle & \\
\star & t & ::= & \langle \pi, u, z, k, b\rangle & \text{Idle state} \\
 & \pi & ::= & \langle \texttt{fetch}, n, \pi\rangle & \text{Code sequences} \\
 & & | & \langle \texttt{pushint}, n, \pi\rangle & \\
 & & | & \langle \texttt{pushclosure}, \pi, \pi\rangle & \\
 & & | & \langle \texttt{return}\rangle & \\
 & & | & \langle \texttt{call}, \pi\rangle & \\
 & & | & \langle \texttt{brtrue}, \pi, \pi\rangle & \\
 & & | & \langle \texttt{sub}, \pi\rangle & \\
 & & | & \langle \texttt{put}, \pi\rangle & \\
 & & | & \langle \texttt{get}, \pi\rangle & \\
\star & & | & \langle \texttt{empty}, \pi\rangle & \\
 & & | & \langle \texttt{spawn}, \pi\rangle & \\
 & & | & \langle \texttt{swap}, \pi\rangle & \\
 & u & ::= & z & \text{Displays} \\
 & z & ::= & \langle\rangle & \text{Stacks} \\
 & & | & \langle v, z\rangle & \\
 & k & ::= & \langle \texttt{halt}\rangle & \text{Continuations} \\
 & & | & \langle \texttt{frame}, \pi, u, z, k\rangle & \\
\star & b & ::= & z & \text{Buffers} \\
 & v & ::= & \langle \texttt{int}, n\rangle & \text{Values} \\
 & & | & \langle \texttt{pid}, p\rangle & \\
 & & | & \langle \texttt{closure}, \pi, u\rangle & \\
 & & | & \langle \texttt{bool}, 0\rangle & \\
 & & | & \langle \texttt{bool}, 1\rangle & \\
 & & | & \langle \texttt{unit}\rangle & \\
\star & m & ::= & \langle \texttt{receiving}\rangle & \text{Activity markers} \\
\star & & | & \langle \texttt{emptying}, v\rangle & \\
\star & & | & \langle \texttt{sending}\rangle & \\
\star & & | & \langle \texttt{spawning}, v\rangle & \\
\star & & | & \langle \texttt{computing}\rangle &
\end{array}
$$

Figure 14: System state syntax

$$
\begin{array}{lll}
 & \overline{[p : s] \ldots} & = [p : \overline{s}] \ldots \\[1ex]
 & \overline{\langle \texttt{start}, \pi \rangle} & = \mathit{start}\, \overline{\pi} \\
\star & \overline{\langle \texttt{computing}, t \rangle} & = \overline{t} \\
\star & \overline{\langle m, t \rangle} & = \langle \overline{m}, \overline{t} \rangle \\
 & \overline{\langle \texttt{stopped} \rangle} & = \mathbf{0} \\[1ex]
\star & \overline{\langle \pi, u, z_1, k, z_2 \rangle} & = \langle (\overline{\pi}\, \overline{u}\, \overline{z_1}\, \overline{k}), \overline{z_2} \rangle \\[1ex]
 & \overline{\langle \texttt{fetch}, n, \pi \rangle} & = \mathit{fetch} \ulcorner n \urcorner \overline{\pi} \\
 & \overline{\langle \texttt{pushint}, n, \pi \rangle} & = \mathit{pushint} \ulcorner n \urcorner \overline{\pi} \\
 & \overline{\langle \texttt{pushclosure}, \pi, \pi' \rangle} & = \mathit{pushclosure}\, \overline{\pi'}\, \overline{\pi} \\
 & \overline{\langle \texttt{return} \rangle} & = \mathit{return} \\
 & \overline{\langle \texttt{call} \rangle, \pi} & = \mathit{call}\, \overline{\pi} \\
 & \overline{\langle \texttt{brtrue} \rangle, \pi_t, \pi_f} & = \mathit{brtrue}\, \overline{\pi}_t\, \overline{\pi}_f \\
 & \overline{\langle \texttt{sub} \rangle, \pi} & = \mathit{sub}\, \overline{\pi} \\
 & \overline{\langle \texttt{put} \rangle, \pi} & = \mathit{put}\, \overline{\pi} \\
 & \overline{\langle \texttt{get} \rangle, \pi} & = \mathit{get}\, \overline{\pi} \\
\star & \overline{\langle \texttt{empty} \rangle, \pi} & = \mathit{empty}\, \overline{\pi} \\
 & \overline{\langle \texttt{spawn} \rangle, \pi} & = \mathit{spawn}\, \overline{\pi} \\
 & \overline{\langle \texttt{swap} \rangle, \pi} & = \mathit{swap}\, \overline{\pi} \\[1ex]
 & \overline{\langle \rangle} & = \langle \rangle \\
 & \overline{\langle v, z \rangle} & = (\overline{v} :: \overline{z}) \\[1ex]
 & \overline{\langle \texttt{halt} \rangle} & = \dagger \\
 & \overline{\langle \texttt{frame}, \pi, u, z, k \rangle} & = \lambda v. \overline{\pi}\, \overline{u} (v :: \overline{z}) \overline{k} \\[1ex]
 & \overline{\langle \texttt{int}, n \rangle} & = \mathit{in}_{\mathit{int}} \ulcorner n \urcorner \\
 & \overline{\langle \texttt{pid}, p \rangle} & = \mathit{in}_{\mathit{pid}}\, c_p \\
 & \overline{\langle \texttt{closure}, \pi, u \rangle} & = \mathit{in}_{\mathit{clo}} (\lambda v \kappa. \overline{\pi} (v :: \overline{u}) \langle \rangle \kappa) \\
 & \overline{\langle \texttt{bool}, 0 \rangle} & = \mathit{in}_{\mathit{bool}} \ulcorner \mathit{false} \urcorner \\
 & \overline{\langle \texttt{bool}, 1 \rangle} & = \mathit{in}_{\mathit{bool}} \ulcorner \mathit{true} \urcorner \\
 & \overline{\langle \texttt{unit} \rangle} & = \mathit{unit} \\[1ex]
\star & \overline{\langle \texttt{receiving} \rangle} & = \langle c_{\mathit{receiving}} \rangle \\
\star & \overline{\langle \texttt{emptying}, v \rangle} & = \langle c_{\mathit{emptying}}\, \overline{v} \rangle \\
\star & \overline{\langle \texttt{sending} \rangle} & = \langle c_{\mathit{sending}} \rangle \\
\star & \overline{\langle \texttt{spawning}, v \rangle} & = \langle c_{\mathit{spawning}}\, \overline{v} \rangle
\end{array}
$$

Figure 15: Denotation of system states. Note that the rule for $\langle \texttt{computing}, t \rangle$ takes precedence over the rule for $\langle m, t \rangle$ which follows it.

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

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

Figure 16: Computation rules

Next we shall define a translator $\mathcal{TP}$ from source programs to abstract machine programs. The translator, shown in Figure 18, is a straightforward modification of the compiler of Figure 12, to produce abstract machine terms instead of CLAM terms. We can then show the following:

Lemma 1 For any program e the following hold:

$$\overline{\langle \texttt{start}, \mathcal{TP}[\![a]\!]\rangle} = \mathit{start}\,\mathcal{CP}[\![a]\!] \tag{6}$$

$$\mathcal{P}[\![a]\!] \simeq [p_0 : \langle \texttt{start}, \mathcal{TP}[\![a]\!]\rangle] \tag{7}$$

Proof: Equation (6) is easily shown to be correct, by induction on the structure of e. From this, the compiler correctness condition (3), and the definition of $\simeq$, the correspondence (7) between translated terms and the original semantics follows immediately. ∎

Of course this lemma doesn't really say anything yet, since we have not yet shown that $\simeq$ is a bisimulation. We now apply the proof techniques of [Wan94b] to CLAM, to show that $\simeq$ is in fact 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 CLAM terms, and then that each observable transition of one system is justified by a corresponding transition in the other. In the following, we write $\Pi \overset{\beta}{\simeq} \overline{Q}$ instead of $\Pi \simeq Q$ for simplicity.

Lemma 2 If $s \rightarrow_1 s'$ then $\overline{s} \rightarrow^+_\beta \overline{s'}$ via a reduction sequence containing at least one leftmost step.

Proof: By inspection of each transition of $\rightarrow_1$, including the default cases. ∎

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

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

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

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

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

Figure 17: Small step semantics

Lemma 3 (Sequential Simulation) If $Q \xrightarrow{\tau} Q'$ then $\overline{Q} \stackrel{\beta}{\simeq} \overline{Q'}$.
Proof: If the transition is due to the first rule of Figure 17, then $\overline{Q} = \overline{Q'}$ because $\overline{\langle \texttt{computing}, t\rangle} = \overline{t}$, so $\overline{Q} \stackrel{\beta}{\simeq} \overline{Q'}$. If the transition is due to the second rule of Figure 17, then we can use the preceding lemma, and again $\overline{Q} \stackrel{\beta}{\simeq} \overline{Q'}$. ∎

Lemma 4 (Communicating Simulation) If $a \neq \tau$, $Q \xrightarrow{a} Q'$ and $\Pi \stackrel{\beta}{\simeq} \overline{Q}$, then there is a Π' with $\Pi \xrightarrow{a} \Pi'$ and $\Pi' \stackrel{\beta}{\simeq} \overline{Q'}$.
Proof: We must examine each observable transition of Q in turn. The proofs for each are all similar; we shall write out the proof for the $\langle receive\rangle^+$ transition. The transition is of the form

$$Q[p : X] \xrightarrow{p:\langle receive\rangle^+} Q[p : \langle\langle \texttt{receiving}\rangle, X\rangle]$$

where $X = \langle\langle \texttt{get}, \pi\rangle, u, z, k, b\rangle$. Call the term on the left L and the term on the right R. We need to show that if $\Pi \stackrel{\beta}{\simeq} \overline{L}$, then there is a Π' with $\Pi \xrightarrow{p:\langle receive\rangle^+} \Pi'$ and $\Pi' \stackrel{\beta}{\simeq} \overline{R}$. Now we have

$$\begin{array}{rcll}
\overline{L} & = & \overline{Q}[p : \overline{X}] & \text{by } \overline{\cdot} \\
\overline{X} & = & \overline{\langle\langle \texttt{get}, \pi\rangle, u, z, k, b\rangle} & \\
 & = & ((get\,\overline{\pi})\,\overline{u}\,\overline{z}\,\overline{k}), \overline{b} & \text{by } \overline{\cdot} \\
 & \stackrel{\beta}{=} & (c_{receive}\,(\lambda v.\overline{\pi}\,\overline{u}\,(v{::}\overline{z})\,\overline{k}), \overline{b} & \text{by } get \text{ and } \beta
\end{array}$$

Call the last term M_2. Since $\Pi \stackrel{\beta}{\simeq} \overline{L}$, Π must be of the form $\Pi_0[p : M_1]$, with $\Pi_0 \stackrel{\beta}{\simeq} \overline{Q}$ and $M_1 \stackrel{\beta}{=} M_2 \stackrel{\beta}{=} X$. Then by the $\langle receive\rangle^+$ rule and the foregoing, we have

$$\begin{array}{rcl}
\Pi & \xrightarrow{p:\langle receive\rangle^+} & \Pi_0[p : \langle c_{receiving}, M_2\rangle] \\
 & \stackrel{\beta}{\simeq} & \overline{Q}[p : \langle\overline{\langle \texttt{receiving}\rangle}, \overline{X}\rangle] \\
 & = & \overline{R}
\end{array}$$

as required. ∎

Lemma 5 (Completeness) If $\Pi \xrightarrow{a} \Pi'$ and $\Pi \stackrel{\beta}{\simeq} Q$ then there is a Q_1 and a Q' with $Q \stackrel{\tau}{\Rightarrow} Q_1 \xrightarrow{a} Q'$ and $\Pi' \stackrel{\beta}{\simeq} Q'$.
Proof: Suppose $\Pi \xrightarrow{p:\langle receive\rangle^+} \Pi'$ and $\Pi = \Pi_0[p : M]$. Then it must be that $M \stackrel{\beta}{=} c_{receive}\,N$. Since $\Pi \stackrel{\beta}{\simeq} \overline{Q}$ we must have $Q = Q_0[p : s]$ where $\Pi_0 \stackrel{\beta}{\simeq} \overline{Q_0}$ and $M \stackrel{\beta}{=} \overline{s}$. Since M has a head normal form, so does $\overline{s}$. Now consider the sequence s_i of machine process states $s = s_0 \rightarrow_1 s_1 \rightarrow_1 \ldots$. By Lemma 2, the terms $\overline{s_i}$ form a quasi leftmost reduction sequence [Bar84]. Since quasi leftmost reduction is complete for head normal forms [Wan94b] and $\overline{s}$ has a head normal form, the sequence s_i must end on a term s_k where $\overline{s_k}$ is a head normal form. Since $\overline{s_k} \stackrel{\beta}{=} \overline{s_0} \stackrel{\beta}{=} M \stackrel{\beta}{=} c_{receive}\,N$, it must be that the instruction component of s_k is a $\langle \texttt{get}, \pi\rangle$ instruction. Thus $Q_0[p : s_k] \xrightarrow{\langle receive\rangle^+} Q'$, for some Q'. Since the s_i represent a sequence of $\rightarrow_1$ transitions, the first two rules of Figure 17 enable us to construct the

required sequence of machine states:

$$\begin{array}{rcll} Q_{2i} &=& Q_0[p : s_i] & (i \le k) \\ Q_{2i+1} &=& Q_0[p : \langle \texttt{computing}, s_i \rangle] & (i < k) \\ Q_{2k+1} &=& Q' & \end{array}$$

so that we have $Q_0 \xrightarrow{\tau} Q_1 \xrightarrow{\tau} \ldots Q_{2k} \xrightarrow{\langle receive \rangle^+} Q'$.

Now since $\overline{s_k} \stackrel{\beta}{=} \overline{s_0} \stackrel{\beta}{=} M$, $\Pi \stackrel{\beta}{\simeq} \overline{Q_0[p : s_k]}$ and Lemma 4 tells us that $\Pi \xrightarrow{\langle receive \rangle^+} \Pi''$ for some Π''. The form of the communication rules is such that the final state of a $\xrightarrow{\langle receive \rangle^+}$ transition is uniquely determined by the initial state, so $\Pi'' = \Pi'$ and therefore Q' is the required term.

The reasoning for the other transitions is similar. In the case of the $^-$ transitions the machine will not need to simulate any β-reductions.

Combining these lemmas gives the following:

Theorem 4 $\simeq$ is a branching bisimulation.

Proof: Suppose we have $\Pi \simeq Q$, so that $\Pi \stackrel{\beta}{\simeq} \overline{Q}$. We then verify the conditions in Definition 3:

1. Suppose $\Pi \xrightarrow{a} \Pi'$.

   (a) The case $a = \tau$ cannot occur, because CLAM terms have only a big step communication semantics.

   (b) If $a \neq \tau$ then Lemma 5 applies.

2. Suppose $Q \xrightarrow{a} Q'$.

   (a) Suppose $a = \tau$. If it is a $^+$ transition, then $\overline{Q} = \overline{Q'}$. If it is a $^-$ transition, then $Q \stackrel{\beta}{\simeq} Q'$ by Lemma 3, so $\Pi \stackrel{\beta}{\simeq} \overline{Q'}$.

   (b) If $a \neq \tau$ then Lemma 4 applies.

With this, we see that condition (7) of Lemma 1 does in fact state the correctness of the translator.

5.5 Summary

We have given a CLAM semantics for the simple functional language Urlang, and derived and proved the correctness of a bytecode compiler for Urlang. In Chapter 6 we present and prove the correctness of an implementation of Urlang.

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

Figure 18: The translator

Chapter 6

An Urlang Implementation

> If justifying a procedure means anything more than proving its consistency, it can only mean determining whether the procedure is successful in fulfilling its purpose. Indeed success is necessary; here, too, it is the highest tribunal, to which everyone submits.
>
> —*David Hilbert*

Our Urlang implementation is written in Concert/C. Section 6.1 describes our idiom of transcribing abstract data types into C, and Section 6.2 provides an introduction to those facilities of Concert/C which we use in the implementation.

Section 6.3 outlines the general plan of attack for our implementation. Section 6.4 describes the gross anatomy of the implementation. Section 6.5 presents the code generator of our Urlang compiler. Section 6.6 explains how the various implementation processes communicate with each other. Section 6.7 gives the programming details of the compiler and console process. Section 6.8 presents the bytecode interpreter.

In order to streamline the presentations in Chapter 5 and Section 6.8, the implementation omits such optimizations as tail recursion and garbage collection. These would, of course, be necessary in a production system.

6.1 Data Structuring Conventions

Much of our implementation's code is concerned with manipulating abstract data types, for which we have developed an encoding idiom for C. We now consider a small example, first showing how it would work in ML, and then showing how to do something similar in C.

If we wanted to represent binary trees with integer leaves, we might write the following in ML:

```
datatype tree =
    leaf of int
  | node of tree * tree
    ;
```

This declaration accomplishes several things:

- it declares that the data type `tree` has two variants, `leaf` and `node`
- it declares that the `leaf` variant consists of an `int`
- it creates a constructor function `leaf` of type `int -> tree`
- it declares that the `node` variant consists of a pair of `tree`s
- it creates a constructor function `node` of type `tree * tree -> tree`

When we write code to deal with a `tree`, we make use of ML's pattern matching to discriminate the cases and destructure the components:

```
fun print_tree (t) =
    case t of
        leaf (i) => print i
      | node (l, r) => (print "(";
                        print_tree l;
                        print ", ";
                        print_tree r;
                        print ")")
            ;
```

C requires that we work at a more concrete level, but we can keep the details from becoming oppressive by adopting a rote method of transcribing abstract data types. We start by declaring the data structure as follows:

```
typedef enum {leaf, node} tree_type;
typedef struct t {
  tree_type type_code;
  union {
    int leaf;
    struct {
      struct t *left;
      struct t *right;
    } node;
  } guts;
} tree_struct;
typedef tree_struct *tree;
```

While the syntax is somewhat daunting, the intent is fairly straightforward:

- `tree_type` is declared to be an enumerated data type, that is, it takes on the values denoted by `leaf` and `node`, which are defined to be integer constants.

- `tree_struct` is declared to be a structure with two components: `type_code`, which is of type `tree_type`, and `guts`, which is a variant, or `union`, type. `type_code` says which variant of `tree` the structure represents, and `guts` contains the data.

  If `type_code` is `leaf`, then the `leaf` part of the `guts` contains an integer. If `type_code` is `node`, then the `node` part of `guts` is filled in with a structure containing a pair of (pointers to) tree structures. This relation between the `type_code` and which part of the `union` is used must be enforced by the programmer.

  This structure is recursive, but C does not really support recursive data structures. The phrase `struct t` after the word `typedef` serves to introduce a local name for the structure being defined, so that the components of the structure can refer to it recursively, via the pointer type `struct t *`. The `typedef` itself serves to give this recursive structure the name `tree_struct`; after that, the type `struct t` need never be mentioned again.

- The type `tree` is defined to be a pointer to a `tree_struct`. This type gets the nicest name because it is the most frequently used.

Now we can write the two constructor functions we need. Note that each function needs to mention `tree_struct` once, because a structure type and not a pointer type is required by the `make_a` macro; this is the only time the abstract data structure itself need be mentioned, other than for forward references.

```
tree Leaf (int i) {
  tree t = make_a (tree_struct);

  t->type_code = leaf;
  t->guts.leaf = i;
  return (t);
}

tree Node (tree l, tree r) {
  tree t = make_a (tree_struct);

  t->type_code = node;
  t->guts.node.left = l;
  t->guts.node.right = r;
  return (t);
}

#define NOTREE ((tree) NULL)
```

The data structure and constructor definitions illustrate the orthography for the names of such: the structure type's name ends in `_struct`, the pointer type's name is the structure type's with the `_struct` removed, and the constructors are capitalized and named after the variants they construct. Where we need a null pointer to indicate the absence of a particular type of structure, its name will be all upper case and consist of the name of the structure preceded by `NO` or `EMPTY` as appropriate.

When we use a `tree` structure, instead of full pattern matching we must content ourselves with C's `switch` statement, which is at least sufficient to discriminate between the variants of an abstract datum. Where further discrimination on the contents of data structure components is needed, we must resort to conditionals. For the present example, the `switch` statement suffices:

```
void print_tree (tree t) {
  switch (t->type_code) {
  case leaf:
    printf ("%d", t->guts.leaf);
    break;
  case node:
    printf ("(");
    print_tree (t->guts.node.left);
    printf (", ");
    print_tree (t->guts.node.right);
    printf (")");
    break;
  }
}
```

We close this section with several comments on the subtler aspects of C's handling of names, some of which may by now be worrying the reader.

The slots of `struct` and `union` types are named, and the names are scoped within the containing type. Thus some other `struct` type could have fields called `left` or `right` without any confusion arising, and it also explains why we were able to use `leaf` and `node` both for the name of the `guts` variants and the elements of the `tree_type` enumeration. We shall use this separation of name spaces even more in our Urlang implementation, to minimize the number of names we have to keep straight, even to the point of naming the discriminator code slot of an abstract data structure with the name of the enumeration type it contains.

The most extreme abuse of this notation will be in the introduction of local structure names when declaring recursive structures: where we wrote `struct t` for a local name for the structure being defined, in future we will reuse the actual name of the structure, on the grounds that it is easier if we do not have to have two names for the same thing.

6.2 Concert/C

Concert/C adds several new constructs to C. Some of these are statements and data types pertaining directly to communication, and some are annotations describing the format of data structures. We shall give examples and explanations of each Concert/C construct used in the Urlang implementation, using the example of the previous section as a basis.

6.2.1 Data Annotation and Transmission

Concert/C's main claim to fame is that it automates the process of marshalling and unmarshalling data structures, that is, it can automatically traverse any data structure in order to produce a linear, portable representation for transmission, and similarly turn a linear representation into a data structure isomorphic to the original.[1] The information needed for the traversal can easily be determined from the user's type declarations without any additional annotation, except in the case of `union` types. For these, the user must annotate the data structure with `switch_is` and `case` annotations, to show where the discriminator field is, and to give the correspondence between its values and various parts of the `union`. We illustrate by decorating the `tree_struct` declaration of the previous section:

```
typedef struct t {
  tree_type type_code;
  [switch_is (type_code)]
  union {
    [case (leaf)]
    int leaf;
    [case (node)]
    struct {
      struct t *left;
      struct t *right;
    } node;
  } guts;
} tree_struct;
```

This works fairly painlessly with our convention for transcribing abstract data types: one annotates the discriminator field as being such, and annotates each case of the `guts` `union` with the appropriate discriminator value.

Once a data type t has been annotated as necessary, it can be sent and received along a communication port of the corresponding type, which is written using Concert/C's `receiveport` declarator as `receiveport`$\{t\}$. Thus we could declare `tree_port` to be a type of port for communicating `tree_struct`s[2] as follows:

[1] It is actually up to the user whether graph isomorphism is to be preserved or whether aliased pointers may lead to copying; in the former case, the user must annotate any potentially aliased pointers. Since our transmissible data structures are never modified, extraneous copying is harmless except for the well-known potential for exponential space blowup. Annotating the aliased pointers seems to cause the Concert/C marshaller to loop; it has not yet determined whether this is a Concert/C bug or whether there is an erroneous annotation in our code.

[2] Because we prefer using pointers to using structures, we would normally want a port for communicating `tree`s. This example is slightly less confusing using `tree_struct`s.

```
typedef receiveport {tree_struct} tree_port;
```

A message is communicated over a port by the `send` and `receive` commands, which may be understood as polymorphic functions. A message of type t is sent by an instance of `send` which takes arguments of types `(receiveport`$\{t\}$ `*)` and t, while it is received by an instance of `receive` which takes arguments of type `receiveport`$\{t\}$ and `(`t `*)`. The following fragment is therefore type-correct:

```
tree_port *outgoing, incoming;

void test_tree () {
  tree_struct t;

  receive (incoming, &t);
  send (outgoing, t);
}
```

The types of the second arguments to `send` and `receive` follow the conventions for the formatted I/O functions `printf` and `scanf`: output functions are only interested in values, while input functions need locations to fill with data. The case of the first argument is rather more obscure. A reasonable intuitive story is this: when a `receiveport` is declared by a process, that process is the only one which can `receive` from that port. When the port gets passed to other processes, they have not an actual `receiveport`, but rather a mechanism which communicates with a `receiveport` in another process. Concert/C gives this mechanism the type of a pointer to a `receiveport`.

It matters little whether this story is correct, or even whether the reader believes it. At worst one can simply take it as an arbitrary parameter-passing convention, and simply insert referencing and dereferencing operations as necessary to make one's Concert/C code compile.

6.2.2 Process Control

When a Concert/C program spawns another UNIX executable as a process, at least one port must be passed between the programs or they will never be able to communicate. There is no difficulty sending ports over ports, but of course one has to have a port to begin with. Concert/C provides a mechanism whereby a designated port of the child process is returned back to the parent at the time the child is spawned.

The child program indicates which of its `receiveport`s is to be returned to its parent by declaring it `initial`. Continuing with our tree example, we might find the following declaration in a tree server:

```
initial tree_port incoming;
```

Now consider the following code fragment from a hypothetical parent program which would spawn a tree server, using the constructs `program`, `cn_prog_init`, `cn_prog_set_name`, and `create` of Concert/C:

```
{
  program child;
  tree_port *outgoing;

  child = cn_prog_init ();
  cn_prog_set_name (child, "/tree/server/name/goes/here");
  create (child, &outgoing);
}
```

To spawn a process, one needs to initialize a variable of type `program` to the value returned by `cn_prog_init`, set the name of the UNIX executable into the `program` with `cn_prog_set_name`, and call `create` on the `program`. `create` returns a pointer to the other process' `initial receiveport`, doing so (for no obvious reason) by side-effecting its second argument, which must therefore be a pointer to a pointer variable.[3] Thus after the `create`, the variable `outgoing` in the parent can be thought of as a pointer to the `receiveport` called `incoming` in the child, despite the fact that they are in two different processes.

The reader may find this example somewhat unrealistic, as the server process has no way to send messages to its parent, or indeed any other process other than those it might spawn. Obviously one needs to be careful about one's choice of `initial receiveport`, so that after an exchange of messages the parent and child have pointers to all of the ports they need. A general solution is for the child's `initial` port to receive records containing all of the ports the parent needs to divulge, and for one of those ports to receive records containing all of the ports the child needs to divulge.

6.3 Implementing Abstract Machines

Before turning to the details of our implementation, we consider the question of what it means for a computer program to represent an abstract machine. Since our calculus CLAM was supposed to reflect our understanding of how concurrent programs really work, we should be able to bring this understanding to bear on the problem of implementing Urlang, which has a CLAM semantics.

As a CLAM program is a collection of processes, our implementation is a collection of Concert/C UNIX Processes. As the semantics of a CLAM program is a description of the possible interleavings of state transitions of concurrently executing processes, we must show how the state transitions of our Concert/C processes give rise to a corresponding interleaving description. Specifically, we have the following obligations:

- We must give an account of the state space of Concert/C programs at an appropriate level of detail.

[3] In our example, there seems to be no reason to have such a thing as a `program`, but that is only because of the way we happen to be using Concert/C. The value of a `program` variable is known as a process handle. There is a `terminate` function, taking a process handle as an argument, which terminates the process without its cooperation, so there is in fact something you can do with a `program` variable after using it to spawn another process.

- We must decide on labels for the state transitions of such programs.
- We must give a relation between abstract machine states and program states, and show that it has some compelling property, in this case that it is a deadlock preserving simulation.

It goes without saying that we are not going to do this for all CLAM programs and all Concert/C programs, but rather that we will restrict our attention to compiled Urlang programs and our Concert/C progrgams purporting to implement them.

We work from the top down. An abstract machine state is a finite map from process identifiers to process states as described in Figure 14. Similarly an implementation state is a collection of Concert/C processes, which we can consider to be a finite map from Concert/C process identifiers to the (as yet unspecified) process states represented by Concert/C processes.

An abstract machine process state is simply an abstract datum, which we could represent in C as explained in Section 6.1. Recall that in our interleaving paradigm, transitions are instantaneous and process states are divided into active and idle states. Durational actions are represented by active states, which consist of an idle state and an activity marker. Now to accomplish an activity, our implementation will actually have to *do* something: a data structure representating a process state cannot be expected to evolve all by itself. As long as our implementation must consist in part of a running program anyway, we can consider the control state of the program to be part of the process state. We therefore take the state of an implementation process to consist of two components: a state variable containing an idle state represented as an abstract datum, and the program counter of the process. Thus the relation between implementation states and abstract machine states will actually be a map, taking a state variable-program counter pair to an abstract machine state.[4] Given the value of the state variable, the program counter location determines whether the process is in the idle state represented by the state variable, or whether the process is in an active state of which the state variable is a part.

The remaining part of the correspondence is the labelling of implementation transitions. Having established the correspondence between implementation states and abstract machine states, we shall take the label of an implementation transition to be the label of the transition between the corresponding abstract machine states.

We must now suspend this discussion until the concrete details of the program have been presented.

6.4 Program Overview

Our Concert/C implementation of Urlang is a compile-and-go system. It consists of two separate programs, namely the bytecode compiler and the runtime bytecode interpreter.

[4] In actual fact, some of the active states will have a little extra information stored in variables. These extra values may be considered to form a tuple with the program counter.

The main program, which the user runs, is the compiler. After compiling the source program, it spawns a server process representing the initial CLAM process, and thereafter serves as a console process, facilitating communication with the user. Each CLAM process is implemented as a Concert/C process running a bytecode interpreter. A CLAM program is implemented as a collection of such processes. Thus when we argue later that this implementation of Urlang is correct, we mean that the collection of independent processes, taken as a whole, is related to the abstract bytecode machine by a deadlock preserving simulation.

Since the details of the compiler front end are not particularly interesting, they have been relegated to Appendix A. Our story picks up with the compiler back end in Section 6.5.

6.5 Code Generation

We now start on the details of establishing the correspondence between implementation states and abstract machine states. Our job here is to generate data structures isomorphic to those created by the function $\mathcal{TE}$ of Figure 18. The representation of the rest of the process state information will be taken up in Section 6.8.

First we define an abstract data type for instructions, following the conventions of the previous sections. Here we break slightly with tradition, in that each instruction variant is a `struct`, even if it has only one slot. This enables us to establish the convention that instructions are linked by their `tail` slots.

```
typedef enum {pushconstant, op2, swap, op1, fetch, pushclosure, ret, call, brtrue,
                empty, get, put, spawn} instruction_code;

typedef struct inst_struct {
  instruction_code instruction_code;
  [switch_is (instruction_code)]
  union {
    [case (pushconstant)]
      struct {value c;
              struct inst_struct *tail;
            } pushconstant;
    [case (op2)]
      struct {ident id;
              struct inst_struct *tail;
            } op2;
    [case (swap)]
      struct {struct inst_struct *tail;
            } swap;
    [case (op1)]
      struct {ident id;
              struct inst_struct *tail;
            } op1;
    [case (fetch)]
      struct {lex addr;
              struct inst_struct *tail;
```

```
            } fetch;
    [case (pushclosure)]
      struct {struct inst_struct *body;
              struct inst_struct *tail;
            } pushclosure;
    [case (call)]
      struct {struct inst_struct *tail;
            } call;
    [case (brtrue)]
      struct {struct inst_struct *true_inst;
              struct inst_struct *false_inst;
            } brtrue;
    [case (empty)]
      struct {struct inst_struct *tail;
            } empty;
    [case (get)]
      struct {struct inst_struct *tail;
            } get;
    [case (put)]
      struct {struct inst_struct *tail;
            } put;
    [case (spawn)]
      struct {struct inst_struct *tail;
            } spawn;
    [default]
      NOTHING;
  } guts;
} inst_struct;
typedef inst_struct *inst;
```

Next we have some definitions for compile-time environments, and auxiliaries for environment lookup and extension.

```
typedef struct symtab_struct {
  string var;
  struct symtab_struct *tail;
} symtab_struct;
typedef symtab_struct *symtab;

#define EMPTYSYMTAB ((symtab) NULL)
#define LEXNOTFOUND (-1)


lex slookup (symtab gamma, string var) {
  lex a;

  for (a = 0;
       gamma != EMPTYSYMTAB && ! (streq (var, gamma->var));
       gamma = gamma->tail, a++);
  if (gamma == EMPTYSYMTAB)
```

```
    return (LEXNOTFOUND);
  else
    return (a);
}

symtab extend (string var, symtab gamma) {
  symtab s = make_a (symtab_struct);

  s->var = var;
  s->tail = gamma;
  return (s);
}
```

We now consider the actual code generator function, TE. Its outer structure is of the form

```
inst TE (source_term t,
         symtab gamma,
         inst pi) {
  switch (t->source_term_code) {

                      ...clauses for each abstract syntax term...

  default:
    fprintf (stderr, "codegen: bogus source term code %d\n", t->source_term_code);
    exit (-1);
    break;
  }
}
```

As with the abstract machine translator $\mathcal{TE}$, TE takes a source phrase, a compile-time environment, and a code sequel as arguments, and returns a code sequence. We now consider each clause of the abstract syntax in turn. The instruction data structures are created by the simple utilities of Section B.2.

$\underline{\mathcal{TE}[\![n]\!]\Gamma\pi = \langle \mathtt{pushint}, n, \pi\rangle}$:
Instead of having a special instruction for each kind of constant data, we have a general instruction, pushconstant, which holds an actual data value. Thus pushint instructions correspond to pushconstant instructions with integer value parts.

```
  case INT_CONSTANT:
    return (Pushconstant (Int (t->guts.integer_constant),
                          pi));
    break;

  case BOOL_CONSTANT:
    return (Pushconstant (Bool (t->guts.boolean_constant),
                          pi));
    break;
```

```
  case NIL_CONSTANT:
    return (Pushconstant (List (NIL),
                          pi));
    break;
```

$\mathcal{TE}[\![i]\!]\Gamma\pi = \langle \texttt{fetch}, (\mathit{slookup}\,\Gamma\, i), \pi\rangle$:
We stop the compilation if the identifier is not found in the compile-time environment. Aside from that the correspondence is direct.

```
  case VARIABLE: {
    lex addr = slookup (gamma, t->guts.variable);
    if (addr == LEXNOTFOUND) {
      fprintf (stderr, "Error: variable %s not bound near line %d\n",
               t->guts.variable, t->line_number);
      exit (-1);
    } else {
      return (Fetch (addr, pi));
    }
  }
    break;
```

$\mathcal{TE}[\![(\mathbf{rec}\ i_1(i_2)\ e)]\!]\Gamma\pi = \langle \texttt{pushclosure}, (\mathcal{TE}[\![e]\!](i_2{::}i_1{::}\Gamma)\langle\texttt{return}\rangle), \pi\rangle$:
Explicit `extend` calls correspond to extending the compile-time environment. Note that we cannot use a `pushconstant` instruction here, because the closure cannot in general be formed without the run-time environment. Also, the instruction is called `ret` because `return` is a reserved word in C.

```
  case PROCEDURE:
    return (Pushclosure (TE (t->guts.procedure.body,
                             extend (t->guts.procedure.argname,
                                     extend (t->guts.procedure.procname,
                                             gamma)),
                             Ret ()),
                         pi));
    break;
```

$\mathcal{TE}[\![(e_1\ e_2)]\!]\Gamma\pi = \mathcal{TE}[\![e_1]\!]\Gamma(\mathcal{TE}[\![e_2]\!]\Gamma\langle\texttt{call}, \pi\rangle)$:
The correspondence is direct.

```
  case APPLICATION:
    return (TE (t->guts.application.function, gamma,
                TE (t->guts.application.argument, gamma,
                    Call (pi))));
    break;
```

$\underline{\mathcal{TE}[\![(\mathbf{if}\ e_1\ e_2\ e_3)]\!]\Gamma\pi = \mathcal{TE}[\![e_1]\!]\Gamma\langle\mathtt{brtrue}, \mathcal{TE}[\![e_2]\!]\Gamma\pi, \mathcal{TE}[\![e_3]\!]\Gamma\pi\rangle}$:
The correspondence is direct.

```
  case IF_THEN_ELSE:
    return (TE (t->guts.if_then_else.test,
                gamma,
                Brtrue (TE (t->guts.if_then_else.true_part,
                            gamma, pi),
                        TE (t->guts.if_then_else.false_part,
                            gamma, pi))));
    break;
```

$\underline{\mathcal{TE}[\![(-\ e_1\ e_2)]\!]\Gamma\pi = \mathcal{TE}[\![e_1]\!]\Gamma(\mathcal{TE}[\![e_2]\!]\Gamma\langle\mathtt{sub}, \pi\rangle)}$:
Here we have general `op2` and `op1` instructions, instead of one for each operator. A parameter of the instruction indicates the operation.

```
  case BINARY_OP:
    return (TE (t->guts.binary_op.first_argument, gamma,
                TE (t->guts.binary_op.second_argument, gamma,
                    Op2 (t->guts.binary_op.operator, pi))));
    break;

  case UNARY_OP:
    return (TE (t->guts.unary_op.argument, gamma,
                Op1 (t->guts.unary_op.operator, pi)));
    break;
```

$\underline{\mathcal{TE}[\![(\mathbf{get})]\!]\Gamma\pi = \langle\mathtt{get}, \pi\rangle}$:
The correspondence is direct.

```
  case GET:
    return (Get (pi));
    break;
```

$\underline{\mathcal{TE}[\![(\mathbf{empty})]\!]\Gamma\pi = \langle\mathtt{empty}, \pi\rangle}$:
The correspondence is direct.

```
  case EMPTY:
    return (Empty (pi));
    break;
```

$\underline{\mathcal{TE}[\![(\mathbf{put}\ e_1\ e_2)]\!]\Gamma\pi = \mathcal{TE}[\![e_1]\!]\Gamma(\mathcal{TE}[\![e_2]\!]\Gamma\langle\mathtt{put}, \pi\rangle)}$:
The correspondence is direct.

```
  case PUT:
    return (TE (t->guts.put.recipient, gamma,
                TE (t->guts.put.message, gamma,
                    Put (pi))));
    break;
```

$\underline{\mathcal{TE}[\![(\textbf{spawn}\ e)]\!]\Gamma\pi = \mathcal{TE}[\![e]\!]\Gamma\langle\texttt{spawn},\pi\rangle}$:
The correspondence is direct.

```
  case SPAWN:
    return (TE (t->guts.spawn, gamma, Spawn (pi)));
    break;
```

$\underline{\mathcal{TP}[\![a]\!] = \mathcal{TE}[\![a]\!]\langle\rangle\langle\texttt{swap},\langle\texttt{call},\langle\texttt{return}\rangle\rangle\rangle}$:
The correspondence is direct.

```
inst generate_code (source_term t) {
  return (TE (t, EMPTYSYMTAB, Swap (Call (Ret ()))));
}
```

6.6 Communication and Process Structure

We give an overview of the communication and process structure of the Urlang implementation, in order to facilitate understanding of the material which follows.

The system consists of two executables, the compiler `urlang` and the bytecode interpreter `bytecodes`. There will be one UNIX process running the bytecode interpreter for each Urlang process, including the initial process.

Each `bytecodes` process has a port, called the *command port*, on which it can receive command messages from other processes. This port is the only means of communicating with a process, and so it serves as the implementation of an Urlang PID.

The `urlang` process provides three ports: the *chat port*, the *result port*, and the *process registry port*. The chat port receives messages which are to be printed out on the console. The result port is the same, except that sending a message to the result port constitutes a declaration that the computation is now finished and that all `bytecodes` processes should be shut down. Upon booting, each process sends itself (that is, its command port) to the process registry port, so that the `urlang` process always has a complete list of `bytecodes` processes. We take the view that the operation of these ports is outside the operational semantics of the concurrent bytecode machine. The side effects of sending messages to the chat and process registry ports are completely invisible to the machine, while sending a message to the result port is simply the programmer's way of turning off the machine. (The details of the Urlang `print` and `result_is` primitives can be found in the function `do_op1`, starting on page 71.)

6.7 The Program `urlang`

The program `urlang` serves two purposes: it compiles and launches the user's program, and it acts as a rudimentary console process in support of the user's processes. In particular, it provides facilities for printing values, for announcing the result of an entire computation, and for keeping track of the set of all processes, so that they can be cleaned up when the program finishes.

The result port and the chat port are both instances of `print_string_ports`, which receive `print_string_msg` structures:

```
typedef struct {int unix_pid;
                string hostname;
                string s;
              } print_string_msg;

typedef receiveport {print_string_msg} print_string_port;

print_string_port result_port, chat_port;
```

The story of command ports and process registry ports is rather more complicated because of mutual recursion: a command port receives commands, one variant of command is a boot command, a boot command contains a pointer to a process registry port, and a process registry port receives command ports. The usual C coding trick for breaking mutual recursion between structures will not work here, because `receiveport`s are not structures per se. To make the trick work, whenever `receiveport`s are involved in such recursions we must wrap them in one-element structures. With profound apologies to the reader, we present the declarations for commands, command ports, and process registry ports:

```
typedef enum {boot, start, message, shutdown, ack} command_code;

typedef struct {
  command_code command_code;
  [switch_is (command_code)]
  union {
    [case (boot)]
      struct {
        struct value_struct *boot_closure;
        print_string_port *result_port;
        print_string_port *chat_port;
        struct command_port_port_struct *process_registry_struct;
      } boot;
    [case (start)]
      struct {
        struct inst_struct *main_program;
        print_string_port *result_port;
        print_string_port *chat_port;
        struct command_port_port_struct *process_registry_struct;
      } start;
    [case (message)]
      struct value_struct *msg;
    [default]
      NOTHING;
  } guts;
  struct command_port_holder_struct *ack_port_struct;
} command_struct;
typedef command_struct *command;

typedef receiveport {command_struct} command_receiveport;

typedef command_receiveport *command_port;
```

```
typedef struct command_port_holder_struct {command_port command_port;}
  command_port_holder_struct;
typedef command_port_holder_struct *command_port_holder;

typedef receiveport {command_port} command_port_port;

typedef struct command_port_port_struct {
  command_port_port *command_port_port;
} command_port_port_struct;

#define NO_ACKNOWLEDGMENT_REQUIRED (command_port_holder)NULL
```

The alert reader will have noticed the `ack_port_structure` slot. If this slot contains a command port, it means that the sender is expecting an `ack` command to be returned once the receiver has processed the comand in question. This is necessary only to synchronize `put` instructions with receipt of the messages they send, as explained on page 74.

The process registry is straightforward: we maintain a stack of registered processes. Registering a process amounts to pushing it onto the stack, and cleaning up consists of sending each registered process a `Shutdown` command.

```
command_port_port process_registry_port;
command_port_port_struct
  process_registry_struct = {&process_registry_port};

command_port_stack registered_processes = EMPTYCOMMANDPORTSTACK;

void register_process (command_port registrant) {
  registered_processes = Command_port_stack (registrant, registered_processes);
}

void shut_down_all_processes () {
  command_port com;
  while (registered_processes != EMPTYCOMMANDPORTSTACK) {
    com = registered_processes->top;
    send_command (com,
                  Shutdown (),
                  NO_ACKNOWLEDGMENT_REQUIRED);
    registered_processes = registered_processes->tail;
  }
}
```

Next we have the portion of `urlang` which deals with running the compiler. The upper-case variables in the conditionals represent command-line options.

```
source_term the_source_program;
inst the_object_code;

main (int argc, char *argv[])
{
  program my_server;
```

```
command_port initial_process, registrant;
print_string_msg print_msg;
int run = 1;

process_arguments (argc - 1, argv + 1);
if (yyparse ()) {
  fprintf (stderr, "Parse failed\n");
  exit (-1);
}
if (PRINT_SOURCE) {
  fprintf (stderr, "\nORIGINAL SOURCE:\n");
  print_source_term (stderr, 0, the_source_program);
}
the_source_program = macroexpand (the_source_program);
if (PRINT_MACROEXPANDED_SOURCE) {
  fprintf (stderr, "\nMACROEXPANDED SOURCE:\n");
  print_source_term (stderr, 0, the_source_program);
}
the_object_code = generate_code (the_source_program);
if (PRINT_OBJECT) {
  fprintf (stderr, "\nOBJECT:\n");
  print_inst (stderr, 0, the_object_code);
}
if (COMPILE_ONLY)
  exit (0);
fprintf (stderr, "\nRUN:\n");
```

To launch the compiled program, we first need to start and register a bytecode interpreter. When a Concert/C process is started by the `create` construct, it has the option to return one port to the parent; in our case, the bytecode interpreter returns its command port, without which further communication would be impossible.

```
my_server = cn_prog_init ();
cn_prog_set_name (my_server, "/home/daveg/phd/clam/concert/bytecodes");
create (my_server, &initial_process);
send (process_registry_port, initial_process);
```

Next we need to send the interpreter a `Start` command, to start the program. The last argument to `send_command` will usually be the null pointer `NO_ACKNOWLEDGMENT_REQUIRED` as it is here, indicating that the sender of this message is not expecting the receiver to acknowledge receipt of the message.

```
send_command (initial_process,
              Start (the_object_code,
                     &result_port,
                     &chat_port,
                     &process_registry_struct
                     ),
              NO_ACKNOWLEDGMENT_REQUIRED);
```

The remaining job of the `urlang` process is to process requests on the chat, result, and process registry ports. The Concert/C `select` construct determines which of the ports has a message, while the `receive` construct copies the message into a given structure. Note that the loop continues until a message is received on the result port, at which time the program is shut down.

```
  while (run) {
    switch (select (result_port, chat_port, process_registry_port)) {
    case 1:
      receive (result_port, &print_msg);
      printf ("RESULT:\n[%s %d]: %s\n",
              print_msg.hostname, print_msg.unix_pid, print_msg.s);
      run = 0;
      break;
    case 2:
      receive (chat_port, &print_msg);
      if (PRINT_CHAT)
        printf ("[%s %d]: %s\n",
                print_msg.hostname, print_msg.unix_pid, print_msg.s);
      break;
    case 3:
      receive (process_registry_port, &registrant);
      register_process (registrant);
      break;
    }
  }
  shut_down_all_processes ();

  exit (0);
}
```

6.8 Bytecode Interpreter

We start by giving some top level variable declarations and utilities which will be used later. The routines are not profound, but we do not wish to cause any undue confusion in the reader by using them before they are defined.

```
initial command_receiveport the_command_port;
command_port_holder_struct
  the_command_port_holder_struct = {&the_command_port};
command_port_holder acknowledgment_required = &the_command_port_holder_struct;

print_string_port *result_port, *chat_port;
char hostname[256];
int mypid;

void result_is (string s) {
  print_string_msg psd;
  psd.unix_pid = mypid;
```

```
  psd.hostname = hostname;
  psd.s = s;
  if (run_state != UNBORN)
    send (*result_port, psd);
}

void display (string s) {
  print_string_msg psd;
  psd.unix_pid = mypid;
  psd.hostname = hostname;
  psd.s = s;
  if (run_state != UNBORN)
    send (*chat_port, psd);
}

command_port_port_struct *process_registry_struct;

char cause_of_death[256] = "Unknown.";

void get_stuck (string s) {
  strcpy (cause_of_death, s);
  run_state = STOPPED;
}
```

Next we give the data structures needed for our representation of machine states. This includes representations of data, environments, stacks, and the machine state itself; instructions were already presented in Section 6.5.[5]

```
typedef enum {integer, bool, pid, closure, list, unit} value_code;

typedef struct {
  struct value_struct *head;
  struct value_struct *tail;
} cons_struct;
typedef cons_struct *cons;

typedef struct env_struct {
  struct value_struct *val;
  struct env_struct *tail;
} env_struct;
typedef env_struct *env;

typedef struct value_struct {
  value_code value_code;
  [switch_is (value_code)]
  union {
    [case (integer)]
      int integer;
```

[5]The reason we write `struct inst_struct *` here instead of `inst` is that these declarations actually precede those for instructions in the source code.

```
    [case (bool)]
      int bool;
    [case (pid)]
      command_port pid;
    [case (closure)]
      struct {struct inst_struct *inst;
              env env;
            } closure;
    [case (list)]
      cons list;
  } guts;
} value_struct;
typedef value_struct *value;

typedef struct stack_struct {
  value top;
  struct stack_struct *tail;
} stack_struct;
typedef stack_struct *stack;

typedef struct command_port_stack_struct {
  command_port top;
  struct command_port_stack_struct *tail;
} command_port_stack_struct;
typedef command_port_stack_struct *command_port_stack;

typedef struct {
  stack head;
  stack tail;
} buffer_struct;
typedef buffer_struct *buffer;

typedef struct frame_struct {
  struct inst_struct *pi;
  env u;
  stack z;
  struct frame_struct *k;
} frame_struct;
typedef frame_struct *frame;

typedef struct {
  struct inst_struct *pi;
  env u;
  stack z;
  frame k;
  buffer buffer;
} state_struct;
typedef state_struct *state;
```

To make coding the interpreter easier, we introduce an abstraction of the interpreter's state:

```
enum {UNBORN, RUNNABLE, RECEIVING, SENDING, HALTED, STOPPED, DEFUNCT} run_state;
```

Aside from the bracketing `UNBORN` and `DEFUNCT` states, `run_state` is an abstraction of the machine state. When the program counter is in `main`, as opposed to when it is in `process_command` or `execute_one_instruction`, `run_state` may be interpreted as follows:

- `UNBORN`: The interpreter has not yet received a `boot` or `start` command. This process is not yet part of the program state.
- `RUNNABLE`: The machine has been booted, and is neither stopped nor waiting for anything. The process is in the idle state represented by the state variable `s`.[6]
- `RECEIVING`: The machine has executed a `get` instruction but the buffer is empty, so the machine is waiting for a message to arrive. The process is in a ⟨`receiving`⟩ state.
- `SENDING`: The machine has executed a `send` instruction and sent the message, and is awaiting acknowledgment of receipt. The process is in a ⟨`sending`⟩ state.
- `HALTED`: The machine has encountered a `HALT` continuation. The process is in the ⟨`stopped`⟩ state.
- `STOPPED`: The machine has erroneously run to a state from which there are no transitions. The process is in the ⟨`stopped`⟩ state.
- `DEFUNCT`: The intepreter has received a `Shutdown` command.

The interpretation of the process states when the program counter is in the functions `process_command` or `execute_one_instruction` will be given when those routines are described.

The variable `s`, which holds the state information for the current process, is declared at top level to be visible to each function of the interpreter.

```
state s;
```

The program `urlang` starts by establishing its UNIX process id and hostname. Initially the `run_state` is `UNBORN`. We enter the outermost loop, which runs until the interpreter is shut down:

```
main ()
{
  static char buf[256];
  command_struct com;

  mypid = getpid ();
  gethostname (hostname, 256);
  {
```

[6] Recall that an idle state is merely one which is not actively engaged in a durational action (see page 22). Here we distinguish idle states which can proceed (are `RUNNABLE`) from those which cannot (`HALTED` or `STOPPED`).

```
    char *p;
    for (p = hostname; *p && *p != '.'; p++);
    *p = '\0';
  }

  run_state = UNBORN;
  while (run_state != DEFUNCT) {
```

The first responsibility of the loop body is to receive incoming messages. Messages are received until such time as either (1) the machine is ready to run and there are no messages immediately available, or (2) the interpreter is shut down. Here we use the Concert/C `poll` construct, which is like `select` except that it returns 0 instead of blocking if there are no messages available:

```
    while (! ((run_state == RUNNABLE && poll (the_command_port) == 0)
              ||
              run_state == DEFUNCT)) {
        receive (the_command_port, &com);
        process_command (&com);
      }
```

The details of `process_command` will be discussed shortly, including the ways it can change the `run_state` in response to received messages.

If the machine is `RUNNABLE` after incoming messages are taken care of, we execute one instruction via `execute_one_instruction`. Afterward we check whether the machine is now in a terminal state, and if so we display appropriate console messages. This concludes the interpreter loop.

```
      if (run_state == RUNNABLE)
        execute_one_instruction ();
      if (run_state == HALTED) {
        if (s->z != EMPTYSTACK) {
          sprintf (buf, "The machine halted with %s on top of the stack.",
                   decode_value (s->z->top));
          display (buf);
        }
        else display ("The machine halted with an empty stack.");
      }
      else if (run_state == STOPPED) {
        sprintf (buf, "The machine got stuck; cause of death is `%s'",
                 cause_of_death);
        display(buf);
      }
  }
  exit (0);
}
```

The procedure `process_command` handles all messages. We consider each case in turn, starting with `boot`. The `boot` message contains the closure to be run, as well as the chat, result, and process registry ports. The initial program state, constructed by the function

`State`, consists of the initial instruction sequence specified by the semantics, an empty environment, a stack consisting of the closure and the PID, a terminal continuation, and an empty buffer. Receipt of the `boot` message causes the run state to be set to `RUNNABLE`. The `boot` message puts the process into its initial idle state. It is not associated with any transitions, because a process in the `unborn` state is not considered to correspond to any state component of the abstract machine.

```
void process_command (command com)
{
  switch (com->command_code) {
  case boot:
    s = State (Call (Ret ()),
               EMPTYENV,
               Stack (Pid (&the_command_port),
                      Stack (com->guts.boot.boot_closure,
                             EMPTYSTACK)),
               HALT,
               Empty_buffer ());
    result_port = com->guts.boot.result_port;
    chat_port = com->guts.boot.chat_port;
    process_registry_struct = com->guts.boot.process_registry_struct;
    run_state = RUNNABLE;
    display ("Booted.");
    break;
```

A `start` command is a minor variant of a `boot` command. Where `boot` contains a closure, `start` contains the compiled main program, which is code for producing the initial closure. Rather than implementing an actual $\langle$`start`$\rangle$ state, we initialize the process with the result of the $\langle$`start`$, \pi\rangle$ computation rule from Figure 16. This is clearly safe because the `start` command is used only to start the initial process.

```
  case start:
    s = State (com->guts.start.main_program,
               EMPTYENV,
               Stack (Pid (&the_command_port),
                      EMPTYSTACK),
               HALT,
               Empty_buffer ());
    result_port = com->guts.start.result_port;
    chat_port = com->guts.start.chat_port;
    process_registry_struct = com->guts.start.process_registry_struct;
    run_state = RUNNABLE;
    display ("Started.");
    break;
```

A `message` message represents a CLAM message. The message is enqueued. If the machine was in a `RECEIVING` state due to a `get` instruction encountering an empty buffer, then this code implements the $\langle receive\rangle^-$ transition, and the process is returned to an idle state. If the process was not waiting in a `RECEIVING` state, then this does not count as a transition of this process; rather it is part of the $\langle send\rangle^-$ transition of the sending process.

```
  case message:
    s = State (s->pi,
               s->u,
               s->z,
               s->k,
               Enqueue (com->guts.msg, s->buffer));
    if (run_state == RECEIVING) {
      buffer b;
      value v;

      v = Head (s->buffer);
      b = Dequeue (s->buffer);
      s = State (s->pi->guts.get.tail,
                 s->u,
                 Stack (v,
                        s->z),
                 s->k,
                 b);
      run_state = RUNNABLE;
    }
    break;
```

A `shutdown` simply sets the `run_state` to `DEFUNCT`, causing the program to exit.

```
  case shutdown:
    run_state = DEFUNCT;
    break;
```

A `ack` message is received by a process in a $\langle$`sending`$\rangle$ state. This case implements the $\langle send\rangle^-$ transition. It sets the `run_state` back to `RUNNABLE`, allowing the interpreter to proceed.

```
  case ack:
    s = State (s->pi->guts.put.tail,
               s->u,
               Stack (Unit(), s->z->tail->tail),
               s->k,
               s->buffer);
    run_state = RUNNABLE;
    break;
  }
```

This concludes the processing of the individual messages. The remaining piece of business is to check whether the sender wants an acknowledgment, and if so, to send one.

```
  if (com->ack_port_struct != NO_ACKNOWLEDGMENT_REQUIRED) {
    send_command (com->ack_port_struct->command_port,
                  Ack (),
                  NO_ACKNOWLEDGMENT_REQUIRED);
  }
}
```

Now we turn to the heart of the interpreter: `execute_one_instruction`, which implements the machine's instructions. Its structure is as follows:

```
void execute_one_instruction () {
    switch (s->pi->instruction_code) {

                    ...clauses for each instruction...

    default:
      sprintf (cause_of_death,
               "unimplemented instruction code %d.", s->pi->instruction_code);
      run_state = STOPPED;
      break;
    }
  }
```

We now consider each instruction in turn, starting with some remarks that apply to all of the purely computational instructions of Figure 16. Recall that these were expressed as a relation $\rightarrow_0$ on the first four components of the state, and then lifted to a relation $\rightarrow_1$ on process states by the rule

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

In our implementation we can see that for each $\rightarrow_0$ transition the buffer component of the state is unchanged.

In Figure 17, the $\rightarrow_1$ relation was lifted to a relation on machine states by the rule

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

The clauses for the individual computation rules have the same general form:

`case` *instruction name*:
  `if (`*erroneous state*`) get_stuck (`*pithy message*`);`
  `else s =` *some function of* `s;`
  `break;`

When the program counter is in one of these `case` clauses, we take the correspondence to be this: the process is in the idle state represented by `s` until entry to the `else` clause, wherupon the process is in a $\langle$`computing`$\rangle$ state. (Obviously if the test fails the process is in a $\langle$`stopped`$\rangle$ state.) The completion of the statement in the `else` clause corresponds to the $^{-}$ transition back to an idle state. This forecloses the possibility of another process sending a message to a process which is in a $\langle$`computing`$\rangle$ state; however, since a process cannot remain in such a state forever, this cannot cause deadlock.

Having said this, we now consider the code implementing each of the $\rightarrow_0$ rules in turn.

$\langle\langle \texttt{fetch}, n, \pi\rangle, u, z, k\rangle \rightarrow_0 \langle \pi, u, \langle (\mathit{clookup}\ u\ n), z\rangle, k\rangle$:
Aside from the details of the environment lookup, the correspondence is direct. For safety, we allow for the possibility that the lookup fails, although the compiler should not generate bad lexical references.

```
case fetch: {
  value v = clookup (s->u, s->pi->guts.fetch.addr);
    if (v == NOVALUE)
      run_state = STOPPED;
    else
      s = State (s->pi->guts.fetch.tail,
                 s->u,
                 Stack (v, s->z),
                 s->k,
                 s->buffer);
  };
  break;
```

The run time environment is implemented as a stack, so the lookup is straightforward:

```
value clookup (env e, lex addr) {
  lex a = addr;

  while (addr > 0 && e != EMPTYENV) addr--, e = e->tail;
  if (e != EMPTYENV && addr == 0)
    return (e->val);
  else {
    sprintf (cause_of_death, "Couldn't look up lexical address %d.", a);
    return (NOVALUE);
  }
}
```

$\langle\langle \texttt{pushint}, n, \pi\rangle, u, z, k\rangle \rightarrow_0 \langle \pi, u, \langle\langle \texttt{int}, n\rangle, z\rangle, k\rangle$:
As remarked earlier, we have a general `pushconstant` instruction which does away with the need for type-specific `push` instructions. Aside from this, the correspondence is direct.

```
case pushconstant:
  s = State (s->pi->guts.pushconstant.tail,
             s->u,
             Stack (s->pi->guts.pushconstant.c, s->z),
             s->k,
             s->buffer);
  break;
```

$\underline{\langle\langle \texttt{pushclosure}, \pi', \pi\rangle, u, z, k\rangle \rightarrow_0 \langle \pi, u, \langle\langle \texttt{closure}, \pi', u\rangle, z\rangle, k\rangle}$:
The correspondence is direct.

```
    case pushclosure:
      s = State (s->pi->guts.pushclosure.tail,
                 s->u,
                 Stack (Closure (s->pi->guts.pushclosure.body, s->u), s->z),
                 s->k,
                 s->buffer);
      break;
```

$\underline{\langle\langle \texttt{return}\rangle, u, \langle v, z\rangle, \langle \texttt{frame}, \pi', u', z', k'\rangle\rangle \rightarrow_0 \langle \pi', u', \langle v, z'\rangle, k'\rangle}$:
The first test checks that there is a saved continuation; if not, the machine halts. The second test checks that the stack is nonempty, that is, that there is a value to return. Aside from this the correspondence is direct.

```
    case ret:
      if (s->k == HALT)
        run_state = HALTED;
      else if (s->z == EMPTYSTACK)
        get_stuck ("ret fails: stack is empty");
      else
        s = State (s->k->pi,
                   s->k->u,
                   Stack (s->z->top, s->k->z),
                   s->k->k,
                   s->buffer);
      break;
```

$\underline{\langle\langle \texttt{call}, \pi\rangle, u, \langle v, \langle\langle \texttt{closure}, \pi', u'\rangle, z\rangle\rangle, k\rangle \rightarrow_0 \langle \pi', \langle\langle \texttt{closure}, \pi', u'\rangle, \langle v, u'\rangle\rangle, \langle\rangle, \langle \texttt{frame}, \pi, u, z, k\rangle\rangle}$:
The three tests implement the constraint that there must be a value and a closure on the stack. Aside from this the correspondence is direct.

```
    case call:
      if (s->z == EMPTYSTACK)
        get_stuck ("call fails: stack is empty");
      else if (s->z->tail == EMPTYSTACK)
        get_stuck ("call fails: no function on stack");
      else if (s->z->tail->top->value_code != closure)
        get_stuck ("call fails: callee is not a closure");
      else
        s = State (s->z->tail->top->guts.closure.inst,
                   Env (s->z->top,
                        Env (s->z->tail->top,
                             s->z->tail->top->guts.closure.env)),
                   EMPTYSTACK,
                   Frame (s->pi->guts.call.tail,
                          s->u,
                          s->z->tail->tail,
                          s->k),
                   s->buffer);
      break;
```

$$\left[\begin{array}{l} \langle\langle \texttt{brtrue}, \pi_t, \pi_f\rangle, u, \langle\langle \texttt{bool}, \mathit{false}\rangle, z\rangle, k\rangle \rightarrow_0 \langle \pi_f, u, z, k\rangle \\ \\ \langle\langle \texttt{brtrue}, \pi_t, \pi_f\rangle, u, \langle\langle \texttt{bool}, \mathit{true}\rangle, z\rangle, k\rangle \rightarrow_0 \langle \pi_t, u, z, k\rangle \end{array}\right]:$$

The two rules are implemented as one clause, plus a test for the boolean value. The conditionals implement the constraint that there must be a boolean value on top of the stack. Aside from this the correspondence is direct.

```
    case brtrue:
      if (s->z == EMPTYSTACK)
        get_stuck ("brtrue fails: stack is empty");
      else if (s->z->top->value_code != bool)
        get_stuck ("brtrue fails: testee is not a boolean");
      else
        s = State ((s->z->top->guts.bool?
                    s->pi->guts.brtrue.true_inst:
                    s->pi->guts.brtrue.false_inst),
                   s->u,
                   s->z->tail,
                   s->k,
                   s->buffer);
      break;
```

$\langle\langle \texttt{sub}, \pi\rangle, u, \langle\langle \texttt{int}, n\rangle, \langle\langle \texttt{int}, m\rangle, z\rangle\rangle, k\rangle \rightarrow_0 \langle \pi, u, \langle\langle \texttt{int}, (m-n)\rangle, z\rangle, k\rangle$:

Analogously to the `pushconstant` instruction, we have a single `op2` instruction for binary primitives, with a field in the instruction indicating the primitive. The auxiliary `do_op2` dispatches on the operator field, does the required type checking, and returns either the operation's value, or `NOVALUE` in the event of a type error. The machine gets stuck as required if the type checks fail.

```
    case op2:
      if (s->z != EMPTYSTACK && s->z->tail != EMPTYSTACK) {
        value v = do_op2 (s->pi->guts.op2.id, s->z->tail->top, s->z->top);
        if (v == NOVALUE)
          run_state = STOPPED;
        else
          s = State (s->pi->guts.op2.tail,
                     s->u,
                     Stack (v, s->z->tail->tail),
                     s->k,
                     s->buffer);
      };
      break;
```

`do_op2` implements the type checking of each primitive, returning the special value `NOVALUE` if the type check fails.

```
value do_op2 (ident code, value v1, value v2) {
  value_code v1_code = v1->value_code, v2_code = v2->value_code;

  if (v1_code == integer && integer == v2_code) {
    if (streq (code, "+"))
```

```
      return (Int (v1->guts.integer + v2->guts.integer));
    else if (streq (code, "-"))
      return (Int (v1->guts.integer - v2->guts.integer));
    else if (streq (code, "*"))
      return (Int (v1->guts.integer * v2->guts.integer));
    else if (streq (code, "/"))
      return (Int (v1->guts.integer / v2->guts.integer));
    else if (streq (code, "="))
      return (Bool (v1->guts.integer == v2->guts.integer));
    else if (streq (code, ">"))
      return (Bool (v1->guts.integer > v2->guts.integer));
    else if (streq (code, "%"))
      return (Int (v1->guts.integer % v2->guts.integer));
  }
  if (streq (code, "cons"))
    return (List (Cons (v1, v2)));
  sprintf (cause_of_death,
           "Unknown or inapplicable op2 code '%s' for value codes %d and %d.",
           code, v1_code, v2_code);
  return (NOVALUE);
}
```

For convenience we have added unary operators to the language; they are straightforward. Note that the operators `print` and `result_is` are called for their extra-semantic side effects of sending messages to the chat or result port, respectively.

```
    case op1:
      if (s->z != EMPTYSTACK) {
        value v = do_op1 (s->pi->guts.op1.id, s->z->top);
        if (v == NOVALUE)
          run_state = STOPPED;
        else
          s = State (s->pi->guts.op1.tail,
                     s->u,
                     Stack (v, s->z->tail),
                     s->k,
                     s->buffer);
      };
      break;


value do_op1 (ident code, value v) {
  value_code v_code = v->value_code;

  if (streq (code, "print")) {
    display (decode_value (v));
    return (Unit ());
  }
  if (streq (code, "result_is")) {
    result_is (decode_value (v));
```

```
    return (Unit ());
  }
  if (streq (code, "bool?"))
      return (Bool (v_code == bool));
  if (streq (code, "integer?"))
      return (Bool (v_code == integer));
  if (streq (code, "PID?"))
      return (Bool (v_code == pid));
  if (streq (code, "closure?"))
      return (Bool (v_code == closure));
  if (streq (code, "list?"))
      return (Bool (v_code == list));
  if (v_code == integer) {
    if (streq (code, "0?"))
      return (Bool (v->guts.integer == 0));
  }
  if (v_code == list) {
    if (streq (code, "car") && v->guts.list != NIL)
      return (v->guts.list->head);
    if (streq (code, "cdr") && v->guts.list != NIL)
      return (v->guts.list->tail);
    if (streq (code, "null?"))
      return (Bool (v->guts.list == NIL));
  }
  sprintf (cause_of_death, "Unknown or inapplicable op1 code `%s'.", code);
  return (NOVALUE);
}
```

$\underline{\langle\langle \texttt{swap}, \pi\rangle, u, \langle v, \langle w, z\rangle\rangle, k\rangle \rightarrow_0 \langle \pi, u, \langle w, \langle v, z\rangle\rangle, k\rangle}$:
The conditionals implement the constraint that there be at least two elements on the stack. Aside from this the correspondence is direct.

```
    case swap:
      if (s->z == EMPTYSTACK)
        get_stuck ("swap fails: stack is empty");
      else if (s->z->tail == EMPTYSTACK)
        get_stuck ("swap fails: only one argument on stack");
      else
        s = State (s->pi->guts.swap.tail,
                   s->u,
                   Stack (s->z->tail->top,
                          Stack (s->z->top,
                                 s->z->tail->tail)),
                   s->k,
                   s->buffer);
      break;
```

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

The correspondence here is the same as for the computational instructions. Again we foreclose the possibility of a message being received during the active state, and again this cannot cause deadlock. The correspondence of the final states is direct.

```
case empty:
  s = State (s->pi->guts.empty.tail,
             s->u,
             Stack (Bool (buffer_is_empty (s->buffer)),
                    s->z),
             s->k,
             s->buffer);
  break;
```

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

The transitions into and out of the ⟨`receiving`⟩ state are marked by the assignments to `run_state`. If the buffer is nonempty, the $\langle receive\rangle^{-}$ transition is handled here. If the buffer is empty the process will be left in the ⟨`receiving`⟩ state upon exit from `execute_one_instruction`, and the $\langle receive\rangle^{-}$ transition will be left for the procedure `process_command`. This allows other processes to send messages to this process while it is in a ⟨`receiving`⟩ state; this is necessary to prevent deadlock, since otherwise the buffer would remain empty forever.

```
case get:
  run_state = RECEIVING;
  if (! buffer_is_empty (s->buffer)) {
    buffer b;
    value v;

    v = Head (s->buffer);
    b = Dequeue (s->buffer);
    s = State (s->pi->guts.get.tail,
               s->u,
               Stack (v,
                      s->z),
               s->k,
               b);
    run_state = RUNNABLE;
  }
  break;
```

$$\left[\begin{array}{c} \dfrac{X = \langle\langle \mathtt{put}, \pi\rangle, u, \langle v, \langle\langle \mathtt{pid}, p'\rangle, z\rangle\rangle, k, b\rangle}{Q[p:X] \xrightarrow{p:\langle send\ p'\rangle^+} Q[p:\langle \mathtt{sending}, X\rangle]} \\[2ex] \dfrac{X = \langle \mathtt{sending}, \langle\langle \mathtt{put}, \pi\rangle, u, \langle v, \langle\langle \mathtt{pid}, p'\rangle, z\rangle\rangle, k, b\rangle\rangle \quad Y = \langle \pi'', u'', z'', k'', b''\rangle}{Q[p:X][p':Y] \xrightarrow{p:\langle send\ p'\rangle^-} \begin{array}{l} Q[p:\langle \pi, u, \langle\langle \mathtt{int}, 1\rangle, z\rangle, k\rangle] \\ [p':\langle \pi'', u'', z'', k'', b''@\langle v, \langle\rangle\rangle\rangle] \end{array}} \\[2ex] \dfrac{X = \langle \mathtt{sending}, \langle\langle \mathtt{put}, \pi\rangle, u, \langle v, \langle\langle \mathtt{pid}, p'\rangle, z\rangle\rangle, k, b\rangle\rangle \quad Y = \langle m, \langle \pi'', u'', z'', k'', b''\rangle\rangle}{Q[p:X][p':Y] \xrightarrow{p:\langle send\ p'\rangle^-} \begin{array}{l} Q[p:\langle \pi, u, \langle\langle \mathtt{int}, 1\rangle, z\rangle, k\rangle] \\ [p':\langle m, \langle \pi'', u'', z'', k'', b''@\langle v, \langle\rangle\rangle\rangle\rangle] \end{array}} \end{array}\right]:$$

Here as before we take the entrance to the `else` to be the $^+$ transition, as indicated by the assignment to `run_state`, but there is no $^-$ transition here. Instead we enter a $\langle$`sending`$\rangle$ state until receipt of the message is acknowledged; thus the $^-$ transition actually occurs in the `process_command` procedure. The correspondence of the final states is direct after the usual stack checking.

```
case put:
  if (s->z == EMPTYSTACK || s->z->tail == EMPTYSTACK)
    get_stuck ("put fails: needed two arguments on stack.");
  else if (s->z->tail->top->value_code != pid)
    get_stuck ("put fails: must send to a pid.");
  else {
    run_state = SENDING;
    send_command (s->z->tail->top->guts.pid,
                  Message (s->z->top),
                  acknowledgment_required);
  }
  break;
```

Note that here the third argument to `send_command` is `acknowledgment_required`, which points to the command port of the current process. This indicates to the receiver that the current process will wait for an `ack` message before proceeding; but for this synchronization, the sending process could violate CLAM semantics by proceeding before the message had been received. It is unlikely that this would cause any problems in scientific applications, but it is very easy to write programs which can detect this impermissible behavior.

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

Here we take the $^+$ transition to be the point at which the `create` returns, since this is when the new PID is established. We take the $^-$ transition to be the end of the `else` as usual, since at that point there are no further observable events required to get the spawned process running.

```
case spawn:
  if (s->z == EMPTYSTACK)
```

```
  get_stuck ("spawn fails: nothing on stack.");
else if (s->z->top->value_code != closure)
  get_stuck ("spawn fails: must fork a closure.");
else {
  program my_server;
  command_port child;

  my_server = cn_prog_init ();
  cn_prog_set_name (my_server, "/home/daveg/phd/clam/concert/bytecodes");
  create (my_server, &child);

  send (*(process_registry_struct->command_port_port), child);

  send_command (child,
                Boot (s->z->top,
                      result_port,
                      chat_port,
                      process_registry_struct
                      ),
                NO_ACKNOWLEDGMENT_REQUIRED);
  s = State (s->pi->guts.spawn.tail,
             s->u,
             Stack (Pid (child), s->z->tail),
             s->k,
             s->buffer);
}
break;
```

6.9 Summary

We have shown how to encode the states of the abstract machine of Section 5.4 in C data structures. We have coded the machine's transitions as a Concert/C program, and argued that the program is in a deadlock-preserving simulation with the abstract machine.

Chapter 7

FINE Language Design

Language is a virus from outer space.
—*William S. Burroughs*

7.1 Introduction

Thus far we have defined CLAM, a concurrent λ-calculus (Chapter 4), used it to define the semantics of a simple concurrent functional language, Urlang, and derive a compiler for Urlang and prove it correct (Chapter 5), and shown Urlang's semantics to be reasonable by implementing it and proving the implementation correct (Chapter 6). It remains to argue that Urlang is a sufficiently powerful language to make the foregoing accomplishments meaningful.

The argument goes like this: we take it for granted that Erlang [AVW93] is sufficiently powerful that, had we implemented it and proved the implementation correct, there would be no question as to the significance of that accomplishment. Then we examine the informal definition of Erlang and note the features of that language which are missing from Urlang. For each feature, we explain what would be required to implement it in the present framework.

Erlang has the following features which are not present in Urlang:

- Error handlers
- Non-local exits (`catch` and `throw`)
- Selective reception of messages by pattern matching
- Function invocation by pattern matching
- Notification of death of child processes

$$
\begin{array}{rcll}
e & ::= & (\textbf{receive}\ c_1 \ldots c_n) & \text{Selective receive} \\
c & ::= & (m \rightarrow e) & \text{Clause (Pattern} \rightarrow \text{expression)} \\
m & ::= & i & \text{Patterns} \\
 & | & (i\ \textbf{suchthat}\ e) & \\
 & | & (i\ \textbf{oftype}\ t) & \\
 & | & (\textbf{equalto}\ e) & \\
 & | & (m\ .\ m) & \\
t & ::= & \textbf{int} & \text{Type specifiers} \\
 & | & \textbf{bool} & \\
 & | & \textbf{list} & \\
 & | & \textbf{cons} & \\
 & | & \textbf{nil} & \\
 & | & \textbf{closure} &
\end{array}
$$

Figure 19: FINE's **receive** construct.

The features may be partitioned according to whether or not they are purely sequential, or have something to do with the concurrent nature of the language. Error handling and non-local jumps are purely sequential, so we will not treat them further; see [SW74, CFW85] for a discussion. This leaves three items:

- Selective reception of messages by pattern matching is discussed in Section 7.2. We show that programs in the language FINE, which is Urlang plus a pattern matching `receive` construct, can be transformed into Urlang programs.
- Function invocation by pattern matching is easily handled by the mechanisms for pattern matching developed in Section 7.2.
- Erlang's death notification mechanism cannot be implemented in CLAM. Section 7.3 sketches the semantic enhancements that would be required, and shows that they are easily compatible with our framework.

7.2 Selective Reception of Messages

We now show how selective reception of messages via pattern matching may be implemented in Urlang. We define the source language FINE to be Urlang extended with the selective **receive** statement whose syntax is given in Figure 19, and we give an informal explanation of the **receive** statement, from the top down. In pseudocode, the statement works like this:

If a message in the buffer matches a clause's pattern
  Remove the message from the buffer
  Apply the message to the clause's result expression

The result of that expression is the value of the **receive** statement
Otherwise try again

The matching algorithm is suggested by the following pseudocode:

```
For each message in the buffer
  For each clause in the receive statement
    If the clause's pattern matches the message, announce success
Announce failure
```

To match a pattern to a value, one destructures the pattern and the value at the same time, checking that the structures match. The pattern bottoms out on a predicate which can simply be applied to the corresponding value. As the matching process proceeds, new variable bindings may be created. We now give an informal explanation of each type of pattern; the word 'datum' here refers to the value being matched to the pattern, and the phrase 'the clause's expression' refers to the expression e in the clause $p \rightarrow e$.

i: The match always succeeds. In the clause's expression, i is bound to the datum.

(i **suchthat** e): i is bound to the datum and e is evaluated. If the value of e is **true**, the match succeeds and in the clause's expression, i is bound to the datum.

(i **oftype** t): This expands into (i **suchthat** ($pred_t$ i)), where $pred_t$ denotes the built-in Urlang predicate which tests for the given type.

(**equalto** e): This expands into (`no_name` **suchthat** (== `no_name` e)). No user-visible bindings result.

(m_1 . m_2): The match succeeds if the datum is a **cons** whose **car** matches m_1 and whose **cdr** matches m_2. In the clause's result expression, any bindings resulting from m_1 are lexically outside those for m_2, that is, the bindings are evaluated from left to right.

We shall presently give a source-to-source transormation from FINE to Urlang, proceeding top down, analogous to our informal description of the **receive** construct. First, however, we must dispose of a few technical details:

- Because our translation is in the nature of a macro, we need a mechanism for generating identifiers which are guaranteed not to appear in the user's program. As the reader my have already seen, Appendix A.3 contains definitions for `let` and `progn` forms, which are purely syntactic extensions to Urlang; they are macros, which are expanded out in the first phase of compilation. In order to generate an identifier which cannot occur in the user's program, the code for `progn` uses a fact about the lexical structure of Urlang's identifiers, namely that they are of nonzero length (see pages 102–104); in our implementation, the empty string is a valid identifier in the abstract syntax tree, although it is not expressible in the concrete syntax. To obtain other identifiers which are guaranteed not to appear in the user's program, we simply use strings which contain spaces; these are also not expressible in the concrete syntax. For readability we write spaces as underscores, as in `no_name`, rather than `no␣name`.

- To make the code our FINE-to-Urlang transformation emits more concise, we make use of a few more syntactic abbreviations. Although we have not implemented these in our Urlang compiler, it would be easy to do so in the manner of the rules for `progn` and `let` in Appendix A.3.

$$
\begin{array}{ll}
(\lambda\ (var)\ body) \Rightarrow (\textbf{rec}\ \texttt{no\_name}\ (var)\ body) & \\
(\lambda\ (var_1 \ldots var_n)\ body) \Rightarrow (\lambda\ (var_1) & \\
\qquad\qquad\qquad\qquad (\lambda\ (var_2 \ldots var_n)\ body)) & \\
(M_1\ M_2 \ldots M_n) \Rightarrow ((M_1\ M_2 \ldots M_{n-1})\ M_n) & (n > 2))
\end{array}
$$

Now, the most serious difference between FINE as described and Urlang is that FINE's **receive** statement needs to be able to examine each message in the input buffer, whereas once an Urlang program retrieves a message via **get**, it has been removed from the input buffer. Rather than changing the semantics of the underlying calculus, we implement FINE in such a way as to maintain an input buffer data structure and manipulate it explicitly.

As each phrase of a FINE program executes, there is the possibility that the phrase (or a subphrase thereof) will alter the contents of the input buffer. We therefore express our FINE-to-Urlang transformation $\mathcal{F}$ as a 'continuation-buffer' semantics, by analogy with the usual environment-continuation semantics for a functional language: each phrase is mapped to a function taking a continuation and a buffer as arguments, and calling the continuation on the phrase's value and a new buffer.[1] The uninteresting part of the function $\mathcal{F}$, for the FINE phrases which do not involve the input buffer, is given in Figure 20. Since the buffer argument would simply be threaded through the various subphrases as they are executed, we make extensive use of η-conversion, replacing terms of the form $(\lambda x.(M\ x))$ with M, thus eliminating the buffer argument. The only item of note is `spawn`, which must take care that the spawned closure be compatible with the CLAM calling conventions, that is, it must run when called on nothing more than its process id.

The Urlang constructs **empty**, **get**, and **receive** manipulate the input buffer. While **empty** and **get** to not do anything conceptually different than their Urlang counterparts, we still need to give a nontrivial translation for them, because now the input buffer is maintained in software; thus it would be wrong to let these constructs use the CLAM input buffer. We leave the actual encoding of the software buffer unspecified, but the interface is as follows:

`empty_buffer`: An empty buffer.

[1]As we have stated, this is not intended as a suggestion for an actual implementation, but rather as a semantic definition. Since FINE has the same order of evaluation as Urlang, and since the buffer is threaded linearly through the computation, one could eliminate the continuations and have the buffer management functions side-effect a single global buffer variable.

`not_a_buffer`: A value which is not a buffer, but which causes a runtime type error if passed to any of the buffer functions, which follow. The buffer functions are required to report an error if passed `not_a_buffer`, while their behavior when passed a value which is neither `not_a_buffer` nor a legitimate buffer is undefined.

(`buffer_empty?` b): Tests whether b represents an empty buffer.

(`update_buffer` b): Append any available messages to buffer b in order.

(`update_buffer_nonempty` b): Append any available messages to buffer b, but don't return until the result would be nonempty.

(`update_buffer_change` b): Append any available messages to buffer b, but don't return until the result would be different from b.

(`buffer_head` b): The first message in b.

(`buffer_tail` b): The rest of the messages in b, after the first.

(`remove_message` b_1 b_2): b_2 was obtained from b_1 by applying `buffer_tail` n times; return a copy of b_1 in which the trailing portion b_2 is replaced by (`buffer_tail` b_2). In short, remove the first message in b_2 from b_1.

Figure 21 gives the portions of $\mathcal{F}$ for the input constructs, which we describe in turn. The **empty** construct updates the software buffer via `update_buffer`, and then tests it with `buffer_empty?`.

The `get` construct also updates the software buffer, but via `update_buffer_nonempty` because the `get` cannot proceed on an empty buffer. The first element of the resulting buffer is returned as the value, and the rest of the buffer is used as the buffer for the continuation.

`receive` constructs are translated into a pair of nested loops. The outer loop is the procedure `try_again`, which simply calls the inner loop on its argument, `the_buffer`. The inner loop is the procedure `scan_messages`, with argument `buffer_remaining` representing the rest of the input buffer to be examined. The main job of the inner loop is to match the clauses of the **receive** construct against the first message in `buffer_remaining`, which is bound to `this_msg`. At this point we run into a conundrum: the patterns which are being matched against the messages in the input buffer are specified as unrestricted FINE expressions; in particular, they may contain constructs which modify the input buffer. Erlang resolves this problem overconservatively by specifying a restricted sublanguage for pattern predicates which does not even allow user-defined procedures. We prefer to resolve the problem at the other end of the spectrum: we arrange that a runtime error will occur if any pattern code attempts to examine or modify the input buffer.

As we shall see in Figure 23, the code for each pattern and the corresponding result expression is transformed by $\mathcal{F}$, so that they take a continuation and an input buffer as arguments. The generated pattern code is called on `identity_continuation`, which simply returns the value on which it is called, and on `not_a_buffer`. Thus the effect of running the pattern code (if it doesn't misbehave and cause a runtime error) will be to return true

$\mathcal{F}[\![n]\!] =$
$\quad(\lambda\ (k)\ (k\ n))$
$\mathcal{F}[\![i]\!] =$
$\quad(\lambda\ (k)\ (k\ i))$
$\mathcal{F}[\![(\textbf{rec}\ f(a)\ body)]\!] =$
$\quad(\lambda\ (k)\ (k\ (\textbf{rec}\ f(a)\ \mathcal{F}[\![body]\!])))$
$\mathcal{F}[\![(rator\ rand)]\!] =$
$\quad(\lambda\ (k)\ (\mathcal{F}[\![rator]\!]\ (\lambda\ (f)\ (\mathcal{F}[\![rand]\!]\ (\lambda\ (a)\ (f\ a\ k))))))$
$\mathcal{F}[\![(\textbf{if}\ test\ iftrue\ iffalse)]\!] =$
$\quad(\lambda\ (k)\ (\mathcal{F}[\![test]\!]\ (\lambda\ (t)\ (\textbf{if}\ t\ (\mathcal{F}[\![iftrue]\!]\ k)\ (\mathcal{F}[\![iffalse]\!]\ k)))))$
$\mathcal{F}[\![(op_1\ arg)]\!] =$
$\quad(\lambda\ (k)\ (\mathcal{F}[\![arg]\!]\ (\lambda\ (a)\ (k\ (op_1\ a)))))$
$\mathcal{F}[\![(op_2\ arg_1\ arg_2)]\!] =$
$\quad(\lambda\ (k)\ (\mathcal{F}[\![arg_1]\!]\ (\lambda\ (a_1)\ (\mathcal{F}[\![arg_2]\!]\ (\lambda\ (a_2)\ (k\ (op_2\ a_1\ a_2))))))$
$\mathcal{F}[\![(\textbf{put}\ addressee\ message]\!] =$
$\quad(\lambda\ (k)\ (\mathcal{F}[\![addressee]\!]\ (\lambda\ (a)\ (\mathcal{F}[\![message]\!]\ (\lambda\ (m)\ (k\ (\textbf{put}\ a\ m)))))$
$\mathcal{F}[\![(\textbf{spawn}\ arg)]\!] =$
$\quad(\lambda\ (k)\ (\mathcal{F}[\![arg]\!]\ (\lambda\ (a)\ (k\ (\textbf{spawn}\ (\lambda\ (\texttt{pid})\ (a\ \texttt{pid}$
$\qquad\qquad\qquad\qquad\qquad\qquad\texttt{empty\_buffer}$
$\qquad\qquad\qquad\qquad\qquad\qquad\texttt{identity\_continuation})))))$

Figure 20: $\mathcal{F}$ for non-input constructs

```
F[[(empty)]] =
  (λ (k b)
    (let ((b (update_buffer b)))
     (k (buffer_empty? b) b)))
F[[(get)]] =
  (λ (k b)
    (let ((b (update_buffer_nonempty b)))
     (k (buffer_first b) (buffer_rest b))))
F[[(receive clauses)]] =
  (λ (k b)
    ((rec try_again (the_buffer)
      ((rec scan_messages (buffer_remaining)
        (if (buffer_empty? buffer_remaining)
            (try_again (update_buffer_change the_buffer))
            (let ((this_msg (buffer_head buffer_remaining))
                  (more_msgs (buffer_tail buffer_remaining))
                  (identity_continuation (λ (v b) v))
                  (real_buffer (remove_message the_buffer buffer_remaining))
                  (real_continuaton k))
             ML[[this_msg, clauses, (scan_messages more_msgs)]])))
       the_buffer))
     (update_buffer b)))
```

Figure 21: $\mathcal{F}$ for input constructs

$\mathcal{ML}[\![datum, [\,], failure]\!] = failure$
$\mathcal{ML}[\![datum, [(m \rightarrow e)::l], failure]\!] =$
$\quad \mathcal{M}[\![datum, m, e, \mathcal{ML}[\![datum, l, failure]\!]]\!]$

Figure 22: $\mathcal{ML}$

$\mathcal{M}[\![datum, i, success, failure]\!] =$
$\quad$ (**let** ((i $datum$)) ($\mathcal{F}[\![success]\!]$ `real_continuation` `real_buffer`)
$\mathcal{M}[\![datum, (i$ **suchthat** $e), success, failure]\!] =$
$\quad$ (**if** (**let** ((i $datum$)) ($\mathcal{F}[\![e]\!]$ `identity_continuation` `not_a_buffer`)
$\qquad$ (**let** ((i $datum$)) ($\mathcal{F}[\![success]\!]$ `real_continuation` `real_buffer`)
$\qquad$ $failure$)
$\mathcal{M}[\![datum, (i$ **oftype** $t), success, failure]\!] =$
$\quad \mathcal{M}[\![datum, (i$ **suchthat** $(pred_t\ i)), success, failure]\!]$
$\mathcal{M}[\![datum,$ (**equalto** $e), success, failure]\!] =$
$\quad \mathcal{M}[\![datum,$ (no_name **suchthat** (== no_name e)), $success, failure]\!]$
$\mathcal{M}[\![datum, (m_1\ .\ m_2), success, failure]\!] =$
$\quad \mathcal{M}[\![datum,$
$\qquad$ (`the_cons` **oftype** **cons**),
$\qquad$ (**let** ((`the_car` (**car** `the_cons`))
$\qquad\qquad$ (`the_cdr` (**cdr** `the_cons`)))
$\qquad\quad \mathcal{M}[\![$ `the_car`$, m_1, \mathcal{M}[\![$ `the_cdr`$, m_2, success, failure]\!], failure]\!]$),
$\qquad failure]\!]$

Figure 23: $\mathcal{M}$

or false, depending on whether the pattern matches. The result expression code, by contrast, is expected to carry on the computation. Therefore it is called on the arguments `real_continuation`, which is the continuation originally passed to the **receive** expression, and on `real_buffer`, which represents the buffer after the matched message has been removed . This explains the values bound in the **let** form, so all that remains is the call to the function $\mathcal{ML}$.

As we shall see in Figure 23, each pattern-action clause is expanded into code which examines a datum against a pattern, and either calls the action or a piece of failure code. The job of $\mathcal{ML}$ (Figure 22) is to take a datum, a list of clauses, and a piece of failure code, and produce code which tries all of the clauses in turn (expanding them with $\mathcal{M}$). Thus the failure code for the last clause is the failure code passed to $\mathcal{ML}$, while the failure code for any clause but the last is the $\mathcal{ML}$ expansion of the remaining clauses.

Figure 23 gives the actual definitions of these matching constructs. The function $\mathcal{M}$ is of type *source* $\times$ *pattern* $\times$ *source* $\times$ *source* $\rightarrow$ *source*: it takes source code representing the datum to examine, the pattern to match against, and source code for the success and failure actions, and produces source code which implements the matching and binding operations specified by the pattern. Since $\mathcal{M}$ calls $\mathcal{F}$ to transform the pattern and action, the functions $\mathcal{M}$, $\mathcal{ML}$, and $\mathcal{F}$ are mutually recursive.

The only remaining issue is the transformation of a top-level FINE program. As we saw in the transformation for `spawn` expressions in Figure 20, we need to put a wrapper in the $\mathcal{F}$-transformed code for the main program abstraction:

$$\begin{aligned}\mathcal{FP}[\![a]\!] = (\lambda\ (\texttt{pid})\\ &(\mathcal{F}[\![a]\!]\ \texttt{pid identity\_continuation empty\_buffer}))\end{aligned}$$

7.3 Death Notification

Erlang's death notification process cannot be implemented in CLAM. At a high level of description, it works as follows:

- A process may establish a bidirectional link with another process.
- If a process terminates abnormally a death notification is sent to all processes having a link to it.
- A process receiving a death notification is terminated abnormally, except that
- A process may elect to trap death notifications, in which case an incoming death notification is received as a normal message.

As things stand, there is simply no way for one process to kill another. We can, however, extend CLAM semantics in a very straightforward way to capture the desired behavior, taking a hint from the c_{send} rule of Figure 7 (page 25). We see that the c_{send} primitive puts its process into a $c_{sending}$ state, and that a process in a $c_{sending}$ state causes a message to appear in another process' input buffer; the other process could just as easily have been removed from the program or replaced by an arbitrary term had that been the desired effect. This observation leads us to the following sketch for defining the semantics of death notification, omitting for now the ability to trap notifications:

1. Add c_{link}, c_{unlink}, and c_{die} operators to CLAM's syntax (Figure 5, page 24).
2. Add activity marker constants $c_{linking}$, $c_{unlinking}$ and c_{dying} to CLAM's syntax.
3. Add a set of linked processes to each process: define Links = PID* and Idle state = Term $\times$ Buffer $\times$ Links in CLAM's syntax.
4. Add $\langle link \rangle$, $\langle unlink \rangle$, and $\langle die \rangle$ events to the event domain (Figure 6, page 24).
5. Write communication rules for c_{link} using c_{send} as a template: when process p links to process q, p should appear in q's Links set and vice versa.

6. Write communication rules for c_{unlink} using c_{send} as a template: when process p unlinks from process q, p should be removed from q's Links set and vice versa.

7. Write a rule sending a c_{die} term into a c_{dying} state, by analogy with c_{send}.

8. Write a rule whereby a c_{dying} process with p in its Links evolves into the same process with p removed, and process p evolves into c_{die}.

9. (Optional) A process in a c_{dying} state with empty Links is removed from the program.

From here it is a simple matter to carry through a compiler derivation like the one in Chapter 5. In addition to wiring up the new primitives, one would need to extend the auxiliaries (Figure 10, page 31) so as to be able to differentiate between normal and abnormal termination.

In the foregoing sketch we have omitted the mechanism for trapping death notifications. Briefly, one adds a Trap flag to each process, one adds primitives for turning the flag on and off, one adds communication rules to implement the primitives, and one makes the behavior of the c_{dying} state conditional on the linked-to process' Trap flag, so that it either evolves to c_{die} or some sort of death notification message appears in its input buffer.

7.4 Summary

We have examined the five features of Erlang that are not present in our language Urlang: error handling, non-local jumps, pattern matching reception of messages, pattern matching function invocation, and death notification. Error handling and non-local jumps were dismissed as being purely sequential. We showed how pattern matching reception of messages could be implemented by a source-to-source transformation. Having done so, we have developed mechanisms which can similarly implement pattern matching function invocation. Lastly, we discussed the straightforward semantic extensions which would be necessary for CLAM to support death notification, and sketch how it would be implemented, showing that it fits neatly into our framework.

Chapter 8

Some Early Difficulties

Once in a while you get shown the light
In the strangest of places if you look at it right.
—*Grateful Dead*

It is generally considered to be easier to implement a concurrent system via message passing than by more general mechanisms like shared memory [MP92]. The reason is quite straightforward: processes only interact where they share data, and the only shared data in a message passing system are communication channels. We therefore hoped that the implementation of Urlang, a message passing language, would be foolproof in Concert/C, which is itself a message passing language; since the sequential part of the language implementation is cut and dried, and the communication aspect is handled by the implementation language, there seemed to be little room for error to creep in. This turned out not to be the case, or rather, there was just a little room, and it was enough.

Our first attempt at implementing an early version of Urlang semantics was based on an appealing but unimplementable CLAM semantics [Gla94]. As with the current CLAM semantics, we started with HOCC [Wan94b]. HOCC programs are, as here, collections of labelled processes; in HOCC, however, communication is synchronous so processes have no need of buffers. The operational semantics specifies two kinds of program steps: the spawning of a new process, and synchronous value passing between a pair of processes. In the work cited, an abstract machine similar to ours is proven correct. Its small-step semantics describes programs taking steps in which a single process (or two communicating processes) takes a single step; the resulting semantics, if taken literally, is of a sort that has been called "massively serial computing" [Ams88]. A multiprocessor implementation of such a semantics would be silly: after going to the trouble of distributing a computation to multiple processors, great care must then be taken to ensure that no two processors execute simultaneously. This should not, however, be construed as an argument against the utility of such interleaving semantic descriptions of uniprocessor languages with concurrent processes.

Naturally, nobody expects a multiprocessor implementation to use mutual exclusion in the way described. By changing the semantics from synchronous to asynchronous buffered communication, one would seem to remove any need for interlocking at all. These ideas led to the following recipe for turning the interleaving semantics of HOCC into a true concurrency semantics for CLAM:

1. Add a buffer to each process.
2. Change `put` so that it puts a value into the buffer of the intended receiving process, without any cooperation on the part of the receiving process (that is, it sends a message).
3. Change `get` so that it removes a value from the process' own buffer (that is, it receives a message).
4. Take the above rules as descriptions of how a single process takes a step, and write down a "power set" transition for entire programs, that is, allow a program step to consist of any number of process steps (at most one per process).

This programme was realized in the following development, which is taken from [Gla94]. Note that here buffers are multisets rather than queues; we could have changed this quite easily, but we decided to leave this material intact in the spirit of full disclosure. Naturally any names redefined here are to be considered local to this chapter.

Our use of multisets in this exposition is minimal, but we define our notation for completeness.

Definition 8 (Multisets[Vog92]) A multiset over a set X is a function $\mu : X \to \mathbb{N}_0$. It is said to be finite if it has finite support, that is if $\mu(x) \neq 0$ for finitely many $x \in X$. It is said to be empty if $\mu(x) = 0$ for all $x \in X$. The set of finite non-empty multisets over X is denoted by $\mathcal{M}(X)$. We write $x \in \mu$ if $\mu(x) \neq 0$.

Convention 2 We use double brackets $\{\!\{\cdot\}\!\}$ to distinguish multisets and multiset comprehensions from sets.

8.1 Original CLAM

We base our concurrent calculus on the λ-calculus with constants. Starting from a set PID of process identifiers, we derive a set of constants, which contains only the following:

c_p	a constant for each process id p
c_{true}, c_{false}	booleans
$c_{spawn}, c_{send}, c_{receive}, c_{empty}$	uninterpreted communication operators

The syntactic domains of our calculus are as follows:

p	$\in$	PID	Process ids
M	$\in$	Term	λ-terms plus constants as above
B	$\in$	Buffer	Bag of closed values
M, B	$\in$	Process	Term $\times$ Buffer
Π	$\in$	Program	PID $\stackrel{\text{fin}}{\to}$ Process

$$\frac{M \overset{\beta}{=} (c_{receive}\ N) \quad V \in B}{\Pi[p : M, B] \overset{p:\langle receive\rangle}{\rightarrow} \Pi[p : (N\ V), B - \{\!\{V\}\!\}]}$$

$$\frac{M \overset{\beta}{=} (c_{empty}\ N) \quad V = \begin{cases} c_{true} & B = \emptyset \\ c_{false} & B \neq \emptyset \end{cases}}{\Pi[p : M, B] \overset{p:\langle empty\rangle}{\rightarrow} \Pi[p : (N\ V), B]}$$

$$\frac{M \overset{\beta}{=} (c_{send}\ c_{p'}\ V\ N) \quad V \text{ a closed value}}{\Pi[p : M, B][p' : M', B'] \overset{p:\langle send\ p'\rangle}{\rightarrow} \Pi[p : (N\ V), B][p' : M', B' \cup \{\!\{V\}\!\}]}$$

$$\frac{M \overset{\beta}{=} (c_{spawn}\ N\ P) \quad p' \notin \text{dom}\ \Pi \cup \{p\}}{\Pi[p : M, B] \overset{p:\langle spawn\ p'\rangle}{\rightarrow} \Pi[p : (P\ c_{p'}), B][p' : (N\ c_{p'}), \emptyset]}$$

Figure 24: Original communication rules

The program transition relation $\rightarrow \subseteq$ Program $\times$ (PID $\times$ Ev) $\times$ Program has labels which consist of a PID and an event descriptor, the latter being given by the following grammar in which p denotes a PID:

$$\begin{array}{rcl} \text{Ev} & ::= & \langle spawn\ p\rangle \\ & | & \langle send\ p\rangle \\ & | & \langle receive\rangle \\ & | & \langle empty\rangle \end{array}$$

The rules for $\rightarrow$ are given in Figure 24.

Our goal is to have the processes constituting a concurrent program be as independent as possible, and for our operational semantics to reflect this. Our intuitive picture of concurrent computation is summarized in the following points:

- An individual process undergoes transitions, both as prescribed by its own program and due to the effects caused by other processes' transitions.
- A process transition may have effects outside the transitioning process.
- A program transition is derived from a subset of the possible transitions of its constituent processes.

As a first step, we reformulate the program transition rules of Figure 24 so that each program $\rightarrow$ transition is derived from an $\rightarrow_0$ transition of a constituent process, where $\rightarrow_0 \subseteq$ Process $\times$ Ev $\times$ Process. This is shown in Figure 25, in which each kind of process transition of is grouped with the corresponding program transition. For $\langle receive\rangle$ and $\langle empty\rangle$, the $\rightarrow$ rule simply says that an $\rightarrow_0$ transition is allowed in any "program context". $\langle send\ p\rangle$ is allowed only in contexts where p labels a process; it has the non-local effect of delivering a

$$\left[\begin{array}{c}\dfrac{M \overset{\beta}{=} (c_{receive}\ N) \quad V \in B}{M, B \overset{\langle receive \rangle}{\longrightarrow_0} (N\ V), B - \{\!\{V\}\!\}} \\[2em] \dfrac{M, B \overset{\langle receive \rangle}{\longrightarrow_0} (N\ V), B - \{\!\{V\}\!\}}{\Pi[p : M, B] \overset{p:\langle receive \rangle}{\longrightarrow} \Pi[p : (N\ V), B - \{\!\{V\}\!\}]}\end{array}\right]$$

$$\left[\begin{array}{c}\dfrac{M \overset{\beta}{=} (c_{empty}\ N) \quad V = \begin{cases} c_{true} & B = \emptyset \\ c_{false} & B \neq \emptyset \end{cases}}{M, B \overset{p:\langle empty \rangle}{\longrightarrow_0} (N\ V), B} \\[2em] \dfrac{M, B \overset{\langle empty \rangle}{\longrightarrow_0} (N\ V), B}{\Pi[p : M, B] \overset{p:\langle empty\ p \rangle}{\longrightarrow} \Pi[p : (N\ V), B]}\end{array}\right]$$

$$\left[\begin{array}{c}\dfrac{M \overset{\beta}{=} (c_{send}\ c_{p'}\ V\ N) \quad V \text{ a closed value}}{M, B \overset{\langle send\ p' \rangle}{\longrightarrow_0} (N\ V)} \\[2em] \dfrac{M, B \overset{\langle send\ p' \rangle}{\longrightarrow_0} (N\ V)}{\Pi[p : M, B][p' : M', B'] \overset{p:\langle send\ p' \rangle}{\longrightarrow} \Pi[p : (N\ V), B][p' : M', B' \cup \{\!\{V\}\!\}]}\end{array}\right]$$

$$\left[\begin{array}{c}\dfrac{M \overset{\beta}{=} (c_{spawn}\ N\ P)}{M, B \overset{\langle spawn\ p' \rangle}{\longrightarrow_0} (P\ c_{p'}), B} \\[2em] \dfrac{M, B \overset{\langle spawn\ p' \rangle}{\longrightarrow_0} (P\ c_{p'}), B \quad p' \notin \mathrm{dom}\ \Pi \cup \{p\}}{\Pi[p : M, B] \overset{p:\langle spawn\ p' \rangle}{\longrightarrow} \Pi[p : (P\ c_{p'}), B][p' : (N\ c_{p'}), \emptyset]}\end{array}\right]$$

Figure 25: Original communication rules derived from process rules

message to the recipient. $\langle spawn\ p\rangle$ is not allowed in contexts where p labels a process; it has the non-local effect of creating a new process.

Having split out the process transitions from the resulting interleaving program transitions, we are in a position to re-use the process transitions to formulate a true-concurrency program transition relation. First we need some notation.

Convention 3 (Projections) The function $(x_1 \ldots x_n) \mapsto x_k$ is denoted by π_k.

Thus if we have some transition $X \xrightarrow{l} Y$, then π_1, π_2, and π_3 would pick out X, l, and Y, respectively.

The function *im* picks out the possible transitions of a process:

Definition 9 (im) $im\ X = \{t \in \rightarrow_0 \mid \pi_1(t) = X\}$

Our fundamental notian of true concurrency is that each program step results from one or more processes taking a step, so:

Definition 10 (Step) A step for a program Π is a nonempty partial function from dom Π to $\rightarrow_0$.

For a step to make sense for a given program, the step should only specify transitions possible for the program's processes, and forks should not collide with pre-existing processes or each other. Formally:

Definition 11 (Enabling) A step S is enabled for a program Π when

$$S(p) \in im\ \Pi(p)$$

$$\text{If } \pi_2(S(p)) = \langle spawn\ p'\rangle \text{ then } p' \notin \text{dom } \Pi$$

$$\text{If } p \neq q,\ \pi_2(S(p)) = \langle spawn\ p'\rangle \text{ and } \pi_2(S(q)) = \langle spawn\ q'\rangle, \text{ then } p' \neq q'$$

We write $\Pi[S\rangle$.

Definition 12 (Program transition) Let an enabled step S for a program Π be given. For concreteness write $\Pi(p_i) = M_i, B_i$, and for $p_i \in$ dom S, $S(p_i) = M_i, B_i \xrightarrow{e_i}_0 M_i', B_i'$. Let Π' be defined by

$$\begin{array}{rcl}
receipts(p) & = & \{\!\{V \mid \exists p_i : e_i = \langle send\ p\rangle \text{ and } M_i = (c_{send}\ c_p\ V\ N)\}\!\} \\
forks & = & \{q \mid \exists p_i : e_i = \langle spawn\ q\rangle\} \\
\text{dom } \Pi' & = & \text{dom } \Pi \cup forks \\
\Pi'(p_i) & = & \begin{cases} M_i, (B_i \cup receipts(p_i)) & \text{if } p_i \in \text{dom } \Pi - \text{dom } S \\ M_i', (B_i' \cup receipts(p_i)) & \text{if } p_i \in \text{dom } S \\ (V\ c_{p_i}), \emptyset & \text{if } p_i \in forks, \text{ so } \exists p_j \text{ with} \\ & \quad e_j = \langle spawn\ p_i\rangle \text{ and} \\ & \quad M_j = (c_{spawn}\ V\ N) \end{cases}
\end{array}$$

Then we write $\Pi[S\rangle\Pi'$.

The $[\ \rangle$ notation and the names of the corresponding concepts are taken from [Vog92], where they are applied to the homologous constructs of Petri nets.

8.2 Original Implementation Sketch

We extend the usual sequential 4-register machine [Oli93, WO92] with a buffer register, to get a 5-register machine. The 5-register machine's transition relation $\Rightarrow_1$ lifts the 4-register machine's transition relation, $\Rightarrow_0$, and provides potential communication transitions:

$$\frac{\pi u \zeta \kappa \Rightarrow_0 \pi' u' \zeta' \kappa'}{\pi u \zeta \kappa B \overset{\tau}{\Rightarrow}_1 \pi' u' \zeta' \kappa' B'}$$

$$(\mathsf{empty}\ \pi) u \zeta \kappa B \overset{\langle empty \rangle}{\Rightarrow_1} \pi u (B = \emptyset :: \zeta) \kappa B$$

$$\frac{v \in B}{(\mathsf{receive}\ \pi) u \zeta \kappa B \overset{\langle receive \rangle}{\Rightarrow_1} \pi u (v :: \zeta) \kappa (B - \{v\})}$$

$$(\mathsf{fork}\ \pi) u (\langle \mathsf{cl}\ \pi'\ u' \rangle :: \zeta) \kappa B \overset{\langle spawn\ p' \rangle}{\Rightarrow_1} \pi u (\langle \mathsf{pid}\ p' \rangle :: \zeta) \kappa B$$

$$(\mathsf{send}\ \pi) u (v :: \langle \mathsf{pid}\ p' \rangle :: \zeta) \kappa B \overset{\langle send\ p' \rangle}{\Rightarrow_1} \pi u (v :: \zeta) \kappa B$$

A concurrent machine is simply a finite map from PIDs to 5-register machines. The definitions of step and enabling are the same as in section 8.1, except of course that the relation in question is $\Rightarrow_1$, the transition relation of 5-register machines. The analog of the program transition relation is different, but only in the details:

Definition 13 (Machine transition) Let an enabled step S for a machine Π be given. For concreteness write $\Pi(p_i) = \pi_i, u_i, \zeta_i, \kappa_i, B_i$, and for each $p_i \in \mathrm{dom}\ S$, write $S(p_i) = \pi_i, u_i, \zeta_i, \kappa_i, B_i \overset{e_i}{\Rightarrow}_1 \pi_i', u_i', \zeta_i', \kappa_i', B_i'$. Let Π' be defined by

$$\begin{array}{rcl}
receipts(p) & = & \{\!\{ v \mid \exists p_i : e_i = \langle send\ p \rangle \text{ and } \zeta_i = v :: \langle \mathsf{pid}\ p \rangle :: \zeta \}\!\} \\
forks & = & \{ q \mid \exists e_i = \langle spawn\ q \rangle \} \\
\mathrm{dom}\ \Pi' & = & \mathrm{dom}\ \Pi \cup forks \\
\Pi'(p_i) & = & \begin{cases}
\pi_i, u_i, \zeta_i, \kappa_i, (B_i \cup receipts(p_i)) & \text{if } p_i \in \mathrm{dom}\ \Pi - \mathrm{dom}\ S \\
\pi_i', u_i', \zeta_i', \kappa_i', (B_i' \cup receipts(p_i)) & \text{if } p_i \in \mathrm{dom}\ S \\
(\mathsf{call\ ret}), & \text{if } p_i \in forks, \text{ so } \exists p_j : \\
\quad \mathsf{emptyenv}, & \quad e_j = \langle spawn\ p_i \rangle \text{ and} \\
\quad (\langle \mathsf{cl}\ \pi\ u \rangle :: \langle \mathsf{pid}\ p_i \rangle :: \langle\rangle), & \quad \zeta_j = \langle \mathsf{cl}\ \pi\ u \rangle :: \zeta \\
\quad \mathsf{initcont}, & \\
\quad \emptyset &
\end{cases}
\end{array}$$

Then we write $\Pi[S\rangle\Pi'$.

8.3 Discussion

The fundamental problem with this model of concurrent computation is that transitions are used to perform durational actions. Notice that we had to take great care to explain the interaction of some simultaneous transitions, in particular we had to decide what should happen when process p_i sends a message to process p_j when process p_j simultaneously takes a step. This is fine as far as it goes, but problems arise as soon as we try to implement it.

Suppose we want to implement a high-level simulator for CLAM. In a multiprocessor implementation, the reductions of the independently-running individual processes would not be synchronized. Thus we immediately have a problem in relating the behavior of the collection of process simulators to the behavior defined by the CLAM semantics: the semantics says how transitions interact if they are synchronized, and it says how they don't interact if they happen on different steps, but the semantics is silent on the issue of what happens if they overlap partially in time.

Similar problems arise if we implement the CLAM processes by lower level abstract machines. Then we would have to consider the interactions of the abstract machine instructions, and again, a multiprocessor implementation would not syncronize the abstract machine steps, so having a semantics which describes the synchronized execution of the machine instructions wouldn't help.

Suppose we were to try to forge ahead anyway, and figure out the desired behavior for a pair of overlapping transitions. We would have to contemplate the microstructure of each transition, asking the question: if at some instant transition 1 is *this* far along, and transition 2 is *that* far along, then what happens? We have thus been led to model the progress of durational transitions as a sequence of states. As long as we are going to have to do this anyway, we might as well reify these states in the operational semantics; we call them active states (as opposed to idle states) to indicate that they correspond to activity (which might otherwise be called a transition). A transition in such a semantics is instantaneous, and merely denotes the fact that a particular state may follow another given state. When we specify the instantaneous state transitions of the system, we have available complete information about the state of the system, including those parts engaged in activity. Since activities are represented as first class entities in the system state, it is much less likely that we will make errors in specifying transitions due to forgetting about possible interfering activities.

We now briefly relate an embarrassing experience we had with the initial version of our Urlang implementation based on the durational transition semantics described previously. In Section 8.4 we give the details of Concert/C's RPC mechanisms; suffice it to say that they are much easier to use than explicit message passing, and we succumbed to temptation. Now it is well known that because RPC calls are implemented as a send followed by a receive, two processes making RPC calls to each other are subject to deadly embrace: each may block forever waiting for the other to respond to the RPC request. We overlooked this possibility in our desire to think of the RPC calls as atomic transitions, despite knowing that they would surely take nonzero time: if one does not think of the durational transitions as being essentially instantaneous, and hence incapable of being interfered with, then one is faced with the harsh fact that the interleaving or even true concurrency operational semantics does not provide a justification for the unsynchronized multiprocessor implementation.

We cite this case as an example of the dangers inherent in the schizophrenic thinking engendered by the durational transition model. The fact that the RPC problem is well known is irrelevant, since there is no shortage of situations in which similar problems could arise.[1]

8.4 RPC in Concert/C

It was remarked in Section 6.2.2 that the initial port of a process will often have a rather complicated type, because of the necessity of the parent and child exchanging sufficient ports for their later communication. Concert/C ameliorates this difficulty somewhat by providing a remote procedure call facility. The `port` declarator can be used with any function definition, after which the function's address can (apparently) be passed to other Concert/C processes where it can be used in a normal function call. The key simplification from the user's point of view is that he doesn't have to mention explicit `receiveport` types, but only function types. What is actually going on is that Concert/C creates two hidden `receiveport` types, one corresponding to the function's input argument tuple, and one corresponding to the function's result type. When a user of the RPC function makes a call, in reality a message containing the arguments is sent to the process defining the function, the function is called, and then a message containing the result is sent back to the remote caller.

Let us consider a simple example. If we wanted to write an add-one server, we might write the following function definition:

```
initial port
int increment (int i) {
  return (i + 1);
}
```

Notice that all we had to do to make the function an RPC function was to annotate the function with `initial port`. However, to allow other processes to call this function, we need to use the Concert/C `accept` construct to effect the 'receive request/call function/return result' cycle:

```
main () {
  accept (increment);
}
```

A parent might spawn and use such a server as follows using the `create` construct to obtain the initial port of the child. Notice that here also, there are no `receiveports`:

```
void use_incrementer ()
{
  program child;
  int (*remote_increment) (int i);
```

[1] In fact, we have designed and implemented RPC facilities for several computer systems, so we cannot claim ignorance of the mechanisms involved. The fact that we managed to avoid thinking about them for so long indicates the seriousness of the cognitive dissonance.

```
  int answer;

  child = cn_prog_init ();
  cn_prog_set_name (child, "/executable/name/goes/here");
  create (child, &remote_increment);

  answer = (*remote_increment) (42);
}
```

Obviously this is a lot easier than the setup of our bytecode interpreter, where we have to go to a lot of trouble to be able to communicate ports explicitly. As we mentioned in the previous section, the initial Urlang implementation used RPC instead of explicit message passing: instead of sending a command to another process which would then call `process_command`, the requester would simply call the `process_command` procedure remotely.

Appendix A

The Compiler Front End

The compiler consists of a front end, implemented using lex and yacc, and a hand-written bytecode generator. The parser returns an abstract syntax tree, encoded as an abstract data type, for use by the code generator.

A.1 Source Terms

The following encoding of the source term data type is typical. It consists of an enumeration type, listing each possible variant of the data type, and a structure to hold the discriminator and the actual data:

```
typedef enum  {INT_CONSTANT, BOOL_CONSTANT, NIL_CONSTANT, VARIABLE, PROCEDURE,
                 APPLICATION, IF_THEN_ELSE, UNARY_OP, BINARY_OP,
                 GET, EMPTY, PUT, SPAWN,
                 LETFORM, TERM_LIST, EMPTY_TERM_LIST, BINDING, PROGNFORM}
source_term_code;

typedef struct source_term_struct {
  source_term_code source_term_code;
  int line_number;
  union {
    int integer_constant;
    int boolean_constant;
    string variable;
    struct {string procname;
            string argname;
            struct source_term_struct *body;
           } procedure;
    struct {struct source_term_struct *function;
            struct source_term_struct *argument;
           } application;
    struct {struct source_term_struct *test;
            struct source_term_struct *true_part;
            struct source_term_struct *false_part;
```

```
          } if_then_else;
    struct {ident operator;
            struct source_term_struct *argument;
          } unary_op;
    struct {ident operator;
            struct source_term_struct *first_argument;
            struct source_term_struct *second_argument;
          } binary_op;
    struct {struct source_term_struct *recipient;
            struct source_term_struct *message;
          } put;
    struct source_term_struct *spawn;
    struct {struct source_term_struct *binding_list;
            struct source_term_struct *body;
          } let;
    struct source_term_struct *progn;
    struct {struct source_term_struct *term;
            struct source_term_struct *terms;
    } term_list;
    struct {string var;
            struct source_term_struct *val;
          } binding;
 } guts;
} source_term_struct;
typedef source_term_struct *source_term;
```

In addition to this data type, we introduce an abbreviation for character strings, and fix an implementation of identifiers and lexical addresses:

```
typedef char *string;
typedef int lex;
typedef string ident;
```

With a few utilities to aid in making the source term data structures, we will be able to move on to the compiler itself.

```
#define make_a(foo) ((foo *) (safe_malloc (sizeof (foo))))



source_term Source_term (source_term_code s) {
  extern int line_number;

  source_term st = make_a (source_term_struct);
  st->source_term_code = s;
  st->line_number = line_number;
  return (st);
}
```

A.2 Parser and Lexical Analyzer

The yacc parser is straightforward, except for a few items of note:

- Top-level programs are syntactically abstractions. This is an aid to the programmer's memory.
- The top-level syntax node is stored in a global, `the_source_program`.
- `let` and `progn` have been added to the source language, as conveniences to the programmer. They are implemented as macros.
- The representations of binding lists and expression lists in `let` and `progn` are constructed reversed from left to right; in the case of the binding list, the parser explicitly reverses the list before returning it.

```
%{
#define YYSTYPE source_term
extern char *yytext;
extern source_term the_source_program;
extern int line_number;
extern int has_result;
%}
%token  LPAREN RPAREN LBRACE RBRACE
%token  NUMBER IDENTIFIER
%token  TRUE FALSE NIL
%token  PLUS MINUS TIMES DIV EQUALS MOD GREATER
%token  ZEROP PRINT RESULT_IS CONS CAR CDR NULLP
%token  REC IF
%token  SEND_TOK GET_TOK BUFEMPTY_TOK FORK_TOK
%token  LET PROGN
%token  LISTP PIDP CLOSUREP BOOLEANP INTEGERP

%%

program :       procedure {
                the_source_program = $1;
                if (! has_result)
                  fprintf (stderr, "Warning: no result_is anywhere.\n");
              }

expr    :       constant
        |       variable
        |       procedure
        |       LPAREN expr expr RPAREN {
                $$ = Source_term (APPLICATION);
                $$->guts.application.function = $2;
                $$->guts.application.argument = $3;
              }
```

```
        |       LPAREN IF expr expr expr RPAREN {
                $$ = Source_term (IF_THEN_ELSE);
                $$->guts.if_then_else.test = $3;
                $$->guts.if_then_else.true_part = $4;
                $$->guts.if_then_else.false_part = $5;
              }
        |       LPAREN unary_op expr RPAREN {
                $$ = $2;
                $$->guts.unary_op.argument = $3;
              }
        |       LPAREN binary_op expr expr RPAREN {
                $$ = $2;
                $$->guts.binary_op.first_argument = $3;
                $$->guts.binary_op.second_argument = $4;
              }
        |       LPAREN GET_TOK RPAREN {$$ = Source_term (GET);}
        |       LPAREN BUFEMPTY_TOK RPAREN  {$$ = Source_term (EMPTY);}
        |       LPAREN SEND_TOK expr expr RPAREN {
                $$ = Source_term (PUT);
                $$->guts.put.recipient = $3;
                $$->guts.put.message = $4;
              }
        |       LPAREN FORK_TOK expr RPAREN {
                $$ = Source_term (SPAWN);
                $$->guts.spawn = $3;
              }
        |       LPAREN LET LPAREN binding_list RPAREN expr RPAREN {
                $$ = Source_term (LETFORM);
                $$->guts.let.binding_list = reverse_source_term_list ($4);
                $$->guts.let.body = $6;
              }
        |       LPAREN PROGN expr_list RPAREN {
                $$ = Source_term (PROGNFORM);
                $$->guts.progn = $3;
              }

procedure:      LPAREN REC variable LPAREN variable RPAREN expr RPAREN {
                $$ = Source_term (PROCEDURE);
                $$->guts.procedure.procname = $3->guts.variable;
                $$->guts.procedure.argname = $5->guts.variable;
                $$->guts.procedure.body = $7;
              }

binding_list:   {$$ = Source_term (EMPTY_TERM_LIST);}
        |       binding_list binding {
                $$ = Source_term (TERM_LIST);
                $$->guts.term_list.terms = $1;
                $$->guts.term_list.term = $2;
```

```
            }

binding :     LPAREN variable expr RPAREN {
              $$ = Source_term (BINDING);
              $$->guts.binding.var = $2->guts.variable;
              $$->guts.binding.val = $3;
            }

expr_list:    {$$ = Source_term (EMPTY_TERM_LIST);}
        |     expr_list expr {
              $$ = Source_term (TERM_LIST);
              $$->guts.term_list.terms = $1;
              $$->guts.term_list.term = $2;
            }

constant:     TRUE           {$$ = Source_term (BOOL_CONSTANT);
                              $$->guts.boolean_constant = 1;}
        |     FALSE          {$$ = Source_term (BOOL_CONSTANT);
                              $$->guts.boolean_constant = 0;}
        |     NIL            {$$ = Source_term (NIL_CONSTANT);}
        |     NUMBER         {$$ = Source_term (INT_CONSTANT);
                              sscanf (yytext, "%d",
                                      &($$->guts.integer_constant));}

variable:     IDENTIFIER     {$$ = Source_term (VARIABLE);
                              fflush (stderr);
                              $$->guts.variable = strdup (yytext);}

unary_op:     ZEROP          {$$ = Source_term (UNARY_OP);
                              $$->guts.unary_op.operator = strdup (yytext);}
        |     PRINT          {$$ = Source_term (UNARY_OP);
                              $$->guts.unary_op.operator = strdup (yytext);}
        |     RESULT_IS      {$$ = Source_term (UNARY_OP);
                              $$->guts.unary_op.operator = strdup (yytext);}
        |     CAR            {$$ = Source_term (UNARY_OP);
                              $$->guts.unary_op.operator = strdup (yytext);}
        |     CDR            {$$ = Source_term (UNARY_OP);
                              $$->guts.unary_op.operator = strdup (yytext);}
        |     NULLP          {$$ = Source_term (UNARY_OP);
                              $$->guts.unary_op.operator = strdup (yytext);}
        |     INTEGERP       {$$ = Source_term (UNARY_OP);
                              $$->guts.unary_op.operator = strdup (yytext);}
        |     BOOLEANP       {$$ = Source_term (UNARY_OP);
                              $$->guts.unary_op.operator = strdup (yytext);}
        |     CLOSUREP       {$$ = Source_term (UNARY_OP);
                              $$->guts.unary_op.operator = strdup (yytext);}
        |     LISTP          {$$ = Source_term (UNARY_OP);
                              $$->guts.unary_op.operator = strdup (yytext);}
```

```
        |       PIDP            {$$ = Source_term (UNARY_OP);
                                 $$->guts.unary_op.operator = strdup (yytext);}

binary_op:      PLUS            {$$ = Source_term (BINARY_OP);
                                 $$->guts.binary_op.operator = strdup (yytext);}
        |       MINUS           {$$ = Source_term (BINARY_OP);
                                 $$->guts.binary_op.operator = strdup (yytext);}
        |       TIMES           {$$ = Source_term (BINARY_OP);
                                 $$->guts.binary_op.operator = strdup (yytext);}
        |       DIV             {$$ = Source_term (BINARY_OP);
                                 $$->guts.binary_op.operator = strdup (yytext);}
        |       EQUALS          {$$ = Source_term (BINARY_OP);
                                 $$->guts.binary_op.operator = strdup (yytext);}
        |       MOD             {$$ = Source_term (BINARY_OP);
                                 $$->guts.binary_op.operator = strdup (yytext);}
        |       GREATER         {$$ = Source_term (BINARY_OP);
                                 $$->guts.binary_op.operator = strdup (yytext);}
        |       CONS            {$$ = Source_term (BINARY_OP);
                                 $$->guts.binary_op.operator = strdup (yytext);}
%%

yyerror (string s) {fprintf (stderr, "Parse error: %s near line %d\n",
                             s, line_number);}


source_term reverse_source_term_list (source_term s) {
  source_term t, u;

  t = Source_term (EMPTY_TERM_LIST);
  while (s->source_term_code != EMPTY_TERM_LIST) {
    assert (s->source_term_code == TERM_LIST);
    u = Source_term (TERM_LIST);
    u->guts.term_list.terms = t;
    u->guts.term_list.term = s->guts.term_list.term;
    t = u;
    s = s->guts.term_list.terms;
  }
  return (t);
}
```

For completeness we show the lexical analyzer, although it contains no surprises.

```
%{
int line_number = 1;
int has_result = 0;
%}

alpha           [a-zA-Z0-9~!@#$%^&*_+=/?\:<>_-]
digit           [0-9]
```

```
whitespace      [ \t]
newline         [\n]

%%
"("             return (LPAREN);
")"             return (RPAREN);
"{"             return (LBRACE);
"}"             return (RBRACE);
"True"          return (TRUE);
"False"         return (FALSE);
"boolean?"      return (BOOLEANP);
"integer?"      return (INTEGERP);
"+"             return (PLUS);
"-"             return (MINUS);
"*"             return (TIMES);
"/"             return (DIV);
"="             return (EQUALS);
"%"             return (MOD);
">"             return (GREATER);
"0?"            return (ZEROP);
"print"         return (PRINT);
"nil"           return (NIL);
"cons"          return (CONS);
"car"           return (CAR);
"cdr"           return (CDR);
"list?"         return (LISTP);
"null?"         return (NULLP);
"PID?"          return (PIDP);
"closure?"      return (CLOSUREP);
"result_is"     {has_result = 1; return (RESULT_IS);}
"rec"           return (REC);
"if"            return (IF);
"get"           return (GET_TOK);
"empty"         return (BUFEMPTY_TOK);
"put"           return (SEND_TOK);
"spawn"         return (FORK_TOK);
"let"           return (LET);
"progn"         return (PROGN);
"begin"         return (PROGN);
{digit}+        return (NUMBER);
{alpha}+        return (IDENTIFIER);
{whitespace}    {}
{newline}       {line_number++;}
";".* {}
.               {fprintf (stderr,
                          "lexer: bad character `%c' (0x%x) in line %d\n",
                          yytext [0], yytext[0], line_number);
                 exit (-1);}
```

%%

A.3 Macroexpansion

As mentioned previously, the source language has been extended with `let` and `progn` forms, which are to be macroexpanded out. Since these are the only two macros, macroexpanding a form generally consists of macroexpanding a form's constituent subexpressions, if any. We therefore present the function `macroexpand` without comment, until the clauses for `let` and `progn`.

```
source_term macroexpand (source_term s) {
  source_term t;
  switch (s->source_term_code) {
  case INT_CONSTANT:
    return (s);
  case BOOL_CONSTANT:
    return (s);
  case NIL_CONSTANT:
    return (s);
  case VARIABLE:
    return (s);
  case PROCEDURE:
    t = Source_term (PROCEDURE);
    t->guts.procedure.procname = s->guts.procedure.procname;
    t->guts.procedure.argname = s->guts.procedure.argname;
    t->guts.procedure.body = macroexpand (s->guts.procedure.body);
    return (t);
  case APPLICATION:
    t = Source_term (APPLICATION);
    t->guts.application.function = macroexpand (s->guts.application.function);
    t->guts.application.argument = macroexpand (s->guts.application.argument);
    return (t);
  case IF_THEN_ELSE:
    t = Source_term (IF_THEN_ELSE);
    t->guts.if_then_else.test = macroexpand (s->guts.if_then_else.test);
    t->guts.if_then_else.true_part = macroexpand (s->guts.if_then_else.true_part);
    t->guts.if_then_else.false_part =
      macroexpand (s->guts.if_then_else.false_part);
    return (t);
  case UNARY_OP:
    t = Source_term (UNARY_OP);
    t->guts.unary_op.operator = s->guts.unary_op.operator;
    t->guts.unary_op.argument = macroexpand (s->guts.unary_op.argument);
    return (t);
  case BINARY_OP:
    t = Source_term (BINARY_OP);
    t->guts.binary_op.operator = s->guts.binary_op.operator;
    t->guts.binary_op.first_argument =
```

```
      macroexpand (s->guts.binary_op.first_argument);
    t->guts.binary_op.second_argument =
      macroexpand (s->guts.binary_op.second_argument);
    return (t);
  case GET:
    return (s);
  case EMPTY:
    return (s);
  case PUT:
    t = Source_term (PUT);
    t->guts.put.recipient = macroexpand (s->guts.put.recipient);
    t->guts.put.message = macroexpand (s->guts.put.message);
    return (t);
  case SPAWN:
    t = Source_term (SPAWN);
    t->guts.spawn = macroexpand (s->guts.spawn);
    return (t);
```

The `let` special form is implemented by the following transformation rules:

$$
\begin{array}{l}
\texttt{(let () } \mathit{body}\texttt{)} \Rightarrow \mathit{body} \\
\texttt{(let ((}v_1\ e_1\texttt{)...(}v_n\ e_n\texttt{)) } \mathit{body}\texttt{)} \Rightarrow \texttt{(let ((}v_2\ e_2\texttt{)...(}v_n\ e_n\texttt{))} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{((rec } \epsilon\texttt{(}v_1\texttt{)} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{body}\texttt{)} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ e_1\texttt{))}
\end{array}
$$

The construction `(rec` ϵ`(x)...)` denotes a recursive procedure whose name is the empty string. Since such a name is not expressible in the source language, this construct can be used to represent anonymous procedures without fear of variable capture. The rules are implemented in the following code. Note that parser explicitly reversed the binding list from the order in which it was constructed, so that the first binding in the list structure is the leftmost one.

```
  case LETFORM: {
    source_term_struct
      *binding_list = s->guts.let.binding_list,
      *body = s->guts.let.body;
    if (binding_list->source_term_code == EMPTY_TERM_LIST)
      return (macroexpand (body));
    else {
      source_term_struct
        *fun = Source_term (PROCEDURE),
        *app = Source_term (APPLICATION),
        *let = Source_term (LETFORM);

      assert (binding_list->source_term_code == TERM_LIST);
      assert (binding_list->guts.term_list.term->source_term_code == BINDING);
      let->guts.let.binding_list = binding_list->guts.term_list.terms;
      let->guts.let.body = body;
```

```
      fun->guts.procedure.procname = "";
      fun->guts.procedure.argname =
        binding_list->guts.term_list.term->guts.binding.var;
      fun->guts.procedure.body = macroexpand (let);
      app->guts.application.function = fun;
      app->guts.application.argument =
        macroexpand (binding_list->guts.term_list.term->guts.binding.val);
      return (app);
    }
  }
```

The progn form is implemented by turning it into a let form, according to the following transformation rule, in which we require $n \geq 1$:

(progn $e_1 \ldots e_n$) $\Rightarrow$ (let ((ϵ e_1)...(ϵ e_n)) ϵ)

```
  case PROGNFORM:
    if (s->guts.progn->source_term_code == EMPTY_TERM_LIST) {
      fprintf (stderr, "macroexpand: empty progn form\n");
      exit (-1);
    } else {
      source_term_struct
        *forms = s->guts.progn,
        *new_bindings = Source_term (EMPTY_TERM_LIST),
        *let = Source_term (LETFORM),
        *body = Source_term (VARIABLE),
        *t;

      body->guts.variable = " ";
      let->guts.let.body = body;
      while (forms->source_term_code == TERM_LIST) {
        t = Source_term (TERM_LIST);
        t->guts.term_list.terms = new_bindings;
        t->guts.term_list.term = Source_term (BINDING);
        t->guts.term_list.term->guts.binding.var = " ";
        t->guts.term_list.term->guts.binding.val = forms->guts.term_list.term;
        new_bindings = t;
        forms = forms->guts.term_list.terms;
      }
      let->guts.let.binding_list = new_bindings;
      return (macroexpand (let));
  }
  default:
    fprintf (stderr, "Cannot macroexpand term code %d\n", s->source_term_code);
    exit (-1);
  }
}
```

Note that the order of the progn clauses was deliberately left reversed by the parser, since that made it easier to construct the let expression as above.

Appendix B

Utilities

B.1 Pretty Printing

In order to debug the parser and macroexpander, one needs to be able to print out the internal form of the source code, either before or after macroexpansion.

```
void print_source_term (FILE *f, int indentation, source_term s) {
  switch (s->source_term_code) {
  case INT_CONSTANT:
    indent (f, indentation);
    fprintf (f, "%d\n", s->guts.integer_constant);
    break;
  case BOOL_CONSTANT:
    indent (f, indentation);
    fprintf (f, "%s\n", s->guts.boolean_constant? "True": "False");
    break;
  case NIL_CONSTANT:
    indent (f, indentation);
    fprintf (f, "nil\n");
    break;
  case VARIABLE:
    indent (f, indentation);
    fprintf (f, "%s\n", s->guts.variable);
    break;
  case PROCEDURE:
    indent (f, indentation);
    fprintf (f, "(rec %s (%s)\n",
             s->guts.procedure.procname, s->guts.procedure.argname);
    print_source_term (f, indentation + 2, s->guts.procedure.body);
    indent (f, indentation + 2);
    fprintf (f, ")\n");
    break;
  case APPLICATION:
    indent (f, indentation);
    fprintf (f, "(\n");
```

```
    print_source_term (f, indentation + 2, s->guts.application.function);
    print_source_term (f, indentation + 2, s->guts.application.argument);
    indent (f, indentation + 2);
    fprintf (f, ")\n");
    break;
  case IF_THEN_ELSE:
    indent (f, indentation);
    fprintf (f, "(if\n");
    print_source_term (f, indentation + 2, s->guts.if_then_else.test);
    print_source_term (f, indentation + 2, s->guts.if_then_else.true_part);
    print_source_term (f, indentation + 2, s->guts.if_then_else.false_part);
    indent (f, indentation + 2);
    fprintf (f, ")\n");
    break;
  case UNARY_OP:
    indent (f, indentation);
    fprintf (f, "(%s\n", s->guts.unary_op.operator);
    print_source_term (f, indentation + 2, s->guts.unary_op.argument);
    indent (f, indentation + 2);
    fprintf (f, ")\n");
    break;
  case BINARY_OP:
    indent (f, indentation);
    fprintf (f, "(%s\n", s->guts.binary_op.operator);
    print_source_term (f, indentation + 2, s->guts.binary_op.first_argument);
    print_source_term (f, indentation + 2, s->guts.binary_op.second_argument);
    indent (f, indentation + 2);
    fprintf (f, ")\n");
    break;
  case GET:
    indent (f, indentation);
    fprintf (f, "(get)\n");
    break;
  case EMPTY:
    indent (f, indentation);
    fprintf (f, "(bufempty)\n");
    break;
  case PUT:
    indent (f, indentation);
    fprintf (f, "(send\n");
    print_source_term (f, indentation + 2, s->guts.put.recipient);
    print_source_term (f, indentation + 2, s->guts.put.message);
    indent (f, indentation + 2);
    fprintf (f, ")\n");
    break;
  case SPAWN:
    indent (f, indentation);
    fprintf (f, "(fork\n");
```

```
    print_source_term (f, indentation + 2, s->guts.spawn);
    indent (f, indentation + 2);
    fprintf (f, ")\n");
    break;
  case LETFORM:
    indent (f, indentation);
    fprintf (f, "(let (\n");
    print_source_term (f, indentation + 7, s->guts.let.binding_list);
    indent (f, indentation + 6);
    fprintf (f, ")\n");
    print_source_term (f, indentation + 2, s->guts.let.body);
    indent (f, indentation + 2);
    fprintf (f, ")\n");
    break;
  case PROGNFORM:
    indent (f, indentation);
    fprintf (f, "(begin\n");
    print_source_term (f, indentation + 2,
                       reverse_source_term_list (s->guts.progn));
    indent (f, indentation + 2);
    fprintf (f, ")\n");
    break;
  case TERM_LIST:
    print_source_term (f, indentation, s->guts.term_list.term);
    print_source_term (f, indentation, s->guts.term_list.terms);
    break;
  case EMPTY_TERM_LIST:
    break;
  case BINDING:
    indent (f, indentation);
    fprintf (f, "(%s\n", s->guts.binding.var);
    print_source_term (f, indentation + 2, s->guts.binding.val);
    indent (f, indentation + 2);
    fprintf (f, ")\n");
    break;
  default:
    fprintf (f, "Bogus term code %d\n", s->source_term_code);
    break;
  }
}
```

B.2 Instruction Constructors

```
inst Pushconstant (value c,
                       inst tail) {
  inst p = make_a (inst_struct);
  p->instruction_code = pushconstant;
  p->guts.pushconstant.c = c;
```

```
  p->guts.pushconstant.tail = tail;
  return (p);
}
inst Op2 (ident id,
                  inst tail) {
  inst p = make_a (inst_struct);
  p->instruction_code = op2;
  p->guts.op2.id = id;
  p->guts.op2.tail = tail;
  return (p);
}
inst Swap (inst tail) {
  inst p = make_a (inst_struct);
  p->instruction_code = swap;
  p->guts.swap.tail = tail;
  return (p);
}
inst Op1 (ident id,
                  inst tail) {
  inst p = make_a (inst_struct);
  p->instruction_code = op1;
  p->guts.op1.id = id;
  p->guts.op1.tail = tail;
  return (p);
}
inst Fetch (lex addr,
                    inst tail) {
  inst p = make_a (inst_struct);
  p->instruction_code = fetch;
  p->guts.fetch.addr = addr;
  p->guts.fetch.tail = tail;
  return (p);
}
inst Pushclosure (inst body,
                    inst tail) {
  inst p = make_a (inst_struct);
  p->instruction_code = pushclosure;
  p->guts.pushclosure.body = body;
  p->guts.pushclosure.tail = tail;
  return (p);
}
inst Call (inst tail) {
  inst p = make_a (inst_struct);
  p->instruction_code = call;
  p->guts.call.tail = tail;
  return (p);
}
```

```
inst Brtrue (inst true_inst,
                     inst false_inst) {
  inst p = make_a (inst_struct);
  p->instruction_code = brtrue;
  p->guts.brtrue.true_inst = true_inst;
  p->guts.brtrue.false_inst = false_inst;
  return (p);
}
inst Empty (inst tail) {
  inst p = make_a (inst_struct);
  p->instruction_code = empty;
  p->guts.empty.tail = tail;
  return (p);
}
inst Get (inst tail) {
  inst p = make_a (inst_struct);
  p->instruction_code = get;
  p->guts.get.tail = tail;
  return (p);
}
inst Put (inst tail) {
  inst p = make_a (inst_struct);
  p->instruction_code = put;
  p->guts.put.tail = tail;
  return (p);
}
inst Spawn (inst tail) {
  inst p = make_a (inst_struct);
  p->instruction_code = spawn;
  p->guts.spawn.tail = tail;
  return (p);
}
inst Ret () {
  inst p = make_a (inst_struct);
  p->instruction_code = ret;
  return (p);
}
```

B.3 General Utilities

```
void *safe_malloc (unsigned u) {
  void *p;

  p = malloc (u);
  assert (p);
  return (p);
}
```

```
void indent (FILE *f, int indentation) {
  while (indentation-- > 0)
    fprintf (f, " ");
}

cons Cons (value head,
           value tail) {
  cons p = make_a (cons_struct);
  p->head = head;
  p->tail = tail;
  return (p);
}

env Env (value val,
         env tail) {
  env p = make_a (env_struct);
  p->val = val;
  p->tail = tail;
  return (p);
}

stack Stack (value top,
             stack tail) {
  stack p = make_a (stack_struct);
  p->top = top;
  p->tail = tail;
  return (p);
}

command_port_stack Command_port_stack (command_port top,
                                       command_port_stack tail) {
  command_port_stack p = make_a (command_port_stack_struct);
  p->top = top;
  p->tail = tail;
  return (p);
}

buffer Empty_buffer () {
  buffer p = make_a (buffer_struct);
  p->head = EMPTYSTACK;
  p->tail = EMPTYSTACK;
  return (p);
}

buffer Enqueue (value item,
                buffer b) {
  buffer p = make_a (buffer_struct);
  p->head = b->head;
  p->tail = Stack (item, b->tail);
  return (p);
}
```

```
int buffer_is_empty (buffer b) {
  int i = b->head == EMPTYSTACK && b->tail == EMPTYSTACK;
  return (i);
}

void normalize_buffer (buffer b) {
  if (b->head == EMPTYSTACK) {
    while (b->tail != EMPTYSTACK) {
      b->head = Stack (b->tail->top, b->head);
      b->tail = b->tail->tail;
    }
  }
}

buffer Dequeue (buffer b) {
  buffer p = make_a (buffer_struct);

  if (buffer_is_empty (b)) {
    display ("queue underflow");
    fprintf (stderr, "Queue underflow");
    exit (-1);
  }
  normalize_buffer (b);
  p->head = b->head->tail;
  p->tail = b->tail;
  return (p);
}

value Head (buffer b) {
  if (buffer_is_empty (b)) {
    fprintf (stderr, "Queue underflow");
    exit (-1);
  };
  normalize_buffer (b);
  return (b->head->top);
}

frame Frame (inst pi,
             env u,
             stack z,
             frame k) {
  frame p = make_a (frame_struct);
  p->pi = pi;
  p->u = u;
  p->z = z;
  p->k = k;
  return (p);
}
```

```
void send_command (command_port p,
                   command com,
                   command_port_holder ack_port_struct) {
  com->ack_port_struct = ack_port_struct;
  send (p, *com);
}


command Boot (value boot_closure,
              print_string_port *result_port,
              print_string_port *chat_port,
              command_port_port_struct *process_registry_struct
              ) {
  command p = make_a (command_struct);
  p->command_code = boot;
  p->guts.boot.boot_closure = boot_closure;
  p->guts.boot.result_port = result_port;
  p->guts.boot.chat_port = chat_port;
  p->guts.boot.process_registry_struct = process_registry_struct;
  return (p);
}

command Start (inst main_program,
               print_string_port *result_port,
               print_string_port *chat_port,
               command_port_port_struct *process_registry_struct
               ) {
  command p = make_a (command_struct);
  p->command_code = start;
  p->guts.start.main_program = main_program;
  p->guts.start.result_port = result_port;
  p->guts.start.chat_port = chat_port;
  p->guts.start.process_registry_struct = process_registry_struct;
  return (p);
}

command Message (value msg) {
  command p = make_a (command_struct);
  p->command_code = message;
  p->guts.msg = msg;
  return (p);
}

command Shutdown () {
  command p = make_a (command_struct);
  p->command_code = shutdown;
  return (p);
}
```

```
command Ack () {
  command p = make_a (command_struct);
  p->command_code = ack;
  return (p);
}

value Unit () {
  value p = make_a (value_struct);
  p->value_code = unit;
  return (p);
}

value Int (int i) {
  value p = make_a (value_struct);
  p->value_code = integer;
  p->guts.integer = i;
  return (p);
}

value Bool (int b) {
  value p = make_a (value_struct);
  p->value_code = bool;
  p->guts.bool = b;
  return (p);
}

value Pid (command_port process_id) {
  value p = make_a (value_struct);
  p->value_code = pid;
  p->guts.pid = process_id;
  return (p);
}

value Closure (inst inst,
               env env) {
  value p = make_a (value_struct);
  p->value_code = closure;
  p->guts.closure.inst = inst;
  p->guts.closure.env = env;
  return (p);
}

value List (cons cell) {
  value p = make_a (value_struct);
  p->value_code = list;
  p->guts.list = cell;
  return (p);
}
```

```
state State (inst pi,
             env u,
             stack z,
             frame k,
             buffer b) {
  state p = make_a (state_struct);
  p->pi = pi;
  p->u = u;
  p->z = z;
  p->k = k;
  p->buffer = b;
  return (p);
}

void decode (value v, char *b, char *end) {
  if (b > end)
    *end = '\0';
  else switch (v->value_code) {
  case integer:
    sprintf (b, "%d", v->guts.integer);
    break;
  case bool:
    strcpy (b, ((v->guts.bool)? "True": "False"));
    break;
  case pid:
    strcpy (b, "<a PID>");
    break;
  case closure:
    strcpy (b, "<a closure>");
    break;
  case list: {
#define ZIPB {b += strlen (b); if (b > end) {*end = '\0'; return;}}
    strcpy (b, "(");
    ZIPB;
    while (v->value_code == list && v->guts.list != NIL) {
      decode (v->guts.list->head, b, end);
      ZIPB;
      v = v->guts.list->tail;
      if (v->value_code == list && v->guts.list != NIL) {
        strcpy (b, " ");
        ZIPB;
      }
    }
    if (v->value_code != list) {
      strcpy (b, " . ");
      ZIPB;
      decode (v, b, end);
      ZIPB;
    }
```

```
    strcpy (b, ")");
  }
    break;
  case unit:
    strcpy (b, "<unit>");
    break;
  default:
    strcpy (b, "<a bogus value>");
    }
  }

char *decode_value (value v) {
  static char b[10000];

  decode (v, b, b + sizeof (b) - 100);
  return (b);
}

int streq (string s1, string s2) {
  return (0 == strcmp (s1,s2));
}

void print_value (FILE *f, value v) {
  fprintf (f, "%s", decode_value (v));
}

void print_inst (FILE *f, int indentation, inst i) {
  indent (f, indentation);
  switch (i->instruction_code) {

  case pushconstant:
    fprintf (f, "pushconstant ");
    print_value (f, i->guts.pushconstant.c);
    fprintf (f, "\n");
    print_inst (f, indentation, i->guts.pushconstant.tail);
    break;
  case op2:
    fprintf (f, "op2 %s\n", i->guts.op2.id);
    print_inst (f, indentation, i->guts.op2.tail);
    break;
  case swap:
    fprintf (f, "swap\n");
    print_inst (f, indentation, i->guts.swap.tail);
    break;
  case op1:
    fprintf (f, "op1 %s\n", i->guts.op1.id);
    print_inst (f, indentation, i->guts.op1.tail);
    break;
  case fetch:
```

```
    fprintf (f, "fetch %d\n", i->guts.fetch.addr);
    print_inst (f, indentation, i->guts.fetch.tail);
    break;
  case pushclosure:
    fprintf (f, "pushclosure\n");
    print_inst (f, indentation + 2, i->guts.pushclosure.body);
    print_inst (f, indentation, i->guts.pushclosure.tail);
    break;
  case ret:
    fprintf (f, "ret\n");
    indent (f, indentation);
    fprintf (f, "-----\n");
    break;
  case call:
    fprintf (f, "call\n");
    print_inst (f, indentation, i->guts.call.tail);
    break;
  case brtrue:
    fprintf (f, "brtrue\n");
    print_inst (f, indentation + 2, i->guts.brtrue.true_inst);
    print_inst (f, indentation + 2, i->guts.brtrue.false_inst);
    break;
  case empty:
    fprintf (f, "empty\n");
    print_inst (f, indentation, i->guts.empty.tail);
    break;
  case get:
    fprintf (f, "get\n");
    print_inst (f, indentation, i->guts.get.tail);
    break;
  case put:
    fprintf (f, "put\n");
    print_inst (f, indentation, i->guts.put.tail);
    break;
  case spawn:
    fprintf (f, "spawn\n");
    print_inst (f, indentation, i->guts.spawn.tail);
    break;
  default:
    fprintf (f, "Unknown code %d\n", i->instruction_code);
  }
}


int
  PRINT_SOURCE = 0,
  PRINT_MACROEXPANDED_SOURCE = 0,
  PRINT_OBJECT = 0,
  PRINT_CHAT = 0,
```

```
  COMPILE_ONLY = 0;

void process_arguments (int argc, char *argv[]) {
  char *p;
  while (argc-- > 0) {
    p = argv[0] + (argv[0][0] == '-'? 1: 0);
    switch (*p) {
      case 's': PRINT_SOURCE = 1;
        break;
      case 'm': PRINT_MACROEXPANDED_SOURCE = 1;
        break;
      case 'o': PRINT_OBJECT = 1;
        break;
      case 'c': PRINT_CHAT = 1;
        break;
      case 'n': COMPILE_ONLY = 1;
        break;
      case 'v':
        PRINT_SOURCE = PRINT_MACROEXPANDED_SOURCE =
          PRINT_OBJECT = PRINT_CHAT = 1;
        break;
      default:
        fprintf (stderr, "Unknown option %s\n", argv[0]);
        exit (-1);
    }
    argv[0]++;
  }
}
```

```
#define NIL ((cons) NULL)
#define EMPTYENV ((env) NULL)
#define EMPTYSTACK ((stack) NULL)
#define EMPTYCOMMANDPORTSTACK ((command_port_stack) NULL)
#define HALT ((frame) NULL)
#define NOVALUE ((value) NULL)
#define NOINST ((inst) NULL)
#define NOSTATE ((state) NULL)
```

Appendix C

An Urlang Example

We present an iterative computation server as an extended example of an Urlang program.

We use the term *engine* to mean a particular kind of function which implements an iterative computation. When called, an engine performs one step of its computation and either returns the answer, or returns another engine which performs the next step. By *running an engine* we mean continually calling the engine returned from each successive engine call. Our iterative computation server will take an engine and run it until either it returns an answer, or another engine is sent to replace the first one.

We implement engines by means of curried functions taking two arguments: the iteration variable and the client's process id. The precise mechanism is best explained in the context of an example. The following function produces an engine which counts from 0 to some specified upper limit:

```
(rec make-counting-engine (upper-limit)
     ((rec counting-engine (count)
           (rec engine (client-pid)
                (progn
                  (print count)
                  (if (= count upper-limit)
                      (put client-pid count)
                    (let ((next (+ count 1)))
                      (counting-engine next))))))
      0))
```

When `counting-engine` is called on the current value of `count`, it returns an engine. When the engine is called on `client-pid`, the computation can proceed. If the upper limit has been reached, the client is informed of the answer. If not, the next value of `count` is computed and passed to `counting-engine`, which returns an engine. `make-counting-engine` makes the initial engine by calling `counting-engine` on 0.

The next function computes factorials. The iteration variable here is a pair consisting of the current count and its factorial:

```
(rec make-fact-engine (upper-limit)
     ((rec fact-engine (count&fact)
           (rec engine (client-pid)
                (let ((count (car count&fact))
                      (fact (cdr count&fact))
                      )
                  (progn
                    (print count&fact)
                    (if (= count upper-limit)
                        (put client-pid fact)
                      (let ((next-count (+ count 1))
                            (next-fact (* fact next-count))
                            (next-count&fact (cons next-count
                                                   next-fact)))
                        (fact-engine next-count&fact)))))))
      (cons 0 1)))
```

Next we have the function that makes a server. The function `server` is the spawnable server function itself. Its outer loop receives a `command`, running it if it is an engine closure and terminating otherwise. The engine is run by repeatedly calling it on the client's process id as long as there is not another command waiting and it keeps returning a closure. We assume, somewhat sloppily, that the engine does not return a closure as its final result.

```
(rec make-server (client-pid)
     (rec server (server-pid)
          (let ((command (get))
                (run-function
                 (rec run-function (fun)
                      (if (empty)
                          (let ((next-fun (fun client-pid)))
                            (if (closure? next-fun)
                                (run-function next-fun)
                              (server server-pid)))
                        (server server-pid))))
                )
            (if (closure? command)
                (run-function command)
              nil))))
```

The following program tests the engines and server. We send the server an engine which counts to 6, followed immediately by an engine which computes 10 factorial, followed by a command which makes the server terminate. The function `get-everything` loops to collect a list of all the messages returned by the server, and prints it out as the answer. Previously defined procedures are replaced by ellipses for brevity.

```
(rec main (root-pid)
     (let (
           (make-counting-engine... )
```

```
          (make-fact-engine... )
          (make-server... )
          (get-everything
           (rec get-everything (ignore)
                (if (empty)
                    nil
                  (cons (get) (get-everything nil)))))
          (server (spawn (make-server root-pid)))
          )
      (progn
        (put server (make-counting-engine 6))
        (put server (make-fact-engine 10))
        (put server nil)
        (result_is (get-everything nil)))))
```

When this program is run, the typical result is as follows:

```
[delphi: ~/phd/clam/concert] > urlang -c < programs/iter-server.l

RUN:
[delphi 18744]: Started.
[delphi 18749]: Booted.
[delphi 18749]: 0
[delphi 18749]: 1
[delphi 18749]: 2
[delphi 18749]: 3
[delphi 18749]: 4
[delphi 18749]: 5
[delphi 18749]: 6
[delphi 18749]: (0 . 1)
[delphi 18749]: (1 . 1)
[delphi 18749]: (2 . 2)
[delphi 18749]: (3 . 6)
[delphi 18749]: (4 . 24)
[delphi 18749]: (5 . 120)
[delphi 18749]: (6 . 720)
[delphi 18749]: (7 . 5040)
[delphi 18749]: (8 . 40320)
[delphi 18749]: (9 . 362880)
[delphi 18749]: (10 . 3628800)
[delphi 18749]: The machine halted with () on top of the stack.
RESULT:
[delphi 18744]: (6 3628800)
[delphi: ~/phd/clam/concert] >
```

As a matter of happenstance, the engines' computations were short enough that each completed before the next command arrived at the server; thus the answer is `(6 3628800)`. If we replace the first engine with `(make-counting-engine 666)`, then a typical result is this:

```
[delphi: ~/phd/clam/concert] > urlang -c < programs/iter-server.l

RUN:
[delphi 19141]: Started.
[delphi 19142]: Booted.
[delphi 19142]: 0
[delphi 19142]: 1
[delphi 19142]: 2
```

... 68 lines deleted ...

```
[delphi 19142]: 71
[delphi 19142]: 72
[delphi 19142]: 73
[delphi 19142]: (0 . 1)
[delphi 19142]: (1 . 1)
[delphi 19142]: (2 . 2)
[delphi 19142]: (3 . 6)
[delphi 19142]: (4 . 24)
[delphi 19142]: (5 . 120)
[delphi 19142]: (6 . 720)
[delphi 19142]: (7 . 5040)
[delphi 19142]: (8 . 40320)
[delphi 19142]: (9 . 362880)
[delphi 19142]: (10 . 3628800)
RESULT:
[delphi 19141]: (3628800)
[delphi: ~/phd/clam/concert] >
```

Here the server is interrupted by the arrival of the factorial engine, so the counting engine does not produce an answer.

Bibliography

[A+94] Joshua 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.

[Ams88] Richard Amster, September 1988. Personal communication.

[AVW93] Joe Armstrong, Robert Virding, and Mike Williams. *Concurrent Programming in Erlang.* Prentice Hall, 1993.

[Bar84] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* North-Holland, 1984.

[BB90] G. Berry and G. Boudol. The chemical abstract machine. In *Proceedings of the 17th Annual ACM Symposium on Principles of Programming Languages*, 1990.

[BMT92] Dave Berry, Robin Milner, and David N. Turner. A semantics for ML concurrency primitives. In *Conference Record of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, 1992.

[CCI87] CCITT Working Party X/1. *Recommendation Z.100 (SDL)*, 1987.

[CFW85] William Clinger, Daniel P. Friedman, and Mitchell Wand. A scheme for a higher-level semantic algebra. In Maurice Nivat and John C. Reynolds, editors, *Algebraic Methods in Semantics*, pages 237–250. Oxford University Press, Cambridge, 1985. Proceedings of US-French Seminar on the Application of Algebra to Language Definition and Compilation, Fontainebleau, France, 1982.

[Cli84] William Clinger. The Scheme 311 compiler: An exercise in denotational semantics. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 356–364, 1984.

[G+93] Al Geist et al. *PVM 3 User's Guide and Reference Manual.* Oak Ridge National Laboratory, May 1993.

[Gla94] David S. Gladstein. Compiler correctness for concurrent languages. Ph.D. thesis proposal, July 1994.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall International, 1985.

[Mil89] Robin Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[Mil92] Robin Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2:119–141, 1992.

[MP92] Z. Manna and A. Pneuli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer-Verlag, 1992.

[Oli93] Dino P. Oliva. *Advice on Structuring Compiler Back Ends and Proving Them Correct.* PhD thesis, Northeastern University, December 1993. Available as Technical Report NU–CCS–94–02.

[Par81] David Park. Concurrency and automata on infinite sequences. In *Theoretical Computer Science: 5^{th}th GI-Conference*, pages 167–183. Gesellschaft für Informatik, Springer Verlag, 1981. LNCS 104.

[PGM90] Sanjiva Prasad, Allessandro Giacalone, and Prateek Mishra. Operational and algebraic semantics for Facile: A symmetric integration of concurrent and functional programming. In *Proceedings of ICALP '90*, pages 765–780. Springer-Verlag, 1990. LNCS 443.

[Rep91] John H. Reppy. CML: A higher-order concurrent language. *ACM SIGPLAN Notices*, 26(6):293–305, June 1991. 1991 ACM Conference on Programming Language Design and Implementation.

[Str93] Bjarne Stroustrup. A history of C++: 1979–1991. In *The Second ACM SIGPLAN History of Programming Languages Conference*, April 1993.

[SW74] Christopher Strachey and C. P. Wadsworth. Continuations: A mathematical semantics for handling full jumps. Technical Monograph PRG-11, Oxford University Computing Laboratory, January 1974.

[Val90] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8), August 1990.

[vGV91] Rob van Glabbeek and Frits 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.

[Vog92] Walter Vogler. *Modular Construction and Partial Order Semantics of Petri Nets.* Springer Verlag, 1992. LNCS 625.

[Wan94a] Mitchell Wand. A proof architecture for languages with communication and concurrency. Seminar notes, April 1994.

[Wan94b] Mitchell Wand. Compiler correctness for parallel languages. In preparation, November 1994.

[WO92] Mitchell Wand and Dino P. Oliva. Proving the correctness of storage representations. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, 1992.

[WS94] Mitchell Wand and Gregory T. Sullivan. A little goes a long way: A simple tool to support denotational compiler-correctness proofs. In preparation, 1994.

Implementation Index