Higher-order Concurrency

John Hamilton Reppy
Ph.D Thesis

TR 92-1852
June 1992

Department of Computer Science
Cornell University
Ithaca, NY 14853

HIGHER-ORDER CONCURRENCY

A Dissertation
Presented to the Faculty of the Graduate School
of Cornell University
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy

by

John Hamilton Reppy

January 1992

HIGHER-ORDER CONCURRENCY

John Hamilton Reppy, Ph.D.
Cornell University 1992

Concurrent programming is a useful technique for structuring many important classes of applications such as interactive systems. This dissertation presents an approach to concurrent language design that provides a new form of linguistic support for constructing concurrent applications. This new approach treats synchronous operations as first-class values in a way that is analogous to the treatment of functions as first-class values in languages such as **ML**. The mechanism is set in the framework of the language **Concurrent ML** (**CML**), which is a concurrent extension of **Standard ML**. **CML** has a domain of first-class values, called *events*, that represent synchronous operations. Synchronous message passing operations are provided as the base-event values, and combinators are provided for constructing more complex events from other event values. This mechanism allows programmers to define new synchronization and communication abstractions that are first-class citizens, which gives programmers the flexibility to tailor their concurrency abstractions to their applications.

The dissertation is organized into three technical parts. The first part describes the design and rationale of **CML** and shows how first-class synchronous operations can be used to implement many of the communication mechanisms found in other concurrent languages. The second part presents the formal operational semantics of first-class synchronous operations and proves that the polymorphic type system used by **CML** is sound. The third part addresses practical issues. It describes the use of **CML** in non-trivial applications, describes the implementation and performance of **CML** on a single-processor computer, and discusses issues related to the use and implementation of **CML** on a shared-memory multiprocessor.

In memory of
David A. Reppy
1962–1988

Acknowledgements

I would first, and foremost, like to thank my parents for all of their support and love during my never-ending education.

My committee members, Tim Teitelbaum, Bard Bloom and Anil Nerode, provided me with feedback on this research. I would especially like to thank Tim, my committee chairman, for teaching me CS100 many years ago; it was this experience that set me on the road to becoming a computer scientist. Robert Cooper, who was on my committee in all but name also provided useful technical criticism. Emden Gansner, Lal George, Tim Griffin, Dave MacQueen, and Cynthia McMillin helped with the proof reading of this dissertation.

I have benefited from a long collaboration with members of the *Advanced Software Department* at AT&T Bell Laboratories. I would like to thank Dave Belanger, Chandra Kintala, and David Korn, who provided support for my projects, and Emden Gansner and Steve North, who were my principal collaborators. Many of the ideas in this dissertation were born out of my work at Bell. I have also spent many hours discussing these, and other, ideas with Andrew Appel, Dave MacQueen, Chet Murthy and Prakash Panangaden. And my work on **SML** implementation has benefited from discussions with Bill Aitken.

There are many other people who have helped me along the way. Pam and Cliff Surko provided me with both encouragement and a home away from home. And I would like to thank Bharati and Chandra Kintala for their hospitality while I was working at Bell Labs. I would like to thank Liz Maxwell for helping me with the difficult task of tracking my advisor, and Jan Batzer, who runs interference with the Cornell bureaucracy for all of the Computer Science graduate students.

The research presented in this dissertation was supported, in part, by the NSF and ONR under NSF grant CCR-85-14862, and by the NSF under NSF grant CCR-89-18233.

Contents

List of Tables

List of Figures

Part I

Introduction

Chapter 1

Introduction

Abstraction is perhaps the most important tool that programmers have for managing the complexity of software design and implementation. There are various language mechanisms for promoting abstraction, such as procedures for hiding the details of computation, abstract data-types for hiding representation information, and modules for grouping related types and operations with an abstract interface. This dissertation describes a new language mechanism for supporting abstraction of communication and synchronization in concurrent programs. My approach is to treat synchronous operations as first-class values in a way that is analogous to the treatment of functions as first-class values in languages such as **ML**. By doing so, a small collection of primitive operations and combinators can support a wide range of different concurrency paradigms. I call this style of programming "*higher-order concurrent programming*," as an analogy with higher-order programming in languages such as **ML**.

This work is set in the context of **Standard ML (SML)** [MTH90]. I have developed a language, called **Concurrent ML (CML)**, that extends **SML** with multiple threads of control and first-class synchronous operations. **CML** is implemented on top of the **Standard ML of New Jersey (SML/NJ)** system [AM87, AM91]. While the discussion of this dissertation uses **CML** as the archetype, the language design principles are easily applied to other higher-order languages (e.g., **Quest** [Car89]), and should also be applicable to object-oriented languages such as **Modula-3**.

1.1 The case for concurrency

Concurrency is often touted as a source of improved performance and rightly so, but it is a subtext of this dissertation that concurrency is an important programming tool independent of the performance benefits from multiprocessing. Certain classes of applications, most notably interactive applications, are naturally structured as concurrent programs. The language design presented in this dissertation is motivated by the need to support the programming of these applications.

Before going any further, it is useful to define a nomenclature. I distinguish between *parallel* and *concurrent* languages by whether they provide implicit or explicit concurrency. For example, the futures found in some dialects of **Lisp** are a parallel language feature, since they only specify the possibility of concurrent computation. Because I am interested in programming systems with explicit concurrency, the focus of this dissertation is on providing support for concurrent programming, and not on parallel programming.[1]

In the remainder of this section, I examine two important classes of applications that benefit from the use of concurrent programming. These applications share the property that flexibility in the scheduling of computation is required. Whereas sequential languages force a total order on computation, concurrent languages permit a partial order, which provides the needed flexibility.

1.1.1 Interactive systems

Providing a better foundation for programming interactive systems, such as programming environments, was the original motivation for this line of research [RG86]. Because of their naturally concurrent structure, interactive systems are one of the most important application areas for **CML**. Concurrency arises in several ways in interactive systems:

User interaction. Handling user input is the most complex aspect of an interactive program. Most interactive systems use an *event-loop* and *call-back* functions. The event-loop receives input events (e.g., mouse clicks) and passes them to the appropriate *event-handler*. This structure is a poor-man's concurrency: the event-handlers are coroutines and the event-loop is the scheduler.

Multiple services. For example, consider a document preparation system that provides both editing and formatting. These two services are independent and can be naturally organized as two separate threads. Threads also provide easy replication of services; if the user opens a new document for editing, then the system just spawns a new edit thread. Multiple views of the same document can also be supported by replicating threads.

Interleaving computation. A user of a document preparation system may want to edit one part of a document while another part is being formatted. Since formatting may

[1] I do examine some of the issues related to a multiprocessor implementation of **CML** in Chapter 12.

take a significant amount of time, providing a responsive interface requires interleaving formatting and editing. If the editor and formatter are separate threads, then interleaving comes for free.

Output-driven applications. Most windowing toolkits (e.g., **Xlib** [Nye90b]) provide an *input-driven* model, in which the application code is occasionally called in response to some external event. But many applications are *output driven.* Consider, for example, a computationally intensive simulation that maintains a graphical display of its current state. This application must monitor window events, such as refresh and resize notifications, so that it can redraw itself when necessary. In a sequential implementation, the handling of these events must be postponed until the simulation is ready to update the displayed information. By separating the display code and simulation code into separate threads, the handling of asynchronous redrawing is easy.

The root cause of these forms of concurrency is computer-human interaction: humans are asynchronous and slow.

While the use of heavy-weight operating-system processes provides some support for multiple services and interleaved computation, it does not address the other two sources of concurrency. Likewise, while event-loops and call-back functions provide flexibility in reacting to user input, they bias the application towards an *input-driven* model and do not provide much support for interleaved computation. A concurrent language, on the other hand, addresses all of these concerns.

1.1.2 Distributed systems

Another application area in which concurrent programming is useful is distributed systems. In fact, many existing distributed programming languages and toolkits provide support for concurrent programming (e.g., **Argus** [LS83], **Isis** [BCJ$^+$90], and **SR** [AOCE88]). Concurrency is needed because interaction with remote processes is slow and naturally asynchronous. Threads provide a useful abstraction for managing outstanding interactions and for reacting to new requests dynamically [LHG86].

1.2 Overview of this dissertation

I believe that there are three important aspects to good language design. First, there should be a real problem that needs solving, and a design that solves it. Second, there should be a firm theoretical foundation for the design. And third, the feasibility and usefulness of the design should be demonstrated in practice. The organization of this dissertation reflects this philosophy. It is divided into five parts: *introduction*, *design*, *theory*, *practice*, and *conclusion*, with the middle three parts addressing the above aspects.

The design part presents the rationale and design of my concurrency mechanism; the theory part provides a formal understanding of the mechanism; and the practice part addresses the issues of feasibility and usefulness of the mechanism. The other two parts are less technical: the introduction part includes this chapter and an introduction to **SML**, which may be skipped by the reader who is familiar with **ML** notation; the conclusion part describes areas for future research and summarizes the results of my research.

1.2.1 Design

The design part starts off with Chapter 3, which surveys existing approaches to concurrent language design. Chapter 4 is the heart of the dissertation; it provides the rationale for first-class synchronous operations and introduces them in the context of **CML**. The following chapter provides several substantial examples of the use of first-class synchronizations to build communication and synchronization abstractions, including several found in other concurrent languages. This part of the dissertation is fairly self contained, although familiarity with **SML** syntax is useful.

1.2.2 Theory

SML has set a precedent of both being a practical language with real implementations and of having a detailed formal semantics. I have developed a dynamic semantics for a small language, called λ_{cv} that models the concurrency features of **CML** [Rep91b]. This dissertation extends the work of [Rep91b], by presenting a static semantics for λ_{cv} and proving that it is *sound* with respect to the dynamic semantics.

Following a brief summary of basic notation, Chapter 6 illustrates the style of formal semantics using a more familiar sequential language, which is a sequential subset of λ_{cv}. Chapter 7 presents the syntax and operational semantics of λ_{cv}. The main results of this part are in Chapter 8, where I present a polymorphic type system for λ_{cv} and show that it is sound with respect to the dynamic semantics of Chapter 7. This result is important, since the implementation of **CML** uses features of **SML/NJ** that are not type-safe. To my knowledge, this is the first proof of the soundness for a polymorphic typing of concurrency primitives.

1.2.3 Practice

In the final analysis, the true worth of a language design can only be determined "in-the-field." Questions about the usefulness and practicality of language features can only be answered by actual experience. I have developed and distributed an implementation of **CML** for single processor computers[2] [Rep90b], which has been used by myself and others to implement several non-trivial applications. This experience demonstrates that **CML** is a useful programming language and that it can have efficient implementations.

Chapter 9 describes the use of **CML** to construct a multi-threaded X window system toolkit, called **eXene** [GR91], and its use to build interactive applications on top of **eXene**. I also briefly discuss the application of **CML** to the programming of distributed systems, and applications of **CML** by other researchers. In Chapter 10, I describe the implementation of **CML** in detail and describe some possible implementation improvements. Chapter 11 presents the results of micro-benchmarks that demonstrate the efficiency of **CML** (including a head-to-head comparison with a **C** thread library). These data support the conclusion that **CML** is competitive with thread libraries implemented in lower-level languages. Finally, Chapter 12 discusses the use of **CML** for parallel programming, possible extensions to better support parallel programming, and sketches the design of a shared-memory multiprocessor implementation of **CML**.

1.3 History

The ideas in this dissertation have been evolving for several years and there have been several instantiations of them in language designs. I first developed this approach in the context of **PML** [Rep88], an **ML**-like language used in the **Pegasus** system at AT&T Bell Laboratories [RG86, GR92]. I reimplemented the concurrency primitives of **PML** on top of **SML/NJ** at Cornell University [Rep89]. This implementation evolved into the current version of **CML** [Rep91a], which is described in this dissertation.

[2]The first version was released in November 1990.

Chapter 2

An Introduction to SML

While the ideas presented in this dissertation are largely language independent, they have been developed in the context of **Standard ML** (**SML**). I use **SML** both as the sequential core of my language design and as the implementation language. This chapter provides an introduction to **SML** that should allow the reader to follow the examples; for a more complete introduction see [Har86] or [Pau91]. The formal definition of **SML** can be found in [MTH90, MT91].

In the remainder of this chapter, I first introduce the basic features of **SML**; then I describe the datatype and pattern matching mechanisms; I follow this by a discussion of the imperative features of **SML**; and finally I present a complete example.

2.1 Basics

SML is an expression language: the traditional statement constructs, such as blocks, conditionals, case statements, assignment, etc., are packaged as expressions. Every expression has a statically determined type and will only evaluate to values of that type (this is called *type soundness*). Computation in **SML** is *value oriented.* Because of the central role of values, there is a much larger range of values than found in more conventional languages.

2.1.1 Basic values and expressions

SML provides a fairly standard collection of ground types and values, which are summarized in Table 2.1. The type `unit`, which has exactly one value (written `()`), is often used as the result type of functions that are executed for their side effects. Negative numbers are denoted using using a leading tilde, which is also the unary negation operator.

In imperative languages, such as **C**, assignment is the principal mechanism used to as-

Table 2.1: **SML** ground types

Type	**Sample literal values**
`unit`	`()`
`bool`	`true`, `false`
`int`	`...`, `~2`, `~1`, `0`, `1`, `2`, `...`
`string`	`"abc"`, `"hello world!\n"`
`real`	`1.0`, `1.0E~6`

sociate values with variables. While **SML** does provide updatable cells (see Section 2.3.1), it uses *binding* as its principal mechanism for associating values with variables. In **SML**, variables are used to name values, and are immutable (this is sometimes called *single assignment*). For example, the binding

```
val x = 1 and y = "I'm a string"
```

establishes bindings for `x` and `y`. The static environment produced by this binding assigns the type `int` to `x` and `string` to `y` (the type information is *inferred* by the compiler). This static environment can be summarized by the following *specification*:

```
val x : int
val y : string
```

The notation of specifications, which comes from the signatures in the module system, is a natural and concise way to describe a set of bindings.

2.1.2 Tuples and records

In addition to these ground types and values, **SML** provides tuples and records. For example, the expression `(1, true)` is a pair of the value `1` and `true`, and has the product type `int * bool`. Records are labeled tuples. For example, `p1` might be defined to be the point $(1, 2)$ by

```
val p1 = {x = 1, y = 2}
```

in which case `p1` has the type `{x : int, y : int}`. Note that the order in which labeled fields appear is insignificant, so that

```
val p2 = {y = 2, x = 1}
```

defines the same point as `p1`. A field labeled l of a record can be selected using the notation `#`l. The example in Section 2.4 further illustrates the use of labeled records.

2.1.3 Functions and polymorphism

Functions play a key role in **SML**. Functions are declared using the leading keyword `fun`; for example, the factorial function can be defined as:

```
fun fact n = if (n = 0) then 1 else n * fact(n-1)
```

which has the specification:

```
val fact : int -> int
```

Tail recursion plays the role of looping in **SML**.[1] For example, the iterative form of the factorial function is written as a tail recursive function:

```
fun fact n = let
      fun loop (i, result) = if (i = 0)
            then result
            else loop(i-1, i*result)
      in
        loop (n, 1)
      end
```

This example also introduces the `let`-expression, which is used for defining local variables (the function `loop` in this case). Note that instead of destructive updates to loop variables, the new values are passed to the next invocation of `iterFact`; each iteration has its own copies of `i` and `result`.

The **SML** compiler uses type inference to determine the types of expressions. In the case of functions, this can often be a family of types. For example, consider the identity function:

```
fun identity x = x
```

The meaning of this function is independent of its argument type. It can be viewed as a function on integers, or strings, or reals, or pairs of integers, etc. Thus, it has the *polymorphic type* $\forall\alpha.(\alpha \rightarrow \alpha)$, where α is a *type variable* ranging over all types. In **SML**, type variables are denoted by a leading apostrophe. For example, the value `identity` has the specification:

```
val identity : 'a -> 'a
```

[1]There is a `while` expression, but it is just syntactic sugar for the application of a tail recursive function.

where the $\forall$ is implicit. The **SML** compiler always infers the most general type for a given expression.

SML is a higher-order language, which means that functions are first-class values; they can be passed as arguments, embedded in data structures and returned as results. A simple example is infix function composition, which is defined in **SML** as:

```
fun o (f, g) = fn x => (f (g x))
infix o
```

The form "`fn x =>` ..." is the way that function values are written in **SML** (for those familiar with the λ-calculus, `fn` can be read as λ). The second line declares `o` to be an infix operator. An infix operator can be used as a normal identifier by prefixing it with the keyword `op` (e.g., `op +`). Function composition can also be defined using a *curried* form:

```
fun o (f, g) x = (f (g x))
```

These two declarations of composition are equivalent, and have the specification:

```
val o : ('a -> 'b) * ('c -> 'a) -> 'c -> 'b
```

As an example of its use, the expression

```
(fn x => (x*x)) o (fn x => (x-1))
```

evaluates to a function that computes $(x-1)^2$.

2.2 Datatypes and pattern matching

In addition to the basic values, **SML** provides recursive data structures and abstract types. Structured values are decomposed using a powerful *pattern matching* notation.

2.2.1 Datatypes

The `datatype` declaration introduces a new, possibly recursive, type. For example, the representation of integer binary trees can be defined as:

```
datatype int_tree
  = Empty
  | Leaf of int
  | Node of (int * int_tree * int_tree)
```

This declaration says that a tree is either *empty*, a *leaf* consisting of an integer value, or a *node* consisting of an integer and two sub-trees. The identifiers `Empty`, `Leaf` and `Node` are called constructors, and are used to construct tree values. Datatype declarations can be parameterized to define *type constructors*. For example, we can define a family of binary tree types by the definition:

```
datatype 'a tree
  = Empty
  | Leaf of 'a
  | Node of ('a * 'a tree * 'a tree)
```

In addition to user defined datatypes, **SML** has a few predefined datatypes. The type `bool` is actually defined as

```
datatype bool = true | false
```

Another important datatype that is predefined by **SML/NJ** is the polymorphic option type:

```
datatype 'a option = NONE | SOME of 'a
```

There is also a list type, which is discussed below.

2.2.2 Pattern matching

The power of the datatype declaration mechanism is enhanced by *pattern matching*. Pattern matching is a mechanism for control-flow, value decomposition, and binding. For example, the boolean negation function can be defined using two clauses:

```
fun not true = false
  | not _ = true
```

The first clause says that if the argument is `true`, then return `false`. The "_" in the second clause is a *wildcard*, which matches anything (in this case, `false` is the only possibility). Pattern matching is the standard binding and value decomposition mechanism in **SML**, and we have already seen some examples of it. For example, the definition of function composition

```
fun o (f, g) x = f (g x)
```

has a tuple pattern as its first argument, which binds `f` to the first element of the pair and `g` to the second. A more interesting example is a function to compute the height of a binary tree:

```
fun height Empty = 0
  | height (Leaf _) = 1
  | height (Node(_, t1, t2)) = max(height t1, height t2) + 1
```

In the third clause, the variables `t1` and `t2` are bound to the subtrees. Unlike pattern matching in **Prolog**, **SML** patterns are *linear* (i.e., a variable can occur at most once in a pattern).

Pattern matching can be used to extract values from records. For example, the following function swaps the x and y coordinates of a point:

```
datatype point = PT of {x : int, y : int}
fun swap (PT{x=x1, y=y1}) = PT{x=y1, y=x1}
```

The pattern binds `x1` to the `x` field and `y1` to the `y` field. There are two shorthand forms for pattern matching records, both of which are illustrated in the following example:

```
fun xCoord (PT{x, ...}) = x
```

Here the field name `x` is being used as shorthand for "`x=x`," and the "`...`" is in lieu of the rest of the fields.

Pattern matching can also be used against literals. For example, the recursive factorial function can be coded as follows:

```
fun fact 0 = 1
  | fact n = n * fact(n-1)
```

In addition to equational definitions of functions, pattern matching is used in a general form of a `case` expression.

2.2.3 Lists

One of the most important recursive types is the polymorphic list type, which is defined as

```
datatype 'a list = nil | :: of ('a * 'a list)
infix 5 ::
```

The datatype declaration defines a list to be either empty (`nil`), or the cons of an element and a list. The `infix` declaration specifies that the cons operator (`::`) is a right associative infix operator with precedence level 5. Because of the importance of lists, **SML** provides special syntax for list patterns and expressions. The syntax

[e_1, e_2, ..., e_n]

is syntactic sugar for

e_1`::`e_2`:: ... ::`e_n`::nil`

and likewise for patterns. The following function, which inserts delimiters between adjacent list elements, is an example of the use of this notation:

```
fun insertDelim delim l = let
      fun insert [] = []
        | insert (s as [_]) = s
        | insert (x::r) = x :: delim :: (insert r)
      in
        insert l
      end
```

The second clause of this function illustrates the *as pattern* form, which, in this case, binds `s` to the single element list matched by "`[_]`."

There are a number of standard list functions that are provided by **SML/NJ**, and used in this dissertation. These are:

```
val length : 'a list -> int
val map    : ('a -> 'b) -> 'a list -> 'b list
val app    : ('a -> 'b) -> 'a list -> unit
val rev    : 'a list -> 'a list
```

The function `length` returns the length of a list; `map` applies a function to a list, returning the list of results; `app` applies a function to a list, discarding the results; and `rev` reverses a list.

2.2.4 Abstract types

The `abstype` declarative form is a variation on the `datatype` declaration that limits the visibility of the type's representation. The time-honored example of an abstract datatype is the stack:

```
abstype 'a stack = STK of 'a list
with
  val empty = STK[]
  fun push (x, STK s) = STK(x::s)
  fun pop (STK(x::r)) = (x, STK r)
end
```

The representation of a stack is only visible in between the `with` and `end`; outside the type `stack` is abstract. I use the `abstype` mechanism in this dissertation in lieu of the **SML** module facility, since it is easier to understand. A more elaborate example of abstract types is given in Section 2.4.

In addition to the abstype declaration, the local declaration can be used to limit the scope of declarations. For example, the stack could be declared as

```
local
  datatype 'a stack = STK of 'a list
in
type 'a stack = 'a stack
val empty = STK[]
fun push (x, STK s) = STK(x::s)
fun pop (STK(x::r)) = (x, STK r)
end
```

There are some technical differences between these two declarations, but they are beyond the scope of this dissertation.

2.3 Imperative features

Although **SML** is mostly applicative, it does have a small collection of imperative features. The most important of these are references and exceptions; in addition, **SML/NJ** provides first-class continuations.

2.3.1 References

References are mutable heap cells. They are created by the function `ref`,[2] updated using `:=`, and examined by the `!` function. As an example, the following binds two functions that share a common reference cell:

```
val (get, put) = let
      val cell = ref 0
      in
        ((fn () => !cell), (fn x => cell := x))
      end
```

The reference operations have the following signature:

```
val ref : '_a -> '_a ref
val ! : 'a ref -> 'a
val := ('a ref * 'a) -> unit
```

[2]The `ref` function is really a constructor and can be used in pattern matching, but that feature is not used in this dissertation.

The notation "'_a" in the type of `ref` means that it has an *imperative* type, which is "less polymorphic" than a similar *applicative* (non-imperative) type. This is a technical restriction that is required to prevent type loopholes. The full technical details of imperative types is beyond the scope of this introduction; Chapter 8 has some discussion of imperative types and Tofte describes them in great detail in [Tof88] and [Tof90].

SML/NJ uses a more flexible scheme for typing polymorphic references, called *weak polymorphism.* The idea is to assign a *rank* (or *strength*) to type variables. Roughly, the rank of a type variable is the number of abstractions that "protect" a reference value of that type; normal type variables have rank ∞. For example, the type of `ref` in **SML/NJ** is

```
val ref : '1a -> '1a ref
```

where the integer in the type-variable name denotes its rank. Since **CML** is implemented on top of **SML/NJ**, its interfaces are presented using weak types. The details of weak polymorphism are not important to this dissertation; it is only necessary to recognize that functions with weak types are not fully polymorphic. The theoretical treatment (Part III), however, uses the more standard imperative type system.

2.3.2 Exceptions

SML has an exception mechanism for signaling run-time errors and other exceptional conditions. There are two aspects to the exception mechanism: the representation of exception packets, and the control-flow of raising and handling exceptions.

The built-in type `exn` is the type of exception packets, which are created using a special kind of datatype constructors. The declaration

```
exception Foo and Bar of int
```

declares two new exception constructors (exception specifications use the same syntax).

Since exception packets are values of a datatype, the handling of exceptions can use the pattern matching mechanism to match exceptions. For example, the following is an implementation of integer division that returns `0` when the divisor is `0`:

```
fun safeDiv (a, b) = (a div b) handle Div => 0
```

An exception is raised using the `raise` expression. For example, the following function, which computes the product of a list of integers, uses the exception `Zero` to short-circuit the evaluation if `0` is encountered:

```
fun product l = let
      exception Zero
      fun loop ([], n) = n
        | loop (0::_, _) = raise Zero
        | loop (i::r, n) = loop (r, i*n)
      in
        (loop (l, 1)) handle Zero => 0
      end
```

Although this dissertation only uses monomorphic exceptions, it is possible to declare polymorphic exception constructors. As with references, fully polymorphic exceptions result in type loopholes; therefore exceptions can only be weakly polymorphic (or have imperative types).

2.3.3 Continuations

SML/NJ provides first-class continuations as an extension, and I use them heavily in the implementation of **CML** (see Chapter 10). A *continuation* is a function that represents the "rest of the program" [Gor79]. The programming language **Scheme** [RC86] makes continuations accessible to the programmer as first-class values.[3] The **Scheme** function *call-with-current-continuation* (*call/cc* for short) calls a function with the current continuation as the argument. First-class continuations are supported in **SML/NJ** via an abstract type and two primitive functions [DHM91]:

```
type 'a cont
val callcc : ('1a cont -> '1a) -> '1a
val throw  : 'a cont -> 'a -> 'b
```

These can be used to implement loops, back-tracking, exceptions and various concurrency mechanisms, such as *coroutines* [Wan80] and *engines* [DH89]. For example, the following is a continuation-based version (from [DHM91]) of the `product` function given in the previous section:

```
fun product l = callcc (
      fn exit => let
        fun loop ([], n) = n
          | loop (0::_, _) = throw exit 0
          | loop (i::r, n) = loop (r, i*n)
        in
          loop (l, 1)
        end)
```

This function uses the continuation `exit` to short-circuit the evaluation if `0` is encountered.

[3]The idea dates back to Landin's **J** operator [Lan65], [Fel87b].

2.4 An example — functional queues

To wrap up this introduction to **SML**, consider the implementation of an abstract FIFO queue type. The signature of this abstraction is:

```
type 'a queue
val empty : 'a queue
val isEmpty : 'a queue -> bool
val insert : 'a * 'a queue -> 'a queue
exception EmptyQ
val remove : 'a queue -> 'a * 'a queue
```

The value `empty` is the empty queue; `isEmpty` returns `true` if its argument is the empty queue; `insert` adds an item to the end of the queue; and `remove` removes the head of the queue. The exception `EmptyQ` is raised if `remove` is applied to an empty queue. This abstraction is functional; i.e., instead of mutating a shared queue object, the operations `insert` and `remove` return new queue values as results.

The implementation of this abstraction is given in Figure 2.1. Internally, a queue is

```
abstype 'a queue = Q of {front : 'a list, rear : 'a list}
with
  val empty = Q{front = [], rear = []}
  fun isEmpty (Q{front=[], rear=[]}) = true
    | isEmpty _ = false
  fun insert (x, Q{front, rear}) = Q{front = front, rear = x::rear}
  exception EmptyQ
  fun remove (Q{front = [], rear = []}) = raise EmptyQ
    | remove (Q{front = [], rear}) = remove(Q{front = rev rear, rear = []})
    | remove (Q{front = x::r, rear}) = (x, Q{front = r, rear = rear})
end (* abstype *)
```

Figure 2.1: A queue implementation.

represented by the constructor `Q` applied to a record of two fields: `front` and `rear`, which are stacks (represented by lists). The `insert` operation pushes a value onto the `rear` stack, and the `remove` operation pops a value from the `front` stack. In the case that the `front` is empty, then `remove` pushes the elements of the `rear` stack onto the `front` in reverse order.

Part II

Design

Chapter 3

Concurrent Programming Languages

In order to understand the trade-offs in language design, it is necessary to know the alternatives. In this chapter, I survey a representative collection of concurrency features and languages.[1] For the purpose of this dissertation, the most important language characteristics are the synchronization and communication primitives. These can be divided into two main classes: *shared memory* primitives and *distributed memory* (or *message-passing*) primitives. In this chapter, following a brief discussion of process creation mechanisms, I focus on these two different classes of concurrent languages, and discuss the appropriateness of the various design alternatives for adding concurrency to **SML**.

There are a number of good surveys of concurrent language design. A comparison of different concurrency mechanisms using two example problems can be found in [BD80]. Andrews and Schneider [AS83] survey a broad range of concurrency mechanisms; Wegner and Smolka compare **CSP**, **Ada** and monitors in [WS83]; Andrews covers concurrent programming using various different languages in [And91]. And a collection of significant reprints of papers on concurrent languages and programming (including [AS83] and [WS83]) can be found in [GM88].

As discussed in Chapter 1, this dissertation is about concurrent language design, therefore I do not survey parallel implementations of lazy languages (e.g., **GAML** [Mar91]), parallel languages (e.g., **Id** [Nik91]), or distributed languages (e.g., **Argus** [LS83], or **SR** [AOCE88]). I also do not discuss concurrent logic-programming and concurrent constraint languages [SR90].

[1]There are literally hundreds of different concurrent programming languages, so a complete survey is impossible.

3.1 Processes and threads

The specification and creation of processes in a concurrent programming language is usually, although not always (see Section 3.3.5), orthogonal to the communication and synchronization mechanisms. Process creation can be either *static*, where the set of processes is fixed by the text of the program, or *dynamic*, where some mechanism is provided for creating new processes on the fly. Each process in a concurrent program has an independent *thread of control*, hence, the term *thread* is often used instead of process. This has the added advantage of avoiding confusion with the other meanings of the word process. I favor the term thread, except in the context of the formal semantics where process is the conventional term.

An example of static process creation is the *cobegin* statement, which has the form[2]

COBEGIN $stmt_1$ || $stmt_2$ || ... || $stmt_n$ COEND

This statement proceeds by executing the n statements in parallel and then synchronizing on the completion of all of the statements. In a language with recursion this statement can be used to create dynamic tree parallelism, but it is still limited in that the lifetimes of processes are tied to their children's lifetimes.

Dynamic process creation usually involves a *fork* operation (sometimes called *spawn*), which takes a statement (or procedure) as an argument and creates a new process to execute it. The fork operation is often accompanied by a *join* operation, which allows the parent to synchronize on the child's termination. Using fork and join, the cobegin construct from above can be implemented as:

```
p1 := FORK stmt1
p2 := FORK stmt2
...
pn := FORK stmtn
JOIN p1
JOIN p2
...
JOIN pn
```

Dynamic process creation allows the flexible use of processes. For example, a server might want to create a new thread to handle each request. In a language with a static set of processes, this requires preallocating a pool of server threads and reusing them. This is awkward and limits the number of simultaneous requests that can be handled, which can

[2]For most of this chapter, I use an **Algol 60** style notation, since most of the languages I discuss have roots in the **Algol** family of languages.

lead to unnecessary delays when handling requests [LHG86]. Writing concurrent programs in a language with static process creation is similar to the problem of writing programs with dynamic data structures in a language that only provides static memory allocation. In conclusion, there does not seem to be any strong reason to use static process creation, and many reasons in favor of dynamic creation.

3.2 Shared-memory languages

Shared-memory languages use mutable shared state (e.g., shared variables) to implement process communication. The key problem in these languages is preventing processes from interfering with each other. This problem can be characterized by the following classic example:

```
x := 1;  COBEGIN x := x+1 || x := x+1 COEND
```

Without some guarantee of atomicity, the resulting value of x is undefined. The assignments are examples of *critical regions*; that is, regions of code that are potential sources of interference without proper concurrency control. Shared-memory languages are distinguished by the mechanisms they use to provide synchronization and concurrency control. To illustrate these, I use a unique ID service as a running example.

3.2.1 Low-level synchronization mechanisms

The most basic synchronization mechanism is the *semaphore*, which is a special integer variable with two operations: **P** and **V**. Given a semaphore s, the execution of $\mathbf{P}(s)$ by a process p forces it to delay until $s > 0$, at which point p executes $s := s - 1$ and proceeds; the test and update of s is done atomically. Execution of $\mathbf{V}(s)$ results in the atomic execution of $s := s + 1$. Using semaphores, the unique ID service can be implemented as

```
VAR x : INTEGER := 0;
    s : SEMAPHORE;
...
PROCEDURE getUId () : INTEGER =
  VAR result : INTEGER
  BEGIN
    P(s);
      result := x;  x := x+1;
    V(s);
    RETURN result
  END
```

with the semaphore s being used to guarantee mutual exclusion on accesses of x. The problem with semaphores is that there is no linguistic support for their correct use. For example, a programmer can easily forget to apply one of the operators, or might forget to protect shared state. Furthermore, implementing patterns of synchronization that are more complicated than mutual exclusion can be tricky.

A restricted form of the semaphore is the *mutex lock*[3] (also called a *binary* semaphore), which is a variable that can be in one of two states, either *locked* or *unlocked*. One of the advantages of mutex locks is that they are naturally supported by the *test-and-set* instruction found on many multiprocessors. The language **Modula-3** [Nel91] supports the use of mutex locks with the special syntax.[4]

`LOCK` *m* `DO` *statements* `END`

This statement is executed by first acquiring the mutex lock m, then executing the statements in the body, and then releasing the lock. If an exception occurs during the execution of the body, the lock is also released. In **Modula-3**, the unique ID service can be implemented as:

```
VAR x : INTEGER := 0;
    m : MUTEX := NEW(MUTEX);
...
PROCEDURE getUId () : INTEGER =
  VAR result : INTEGER
  BEGIN
    LOCK m DO
      result := x;  x := x+1;
    END;
    RETURN result
  END
```

Each call to `getUId` first acquires the mutex lock, executes the critical section, and then releases the lock before returning.

Mutex locks are sufficient for insuring mutual exclusion in critical regions, but do not provide a general synchronization mechanism. For example, consider producer and consumer processes that share a fixed size buffer. If the buffer is empty, then the consumer must wait for the producer to add something to it; likewise, if the buffer is full, the producer must wait for the consumer to remove something. Using mutex locks, this requires polling the buffer, which is inefficient. To alleviate this problem, **Modula-3** provides *condition variables*, which allow processes to wait for specific conditions (e.g., the buffer is non-empty).

[3] *Mutex* is a contraction of *mutual exclusion*.

[4] **Modula-3** inherits these primitives from **Modula-2+** [RLW85].

Condition variables, in effect, reintroduce the counting power of general semaphores that was lost when moving to mutex locks. The **C-threads** package built on top of the MACH operating system also provides this style of concurrency support [CD88].

3.2.2 Monitors

A monitor is a module that encapsulates shared state, providing a set of exported procedures for controlled access to the state [Hoa74]. Monitors provide a more structured form of mutual exclusion than mutex locks. Each monitor has an implicit mutex lock that is acquired on entry and released on exit by every monitor procedure. This guarantees that a monitor-procedure call is mutually exclusive with any other call. Using a monitor, the unique ID server can be coded as follows:

```
MONITOR UId IS
  VAR x : INTEGER
  PROCEDURE getUId () : INTEGER =
    VAR result : INTEGER
    BEGIN
      result := x;  x := x+1;
      RETURN result
    END
  BEGIN
    x := 0
  END
```

The extra syntactic support provided by monitors leaves less room for programmer error than in the case of semaphores or mutex locks. As with mutex locks, condition variables are used to avoid polling.

A number of languages, such as **Concurrent Pascal** [Bri77], **Concurrent Euclid** [Hol83b], and **Mesa** [MMS79, LR80] provide monitors along with condition variables. It is interesting to note that there is a trend in concurrent language design away from the syntactic sugar of monitors and towards explicit mutex locks (e.g., from **Mesa** to **Modula-2+** and **Modula-3**). This trend represents a simplification of language design, since it separates two orthogonal language features (i.e, modules and mutual exclusion).

3.2.3 Shared-memory concurrency and ML

Shared-memory concurrent languages rely on mutable state for inter-process communication. This leads to an imperative programming style, which goes against the traditional, mostly applicative, style of **ML** programs. For this reason, shared-memory primitives are notationally unsuitable as a general purpose concurrency extension to **ML** (although they are useful for low-level implementation work). In contrast, as I show below, message passing fits quite naturally with the **ML** programming style.

Cooper and Morrisett, at Carnegie-Mellon University, have developed a concurrency package, called **ML-threads**, which provides threads, mutex locks and condition variables [CM90]. The design of **ML-threads** is owed to the **C-threads** package [CD88], which in turn owes its design to [RLW85]. The goals and approach of their work are significantly different from those of my research. For example, one of the principal applications of **ML-threads** is the construction of low-level operating system services, which requires heavy use of shared state [CHL91]. **ML-threads** has also been used to implement a subset of **CML**'s primitives. There is also an implementation of **ML-threads** for the SGI 4D/380 multiprocessor [Mor].

3.3 Distributed-memory languages

The other major class of concurrency primitives is distributed-memory (also called *message passing*). The basic operations in message passing are "*send a message*" and "*accept a message*," and are used for both communication and synchronization. Message-passing languages are distinguished by the naming mechanism for the communication medium and the amount of synchronization involved in sending a message.

The naming mechanism must specify both ends of the communication (i.e., sender and receiver). The simplest naming convention uses process names to designate the communication partner. A slightly more general scheme introduces multiple *communication ports* associated with the receiver. This can be further generalized by making port names into independent values, called *channels*. Any process that has access to a channel may use it to send or accept messages (a variant on this scheme differentiates between input and output access). As with the process structure, the naming mechanism can be either static or dynamic. Although static naming is common in a number of languages, it has severe limitations. For example, it is impossible to write procedures parameterized by a sender or receiver name. When adding message passing to **ML** the communication medium must be strongly typed. This requirement means that the use of process names to name communications is too restrictive, since under such a scheme, each process can only receive messages of one fixed type. Using ports or channels to name communications avoids this problem, since each port (or channel) can have its own message type. Given the dynamic nature of **ML** values, it seems that a dynamic port or channel creation mechanism is most suitable.

The message accept operation is usually blocking, but some languages and systems provide a *polling* mechanism to check for incoming messages. There are three basic choices for message sending semantics: *non-blocking send* (or *asynchronous* send), *blocking send* (or *synchronous send*), and *send-reply*. The first two of these are unidirectional, while the last is bidirectional. I discuss each of these below in increasing order of synchronization.

3.3.1 Asynchronous message passing

In asynchronous message passing the communication medium is buffered and the send operation is non-blocking.[5] Figure 3.1 gives a pictorial description of asynchronous communication between two processes P and Q. In this diagram, each process has a "time-line,"

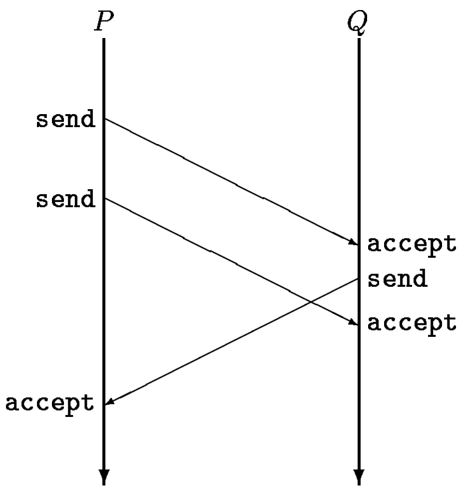


Figure 3.1: Asynchronous message passing

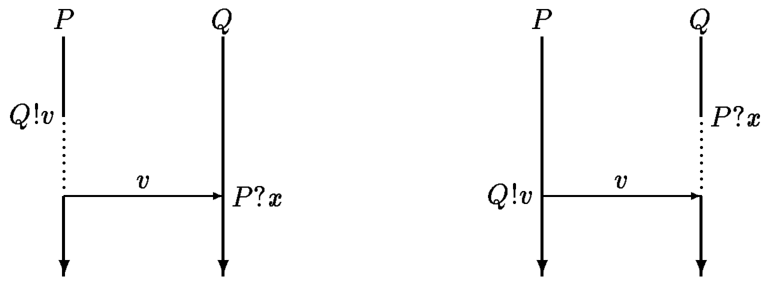
running down the page; a communication is represented by an arrow from the sender's time-line to the accepter's time-line. Notice that, in this picture, P and Q have different views of the order of events.

Actor languages are an example of programming languages based on asynchronous message passing [Agh86]. Message passing in distributed systems is also usually asynchronous. In fact, in systems with arbitrary message delays and failure it is not possible to distinguish between the failure of a communication partner and a slow line, and thus synchronous communication is impossible [FLP85].

[5]Some systems use finite buffers, in which case the send operation will block if the buffer is full.

3.3.2 Synchronous message passing

Hoare's seminal paper [Hoa78] introduced the notion of a set of sequential processes running in parallel and communicating by synchronous message passing. Hoare's language, called **CSP** (for *Communicating Sequential Processes*), provides input operations, $P?x$ (read a value from process P and assign it to x), output operations, $P!e$ (send the value of expression e to process P), and a labeled cobegin statement for process creation. Both the process and communication structures in **CSP** programs are static, since there is no dynamic process creation and process names are used to name communications. If a process P executes $Q!v$, it must block until process Q executes $P?x$ (and *vice versa*). The matching of communications is called *rendezvous*, and is illustrated in Figure 3.2. The dotted time-lines



(a) P waits for Q (b) Q waits for P

Figure 3.2: Rendezvous

in this figure represent idle periods while waiting for a matching communication.

One of the key ideas found in **CSP** is the notion of *selective communication* (also called *guarded communication*). In [Hoa78], selective communication is presented as a generalization of Dijkstra's guarded commands [Dij75], with input operations allowed as guards. When an input guard is matched, its action may be chosen; if more than one input guard is matched, then one is chosen nondeterministically. This mechanism provides the ability for a process to communicate with multiple partners when the order of communications is unknown. For example, a server process that has multiple clients may not know which client will send it the next request. Languages can provide polling as an alternative to selective communication, but the use of polling can result in busy waiting and so should be avoided [GC84].

A natural generalization of the selective communication mechanism of **CSP** is to allow both input and output operations. This is called *generalized* (or *symmetric*) selective communication. As a simple example of why this is useful, consider a system with three processes, A, B and C, where A is supposed to send a message to both B and C. Without generalized select, A must *a priori* choose which process to send the message to first. If B sends a message to C before accepting a message from A and C is waiting for the message from A before accepting a message from B, then A must send to C first to avoid deadlock. In other words, the implementation of A depends on the communication patterns of B and C. This example illustrates that the lack of generalized selective communication has a negative impact on program modularity.[6] Other arguments for the usefulness of selective communication can be found in [Hoa78], [FY85], [Rep91a], and Section 5.1. The only significant argument against generalized selective communication is the difficulty of implementing it on multiprocessor machines [KS79] (Section 12.2.2 discusses this problem in more detail).

The language **occam** [INM84, Bur88] is derived from **CSP**, but includes channels and a limited form of dynamic process creation. And the higher-order language **Amber** [Car86] provides generalized selective communication on typed channels, as well as dynamic process and channel creation. Other languages that owe an intellectual debt to **CSP** include **Joyce** [Bri89] and **Pascal-m** [AB86]. A pared down version of **CSP**, called **TCSP**, has been used for theoretical study of concurrent systems [Hoa85].

3.3.3 Asynchronous vs. synchronous message passing

At first glance, asynchronous communication may seem to be the best choice for a distributed-memory concurrent language, since it minimizes interprocess synchronization and does not restrict parallelism (e.g., in Figure 3.2(a), P must wait for Q). But if the language has dynamic thread creation, then it is possible to efficiently implement an asynchronous channel by using a thread to buffer communication (cf., Section 5.1). The big problem with asynchronous communication is that the sender has no way of knowing when a message has actually been received; introducing acknowledgement messages loses the parallelism that was the main benefit of asynchronous communication. In synchronous message passing, the sender and receiver have common knowledge of the message transmission (e.g., the sender knows that the receiver knows that the sender knows that the message was accepted). This property makes synchronous message passing easier to reason about [AS83].

This is also reflected in the typical failure modes of erroneous programs. In asynchronous systems, the typical failure mode is an overflow of the memory used to buffer messages, which is likely to be far removed in time (and possibly place) from the source of the problem. In

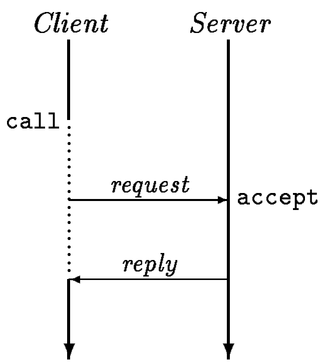
[6] In a language with dynamic thread creation, this example could be programmed by A forking two threads to send the messages, but there are other examples where dynamic process creation is not sufficient.

synchronous systems, the typical failure mode is deadlock, which is immediate and easily detected. Thus, detecting and fixing bugs is easier in a synchronous system.[7]

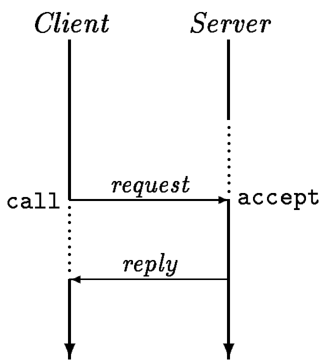
Using asynchronous message passing also increases the likelihood of timing sensitivity and race conditions. In a producer-consumer protocol, for example, if the producer is faster than the consumer, then the number of buffered messages can grow arbitrarily. If the buffer is finite, the system eventually degrades to a synchronous system; while, if the buffers are unbounded, memory overflow may occur. This means that additional acknowledgment messages must be used, which reduces the efficiency gains from using asynchronous communication.

3.3.4 Request-reply message passing

A procedure call style interaction, called *remote procedure call* [Nel81], can be implemented using asynchronous or synchronous message passing. The procedure entry corresponds to a request message from the client to the server and the procedure return corresponds to the reply message from the server to the client. Figure 3.3 shows the timing diagrams for this mechanism (assuming synchronous message passing). While the server is handling a call it



(a) The client waits



(b) The server waits

Figure 3.3: Request-reply rendezvous

cannot accept other requests; thus, calls are necessarily mutually exclusive. Some languages, such as **Ada** [DoD83] and **Concurrent C** [GR86], as well as concurrency libraries such as the **μSystem** [BS90], use this style of bidirectional message passing as their communication

[7]The author's personal experience backs this up. An early version of the **Pegasus** system [RG86] used asynchronous message passing, but we had great difficulty in debugging our programs. Our experience with the implementation of **eXene** [GR91], on the other hand, demonstrates that large synchronous message-passing programs can be debugged fairly easily, even without debugging tools.

mechanism. In these languages, a server thread plays a role very similar to a monitor (see Section 3.2.2). In **Ada**, for example, a task (the **Ada** term for process) exports a collection of transactions (called *entries* in **Ada**) that clients can invoke like normal procedure calls. The server uses a SELECT statement to enable multiple entries simultaneously (this is similar to the **CSP** selective communication in that only input operations are allowed in a select statement). To illustrate, the unique ID server example from Section 3.2 can be programmed in **Ada** as follows:

```
TASK BODY UniqueId IS
  x : INTEGER;
BEGIN
  LOOP
    SELECT
      ACCEPT getUniqueId (result : OUT INTEGER) DO
          result := x;  x := x+1;
        END getUniqueId;
    END SELECT;
  END LOOP;
END UniqueId
```

Since this example only has one operation, the SELECT statement is not really necessary, but if the server supported other operations, then additional accept clauses would be added.

The language **Concurrent C** provides a richer form of **Ada**'s select mechanism. The syntax of an entry clause in a select statement is

$$\texttt{accept}\ \mathit{entry}\ [\,\texttt{suchthat}\ \mathit{pred}\,]\ [\,\texttt{by}\ e\,]\ \mathit{statment}$$

where the phrases enclosed in [ ] are optional, *pred* is a boolean expression and e is an integer expression. If the optional suchthat clause is present then only those requests that satisfy *pred* are accepted. If the optional by clause is present then the expression e is evaluated for each outstanding request and the request with the minimum value is selected.

While the request-reply paradigm is quite useful in concurrent programming, I believe that it is too heavy-weight a mechanism to be the basis of a concurrent language. For example, if one needs to program unidirectional communication, then a bidirectional mechanism is unnatural. Of course, one might argue that programming bidirectional communication using unidirectional message passing is unnecessarily complex, but, as I show in the next chapter, it is possible to support higher-level abstractions, such as RPC, as first-class citizens in a language based on unidirectional message passing. The key is to provide a flexible mechanism for building new communication and synchronization abstractions.

3.3.5 Futures

Various concurrent dialects of **Lisp** and **Scheme** use a mechanism called *futures* [Hal85, KH88] for specifying the parallel evaluation of expressions. Futures combine thread creation, communication and synchronization into a single mechanism. The **Lisp** expression

```
(let
  ((x (future exp)))
  body)
```

evaluates by first spawning a thread to evaluate *exp* and binding a placeholder to *x* in *body*. When the computation of *exp* is complete, the result is put into the placeholder. When a thread attempts to access *x* (called *touching*), it must synchronize on the availability of the value. An important aspect of this mechanism is that any variable can be bound to a future (i.e., touches are implicit), and thus a run-time check is required on every variable access (although compiler optimization can reduce this cost).

Futures are not designed to support concurrent programming, rather they are designed to be a parallel programming mechanism. Their main limitation as a concurrent programming notation is that they only provide one chance for communication and synchronization between the parent and child threads. **Multilisp** [Hal85] provides shared memory and low-level locking mechanisms (essentially test-and-set) for supporting other patterns of communication and synchronization. Using other communication and synchronization mechanisms in conjunction with futures can lead to problems, since some implementations consider it optional as to whether a new thread actually gets spawned for each future (for example, **Mul-T** [KH88]). Although futures might be a useful addition to **ML** to support parallel programming (see Chapter 12), they are not a reasonable base for a concurrent language design.

3.3.6 Message passing and ML

In addition to my own work, there have been several other efforts to integrate message passing and **ML**; e.g., [Hol83a], [Mat89] and [Ram90]. All of these have supported **CSP**-style message passing (i.e., synchronous). Message passing is a useful base for concurrent programming, because it can support the two most common styles of concurrency: *pipelining*, in which threads are arranged in a data-flow network [KM77], and *server-client* interactions. By treating communication channels as infinite streams, the individual threads can be written in an applicative style (e.g., [AB80]), which is consistent with the **ML** style of programming. In fact, **CML** programs tend to use far fewer references than sequential **SML** programs; this point is illustrated in the following two chapters.

A common argument against message passing is that, compared to shared-memory primitives, it provides inferior performance. While this is true for single-processor systems, there is recent empirical evidence that suggests message-passing programs can provide better performance on shared-memory multiprocessors [LS90]. The reason for this is that message-passing programs typically have better locality, and thus map better onto the non-uniform memory structure of modern shared-memory multiprocessors.

3.4 Summary

There are a number of design criteria that can be drawn from the above discussion. A concurrent extension to **ML** should have the following characteristics:

- Dynamic thread creation.
- Synchronous communication on typed channels or ports. Since channels are more general than ports, we prefer them.
- Dynamic channel creation.
- Support for generalized selective communication.

The way existing languages support these mechanisms is not completely satisfactory. The problem is that they support communication by special operations (and often with special syntax) without providing any mechanism for building new communication and synchronization abstractions. In the following chapter, I describe these limitations in more detail, and present my approach to concurrent language design, which addresses them.

Chapter 4

First-class Synchronous Operations

This chapter describes the central result of this dissertation: a new approach to concurrent language design in which synchronous operations are treated as first-class values. I first developed this approach as part of the design of the concurrent language **PML** [Rep88]. **PML** provided a collection of concurrency features similar to those found in **Amber** [Car86]: typed channels, dynamic thread and channel creation, and rendezvous with generalized selective communication. The design of **PML** broke new ground, however, by providing *first-class synchronous operations.*

The basic idea of first-class synchronous operations is to introduce a domain of first-class values, called *events*, for representing synchronous operations. Constructor functions are provided to build base-event values that represent primitive operations such as channel I/O, and combinators are provided to combine event values into higher-level synchronous operations. The design of **CML** [Rep90b, Rep91a] builds on this approach by providing a more powerful version of events. In addition, **CML** provides a number of other features not found in **PML**, such as garbage collection of threads and integrated I/O support.

This chapter is organized chronologically; that is, according to the historical evolution of the language design. First, I introduce a basic set of concurrency primitives, which are similar to what is found in **Amber**. I then motivate and present a subset of **CML**, called **PML** events, that is sufficient to implement the primitives found in **CSP**-style languages. The **PML** subset has limitations, which I use to motivate the extensions that I have developed as part of **CML**. Finally, I summarize the features of **CML** to provide a basis for the examples found in later chapters. I leave the presentation of extended examples to the next chapter, where I present a series of examples of the use of events to build higher-level synchronization and communication abstractions.

4.1 Basic concurrency primitives

We start with a discussion of the basic concurrency operations provided by **CML**. A running **CML** program consists of a collection of *threads*, which use synchronous message passing on typed channels to communicate and synchronize. In keeping with the flavor of **SML**, both threads and channels are created dynamically (initially, a program consists of a single thread). The signature of the basic thread and channel operations is given in Figure 4.1. The function `spawn` takes a function as an argument and creates a new thread

```
val spawn   : (unit -> unit) -> thread_id

val channel : unit -> '1a chan

val accept  : 'a chan -> 'a
val send    : ('a chan * 'a) -> unit
```

Figure 4.1: Basic concurrency primitives

to evaluate the application of the function to the unit value. Channels are also created dynamically using the function `channel`, which is weakly polymorphic.[1] The functions `accept` and `send` are the synchronous communication operations. When a thread wants to communicate on a channel, it must *rendezvous* with another thread that wants to do a complementary communication on the same channel (this is the mechanism described in Section 3.3.2). **SML**'s lexical scoping is used to share channels between threads, and to hide channels from other threads (note, however, that channels can be passed as messages).

A simple example of these primitives is the unique ID service used in the previous chapter. In **CML**, this can be implemented as follows:

```
abstype unique_id_src = UID of int chan
with
  fun makeUIdSrc () = let
        val ch = channel()
        fun loop i = (send(ch, i); loop(i+1))
        in
          spawn (fn () => loop 0);
          UID ch
        end
  fun getUId (UID ch) = accept ch
end
```

This abstraction provides a function for creating a new source of unique IDs (`makeUIdSrc`)

[1] The weak polymorphism is necessary to avoid loop-holes in the type system (see Chapter 8 for details).

and an operation for getting a unique ID from a source (`getUId`). A source of unique IDs is represented by a channel; the function `makeUIdSrc` dynamically creates this channel, and also a thread that sends a stream of unique IDs on the channel. The function `getUId` reads the next ID in the stream. The implementation is an example of how threads can be used to encapsulate state; note that the only side-effects are in the concurrency operations. This style of programming is much more applicative than that of shared-memory primitives (cf., Section 3.2).

4.2 Selective communication vs. abstraction

In Section 3.3, I discussed the arguments for providing generalized selective communication; in this section, I describe a significant limitation with the forms of selective communication found in existing languages.

The problem is that there is a fundamental conflict between selective communication and abstraction. For example, consider a server thread that provides a service via a *request-reply* (or RPC) protocol. The server side of this protocol is something like:

```
fun serverLoop () = if serviceAvailable()
      then let
        val request = accept reqCh
        in
          send (replyCh, doit request);
          serverLoop ()
        end
      else doSomethingElse()
```

where the function `doit` actually implements the service. Note that the service is not always available. This protocol requires that clients obey the following two rules:

1. A client must send a request before trying to read a reply.
2. Following a request the client must read exactly one reply before issuing another request.

If all clients obey these rules, then we can guarantee that each request is answered with the correct reply, but if a client breaks one of these rules, then the requests and replies will be out of sync. An obvious way to improve the reliability of programs that use this service is to bundle the client-side protocol into a function that hides the details, thus ensuring that the rules are followed. The following code implements this abstraction:

```
fun clientCall x = (send(reqCh, x); accept replyCh)
```

While this insures that the protocol is observed, it hides too much. If a client blocks on a call to `clientCall` (e.g., if the server is not available), then it cannot respond to other communications. Avoiding this situation requires using selective communication, but the client cannot do this because the function abstraction hides the synchronous aspect of the protocol. This is the fundamental conflict between selective communication and the existing forms of abstraction. If we make the operation abstract, we lose the flexibility of selective communication; but if we expose the protocol to allow selective communication, we lose the safety and ease of maintenance provided by abstraction. To resolve this conflict requires introducing a new abstraction mechanism that preserves the synchronous nature of the abstraction. First-class synchronous operations provide this new abstraction mechanism.

4.3 First-class synchronous operations

The traditional `select` construct has four facets: the individual I/O operations, the actions associated with each operation, the nondeterministic choice, and the synchronization. The approach of this dissertation is to unbundle these facets by introducing a new type of values, called *events*, that represent synchronous operations. By starting with *base-event* values to represent the communication operations, and providing combinators to associate actions with events and to build nondeterministic choices of events, a flexible mechanism for building new synchronization and communication abstractions is realized. Event values provide a mechanism for building an abstract representation of a protocol without obscuring its synchronous aspect.

To make this concrete, consider the following loop (using an **Amber** style `select` construct [Car86]), which implements the body of an accumulator that accepts either addition or subtraction input commands and offers its contents:

```
fun accum sum = (
      select addCh?x => accum(sum+x)
          or subCh?x => accum(sum-x)
          or readCh!sum => accum sum)
```

The `select` construct consists of three I/O operations: `addCh?x`, `subCh?x`, and `readCh!sum`. For each of these operations there is an associated action on the right hand side of the `=>`. Taken together, each I/O operation and associated action define a clause in a nondeterministic synchronous choice. It is also worth noting that the input clauses define a scope; the input operation binds an identifier to the incoming message, which has the action as its scope.

Figure 4.2 gives the signature of the event operations corresponding to the four facets of generalized selective communication. The functions `receive` and `transmit` build base-

```
val receive  : 'a chan -> 'a event
val transmit : ('a chan * 'a) -> unit event

val choose   : 'a event list -> 'a event
val wrap     : ('a event * ('a -> 'b)) -> 'b event

val sync     : 'a event -> 'a
```

Figure 4.2: Basic event operations

event values that represent channel I/O operations. The `wrap` combinator binds an action, represented by a function, to an event value. And the `choose` combinator composes event values into a nondeterministic choice. The last operation is `sync`, which forces synchronization on an event value. I call this set of operations "**PML** events," since they constitute the mechanism that I originally developed in **PML** [Rep88].

The simplest example of events is the implementation of the synchronous channel I/O operations that were described in the previous section. These are defined using function composition, `sync` and the channel I/O event-value constructors:

```
val accept = sync o receive
val send   = sync o transmit
```

A more substantial example is the accumulator loop from above, which is implemented as:

```
fun accum sum = sync (
      choose [
          wrap (receive addCh, fn x => accum (sum+x)),
          wrap (receive subCh, fn x => accum (sum-x)),
          wrap (transmit (readCh, sum), fn () => accum sum)
        ])
```

Notice how `wrap` is used to associate actions with communications.

The great benefit of this approach to concurrency is that it allows the programmer to create new first-class synchronization and communication abstractions. For example, we can define an event-valued function that implements the client-side of the RPC protocol given in the previous section as follows:

```
fun clientCallEvt x = wrap (transmit(reqCh, x), fn () => accept replyCh)
```

Applying `clientCallEvt` to a value v does not actually send a request to the server, rather it returns an event value that can be used to send v to the server and then accept the server's reply. This event-value can be used in a `choose` expression with other communications; in which case the `transmit` base-event value is in selecting the event. This example shows that we can use first-class synchronous operations to abstract away from the details of the client-server protocol, without hiding the synchronous nature of the protocol.

This approach to synchronization and communication leads to a new programming paradigm, which I call *higher-order concurrent programming.* To understand the higher-order nature of this mechanism, it is helpful to draw an analogy with first-class function values. Table 4.1 relates the features of these two higher-order mechanisms. Values of

Table 4.1: Relating first-class functions and events

Property	**Function values**	**Event values**
Type constructor	`->`	`event`

Introduction	λ-abstraction	`receive`
		`transmit`
		etc.

Elimination	application	`sync`

Combinators	`op o`
	`map`
	etc.
		`choose`
		`wrap`
		etc.

function type are introduced by λ abstraction, while event values are created by the base-event constructors. Function values are eliminated by application, analogously event values are eliminated by the `sync` operator.[2] And both types have combinators for building new values. This analogy does not hold completely, since the various function combinators are derived forms, while the event-value combinators are primitive.

4.4 Other synchronous operations

The event type provides a natural framework for accommodating other primitive synchronous operations.[3] There are three examples of this in **CML**: synchronization on thread termination (sometimes called *process join*), low-level I/O support and time-outs. Figure 4.3 gives the signature of the **CML** base-event constructors for these other synchronous operations. The function `wait` produces an event for synchronizing on the termination of another thread. This is often used by servers that need to release resources allocated to a client in the case that the client terminates unexpectedly. Support for low-level I/O is provided by the functions `syncOnInput` and `syncOnOutput`, which allow threads to synchronize on

[2] "Introduction" and "elimination" are being used in a type theoretic sense. They refer to the syntactic constructs that introduce or eliminate the type constructor.

[3] This is the reason that the I use the term "event" to refer to first-class synchronous operations instead of using "communication."

```
val wait : thread_id -> unit event

val syncOnInput  : int -> unit event
val syncOnOutput : int -> unit event

val waitUntil : time -> unit event
val timeout   : time -> unit event
```

Figure 4.3: Other primitive synchronous operations

the status of file descriptors [UNI86]. These operations are used in **CML** to implement a multi-threaded I/O stream library (Section 10.5.2). There are two functions for synchronizing with the clock: `waitUntil` and `timeout`. The function `waitUntil` returns an event that synchronizes on an absolute time, while `timeout` implements a relative delay. The function `timeout` can be used to implement a timeout in a choice. The following code, for example, defines an event that waits for up to a second for a message on a channel:

```
choose [
    wrap (receive ch, SOME),
    wrap (timeout(TIME{sec=1, usec=0}), fn () => NONE)
  ]
```

By having a uniform mechanism for combining synchronous operations, **CML** provides a great deal of flexibility with a fairly terse mechanism. As a comparison, **Ada** has two different timeout mechanisms: a time entry call for clients and delay statement that servers can include in a select.

4.5 Extending PML events

Thus far, I have described the **PML** subset of first-class synchronous operations. In this section, I motivate and describe two significant extensions to **PML** events that are provided in **CML**.

Consider a protocol consisting of a sequence of communications: $c_1; c_2; \cdots; c_n$. When this protocol is packaged up in an event value, one of the c_i is designated as the *commit point*, the communication by which this event is chosen in a selective communication (e.g., the message send operation in the `clientCallEvt` abstraction above). In **PML** events, the only possible commit point is c_1. The `wrap` construct allows one to tack on $c_2; \cdots; c_n$ after c_1 is chosen, but there is no way to make any of the other c_i the commit point. This asymmetry is a serious limitation to the original mechanism.

A good illustration of this problem is a server that implements an input stream abstraction. Since this abstraction should be smoothly integrated into the concurrency model, the input operations should be event-valued. For example, the function

```
val input : instream -> string event
```

is used to read a single character. In addition, there are other input operations such as `input_line`. Let us assume that the implementation of these operations uses a request-reply protocol; thus, a successful `input` operation involves the communication sequence

send (ch_{req}, REQ_INPUT); accept(ch_{reply})

Packaging this up as an event (as we did in Section 4.3) will make the `send` communication be the commit point, which is the wrong semantics. To illustrate the problem with this, consider the case where a client thread wants to synchronize on the choice of reading a character and a five second timeout:

```
sync (choose [
    wrap (timeout(TIME{sec=5, usec=0}), fn () => raise Timeout),
    input instream
  ])
```

The server might accept the request within the five second limit, even though the wait for input might be indefinite. The right semantics for the `input` operation requires making the `accept` be the commit point, which is not possible using only the **PML** subset of events. To address this limitation, **CML** provides the `guard` combinator.

4.5.1 Guards

The `guard` combinator is the dual of `wrap`; it bundles code to be executed *before* the commit point; this code can include communications. It has the type

```
val guard : (unit -> 'a event) -> 'a event
```

A guard event is essentially a suspension that is forced when `sync` is applied to it. As a simple example of the use of `guard`, the `timeout` function, described above, is actually implemented using `waitUntil` and a guard:

```
fun timeout t = guard (
      fn () => waitUntil (add_time (t, currentTime()))
```

where `currentTime` returns the current time. Some languages support guarded clauses in selective communication, where the guards are boolean expressions that must evaluate to true in order that the communication be enabled. **CML** guards can be used for this purpose too, as illustrated by the following code skeleton:

```
sync (choose [
    ...
    guard (fn () => if pred then evt else choose[])
    ...
  ])
```

Here *evt* is part of the choice only if *pred* evaluates to `true`. Note that the evaluation of *pred* occurs each time the guard function is evaluated.

Returning to the RPC example from above, we can now build an abstract RPC operation with the reply as the commit point. The two different versions are:

```
fun clientCallEvt1 x = wrap (transmit(reqCh, x), fn () => accept replyCh)

fun clientCallEvt2 x = guard (fn () => (send(reqCh, x); receive replyCh)
```

where the `clientCallEvt1` version commits on the server's acceptance of the request, while the `clientCallEvt2` version commits on the server's reply to the request. Note the duality of `guard` and `wrap` with respect to the commit point. Using guards to generate requests like this raises a couple of other problems. First of all, if the server cannot guarantee that requests will be accepted promptly, then evaluating the guard may cause delays. The solution to this is to spawn a new thread to issue the request asynchronously:

```
fun clientCallEvt3 x = guard (fn () => (
      spawn(fn () => send(reqCh, x));
      receive replyCh)
```

Another alternative is for the server to be a clearing-house for requests; spawning a new thread to handle each new request.

The other problem is more serious: what if this RPC event is used in a selective communication and some other event is chosen? How does the server avoid blocking forever on sending a reply? For idempotent services, this can be handled by having the client create a dedicated channel for the reply and having the server spawn a new thread to send the reply. The client side of this protocol is

```
fun clientCallEvt4 x = guard (fn () => let
      val replyCh = channel()
      in
        spawn(fn () => send(reqCh, (replyCh, x)));
        receive replyCh
      end)
```

When the server sends the reply it evaluates

```
spawn (fn () => send(replyCh, reply))
```

If the client has already chosen a different event, then this thread blocks and will be garbage collected. For services that are not idempotent, this scheme is not sufficient; the server needs a way to *abort* the transaction. The `wrapAbort` combinator provides this mechanism and is described in the next section.

4.5.2 Abort actions

The `wrapAbort` combinator associates an *abort action* with an event value. The semantics are that if the event is *not* chosen in a `sync` operation, then a new thread is spawned to evaluate the abort action. The type of this combinator is:

```
val wrapAbort : ('a event * (unit -> unit)) -> unit
```

where the second argument is the abort action. This combinator is the complement of `wrap` in the sense that if you view every base event in a choice as having both a wrapper and an abort action, then, when `sync` is applied, the wrapper of the chosen event is called and threads are spawned for each of the abort actions of the other base events.

Using `wrapAbort`, we can now implement the RPC protocol for non-idempotent services. The client code for the RPC using abort must allocate two channels; one for the reply and one for the abort message:

```
fun clientCallEvt5 x = guard (fn () => let
      val replyCh = channel()
      val abortCh = channel()
      fun abortFn () = send (abortCh, ())
      in
        spawn(fn () => send (reqCh, (replyCh, abortCh, x)));
        wrapAbort (receive replyCh, abortFn)
      end)
```

When the server is ready to reply (i.e., commit the transaction), it synchronizes on the following event value:

```
choose[
    wrap (receive abortCh, fn () => abort the transaction),
    wrap (transmit (replyCh, reply), fn () => commit the transaction)
  ]
```

This mechanism is used to implement the concurrent stream I/O library in **CML** (see Section 10.5.2).

4.6 CML summary

So far, I've touched on the highlights of **CML**'s concurrency mechanisms. In this section, I give a summary of the features of **CML**. This provides the background for the rest of this dissertation. This section is not a language tutorial; for such a discussion see [Rep90b]. Figure 4.4 gives the signature of most of the **CML** concurrency operations, including those already described above.

```
val spawn : (unit -> unit) -> thread_id

val channel : unit -> '1a chan

val sameThread  : (thread_id * thread_id) -> bool
val sameChannel : (channel * channel) -> bool

val accept  : 'a chan -> 'a
val send    : ('a chan * 'a) -> unit

val choose      : 'a event list -> 'a event
val guard       : (unit -> 'a event) -> 'a event
val wrap        : ('a event * ('a -> 'b)) -> 'b event
val wrapHandler : ('a event * (exn -> 'a)) -> 'a event
val wrapAbort   : ('a event * (unit -> unit)) -> 'a event

val sync   : 'a event -> 'a
val select : 'a event list -> 'a
val poll   : 'a event -> 'a option

val always : 'a -> 'a event

val receive   : 'a chan -> 'a event
val transmit  : ('a chan * 'a) -> unit event

val waitUntil : time -> unit event
val timeout   : time -> unit event

val syncOnInput  : int -> unit event
val syncOnOutput : int -> unit event
```

Figure 4.4: **CML** concurrency operations

The two functions `sameThread` and `sameChannel` can be used to test equality of thread IDs and channels. In addition to the `wrap` combinator, the combinator `wrapHandler` wraps an exception handler around an event. For example, `syncOnInput` raises an exception if the file specified by its argument has been closed. Using `wrapHandler`, a more robust version of syncOnInput is defined as:

```
fun waitForInput fd = wrapHandler (
      wrap (syncOnInput fd, fn () => true),
      fn _ => false)
```

Upon synchronization, this returns true if input is available and false if the file is closed.

The operation select is a short-hand for the common idiom of applying sync to a choice of events; i.e.,

```
val select = sync o choose
```

The operation poll is a non-blocking form of sync; it returns NONE in the case that sync would have blocked. This form of polling is different from those of [Rep88], [Rep89] and [Rep91a]. In these earlier versions, polling was handled by constructing polling event values.[4] The semantics of these approaches is more difficult to specify and the implementation is more complicated; furthermore, in practice, the few rare uses of polling have always been in combination with immediate application of sync. For these reasons, I have adopted the simpler polling operation.

The base-event constructor always takes an argument and builds an event that is always available with the argument as its synchronization result. For example, an infinite stream of 1s can be implemented as (always 1). It is useful to compare the function

```
fun poll' evt = select [
        always NONE,
        wrap (evt, SOME)
      ]
```

with poll when supplied to the following function:

```
fun pollLoop pollfn = let
      fun loop () = (case (pollfn (always 1))
           of NONE => loop ()
            | (SOME _) => ())
      in
        loop ()
     end
```

Applying pollLoop to poll' can result in infinite execution sequences, while applying it to poll will always terminate. Section 7.5.5 describes the semantics of the poll function.

[4] Specifically, in [Rep88] and [Rep89] this was done by a special base-event value called anyevent, which was lower priority than other events. In [Rep91a] this was done by a special event-value constructor.

4.6.1 Thread garbage collection

An important property of **CML** programs is the automatic reclamation of concurrency objects (i.e., threads and channels). In general, a thread that communicates infinitely often will block and be garbage collected if it is disconnected from the active part of the system. This property has two benefits. First, it allows threads to be used to implement objects, such as the unique ID source above, without having to worry about termination protocols. If the object representation (i.e., the channels connecting to it) are discarded, then the channels and thread are reclaimed by the garbage collector. Second, it allows use of speculative message passing in complex protocols; i.e., the spawning of a thread to send a message that may never be accepted. If the channel is local to the instance of the protocol, then it is guaranteed to be garbage collected (e.g., `clientCallEvt4` in Section 4.5.1).

4.6.2 Stream I/O

CML provides a concurrent version of the **SML** stream I/O primitives. Input operations in this version are event-valued, which allows them to be used in selective communication. For example, an application may give a user at most 60 seconds to supply a password. This can be programmed as:

```
fun getpasswd () = sync (choose [
        wrap (timeout(TIME{sec=60, usec=0}),
          fn () => NONE),
        wrap (input_line std_in, SOME)
      ])
```

This will return `NONE`, if the user fails to respond within 60 seconds; otherwise it wraps `SOME` around the user's response.

The I/O streams are implemented on top of the other primitives described in this chapter; Section 10.5.2 describes their implementation in some detail.

Chapter 5

Building Concurrency Abstractions

Different applications require different abstractions and programming styles. Modern programming languages provide mechanisms that allow programmers to design and implement the appropriate data and procedural abstractions for their applications, but when it comes to concurrency operations, programmers are stuck with the decisions of the language designer. First-class synchronous operations allow programmers the flexibility to design and implement the right concurrency abstractions for their applications. In this chapter, I demonstrate the utility of first-class synchronous operations by showing how various useful abstractions can be implemented. These abstractions include mechanisms found in other languages, such as asynchronous channels, **Ada**-style rendezvous, and futures. In addition, I present some other abstractions that have proven useful in real applications. These examples also provide further illustration of the use of **CML** as a programming notation. Chapter 9 includes additional examples from applications that are implemented in **CML**.

5.1 Buffered channels

Buffered channels provide a mechanism for asynchronous communication that is similar to the actor mailbox [Agh86]. The source code for this abstraction is given in Figure 5.1. The function `buffer` creates a new buffered channel, which consists of a buffer thread, an input channel and an output channel; the function `bufferSend` is an asynchronous send operation; and the function `bufferReceive` is an event-valued receive operation. The buffer is represented as a queue of messages, which is implemented as a pair of stacks (lists). This example illustrates several key points:

```
abstype 'a buffer_chan = BC of {
    inch : 'a chan,
    outch : 'a chan
  }
with
  fun buffer () = let
        val inCh = channel() and outCh = channel()
        fun loop ([], []) = loop([accept inCh], [])
          | loop (front as (x::r), rear) = select [
                wrap (receive inCh,
                  fn y => loop(front, y::rear)),
                wrap (transmit(outCh, x),
                  fn () => loop(r, rear))
              ]
          | loop ([], rear) = loop(rev rear, [])
        in
          spawn (fn () => loop([], []));
          BC{inch=inCh, outch=outCh}
        end
  fun bufferSend (BC{inch, ...}, x) = send(inch, x)
  fun bufferReceive (BC{outch, ...}) = receive outch
end (* abstype *)
```

Figure 5.1: **CML** implementation of buffered channels

- Buffered channels are a new communication abstraction, which have first-class citizenship. A thread can use the `bufferReceive` function in any context that it could use the built-in function `receive`, such as selective communication.

- The buffer loop uses both input and output operations in its selective communication. This is an example of the necessity of generalized selective communication. If we have only a multiplexed input construct (e.g., **occam**'s `ALT`), then we must to use a request/reply protocol to implement the server side of the `bufferReceive` operation (see pp. 37–41 of [Bur88], for example). But if a request/reply protocol is used, then the `bufferReceive` operation cannot be used in a selective communication by the client.

- The buffer thread is a good example of a common **CML** programming idiom: using threads to encapsulate state. This style has the additional benefit of hiding the state of the system in the concurrency operations, which makes the sequential code cleaner. These threads serve the same role that *monitors* do in some shared-memory concurrent languages.

- This implementation exploits the fact that unreachable blocked threads are garbage collected. If the clients of this buffer discard it, then the buffer thread and channels will be reclaimed by the garbage collector. This improves the modularity of the abstraction, since clients do not have to worry about explicit termination of the buffer thread.

A more complete version of this abstraction is included in the **CML** distribution and is used in a number of applications.

5.2 Multicast channels

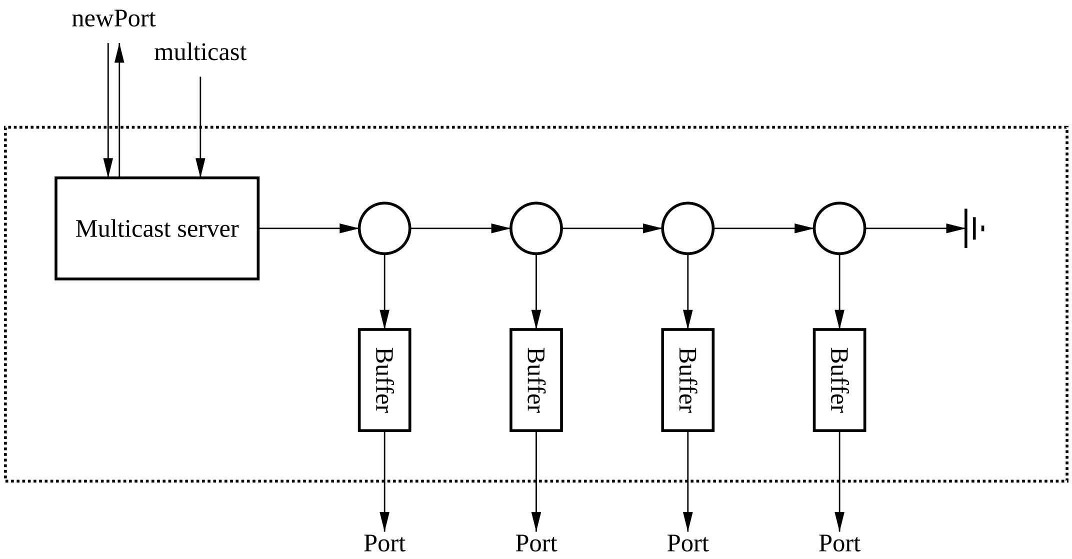
Another useful abstraction is a *buffered multicast channel*, which builds on buffered channels by providing *fan-out*. A multicast channel has a number of *output ports*. When a thread sends a message on a multicast channel, it is replicated once for each output port. In addition to the standard channel operations (create, send and accept), there is an operation to create new ports. The following signature gives the multicast channel interface:

```
type 'a mchan

val mChannel : unit -> '1a mchan
val newPort : 'a mchan -> 'a event
val multicast : ('a mchan * 'a) -> unit
```

New multicast channels are created using `mChannel` and new ports using `newPort`. The `multicast` operation asynchronously broadcasts a message to the ports of a multicast channel. A port is represented by an event value; synchronizing on a port event will return the next multicast message.

A multicast channel consists of a server thread, which initiates the broadcast and creates new ports and a chain of ports. Each port consists of a buffer and a "*tee*" thread that inserts the incoming message in the buffer and propagates it to the next port. The port buffer is implemented using the buffered channel from above. The following picture gives a schematic view of a multicast channel with four ports:



The implementation of the multicast abstraction is given in Figure 5.2. The function `mChannel` is the most interesting, as it includes the code for the server thread. A multicast channel value is represented by a request/reply channel pair that provides an interface to the server thread. A request is either a message to be broadcast, or a request for a new port. The interface between the server thread and the first port in the chain and the interface between a tee thread and the next port is an output function. The output function at the end of the chain is a sink.

5.3 Condition variables

A simple new abstraction is the *condition variable*, which is a *write once* variable.[1] A condition variable is initially *empty*; after a thread writes a value to it, it is *full*. Reading an empty condition variable is a blocking operation, while writing to a full one is an error. In **CML** condition variables have the following interface:

```
type 'a cond_var

val condVar : unit -> '1a cond_var

val readVarEvt : 'a cond_var -> 'a event
val readVar    : 'a cond_var -> 'a

val writeVar : ('a cond_var * 'a) -> unit
exception WriteTwice
```

[1]The name is motivated by the `conditions` found in some shared-memory concurrent languages.

```
abstype 'a mchan = MChan of ('a request chan * 'a event chan)
    and 'a request = Message of 'a | NewPort
with
  fun mChannel () = let
        val reqCh = channel() and respCh = channel()
        fun mkPort outFn = let
              val buf = buffer()
              val inCh = channel()
              fun tee () = let val m = accept inCh
                    in
                      bufferSend(buf, m);
                      outFn m;
                      tee()
                    end
              in
                spawn tee;
                (fn m => send(inCh, m), bufferReceive buf)
              end
        fun server outFn = let
              fun handleReq NewPort = let
                    val (outFn', port) = mkPort outFn
                    in
                      send (respCh, port);
                      outFn'
                    end
                | handleReq (Message m) = (outFn m; outFn)
              in
                server (sync (wrap (receive reqCh, handleReq)))
              end
        in
          spawn (fn () => server (fn _ => ()));
          MChan(reqCh, respCh)
        end

  fun newPort (MChan(reqCh, respCh)) = (
        send (reqCh, NewPort);
        accept respCh)

  fun multicast (MChan(ch, _), m) = send (ch, Message m)

end
```

Figure 5.2: **CML** implementation of multicast channels

Since reading a condition variable is a synchronous operation, an event-valued form of the operation is provided. The exception `WriteTwice` is raised when a thread attempts to write to a full variable. Condition variables are an example of what are called *I-structures* in the parallel language **Id** [ANP89, Nik91]; they can also be regarded as a weak form of *logic variable*.

A condition variable can be implemented in **CML** using a thread to hold the state of the variable, as shown in Figure 5.3. Recent versions of **CML** provide condition variables

```
datatype 'a cond_var = CV of {
    put_ch : 'a chan,
    get_ch : 'a chan
  }

fun condVar () = let
      val putCh = channel() and getCh = channel()
      fun cell () = let
            val v = accept putCh
            fun loop () = (send (getCh, v); loop())
            in
              loop ()
            end
      in
        spawn cell;
        CV{put_ch = putCh, get_ch = getCh}
      end

exception WriteTwice
fun writeVar (CV{put_ch, get_ch}, x) = select [
        wrap (receive get_ch, fn _ => raise WriteTwice),
        transmit (put_ch, x)
      ]

fun readVarEvt (CV{get_ch, ...}) = receive get_ch
fun readVar (CV{get_ch, ...}) = accept get_ch
```

Figure 5.3: **CML** implementation of condition variables

as a primitive concurrency object. This is in part because they are used internally to implement `threadWait`, but also because they provide a significant performance boost (see Chapter 11) in the case that exactly one message must be sent (e.g., for abort messages, see Section 10.5.2).

5.4 Ada-style rendezvous

In this section I describe the implementation of the communication mechanisms found in **Ada** and **Concurrent C**. As described in Section 3.3.4, the basic operation is the extended rendezvous, which consists of an entry call by a client to a server thread. In **Ada** (and **Concurrent C**) this call is asymmetric; i.e., the server can nondeterministically select from a choice of accept clauses, but a client's entry call cannot be involved in a selective communication. There is no problem with supporting entry calls in selective communication in **CML**, but there is the question of which of the two synchronous operations (i.e., sending the request and accepting the reply) should be the commit point. The various alternatives are discussed in some detail in Section 4.5; in this example, I arbitrarily choose the sending of the request as the commit point. Figure 5.4 gives the **CML** code for an abstraction of the basic **Ada** communication mechanism. This implementation is more general than the

```
abstype ('a, 'b) entry = ENTRY of (('a * 'b chan) chan)
with
  fun entry () = ENTRY(channel())
  fun entryCall (ENTRY reqCh) x = guard (fn () => let
        val replyCh = channel()
        in
          wrap (transmit(reqCh, (x, replyCh)), fn () => accept replyCh)
        end)
  fun entryAccept (ENTRY reqCh) =
        wrap (receive reqCh, fn (x, replyCh) => (x, fn y => send(replyCh, y)))
end
```

Figure 5.4: **CML** implementation of **Ada** rendezvous

Ada mechanism in several ways. It allows nested transactions, since the reply channels are dynamically allocated, and it permits selective entry calls, since `entryCall` is an event-valued function. It also allows multiple servers for a given entry. The interface of this abstraction is

```
type ('a,'b) entry
val entry : unit -> ('1a,'1b) entry
val entryCall : ('a,'3b) entry -> 'a -> '3b event
val entryAccept : ('a,'b) entry -> ('a * ('b -> unit)) event
```

Note that the `entryAccept` function returns the entry-call argument and the reply function. As an example of the use of this abstraction, the following is the **CML** version of the implementation of the unique ID server given in Section 3.3.4:

```
fun mkUIdServer () = let
      val e = entry()
      fun loop x = let
            val ((), reply) = sync (entryAccept e)
            in
              reply x; loop (x+1)
            end
      in
        spawn (fn () => loop 0);
        entryCall e
      end
```

In systems programming it is often necessary to deal with the possibility that some expected event might not actually occur. To this end, **Ada** supports several variations on the basic rendezvous mechanism; namely, a *delay* clause in the server's select statement, a *timed* entry call, and a *conditional* entry call. These can be easily implemented in **CML**. A `timeout` event can be used to implement a delay clause; a choice of an entry call and a `timeout` event implements a timed entry call; and applying poll to an entry call implements a conditional entry call.

The language **Concurrent C** provides a couple of additional twists on **Ada**'s rendezvous mechanism. Recall from Section 3.3.4 that **Concurrent C** entry clauses may include a predicate on requests and/or a priority ordering of requests. It is possible to implement these operations in **CML**, but, for reasons discussed below, it is not possible to do so while supporting entry calls in generalized selective communication. As an illustration, I describe the implementation of an entry abstraction that supports conditional acceptance of entry calls based on the argument value. The signature of this abstraction is:

```
type ('a,'b) entry

val entry      : unit -> ('1a,'1b) entry
val condAccept : ('3a,'3b) entry -> ('3a -> bool)
                   -> ('3a * ('3b -> unit)) event
val call       : ('a,'2b) entry -> 'a -> '2b
```

The function `entry` builds a new entry object, the function `condAccept` take an entry object and a predicate and returns an entry event, and the function `call` is used by clients to call an entry. The *lock manager* given in Figure 5.5 is an example of the use of conditional accept (taken from [GR86]). A request to acquire a lock is only accepted if the lock is not currently held. The status of the locks is represented by a list of lock IDs of the currently held locks.

The implementation of the conditional accept abstraction is given in Figure 5.6. An entry object is realized as a buffer thread that matches calls with conditional accept offers.

```
fun lockServer () = let
      val lockEntry = entry()
      val lockReqEvt = condAccept lockEntry
      fun serverLoop locks = let
            fun isLocked id = is id in locks?
            fun unlock id = remove id from locks.
            in
              select [
                  wrap (lockReq isLocked,
                    fn (id, reply) => (reply(); serverLoop (id::locks))),
                  wrap (receive unlockCh,
                    fn id => serverLoop (unlock id))
               ]
            end
      in
        spawn (fn () => (serverLoop []));
        { acquireLock = call lockReqEntry,
          releaseLock = fn id => send(unlockCh, id) }
      end
```

Figure 5.5: A lock manager using conditional accept

Two channels are used to communicate with the buffer thread: clients use the **call_ch** to request an entry call, and the servers[2] use **offer_ch** to offer a conditional acceptance. A call consists of an argument and a reply operation (a curried application of **send** to a reply channel), while an acceptance offer consists of a predicate, a channel for sending a call to the server, and an abort event for notifying the buffer that the offer has been withdrawn. An acceptance offer *matches* a call if the predicate contained in the offer returns true when applied to the argument of the call. The buffer keeps a list of outstanding calls and outstanding offers with the invariant that none of the buffered calls and offers match. When a new call comes in, an attempt is made to match it against an outstanding offer; likewise, when a new offer comes in, an attempt is made to match it against an outstanding call. If a match is actually found, the buffer must also check to see if the offer has been withdrawn. This is done by the function **doMatch**, which synchronizes on the choice of the offer's abort event and transmitting the call to the server. The semantics of abort actions guarantee that exactly one of this choices will be available for selection. The other point of interest is that the buffer thread's main loop synchronizes on the choice of receiving a new call, receiving a new offer, or being notified of the withdrawal of an offer.

This implementation can easily be extended to order requests by some priority function. The timed entry call can also be supported by allowing clients to request that their call be

[2]This abstraction allows multiple servers to share the same entry object.

```
local
  datatype ('a, 'b) offer_t = OFFER of {
      pred      : 'a -> bool,
      req_ch    : ('a * ('b -> unit)) chan,
      abort_evt : unit event
    }
in

abstype ('a, 'b) entry = ENTRY of {
    accept_ch : ('a, 'b) offer_t chan,
    call_ch   : ('a * ('b -> unit)) chan
  }
with
  fun entry () = let
        val acceptCh = channel() and callCh = channel()
        exception NoMatch
        fun buffer (calls, offers) = let
              fun doMatch (call, OFFER{req_ch, abort_evt, ...}) = select [
                      wrap(abort_evt, fn () => false),
                      wrap(transmit(req_ch, call), fn () => true)
                    ]
              fun handleOffer (offer as OFFER{pred, ...}) = let
                    fun matchCall [] = raise NoMatch
                      | matchCall ((call as (x, _)) :: r) = if (pred x)
                          then if (doMatch (call, offer)) then r else (matchCall r)
                          else call :: (matchCall r)
                    val arg = (matchCall calls, offers)
                          handle NoMatch => (calls, offers@[offer])
                    in
                      buffer arg
                    end
              fun handleCall (call as (x, _)) = let
                    fun matchOffer [] = raise NoMatch
                      | matchOffer ((offer as OFFER{pred, ...})::r) =
                          if (pred x)
                            then if (doMatch (call, offer))
                              then r
                              else (matchOffer r)
                            else offer :: (matchOffer r)
                    val arg = (calls, matchOffer offers)
                          handle NoMatch => (calls@[call], offers)
                    in
                      buffer arg
                    end
```

continued...

Figure 5.6: **CML** implementation of conditional entry abstraction

Figure 5.6 (continued)

```
            fun withdraw (OFFER{req_ch = reqCh, ...}) = let
                  fun remove ((off as OFFER{req_ch, ...}) :: r) =
                        if (sameChannel(reqCh, req_ch))
                          then rest
                          else off :: (remove r)
                  in
                    remove offers
                  end
            fun withdrawEvt (offer as OFFER{abort_evt, ...}) =
                  wrap (abort_evt,
                    fn () => buffer (calls, withdraw offer))
          in
            select [
                wrap (receive acceptCh, handleOffer),
                wrap (receive callCh, handleCall),
                choose (map withdrawEvt offers)
              ]
          end
      in
        ENTRY{accept_ch = acceptCh, call_ch = callCh}
      end

  fun condAccept (ENTRY{accept_ch, ...}) pred = guard (
        fn () => let
          val reqCh = channel() and abortCh = channel()
          in
            send (accept_ch, OFFER{
                pred = pred,
                req_ch = reqCh,
                abort_evt = receive abortCh
              });
            wrapAbort (receive reqCh,
              fn () => send(abortCh, ()))
          end)

  fun call (ENTRY{call_ch, ...}) x = let
        val replyCh = channel()
        in
          send (call_ch, (x, fn y => send(replyCh, y)));
          accept replyCh
        end
end (* abstype *)
end (* local *)
```

withdrawn after a timeout. When the buffer receives such a request, if it has not already matched the call with an offer, then it would discard the outstanding call.

The lock manager example given above is cited in [GR86] as an example of the need for the `suchthat` clause. The claim is that without this mechanism, the lock manager requires a separate thread and several transactions per lock and unlock request. The real problem is that **Concurrent C**'s (and **Ada**'s) rendezvous mechanism does not allow a server thread to accept more than one entry call at a time. In **CML**, however, this is not a problem; the lock server can keep a list of pending lock requests for each lock. Thus, the conditional entry abstraction is not a particularly useful one, but as an example it illustrates some interesting points, including the most serious limitation with **CML**'s primitives.

For the basic **Ada** rendezvous, it is possible to implement `entryCall` as an event-valued function, and it would be nice to do the same for this richer abstraction. Unfortunately, this requires a way to insure that three threads (i.e., the client, buffer and server) simultaneously reach agreement (i.e., rendezvous) on the acceptance of a particular call. But, if both the client and server are involved in selective communication, then either might back out at the last minute (i.e., by selecting some other choice). This limitation is not inherent in the mechanism of first-class synchronous operations, but rather is because synchronous channel communication provides only a 2-way rendezvous. If a primitive synchronous operation is supplied for multiway rendezvous[3] [Cha87], then abstractions such as the conditional accept can be supported for generalized communication.

While I have argued that conditional accept is not a useful abstraction, there are other examples where this problem arises. Typically, they involve using a thread to implement a synchronous channel with richer semantics. For example, it might be nice to have a version of channels that logged all messages for debugging purposes. The natural way to do this is to use a thread to implement the logging channel abstraction, but without a 3-way rendezvous the logging channels cannot support generalized selective communication. This problem is a topic for future research.

5.5 Futures

The final example of this chapter is the future mechanism of **Multilisp** (see Section 3.3.5). Since touching a future is a synchronous operation, we represent futures directly as event values. The `future` operation has the type:

```
val future : ('a -> '2b) -> 'a -> '2b event
```

[3]Multiway rendezvous is an instance of the *committee coordination* problem [CM88].

and sync is the touch operator. The implementation of future (see Figure 5.7) is straightforward: we spawn a new thread to evaluate the application and create a condition variable for reporting the result. Since the evaluation of a future might result in a raised exception,

```
fun future f x = let
      datatype 'a msg_t = RESULT of 'a | EXN of exn
      val resVar = condVar()
      in
        spawn (
          fn () => writeVar (resVar, RESULT(f x) handle ex => EXN ex));
        wrap (
          readVarEvt resVar,
          fn (RESULT x) => x | (EXN ex) => raise ex)
      end
```

Figure 5.7: **CML** implementation of futures

the result condition variable (resVar) holds either the result or an exception.

Part III

Theory

Chapter 6

Theory Preliminaries

This part of the dissertation focuses on a small concurrent λ-calculus, called λ_{cv}, that models the significant concurrency mechanisms of **CML**. I present both a dynamic and static semantics for λ_{cv} and prove that the static semantics, which is a polymorphic type discipline, is sound with respect to the dynamic semantics.

Before diving into the semantics of λ_{cv}, it is necessary to review notation. This chapter first describes the basic notation used in this part, a mix of the notations found in [Tof88] and [WF91b], and then introduces the style of semantic specification used by defining the semantics of λ_v, a sequential subset of λ_{cv}.

6.1 Notation

If A and B are sets, then $A \cup B$ is their union, $A \cap B$ is their intersection, and $A \setminus B$ is their difference. The notation $A \xrightarrow{\text{fin}} B$ denotes the set of *finite maps* from A to B (i.e., partial functions with finite domains). If f is a map, then the *domain* and *range* of f are defined as

$$\begin{aligned} \text{dom}(f) &= \{x \mid f(x) \text{ is defined}\} \\ \text{rng}(f) &= \{f(x) \mid x \in \text{dom}(f)\} \end{aligned}$$

The notation

$$\{a_1 \mapsto b_1, \ldots, a_n \mapsto b_n\}$$

denotes a finite map with domain $\{a_1, \ldots, a_n\}$, such that a_i is mapped to b_i; we write $\{\}$ for the map with the empty domain. If f and g are maps, then $f \circ g$ is their *composition*, and $f \pm g$, called f *modified by* g, is a map with domain $\text{dom}(f) \cup \text{dom}(g)$, such that

$$(f \pm g)(x) = \begin{cases} g(x) & \text{if } x \in \text{dom}(g) \\ f(x) & \text{otherwise} \end{cases}$$

The symbol $\pm$ is used because something is added and, when $\text{dom}(g) \cap \text{dom}(f) \neq \emptyset$, something is taken away [Tof88]. It is sometimes useful to view maps as sets of ordered pairs and write $f \subseteq g$, if $\text{dom}(f) \subseteq \text{dom}(g)$ and for $x \in \text{dom}(f)$, $f(x) = g(x)$. In this case, g is called an *extension* of f. If S is a set, then we write $S{+}x$ for $S \cup \{x\}$. When S is a map, and $x = (a, b)$, then $S{+}x$ is only defined if $a \notin \text{dom}(S)$. The operator $+$ associates to the left, so $S{+}y{+}z$ is read as $(S{+}y){+}z$. The notation Fin(S) denotes the set of finite subsets of S (the *finite power set* of S). If ρ is a binary relation, then ρ^* is the reflexive transitive closure of ρ.

6.2 Formal semantics

To further introduce the notation of the following chapters, as well as to survey the important concepts and results related to this style of semantics, the rest of this chapter presents the syntax and semantics of a simple call-by-value λ-calculus. This calculus, which is essentially a polymorphically typed version of Plotkin's λ_v calculus [Plo75], forms the core of the concurrent language presented in the following chapters. I use a style of semantics developed by Wright and Felleisen [WF91b]; other versions of this presentation can be found in many places, such as [DM82] and [Tof88]. The presentation proceeds by first defining the syntax and dynamic semantics of the untyped λ_v calculus, and then defining the standard Hindley-Milner polymorphic type inference system for λ_v [DM82]. Finally, I state, without proof, the standard theorems that relate the static and dynamic semantics.

6.2.1 Syntax of λ_v

The ground terms of λ_v are *variables*, *base constants* and *function constants*:

$$\begin{array}{lcll}
x & \in & \textsc{Var} & \text{variables} \\
b & \in & \textsc{Const} = \textsc{BConst} \cup \textsc{FConst} & \text{constants} \\
 & & \textsc{BConst} = \{\texttt{()}, \texttt{true}, \texttt{false}, 0, 1, \ldots\} & \text{base constants} \\
 & & \textsc{FConst} = \{\texttt{+}, \texttt{-}, \ldots\} & \text{function constants}
\end{array}$$

There are two syntactic classes of terms, expressions ($e \in \textsc{Exp}$) and values ($v \in \textsc{Val} \subset \textsc{Exp}$), defined by the following grammar:

$$\begin{array}{rcll}
e & ::= & v & \text{value} \\
 & | & e_1\ e_2 & \text{application} \\
 & | & \texttt{let}\ x = e_1\ \texttt{in}\ e_2 & \text{let} \\
 \\
v & ::= & b & \text{constant} \\
 & | & x & \text{variable} \\
 & | & \lambda x\,(e) & \lambda\text{-abstraction}
\end{array}$$

Values are the *irreducible* (or *canonical*) terms in the dynamic semantics. The *free variables* of a term are defined inductively:

$$
\begin{array}{rcl}
\mathrm{FV}(b) & = & \emptyset \\
\mathrm{FV}(x) & = & \{x\} \\
\mathrm{FV}(e_1\ e_2) & = & \mathrm{FV}(e_1) \cup \mathrm{FV}(e_2) \\
\mathrm{FV}(\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2) & = & \mathrm{FV}(e_1) \cup (\mathrm{FV}(e_2) \setminus \{x\}) \\
\mathrm{FV}(\lambda x(e)) & = & \mathrm{FV}(e) \setminus \{x\}
\end{array}
$$

A term e is *closed* if $\mathrm{FV}(e) = \emptyset$. A variable is *bound* in a term if it appears as the variable of a `let` or λ. We identify terms up to α-conversion of bound identifiers; for example,

$$\lambda x(x) =_\alpha \lambda x'(x')$$

The *substitution* of a term e' for a variable x' in a term e, where x' is not bound in e, is written as $e[x' \mapsto e']$, and is defined inductively as

$$
\begin{array}{rcl}
b[x' \mapsto e'] & = & b \\
x'[x' \mapsto e'] & = & e' \\
x[x' \mapsto e'] & = & x \quad (x \neq x') \\
(e_1\ e_2)[x' \mapsto e'] & = & e_1[x' \mapsto e']\ e_2[x' \mapsto e'] \\
(\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2)[x' \mapsto e'] & = & \mathtt{let}\ x = e_1[x' \mapsto e']\ \mathtt{in}\ e_2[x' \mapsto e'] \\
(\lambda x(e))[x' \mapsto e'] & = & \lambda x(e[x' \mapsto e'])
\end{array}
$$

Note that, because of the assumption that x' is not bound in e, $x \neq x'$ in the last two cases. This is a reasonable assumption, since bound variables can be renamed. In general, to avoid the problems of free variable capture, we adopt Barendregt's variable convention:

> If $M_1, \ldots, M_n$ occur in a certain mathematical context (e.g., definition, proof), then in these terms all bound variables are chosen to be different from the free variables. (p. 26 of [Bar84])

6.2.2 Dynamic semantics of λ_v

There are a number of different ways to specify the dynamic semantics of programming languages. I use the style of *operational semantics* developed by Felleisen and Friedman [FF86], because it provides a good framework for proving type soundness results [WF91b]. In this approach, the objects of the dynamic semantics are the syntactic terms in Exp.

The meaning of the function constants is defined by a partial function

$$\delta : (\text{FConst} \times \text{BConst}) \rightarrow \text{Const}$$

For example, assuming that $\{\mathtt{not}, \mathtt{+}, \mathtt{1+}\} \subseteq \text{FC}\textsc{onst}$, then

$$\begin{aligned}
\delta(\mathtt{not}, \mathtt{true}) &= \mathtt{false} \\
\delta(\mathtt{+}, \mathtt{1}) &= \mathtt{1+} \\
\delta(\mathtt{1+}, \mathtt{1}) &= \mathtt{2}
\end{aligned}$$

Here `1+` is a special function constant that represents the partial application of `+` to `1`.

An *evaluation context* is a single-hole context where the hole marks the next redex (or is at the top if the term is irreducible) The evaluation contexts of λ_v are defined by the following grammar:

$$E ::= [\,] \mid E\ e \mid v\ E \mid \mathtt{let}\ x = E\ \mathtt{in}\ e$$

The evaluation relation is defined in terms of these contexts.

Definition 6.1 ($\overset{v}{\longmapsto}$) The *evaluation relation* is the smallest relation satisfying the following three rules:

$$\begin{array}{lll r}
E[b\ v] & \overset{v}{\longmapsto} & E[\delta(b, v)] & (\lambda_v\text{-}\delta) \\
E[\lambda x\,(e)\ v] & \overset{v}{\longmapsto} & E[e[x \mapsto v]] & (\lambda_v\text{-}\beta) \\
E[\mathtt{let}\ x = v\ \mathtt{in}\ e] & \overset{v}{\longmapsto} & E[e[x \mapsto v]] & (\lambda_v\text{-}\mathbf{let})
\end{array}$$

It is easily shown that a given expression has a unique evaluation context under these rules, which results in left-to-right call-by-value evaluation; i.e., a function application is evaluated by first evaluating the function position, then the argument position and lastly by applying the function, and similarly for `let` expressions. For example, in the expression

$$\lambda x\,(x\ 1)\,(\lambda y\,(y)\ \lambda z\,(z))$$

the evaluation context is

$$\lambda x\,(x\ 1)\,([\,])$$

and the redex is

$$\lambda y\,(y)\ \lambda z\,(z)$$

As an example of evaluation, consider the following evaluation, where $[\cdots]$ is used to mark the context/redex boundary.

$$\begin{array}{ll}
 & \lambda x\,(1)\ ([\lambda x\,(x\ 10)\ \lambda y\,(y)]) \\
\overset{v}{\longmapsto} & \lambda x\,(1)\ ([\lambda y\,(y)\ 10]) \\
\overset{v}{\longmapsto} & [\lambda x\,(1)\ 10] \\
\overset{v}{\longmapsto} & [1]
\end{array}$$

6.2.3 Typing λ_v

This section describes a standard polymorphic type system for λ_v. The purpose of the type system is to provide a static characterization of the possible results of a computation (e.g., "the expression e evaluates to an integer"). The type system is a deductive proof system that assigns types to λ_v terms. The most interesting aspect of this system is the rule for `let`, which is the source of polymorphism. I start by defining the set of types, then I present the type system and discuss the rule for `let`. Finally, I describe the standard soundness results that hold for this system.

Type terms are built up from *type constants* and *type variables*:

$$\begin{array}{rcll} \alpha & \in & \textsc{TyVar} & \text{type variables} \\ \iota & \in & \textsc{TyCon} = \{\texttt{bool}, \texttt{int}, \ldots\} & \text{type constants} \end{array}$$

The set of types ($\tau \in \textsc{Ty}$) is defined by:

$$\begin{array}{rcll} \tau & ::= & \iota & \text{type constant} \\ & | & \alpha & \text{type variable} \\ & | & (\tau_1 \to \tau_2) & \text{function type} \end{array}$$

and the set of *type schemes* ($\sigma \in \textsc{TyScheme}$) is defined by:

$$\begin{array}{rcl} \sigma & ::= & \tau \\ & | & \forall \alpha . \sigma \end{array}$$

The type schema $\sigma = \forall \alpha_1 . \forall \alpha_2 \cdots \forall \alpha_n . \tau$ is abbreviated as $\forall \alpha_1 \alpha_2 \cdots \alpha_n . \tau$. The type variables $\alpha_1, \ldots, \alpha_n$ are said to be *bound* in σ. A type variable that occurs in τ and is not bound is said to be *free* in σ. We write $\mathrm{FTV}(\sigma)$ for the free type variables of σ. If $\mathrm{FTV}(\tau) = \emptyset$, then τ is said to be a *monotype*. A *type environment* is a finite map from variables to type schemes

$$\mathrm{TE} \in \textsc{TyEnv} = \textsc{Var} \xrightarrow{\text{fin}} \textsc{TyScheme}$$

It is also useful to view a type environment as a finite set of *assumptions* about the types of variables. The set of *free type variables* of a type environment TE is defined to be

$$\mathrm{FTV}(\mathrm{TE}) = \bigcup_{\sigma \in \mathrm{rng}(\mathrm{TE})} \mathrm{FTV}(\sigma)$$

The *closure*, with respect to a type environment TE, of a type τ is defined as

$$\textsc{Clos}_{\mathrm{TE}}(\tau) = \forall \alpha_1 \cdots \alpha_n . \tau$$

where $\{\alpha_1, \ldots, \alpha_n\} = \mathrm{FTV}(\tau) \setminus \mathrm{FTV}(\mathrm{TE})$.

A *substitution* is a map from type variables to types. A substitution S can be naturally extended to map types to types as follows:

$$\begin{array}{rcl} S\iota & = & \iota \\ S\alpha & = & S(\alpha) \\ S(\tau_1 \rightarrow \tau_2) & = & (S\tau_1 \rightarrow S\tau_2) \end{array}$$

Application of a substitution to a type schema respects bound variables and avoids capture. It is defined as:

$$S(\forall\alpha_1 \cdots \alpha_n.\tau) = \forall\beta_1, \ldots, \beta_n.S(\tau[\alpha_i \mapsto \beta_i])$$

where $\beta_i \notin \mathrm{dom}(S) \cup FTV(\mathrm{rng}(S))$. Application of a substitution S to a type environment TE is defined as $S(\mathrm{TE}) = S \circ \mathrm{TE}$. A type τ' is an *instance* of a type scheme $\sigma = \forall\alpha_1 \cdots \alpha_n.\tau$, written $\sigma \succ \tau'$, if there exists a finite substitution, S, with $\mathrm{dom}(S) = \{\alpha_1, \ldots \alpha_n\}$ and $S\tau = \tau'$. If $\sigma \succ \tau'$, then we say that σ is a *generalization* of τ'. Some examples are:

$$\begin{array}{rcl} \forall\alpha.\alpha & \succ & \tau, \text{ for any } \tau \in \textsc{Ty} \\ \forall\alpha,\beta.(\alpha \rightarrow \beta) & \succ & (\alpha \rightarrow \alpha) \\ \forall\alpha,\beta.(\alpha \rightarrow \beta) & \succ & (\alpha \rightarrow \mathtt{int}) \end{array}$$

The typing system is given as a set of rules from which sentences of the form "$\mathrm{TE} \vdash e : \tau$" can be inferred. This sentence is read as "e has the type τ under the set of typing assumptions TE." We write $\vdash e : \tau$ for $\{\} \vdash e : \tau$. To associate types with the constants, we assume the existence of a function

$$\mathrm{TypeOf} : \textsc{Const} \rightarrow \textsc{TyScheme}$$

Figure 6.1 contains the typing rules for λ_v. The rule (τ-**let**), called the *Milner let* rule, plays an important role in this system. It is the rule that introduces polymorphism (via the closure operation), which is the reason for including the `let` construct in λ_v. For example, although the operational semantics of λ_v equates

$$\lambda f((f\ f)\ \mathtt{1})\ \lambda x(x)$$

with

$$\mathtt{let}\ f = \lambda x(x)\ \mathtt{in}\ (f\ f)\ \mathtt{1}$$

The former is untypable, while the latter has the typing

$$\{\} \vdash (\mathtt{let}\ f = \lambda x(x)\ \mathtt{in}\ (f\ f)\ \mathtt{1}) : \mathtt{int}$$

The reason for this is that (τ-**let**) rule assigns to f the type schema $\forall\alpha.(\alpha \rightarrow \alpha)$, which is instantiated to both $(\mathtt{int} \rightarrow \mathtt{int})$ and $((\mathtt{int} \rightarrow \mathtt{int}) \rightarrow (\mathtt{int} \rightarrow \mathtt{int}))$ in the body of the `let`.

$$\frac{\mathrm{TypeOf}(b) \succ \tau}{\mathrm{TE} \vdash b : \tau} \qquad (\tau\textbf{-const})$$

$$\frac{x \in \mathrm{dom}(\mathrm{TE}) \quad \mathrm{TE}(x) \succ \tau}{\mathrm{TE} \vdash x : \tau} \qquad (\tau\textbf{-var})$$

$$\frac{\mathrm{TE} \vdash e_1 : (\tau' \to \tau) \quad \mathrm{TE} \vdash e_2 : \tau'}{\mathrm{TE} \vdash e_1\ e_2 : \tau} \qquad (\tau\textbf{-app})$$

$$\frac{\mathrm{TE} \pm \{x \mapsto \tau\} \vdash e : \tau'}{\mathrm{TE} \vdash \lambda x(e) : (\tau \to \tau')} \qquad (\tau\textbf{-abs})$$

$$\frac{\mathrm{TE} \vdash e_1 : \tau' \quad \mathrm{TE} \pm \{x \mapsto \textsc{Clos}_{\mathrm{TE}}(\tau')\} \vdash e_2 : \tau}{\mathrm{TE} \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau} \qquad (\tau\textbf{-let})$$

Figure 6.1: Type inference rules for λ_v

For this type system to make sense with respect to the set of constants, we place the following restriction on the definition of δ:

If $\mathrm{TypeOf}(b) \succ (\tau' \to \tau)$ and $\vdash v : \tau'$, then $\delta(b, v)$ is defined and $\vdash \delta(b, v) : \tau$.

This restriction insures that any well-typed application of a function constant has a δ reduction. Unfortunately, this restriction rules out some useful function constants, such as integer division, that are not total. In a calculus with exceptions this restriction is unnecessary (see Section 7.5.3 or [WF91b]).

It is worth noting that there is exactly one typing rule for each syntactic form; thus, if we have a proof of $\mathrm{TE} \vdash e : \tau$, for some e, the form of e uniquely specifies which typing rule was the last applied in the deduction. This is the formulation of [Tof88] and differs from the system of [DM82], which has judgements that infer type schemas for expressions and rules for instantiating and generalizing type schemas. A proof of the equivalence of these two systems can be found in [CDDK86].

This type inference system is decidable; there exists an algorithm, called algorithm **W** [DM82] that infers the *principal type* (i.e., most general under the relation $\succ$) of an expression. Algorithm **W** is both sound and complete with respect to the inference system. See [DM82] or [Tof88] for details and proofs.

6.2.4 Properties of typed λ_v

The purpose of static typechecking is to provide compile-time guarantees about the run-time behavior of a program. The most important property of the typing system for λ_v is *type soundness*; i.e., well-typed programs do not have run-time type errors. As with the dynamic semantics, I follow Wright and Felleisen's approach [WF91b], which is a purely syntactic treatment (recall that the objects of the dynamic semantics in Section 6.2.2 are the syntactic terms). Other approaches to this problem can be found in [DM82], [Dam85], and [Tof88]. The key result in the approach of [WF91b] is proving that evaluation preserves types. This is stated in the following *type preservation* lemma:

Lemma 6.1 (Type preservation) If $\mathrm{TE} \vdash e : \tau$ and $e \overset{v}{\longmapsto} e'$, then $\mathrm{TE} \vdash e' : \tau$.

This lemma is also known as *subject reduction.*

An expression $e \notin \mathrm{V{\scriptsize AL}}$ is said to be *stuck* if there is no e' such that $e \overset{v}{\longmapsto} e'$. Because the notion of stuck expressions is a semantic one, Wright and Felleisen define a syntactic notion that is a conservative approximation of the potentially stuck expressions. An expression is *faulty* if it contains a subexpression of the form "$b\ v$," where $\delta(b, v)$ is not defined.[1] The following expression is an example of a faulty expression that cannot become stuck:

$$\lambda x(\mathtt{1})\ \lambda y(\mathtt{true\ 2})$$

Faulty expressions are shown to be untypable in [WF91b].

We say an expression e *diverges*, written $e\Uparrow$, if, for all e' such that $e \overset{v}{\longmapsto}{}^{*} e'$, there exists an e'' such that $e' \overset{v}{\longmapsto} e''$. An expression e *converges* to a value v, written $e\Downarrow v$, if $e \overset{v}{\longmapsto}{}^{*} v$. Given these definitions, the behavior of evaluation is characterized by the following lemma:

Lemma 6.2 (Uniform evaluation) For any closed expression e, either $e\Downarrow v$, $e\Uparrow$, or $e \overset{v}{\longmapsto}{}^{*} e'$, with e' being faulty.

The subject reduction and uniform evaluation lemmas then give us the following theorem:

Theorem 6.3 (Syntactic soundness) If $\vdash e : \tau$, then either $e\Uparrow$, or $e\Downarrow v$ and $\vdash v : \tau$.

To state the soundness of the type system in the traditional way, we define the partial function eval by:

$$\mathrm{eval}(e) = \begin{cases} \mathrm{WRONG} & \text{if } e \overset{v}{\longmapsto}{}^{*} e', \text{ with } e' \text{ being faulty} \\ v & \text{if } e\Downarrow v \end{cases}$$

[1]Examining the evaluation relation and the definition of evaluation contexts, it is clear that the only way that an expression can be stuck is if it has the form $E[b\ v]$, where $\delta(b, v)$ is undefined.

Note that if $e\Uparrow$, then eval(e) is undefined. Using this definition, we can state strong and weak soundness results, which are corollaries of Theorem 6.3.

Theorem 6.4 (Soundness) If $\vdash e : \tau$, then the following hold:

(Strong soundness) if $\text{eval}(e) = v$, then $\vdash v : \tau$

(Weak soundness) $\text{eval}(e) \neq \text{WRONG}$

This theorem means that well-typed programs produce results of the right type (if they terminate) and do not have run-time type errors. See [WF91b] for proof details.

Chapter 7

The Operational Semantics of λ_{cv}

In this chapter, I present the syntax and dynamic semantics of a small concurrent language with first-class synchronous operations. This language, which I call λ_{cv}, is λ_v (from the previous chapter) extended with pairs and the concurrency primitives of **CML**. While λ_{cv} lacks a number of features of **SML** (and thus of **CML**), it embodies the essential concurrency mechanisms of **CML**. In particular, it includes events, channels, the channel I/O event constructors, and the `choose`, `wrap`, `guard`, `wrapAbort` combinators. I also discuss how λ_{cv} might be extended to model additional features found in **CML**, such as exceptions and polling. In Chapter 8, I present the static semantics of λ_{cv}.

7.1 Syntax

As with λ_v, the ground terms consist of variables, base constants and function constants; in addition there are *channel names*. The ground terms are:

$$
\begin{array}{rcll}
x & \in & \textsc{Var} & \text{variables} \\
b & \in & \textsc{Const} = \textsc{BConst} \cup \textsc{FConst} & \text{constants} \\
 & & \textsc{BConst} = \{\,\texttt{()}, \texttt{true}, \texttt{false}, 0, 1, \ldots\} & \text{base constants} \\
 & & \textsc{FConst} = \{\texttt{+}, \texttt{-}, \texttt{fst}, \texttt{snd}, \ldots\} & \text{function constants} \\
\kappa & \in & \textsc{Ch} & \text{channel names}
\end{array}
$$

The sets $\textsc{Var}$, $\textsc{Const}$, and $\textsc{Ch}$ are assumed to be pairwise disjoint. The set $\textsc{FConst}$ includes the following event-valued combinators and constructors:

`choose`, `guard`, `never`, `receive`, `transmit`, `wrap`, `wrapAbort`

In addition to the syntactic classes of expressions, $e \in \textsc{Exp}$, and values, $v \in \textsc{Val}$, λ_{cv} has a syntactic class of *event values*, $ev \in \textsc{Event} \subset \textsc{Val}$. The terms of λ_{cv} are defined by the grammar in Figure 7.1. Pairs have been included to make the handling of two-argument

$$
\begin{array}{rcll}
e & ::= & v & \text{value} \\
 & | & e_1\ e_2 & \text{application} \\
 & | & (e_1 \,.\, e_2) & \text{pair} \\
 & | & \texttt{let}\ x = e_1\ \texttt{in}\ e_2 & \text{let} \\
 & | & \texttt{chan}\ x\ \texttt{in}\ e & \text{channel creation} \\
 & | & \texttt{spawn}\ e & \text{process creation} \\
 & | & \texttt{sync}\ e & \text{synchronization} \\
 \\
v & ::= & b & \text{constant} \\
 & | & x & \text{variable} \\
 & | & (v_1 \,.\, v_2) & \text{pair value} \\
 & | & \lambda x\,(e) & \lambda\text{-abstraction} \\
 & | & \kappa & \text{channel name} \\
 & | & ev & \text{event value} \\
 & | & (\mathbf{G}\ e) & \text{guarded event function} \\
 \\
ev & ::= & \Lambda & \text{never} \\
 & | & \kappa!v & \text{channel output} \\
 & | & \kappa? & \text{channel input} \\
 & | & (ev \Rightarrow v) & \text{wrapper} \\
 & | & (ev_1 \oplus ev_2) & \text{choice} \\
 & | & (ev \mid v) & \text{abort wrapper}
\end{array}
$$

Figure 7.1: Grammar for λ_{cv}

functions easier. Note that the syntactic class of the term $(v_1 \,.\, v_2)$ is either Exp or Val; this ambiguity is resolved in favor of Val. There are three binding forms in this term language: let binding, λ-abstraction and channel creation. Unlike **CML**, new channels are introduced by the special binding form for channel creation. This is done to simplify the presentation of the next chapter, and the `channel` function of **CML** can be defined in terms of λ_{cv} (see Section 7.1.1). The set Val$^{\circ}$ is the set of *closed* value terms (i.e., those without free variables); note, however, that closed values may contain free channel names. The free channel names of an expression e are denoted by FCN(e). Note that, since there are no channel name binding forms, FCN(e) is exactly the set of channel names that appear in e.

Channel names and event values are not part of the concrete syntax of the language; rather, they appear as the intermediate results of evaluation. A *program* is a closed term, which does not contain any guarded event functions (i.e., $(\mathbf{G}\,e)$ terms), or any subterms in the syntactic classes Event or Ch. In other words, programs do not contain intermediate values.

7.1.1 Syntactic sugar

The syntax of λ_{cv} differs from **CML** in several ways, but in many cases the **CML** syntax can be viewed as syntactic sugar for λ_{cv} terms.

CML uses the function `channel` to allocate new channels and provides the more traditional synchronous operations `send` and `accept`. These functions can be used by embedding a λ_{cv} term e in the following context:

```
let channel = λx(chan k in k) in
  let send = λx(sync (transmit x)) in
    let accept = λx(sync (receive x)) in
      [e]
```

The `choose` and `select` functions of **CML** work on lists of events (instead of just pairs). Although λ_{cv} does not have **SML**'s recursive datatypes, event lists can be implemented using the following translation:

$$
\begin{aligned}
[\![\texttt{nil}]\!] &= \texttt{never()} \\
[\![ev\texttt{::}r]\!] &= \texttt{choose}\ (ev\,\texttt{.}\,[\![r]\!])
\end{aligned}
$$

There is no term for sequencing, but we use "$(e_1\texttt{;}\ e_2)$" as syntactic sugar for the term "$\texttt{snd}\ (e_1\texttt{.}e_2)$." Since λ_{cv} uses a left-to-right call-by-value evaluation order, this has the desired semantics.

7.2 Dynamic semantics

The dynamic semantics of λ_{cv} is defined by two evaluation relations: a sequential evaluation relation "$\longmapsto$," and a concurrent evaluation relation "$\Longrightarrow$." The relation "$\longmapsto$" is "$\overset{v}{\longmapsto}$" with a richer δ function and a reduction rule for pairs. Concurrent evaluation is an extension of sequential evaluation to finite sets of processes.

7.2.1 Sequential evaluation

As before, the meaning of the function constants is given by the partial function

$$\delta : \text{FC}\textsc{onst} \times \text{V}\textsc{al}^{\circ} \rightarrow \text{V}\textsc{al}^{\circ}$$

Since a closed value $v \in \text{V}\textsc{al}^{\circ}$ can have free channel names in it, we require, that if $b \in \text{FC}\textsc{onst}$ and $\delta(b, v)$ is defined, then

$$\text{FCN}(\delta(b, v)) \subseteq \text{FCN}(v)$$

In other words, δ is not allowed to introduce new channel names. For the standard built-in function constants, the meaning of δ is the expected one. For example:

$$\begin{array}{rcl}
\delta(\texttt{+}, (\texttt{0}\,.\,\texttt{1})) &=& \texttt{1} \\
\delta(\texttt{+}, (\texttt{1}\,.\,\texttt{1})) &=& \texttt{2} \\
\delta(\texttt{fst}, (v_1\,.\,v_2)) &=& v_1 \\
\delta(\texttt{snd}, (v_1\,.\,v_2)) &=& v_2
\end{array}$$

The meaning of δ is straightforward for most of the event-valued combinators and constructors:

$$\begin{array}{rcl}
\delta(\texttt{never}, (\texttt{)}) &=& \Lambda \\
\delta(\texttt{transmit}, (\kappa\,.\,v)) &=& \kappa!v \\
\delta(\texttt{receive}, \kappa) &=& \kappa? \\
\delta(\texttt{wrap}, (ev\,.\,v)) &=& (ev \Rightarrow v) \\
\delta(\texttt{choose}, (ev_1\,.\,ev_2)) &=& (ev_1 \oplus ev_2) \\
\delta(\texttt{wrapAbort}, (ev\,.\,v)) &=& (ev \mid v)
\end{array}$$

The only complication arises in the case of guarded-event values:

$$\begin{array}{rcl}
\delta(\texttt{guard}, v) &=& (\mathbf{G}\;(v\;(\texttt{)})) \\
\delta(\texttt{wrap}, ((\mathbf{G}\,e)\,.\,v)) &=& (\mathbf{G}\;(\texttt{wrap}\;(e\,.\,v))) \\
\delta(\texttt{choose}, ((\mathbf{G}\,e_1)\,.\,(\mathbf{G}\,e_2))) &=& (\mathbf{G}\;(\texttt{choose}\;(e_1\,.\,e_2))) \\
\delta(\texttt{choose}, ((\mathbf{G}\,e_1)\,.\,ev_2)) &=& (\mathbf{G}\;(\texttt{choose}\;(e_1\,.\,ev_2))) \\
\delta(\texttt{choose}, (ev_1\,.\,(\mathbf{G}\,e_2))) &=& (\mathbf{G}\;(\texttt{choose}\;(ev_1\,.\,e_2))) \\
\delta(\texttt{wrapAbort}, ((\mathbf{G}\,e)\,.\,v)) &=& (\mathbf{G}\;(\texttt{wrapAbort}\;(e\,.\,v)))
\end{array}$$

These rules reflect **guard**'s role as a delay operator; when another event constructor is applied to a guarded event value, then the guard operator $(\mathbf{G})$ is pulled out to delay the event construction.[1]

Like λ_v, evaluation of λ_{cv} is call-by-value, but there is the additional constraint that pairs are evaluated left-to-right. This leads to the following grammar for the evaluation contexts of λ_{cv}:

$$\begin{array}{rcl}
E &::=& [\,] \mid E\;e \mid v\;E \mid (E\,.\,e) \mid (v\,.\,E) \\
&\mid& \texttt{let}\;x\,\texttt{=}\,E\;\texttt{in}\;e \mid \texttt{spawn}\;E \mid \texttt{sync}\;E
\end{array}$$

The following fact about terms and contexts is useful in Chapter 8:

Lemma 7.1 If $E[e]$ is a closed term, then e is a closed term.

Proof. Examining the above definition, it is clear that if x is free in e, then x must also be free in $E[e]$. Hence, $\mathrm{FV}(e) \subseteq \mathrm{FV}(E[e]) = \emptyset$. ■

Definition 7.1 ($\longmapsto$) The *sequential evaluation relation* is the smallest relation "$\longmapsto$" satisfying the following four rules:

[1] In **Algol 60** terminology, $(\mathbf{G}\,e)$ is a *thunk*.

$$
\begin{array}{llll}
E[b\ v] & \longmapsto & E[\delta(b, v)] & (\lambda_{cv}\text{-}\delta) \\
E[\lambda x\,(e)\ v] & \longmapsto & E[e[x \mapsto v]] & (\lambda_{cv}\text{-}\beta) \\
E[\texttt{let}\ x = v\ \texttt{in}\ e] & \longmapsto & E[e[x \mapsto v]] & (\lambda_{cv}\text{-}\mathbf{let}) \\
E[\texttt{sync}\ (\mathbf{G}\, e)] & \longmapsto & E[\texttt{sync}\ e] & (\lambda_{cv}\text{-}\mathbf{guard})
\end{array}
$$

Note that the rule (λ_{cv}-**guard**) forces the expression delayed by `guard`. As usual, $\longmapsto^*$ is the transitive closure of $\longmapsto$. The evaluation of the other new forms (e.g., `spawn`) is defined as part of the concurrent evaluation relation in Section 7.2.3.

7.2.2 Event matching

The key concept in the semantics of concurrent evaluation is the notion of *event matching*, which captures the semantics of rendezvous and communication. Informally, if two processes synchronize on matching events, then they can exchange values and continue evaluation. Before we can make this more formal, we need an auxiliary definition

Definition 7.2 The *abort action* of an event value ev is an expression, which, when evaluated, spawns the abort wrappers of ev. The map

$$\text{AbortAct} : \text{EVENT} \rightarrow \text{EXP}$$

maps an event value to its abort action, and is defined inductively as follows:

$$
\begin{array}{rcl}
\text{AbortAct}(\Lambda) & = & \texttt{()} \\
\text{AbortAct}(\kappa?) & = & \texttt{()} \\
\text{AbortAct}(\kappa!v) & = & \texttt{()} \\
\text{AbortAct}(ev \Rightarrow e) & = & \text{AbortAct}(ev) \\
\text{AbortAct}(ev_1 \oplus ev_2) & = & \texttt{(}\text{AbortAct}(ev_1)\texttt{;}\ \text{AbortAct}(ev_2)\texttt{)} \\
\text{AbortAct}(ev \mid v) & = & \texttt{(}\text{AbortAct}(ev)\texttt{;}\ \texttt{spawn}\ v\texttt{)}
\end{array}
$$

With this definition we can formally define the matching of event values:

Definition 7.3 (Event matching) The matching of event values is defined as a family of binary symmetric relations (indexed by CH). For $\kappa \in \text{CH}$, define

$$ev_1 \overset{\kappa}{\frown} ev_2 \text{ with } (e_1, e_2)$$

(pronounced *"ev_1 matches ev_2 on channel κ with respective results e_1 and e_2*) as the smallest relation satisfying the six inference rules given in Figure 7.2. This relation is abbreviated to $ev_1 \overset{\kappa}{\frown} ev_2$ when the results are unimportant.

An example of event matching is:

$$(\kappa? \Rightarrow \lambda x\texttt{((}x\,\texttt{.}\,x\texttt{))}) \overset{\kappa}{\frown} (\kappa!\texttt{17} \oplus (\kappa? \Rightarrow \lambda x\texttt{()})) \text{ with } (\lambda x\texttt{((}x\,\texttt{.}\,x\texttt{))}\ \texttt{17}, \texttt{()})$$

$$\frac{}{\kappa!v \overset{\kappa}{\frown} \kappa? \text{ with } ((), v)}$$

$$\frac{ev_1 \overset{\kappa}{\frown} ev_2 \text{ with } (e_1, e_2)}{ev_2 \overset{\kappa}{\frown} ev_1 \text{ with } (e_2, e_1)}$$

$$\frac{ev_1 \overset{\kappa}{\frown} ev_2 \text{ with } (e_1, e_2)}{ev_1 \overset{\kappa}{\frown} (ev_2 \Rightarrow v) \text{ with } (e_1, v\ e_2)}$$

$$\frac{ev_1 \overset{\kappa}{\frown} ev_2 \text{ with } (e_1, e_2)}{ev_1 \overset{\kappa}{\frown} (ev_2 \oplus ev_3) \text{ with } (e_1, (\text{AbortAct}(ev_3);\ e_2))}$$

$$\frac{ev_1 \overset{\kappa}{\frown} ev_2 \text{ with } (e_1, e_2)}{ev_1 \overset{\kappa}{\frown} (ev_3 \oplus ev_2) \text{ with } (e_1, (\text{AbortAct}(ev_3);\ e_2))}$$

$$\frac{ev_1 \overset{\kappa}{\frown} ev_2 \text{ with } (e_1, e_2)}{ev_1 \overset{\kappa}{\frown} (ev_2 \mid v) \text{ with } (e_1, e_2)}$$

Figure 7.2: Rules for event matching

Informally, if two processes attempt to synchronize on matching event values, then we can replace the applications of sync with the respective results. This is made more precise in the next section where the concurrent evaluation relation is defined.

Note that event matching is nondeterministic; for example, both

$$\kappa? \overset{\kappa}{\frown} (\kappa!17 \oplus \kappa!29) \text{ with } (17, ())$$

and

$$\kappa? \overset{\kappa}{\frown} (\kappa!17 \oplus \kappa!29) \text{ with } (29, ())$$

It is also worth noting that even if one of the wrappers of an event value is nonterminating, the necessary abort actions for that event will be executed (assuming fair evaluation). This property is important because a common **CML** idiom is to have tail-recursive calls in wrappers (e.g., the buffered channel abstraction in Section 5.1).

7.2.3 Concurrent evaluation

Concurrent evaluation is defined as a transition system between finite sets of process states. This is similar to the style of the "Chemical Abstract Machine" [BB90], except that there are no "cooling" and "heating" transitions (the process sets of this semantics can be thought of as perpetually "hot" solutions). The concurrent evaluation relation extends "$\longmapsto$" to finite sets of terms (i.e., processes) and adds additional rules for process creation, channel creation, and communication. We assume a set of *process identifiers*, and define the set of *processes* and *process sets* as:

$$\begin{array}{rcll} \pi & \in & \textsc{ProcId} & \text{process IDs} \\ p = \langle \pi;\, e \rangle & \in & \textsc{Proc} = (\textsc{ProcId} \times \textsc{Exp}) & \text{processes} \\ \mathcal{P} & \in & \text{Fin}(\textsc{Proc}) & \text{process sets} \end{array}$$

We often write a process as $\langle \pi;\, E[e] \rangle$, where the evaluation context serves the role of the program counter, marking the current state of evaluation.

Definition 7.4 A process set $\mathcal{P}$ is *well-formed* if for all $\langle \pi;\, e \rangle \in \mathcal{P}$ the following hold:

- $\text{FV}(e) = \emptyset$ (e is *closed*), and
- there is no $e' \neq e$, such that $\langle \pi;\, e' \rangle \in \mathcal{P}$.

It is occasionally useful to view well-formed process sets as finite maps from $\textsc{ProcId}$ to $\textsc{Exp}$. If $\mathcal{P}$ is a finite set of process states and $\mathcal{K}$ is a finite set of channel names, then $\mathcal{K}, \mathcal{P}$ is a *configuration.*

Definition 7.5 A configuration $\mathcal{K}, \mathcal{P}$ is *well-formed*, if $\text{FCN}(\mathcal{P}) \subseteq \mathcal{K}$ and $\mathcal{P}$ is well-formed.

The concurrent evaluation relation "$\Longrightarrow$" extends "$\longmapsto$" to configurations, with additional rules for the concurrency operations. It is defined by four inference rules that define single step evaluations. Each concurrent evaluation step affects one or two processes, called the *selected* processes. I first describe each of these rules independently, and then state the formal definition.

The first rule extends the sequential evaluation relation ($\longmapsto$) to configurations:

$$\frac{e \longmapsto e'}{\mathcal{K}, \mathcal{P}+\langle \pi;\, e \rangle \Longrightarrow \mathcal{K}, \mathcal{P}+\langle \pi;\, e' \rangle} \qquad (\lambda_{cv}\text{-}\mapsto)$$

The selected process is π.

The creation of channels requires picking a new channel name and substituting for the variable bound to it:

$$\frac{\kappa \notin \mathcal{K}}{\mathcal{K}, \mathcal{P}+\langle \pi;\, E[\texttt{chan}\ x\ \texttt{in}\ e] \rangle \Longrightarrow \mathcal{K}+\kappa, \mathcal{P}+\langle \pi;\, E[e[x \mapsto \kappa]] \rangle} \qquad (\lambda_{cv}\text{-}\mathbf{chan})$$

Again, π is the selected process.

Process creation requires picking a new process identifier:

$$\frac{\pi' \notin \text{dom}(\mathcal{P}){+}\pi}{\mathcal{K}, \mathcal{P}{+}\langle \pi;\ E[\texttt{spawn}\, v]\rangle \Longrightarrow \mathcal{K}, \mathcal{P}{+}\langle \pi;\ E[\texttt{()}]\rangle{+}\langle \pi';\ v\ \texttt{()}\rangle} \qquad (\lambda_{cv}\textbf{-spawn})$$

This rule has two selected processes: π and π'.

The most interesting rule describes communication and synchronization. If two processes are attempting synchronization on matching events, then they may rendezvous — i.e., exchange a message and continue evaluation:

$$\frac{ev_1 \overset{\kappa}{\frown} ev_2 \text{ with } (e_1, e_2)}{\begin{array}{c}\mathcal{K}, \mathcal{P}{+}\langle \pi_1;\ E_1[\texttt{sync}\ ev_1]\rangle{+}\langle \pi_2;\ E_2[\texttt{sync}\ ev_2]\rangle \\ \Longrightarrow \mathcal{K}, \mathcal{P}{+}\langle \pi_1;\ E_1[e_1]\rangle{+}\langle \pi_2;\ E_2[e_2]\rangle\end{array}} \qquad (\lambda_{cv}\textbf{-sync})$$

The selected processes for this rule are π_1 and π_2. We say that κ is *used* in this transition.

More formally, concurrent evaluation is defined as follows:

Definition 7.6 ($\Longrightarrow$) The *concurrent evaluation relation* is the smallest relation "$\Longrightarrow$" satisfying the rules: (λ_{cv}-$\longmapsto$), (λ_{cv}-**chan**), (λ_{cv}-**spawn**), and (λ_{cv}-**sync**).

Under these rules, processes live forever; i.e., if a process evaluates to a value, it will never again be selected, but it remains in the process set. We could add the following rule, which is similar to the *evaporation* rule of [BB90]:

$$\mathcal{K}, \mathcal{P}{+}\langle \pi;\ [v]\rangle \Longrightarrow \mathcal{K}, \mathcal{P}$$

This rule is not included because certain results are easier to state and prove if the process set is monotonicly increasing.

7.3 Traces

Unlike in the sequential semantics of Section 6.2.2, a program can have many (often infinitely many) different evaluations. Furthermore, there are many interesting programs that do not terminate. Thus some new terminology and notation for describing evaluation sequences is required. This is used to describe some reasonable fairness constraints (see Section 7.4) and to state type soundness results for λ_{cv} (see Chapter 8).

First we note the following properties of $\Longrightarrow$:

Lemma 7.2 If $\mathcal{K}, \mathcal{P}$ is well-formed and $\mathcal{K}, \mathcal{P} \Longrightarrow \mathcal{K}', \mathcal{P}'$ then the following hold:

1. $\mathcal{K}', \mathcal{P}'$ is well-formed

2. $\mathcal{K} \subseteq \mathcal{K}'$

3. $\text{dom}(\mathcal{P}) \subseteq \text{dom}(\mathcal{P}')$

Proof. By examination of the rules for $\Longrightarrow$. ■

Corollary 7.3 The properties of Lemma 7.2 hold for $\Longrightarrow^*$.

Proof. By induction on the length of the evaluation sequence. ■

Note that property (1) implies that evaluation preserves closed terms.

Definition 7.7 A *trace* T is a (possibly infinite) sequence of well-formed configurations

$$T = \langle\!\langle \mathcal{K}_0, \mathcal{P}_0;\ \mathcal{K}_1, \mathcal{P}_1;\ \ldots \rangle\!\rangle$$

such that $\mathcal{K}_i, \mathcal{P}_i \Longrightarrow \mathcal{K}_{i+1}, \mathcal{P}_{i+1}$ (for $i < n$, if T is finite with length n). The *head* of T is $\mathcal{K}_0, \mathcal{P}_0$.

Note that if a configuration $\mathcal{K}_0, \mathcal{P}_0$ is well-formed, then any sequence of evaluation steps starting with $\mathcal{K}_0, \mathcal{P}_0$ is a trace (by Corollary 7.3).

The possible states of a process with respect to a configuration are given by the following definition.

Definition 7.8 Let $\mathcal{P}$ be a well-formed process set and let $p \in \mathcal{P}$, with $p = \langle \pi; e \rangle$. The *state* of π in $\mathcal{P}$ is either *zombie*, *blocked*, or *ready*, depending on the form of e:

- if $e = [v]$, then p is a *zombie*,
- if $e = E[\texttt{sync}\ ev]$ and there does not exist a $\langle \pi'; E'[\texttt{sync}\ ev'] \rangle \in (\mathcal{P} \setminus \{p\})$, such that $ev \overset{\mathcal{K}}{\frown} ev'$, then π is *blocked* in $\mathcal{P}$.
- otherwise, π is *ready* in $\mathcal{P}$.

We define the set of ready processes in $\mathcal{P}$ by

$$\text{Rdy}(\mathcal{P}) = \{\pi \mid \pi \text{ is ready in } \mathcal{P}\}$$

A configuration $\mathcal{K}, \mathcal{P}$ is *terminal* if $\text{Rdy}(\mathcal{P}) = \emptyset$. A terminal configuration with blocked processes is said to be *deadlocked*.

Definition 7.9 A trace is a *computation* if it is maximal; i.e., if it is infinite or if it is finite and ends in a terminal configuration. If e is a program, then we define the computations of e to be

$$\mathrm{Comp}(e) = \{T | T \text{ is a computation with head } \langle \pi_0;\ e \rangle\}$$

Note, I follow the convention of using π_0 as the process identifier of the initial process in a computation of a program.

Definition 7.10 The *set of processes of a trace* T is defined as

$$\mathrm{Procs}(T) = \{\pi \mid \exists \mathcal{K}_i, \mathcal{P}_i \in T \text{ with } \pi \in \mathrm{dom}(\mathcal{P}_i)\}$$

Since a given program can evaluate in different ways, the sequential notions of convergence and divergence are inadequate. Instead, we define convergence and divergence relative to a particular computation of a program.

Definition 7.11 A process $\pi \in \mathrm{Procs}(T)$ *converges* to a value v in T, written $\pi \Downarrow_T v$, if $\mathcal{K}, \mathcal{P} + \langle \pi;\ v \rangle \in T$. We say that π *diverges* in T, written $\pi \Uparrow_T$, if for every $\mathcal{K}, \mathcal{P} \in T$, with $\pi \in \mathrm{dom}(\mathcal{P})$, π is ready or blocked in $\mathcal{P}$.

Divergence includes deadlocked processes and terminating processes that are not evaluated often enough to reach termination, as well as those with infinite loops. It does not include processes with run-time type errors, which are called *stuck* (see Section 8.2.3).

7.4 Fairness

The semantics presented above admits unfair traces, and thus is not adequate as a specification of **CML** implementations. It is necessary to distinguish the acceptable traces. Informally, we require that ready processes make progress and that communication on a single channel is fair (see [Kwi89] for a survey of fairness issues).

A couple of definitions are required before formalizing the notions of fairness. I have already defined the notion of a process being ready in a configuration; a similar definition is required for channels.

Definition 7.12 A channel κ is *enabled* in a configuration $\mathcal{K}, \mathcal{P}$ if there are two distinct processes $\langle \pi;\ E[\texttt{sync}\ ev] \rangle \in \mathcal{P}$ and $\langle \pi';\ E'[\texttt{sync}\ ev'] \rangle \in \mathcal{P}$, such that $ev \overset{\kappa}{\frown} ev'$.

The acceptable computations of a program are defined in terms of fairness restrictions.

Definition 7.13 A computation T is *acceptable* if it ends in a terminal configuration, or if T satisfies the following fairness constraints:

(1) Any process that is enabled infinitely often is selected infinitely often.

(2) Any channel that is enabled infinitely often is used infinitely often.

In the taxonomy of [Kwi89], the first restriction is *strong process fairness* and the second is *strong event fairness*.

An implementation of **CML** should prohibit the possibility of unacceptable computations. In practice this requires that an implementation satisfy some stronger property on finite traces. As an example, consider the following property.

Definition 7.14 A finite trace T of length n is *k–bounded fair* (for k a fixed positive integer), if every intermediate configuration $\mathcal{K}_i, \mathcal{P}_i$, satisfies one of the following (where $m = i + k\,|\mathrm{Rdy}(\mathcal{P}_i)|$):

- $m > n$, or
- for every $\pi \in \mathrm{Rdy}(\mathcal{P}_i)$, π is a selected process at least once in the evaluation subsequence $\mathcal{K}_i, \mathcal{P}_i \Longrightarrow \cdots \Longrightarrow \mathcal{K}_m, \mathcal{P}_m$.

An infinite trace T is *k–bounded fair*, if every finite prefix of T is k–bounded fair.

A k–bounded fair trace obviously satisfies restriction (1) (but not necessarily (2)). The k–bounded fairness restriction is realizable using fairly standard implementation techniques. For example, an implementation that uses fair preemptive scheduling[2] and FIFO queues for the process ready queue and for channel waiting queues will produce only k–bounded fair sequences, where k is determined by the length of the time-slice and speed of the processor. Similar notions can be defined for event fairness.

7.5 Extending λ_{cv}

The language λ_{cv} lacks a number of features found in **CML**; in this section I show how λ_{cv} might be extended to model some of these features. This is not meant to be a complete development of the formal semantics of a more complicated language, rather it is to illustrate that a formal treatment of full **CML** is possible.

[2] By *fair*, I mean that a thread is guaranteed some progress before being preempted.

7.5.1 Recursion

Dynamic process and channel creation is powerful enough to implement the call-by-value $\mathbf{Y}_v$ combinator. This combinator has the following evaluation rule:

$$E[\mathbf{Y}_v\, v] \longmapsto E[v\; \lambda x\,((\mathbf{Y}_v\; v)\;\; x)]$$

The following **CML** code implements $\mathbf{Y}_v$ using only those features found in λ_{cv} (it is adopted from [GMP89]):

```
val Yv  = fn f => let
      val a = channel()
      val g = fn v => let val h' = accept a
                 in
                   spawn (fn () => send(a, h'));
                   f h' v
                 end
      in
        spawn (fn () => send(a, g));
        let val h = accept a
        in
          spawn (fn () => send(a, h));
          f h
        end
      end
```

This code is somewhat mysterious, but what it actually does is fairly simple. The channel a is used to cache the function g for the next iteration of `f`; each time g (renamed `h`) is read from a, a new thread is spawned to send the copy for the next iteration. For **CML**, which is statically typed (see Chapter 8), this definition implements recursion at all imperative types. As an alternative, we could add the $\mathbf{Y}_v$ combinator as a built-in term constructor (as is done in [WF91b]), which would provide recursion at all types.

7.5.2 References

It is well-known that processes and channels can be used to mimic updatable references. The standard technique is to use a process (or thread) to hold the state of the reference cell, with messages to implement reading and writing of the cell. Figure 7.3 gives the **CML** code for this. One can define a formal translation from programs with references to programs using this scheme. This is done in [BMT92], and the translation is shown to be faithful to the expected semantics of references.

The implementation of $\mathbf{Y}_v$ described in Section 7.5.1 is similar to the imperative **Y**–combinator ($\mathbf{Y}_!$) defined by Felleisen [Fel87a]. This suggests the following implementation

```
datatype 'a ref = REF of ('a chan * 'a chan)

fun mkRef initX = let
      val inCh = channel() and outCh = channel()
      fun cell x = sync (choose [
              wrap (transmit (outCh, x), fn () => cell x),
              wrap (receive inCh, fn newX => cell newX)
            ])
      in
        spawn (fn () => cell initX);
        REF(inCh, outCh)
      end

fun assign (REF(inCh, _), x) = send (inCh, x)

fun deref (REF(_, outCh)) = accept outCh
```

Figure 7.3: Implementing references

of references, which uses channels to represent references directly and does not require explicit recursion. Figure 7.4 gives this alternative representation of references. Note that this version of references can be directly coded in λ_{cv}.

```
fun mkRef initX = let val ch = channel()
      in
        spawn (fn () => send (ch, initX));
        ch
      end

fun assign (ch, x) = (accept ch; spawn (fn () => send (ch, x)))

fun deref ch = let val x = accept ch
      in
        spawn (fn () => send (ch, x));
        x
      end
```

Figure 7.4: Implementing references without recursion

7.5.3 Exceptions

One of the most important features of **SML** (and **CML**) is the exception mechanism. **CML** adds further support for exceptions with the `wrapHandler` event-value combinator, which handles exceptions that are raised during evaluation of an event's wrappers. Exceptions are another feature that requires imperative types to achieve sound typing.

Wright and Felleisen provide a semantics of **SML**'s exception mechanism in [WF91b], but applying this technique to λ_{cv} requires some care. The problem is that the soundness of their semantics relies on limiting the scope of exception identifiers to within the scope of their binding site (the rewrite rules allow the binding sites to migrate up to the top of the term, thus expanding the scope of the binding). Since processes can include exceptions in messages, and thus send them out of scope, a different approach is needed. The best approach seems to be to bind exception identifiers in an implicit global environment (as is done with channel names). In the remainder of this section, I sketch the changes to the syntax and semantics of λ_{cv} that are required to support exceptions.

Adding exceptions to λ_{cv} requires a set of *exception names*:

$$ex \in \textsc{ExnName}$$

The syntax of λ_{cv} must be extended to support the declaration, raising, and handling of exceptions. A raised exception is represented by an *exception packet* ($\textsc{Exn} \subset \textsc{Exp}$). Exception packets are irreducible terms, but for technical reasons they are not values. The syntactic extensions are:

$$\begin{array}{rcll}
e & ::= & \texttt{exception}\, x \;\texttt{in}\; e & \text{exception binding} \\
 & | & \texttt{raise}\; e_1\; e_2 & \text{raise exception} \\
 & | & e_1\; \texttt{handle}\; x\; \texttt{with}\; e_2 & \text{exception handler} \\
 & | & exn & \text{exception packet} \\
 & | & \ldots & \\
 & & & \\
exn & ::= & \texttt{[}ex\texttt{,}\; v\texttt{]} & \text{exception packet} \\
 & & & \\
v & ::= & ex & \text{exception name} \\
 & | & \ldots & \\
 & & & \\
ev & ::= & (ev\, \mathbf{H}\, v) & \text{wrapped handler} \\
 & | & \ldots &
\end{array}$$

The terms for exception packets, exception names and wrapped handlers are intermediate forms; i.e., they do not appear in programs.

Sequential evaluation is extended in several ways. Since there is no pattern matching in λ_{cv}, exception matching must be explicitly coded in the semantics. For `wrapHandler`, this means that the wrapper is a pair of the exception name and the handler. This is reflected in the δ–rule for `wrapHandler`:

$$\delta(\texttt{wrapHandler}, \texttt{(}ev\,\texttt{.}\,\texttt{(}ex\,\texttt{.}\,v\texttt{))}) \;=\; (ev\,\mathbf{H}\,\texttt{(}ex\,\texttt{.}\,v\texttt{)})$$

The presence of an exception mechanism means that function constants such as `div` can be supported. Assuming the existence of the exception name $Div \in$ ExnName, then integer division can be defined by:

$$\begin{array}{rcl} \delta(\mathtt{div}, (x\,.\,0)) & = & [Div,\ ()] \\ \delta(\mathtt{div}, (x\,.\,y)) & = & \left\lfloor \frac{x}{y} \right\rfloor \end{array}$$

Additional evaluation contexts for the new syntactic terms are required:

$$\begin{array}{rcl} E & ::= & \mathtt{raise}\ E\ e \ \mid\ \mathtt{raise}\ ex\ E \\ & \mid & e\ \mathtt{handle}\ ex\ \mathtt{with}\ E \ \mid\ E\ \mathtt{handle}\ ex\ \mathtt{with}\ v \\ & \mid & \ldots \end{array}$$

Note that the handler of a `handle` term is evaluated before the body. The sequential evaluation relation (Definition 7.1) must be extended. Most of the new clauses for "$\longmapsto$" are for short circuiting evaluation when an exception is raised and propagating the resulting packet up to a handler. A sampling of these is:

$$\begin{array}{lcl} E[exn\ e] & \longmapsto & E[exn] \\ E[v\ exn] & \longmapsto & E[exn] \\ E[e\ \mathtt{handle}\ ex\ \mathtt{with}\ exn] & \longmapsto & E[exn] \end{array}$$

There are similar rules for pairs, `let`, `sync` and `raise`. The other new clauses have to do with the raising and catching of exception packets:

$$\begin{array}{lcl} E[\mathtt{raise}\ ex\ v] & \longmapsto & E[[ex,\ v]] \\ E[[ex,\ v]\ \mathtt{handle}\ ex\ \mathtt{with}\ v'] & \longmapsto & E[v'\ v] \\ E[[ex,\ v]\ \mathtt{handle}\ ex'\ \mathtt{with}\ v'] & \longmapsto & E[[ex,\ v]] \quad (ex \neq ex') \end{array}$$

As is the case with channel names, the binding of new exception names is left to the concurrent evaluation relation.

The event matching relation (Definition 7.3) must be extended with a clause for wrapped handlers:

$$\frac{ev_1 \overset{\kappa}{\frown} ev_2 \text{ with } (e_1, e_2)}{ev_1 \overset{\kappa}{\frown} (ev_2\ \mathbf{H}\ (ex\,.\,v)) \text{ with } (e_1, e_2\ \mathtt{handle}\ ex\ \mathtt{with}\ v)}$$

Configurations must now include a set of bound exception names. They have the form $\mathcal{K}, \mathcal{X}, \mathcal{P}$, where $\mathcal{X} \subset$ ExnName is a finite set of exception names. A configuration $\mathcal{K}, \mathcal{X}, \mathcal{P}$ is *well-formed*, if $\mathrm{FCN}(\mathcal{P}) \subseteq \mathcal{K}$, $\mathcal{P}$ is well-formed, and any exception name that occurs in $\mathcal{P}$ is in $\mathcal{X}$.

The inference rules for concurrent evaluation relation (Definition 7.6) are modified in light of the new form of configurations. In addition, the concurrent evaluation relation is extended to allow the declaration of exceptions:

$$\frac{ex \notin \mathcal{X}}{\mathcal{K}, \mathcal{X}, \mathcal{P}+\langle\pi;\ E[\mathtt{exception}\ x\ \mathtt{in}\ e]\rangle \Longrightarrow \mathcal{K}, \mathcal{X}+ex, \mathcal{P}+\langle\pi;\ E[e[x \mapsto ex]]\rangle}$$

7.5.4 Process join

CML provides the event constructor `threadWait` that creates an event for synchronizing on the termination of a given thread. There are a couple of ways to extend λ_{cv} to model this. One approach is to define a distinguished set of channel names, $\{\kappa_\pi \mid \pi \in \textsc{ProcId}\}$, to represent process IDs in the dynamic semantics. In this approach, the rule for process creation wraps the body of a process π with code to repeatedly send `()` on the channel κ_π:

$$\frac{\pi' \notin \mathrm{dom}(\mathcal{P})}{\mathcal{K}, \mathcal{P}+\langle\pi;\ E[\mathtt{spawn}\, v]\rangle \Longrightarrow \mathcal{K}+\kappa_{\pi'}, \mathcal{P}+\langle\pi;\ E[\kappa_{\pi'}]\rangle+\langle\pi';\ \mathrm{Fork}(\pi', v)\rangle}$$

where

$$\mathrm{Fork}(\pi, v) = (v\ \mathtt{()}\,\mathtt{;}\ \mathbf{Y}_v\ \lambda f(\mathtt{send}\ (\kappa_\pi\,\mathtt{.}\,\mathtt{()})\,\mathtt{;}\ f\ \mathtt{()})\ \mathtt{()})$$

Waiting for a process' termination is implemented in this scheme by waiting for a message on the process' channel; i.e., `threadWait` is implemented directly by `receive`. While this is a reasonable implementation technique, it has the disadvantage that it becomes hard to distinguish the zombie processes.

A better approach is to support `threadWait` directly. As a side effect of this approach, the event constructor `always` can be directly supported. The direct approach requires adding $\textsc{ProcId}$ to the domain of values and adding two new event value terms:

$$\begin{array}{rcl} v & ::= & \pi \mid \cdots \\ ev & ::= & (\mathbf{W}\,\pi) \mid \mathbf{A} \mid \cdots \end{array}$$

The implementation of the `always` function is defined by the following δ–rule:

$$\delta(\mathtt{always}, v) = (\mathbf{A} \Rightarrow \lambda x\,(v))$$

Matching a base event created by `threadWait` or `always` differs from rendezvous in that only one process is selected. This requires a new relation between an event and a set of processes.

Definition 7.15 Define the ternary relation

$$ev \overset{\mathcal{P}}{\rhd} e$$

(pronounced as "*ev is matched in $\mathcal{P}$ with result e*") as the smallest relation satisfying the inference rules in Figure 7.5.

The concurrent evaluation relation is changed slightly in the case of `spawn`, which now returns the identifier of the new process:

$$\frac{\pi' \notin \mathrm{dom}(\mathcal{P})}{\mathcal{K}, \mathcal{P}+\langle\pi;\ E[\mathtt{spawn}\, v]\rangle \Longrightarrow \mathcal{K}, \mathcal{P}+\langle\pi;\ E[\pi']\rangle+\langle\pi';\ v\ \mathtt{()}\rangle}$$

$$\frac{}{\mathbf{A} \overset{\mathcal{P}}{\rhd} \texttt{()}}$$

$$\frac{\langle\pi;\ [v]\rangle \in \mathcal{P}}{(\mathbf{W}\,\pi) \overset{\mathcal{P}}{\rhd} \texttt{()}}$$

$$\frac{ev \overset{\mathcal{P}}{\rhd} e}{(ev \Rightarrow v) \overset{\mathcal{P}}{\rhd} \texttt{(}v\ e\texttt{)}}$$

$$\frac{ev \overset{\mathcal{P}}{\rhd} e}{(ev \oplus ev') \overset{\mathcal{P}}{\rhd} \texttt{(}\mathrm{AbortAct}(ev')\texttt{;}\ e\texttt{)}}$$

$$\frac{ev \overset{\mathcal{P}}{\rhd} e}{(ev' \oplus ev) \overset{\mathcal{P}}{\rhd} \texttt{(}\mathrm{AbortAct}(ev')\texttt{;}\ e\texttt{)}}$$

$$\frac{ev \overset{\mathcal{P}}{\rhd} e}{(ev \mid v) \overset{\mathcal{P}}{\rhd} e}$$

Figure 7.5: Rules for matching events in process sets

And there is an additional concurrent evaluation rule for `sync` that handles the matching of `threadWait` and `always` events:

$$\frac{ev \overset{\mathcal{P}}{\rhd} e}{\mathcal{K}, \mathcal{P}+\langle\pi;\ \texttt{sync}\ ev\rangle \Longrightarrow \mathcal{K}, \mathcal{P}+\langle\pi;\ e\rangle}$$

7.5.5 Polling

As noted in Section 4.6, **CML** supports a polling mechanism. Recall that the `poll` operation is a non-blocking form of `sync`, which returns `NONE` if `sync` would have blocked, and `SOME` wrapped around the synchronization result otherwise.

It is fairly straightforward to add `poll` to λ_{cv}. To start with, the syntax of expressions and the definition of evaluation contexts is extended with a new form:

$$\begin{array}{rcl} e & ::= & \texttt{poll}\ e \mid \cdots \\ E & ::= & \texttt{poll}\ E \mid \cdots \end{array}$$

Since λ_{cv} does not have the `option` datatype, we need another way to encode the result of polling an event value. To do this, the `poll` operation takes two arguments: an event value to poll and a pair of functions. Informally, the evaluation of $\texttt{poll(}ev\texttt{.(}f\texttt{.}g\texttt{))}$ will either apply f to the result of matching ev, or else it will apply g to `()`.

Since polling is supposed to be non-blocking, we need a formal notion of when synchronizing on an event would block. The following two definitions do this.

Definition 7.16 An event value ev is *offered* by π in a configuration $\mathcal{K}, \mathcal{P}$, if $\mathcal{P}(\pi)$ is of the form $E[\texttt{sync}\ ev]$ or $E[\texttt{poll(}ev\texttt{.}v\texttt{)}]$. The set of offered events in $\mathcal{P}$ is defined to be

$$\mathrm{Offered}(\mathcal{P}) = \{ev \mid \exists \pi \in \mathrm{dom}(\mathcal{P}) \text{ such that } \pi \text{ offers } ev \text{ in } \mathcal{P}\}$$

Definition 7.17 The set of *matched* events in a set of processes $\mathcal{P}$ is defined to be

$$\mathrm{Match}(\mathcal{P}) = \{ev \mid \ \exists ev' \in \mathrm{Offered}(\mathcal{P}) \text{ such that } ev \overset{\kappa}{\frown} ev', \text{ for some } \kappa\ \}$$

And, we need three additional concurrent evaluation rules. The first two handle the transition in which the event is matched by some other process, the third handles the transition for when `sync` would have blocked:

$$\frac{ev \overset{\kappa}{\frown} ev' \text{ with } (e, e')}{\begin{array}{c}\mathcal{K}, \mathcal{P}{+}\langle\pi;\ E[\texttt{poll(}ev\texttt{.(}v_1\texttt{.}v_2\texttt{))}]\rangle{+}\langle\pi';\ E'[\texttt{sync}\ ev']\rangle \\ \Longrightarrow \mathcal{K}, \mathcal{P}{+}\langle\pi;\ E[v_1\ e]\rangle{+}\langle\pi';\ E'[e']\rangle\end{array}}$$

$$\frac{ev \overset{\kappa}{\frown} ev' \text{ with } (e, e')}{\begin{array}{c}\mathcal{K}, \mathcal{P}{+}\langle\pi;\ E[\texttt{poll(}ev\texttt{.(}v_1\texttt{.}v_2\texttt{))}]\rangle{+}\langle\pi';\ E'[\texttt{poll(}ev'\texttt{.(}v_1'\texttt{.}v_2'\texttt{))}]\rangle \\ \Longrightarrow \mathcal{K}, \mathcal{P}{+}\langle\pi;\ E[v_1\ e]\rangle{+}\langle\pi';\ E'[v_1'\ e']\rangle\end{array}}$$

$$\frac{ev \notin \mathrm{Match}(\mathcal{P})}{\mathcal{K}, \mathcal{P}{+}\langle\pi;\ E[\texttt{poll(}ev\texttt{.(}v_1\texttt{.}v_2\texttt{))}]\rangle \Longrightarrow \mathcal{K}, \mathcal{P}{+}\langle\pi;\ E[v_2\ \texttt{()}]\rangle}$$

To make these rules sensible requires the following fairness constraint:

> If $p = \langle\pi;\ E[\texttt{poll(}ev\texttt{.}v\texttt{)}]\rangle$, then a transition $\mathcal{K}, \mathcal{P}{+}p \Longrightarrow \mathcal{K}', \mathcal{P}'{+}p$, is *acceptable* if:
>
> - $ev \in \mathrm{Match}(\mathcal{P})$ and $ev \in \mathrm{Match}(\mathcal{P}')$, or
> - $ev \notin \mathrm{Match}(\mathcal{P})$ and $ev \notin \mathrm{Match}(\mathcal{P}')$.

This constraint captures the notion that `poll` is non-blocking by forcing the polling operation to complete before the state of the polled event can change.

Chapter 8

Typing λ_{cv}

In this chapter, I present a polymorphic type discipline for λ_{cv} and prove that it is sound with respect to the operational semantics presented in the previous chapter. Proofs of the main results are provided in this chapter; additional proof details can be found in the appendix.

CML uses **SML**'s polymorphic type inference system, which is an extension of the one presented in Section 6.2.3 for λ_v. It has been long known that the naïve extension of this system for polymorphic references is unsound [GMW79, Dam85, Tof88]. For example, under the assumptions

$$\begin{array}{rcl} \texttt{ref} & \mapsto & \forall\alpha.(\alpha \rightarrow \alpha\,\texttt{ref}) \\ \texttt{:=} & \mapsto & \forall\alpha.((\alpha \times \alpha\,\texttt{ref}) \rightarrow \texttt{unit}) \\ \texttt{!} & \mapsto & \forall\alpha.(\alpha\,\texttt{ref} \rightarrow \alpha) \end{array}$$

the following erroneous program has the type `bool`:

```
let val r = ref (fn x => x)
in
  r := (fn x => x + 1);
  (! r) true
end
```

Tofte, in [Tof88] and [Tof90], shows that the source of the problem is the rule for `let` bindings. Recall from Figure 6.1 that this rule is

$$\frac{\mathrm{TE} \vdash e_1 : \tau' \quad \mathrm{TE} \pm \{x \mapsto \mathrm{C{\scriptstyle LOS}}_{\mathrm{TE}}(\tau')\} \vdash e_2 : \tau}{\mathrm{TE} \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau} \qquad (\tau\text{-}\mathbf{let})$$

Tofte points out that the closure operation generalizes too many type variables. In particular, there are type variables that are free in the implicit typing of the store, but which are being generalized in the rule for `let`. For example, in the code above, closure causes `r` to be assigned the type scheme $\forall\alpha.(\alpha \rightarrow \alpha)\ \texttt{ref}$ in the body of the `let`, which is instantiated to both $(\texttt{int} \rightarrow \texttt{int})\ \texttt{ref}$ and $(\texttt{bool} \rightarrow \texttt{bool})\ \texttt{ref}$.

Since the typing of the store is undecidable at compile time, a more conservative scheme is necessary to avoid generalization of variables that are free in the store typing. Tofte proposed a system that distinguishes between *applicative* and *imperative* type variables, and between `let` bindings that are *expansive* (i.e., may introduce new store objects) and those that are not. Expansiveness is a syntactic property that conservatively approximates those expressions that introduce new store objects. Basically, irreducible terms, such as abstractions and constants, are non-expansive, and all other terms are expansive. For example, using **SML** notation,

```
let val x = ref 1 in ... end
```

and

```
let val x = 1 + 2 in ... end
```

are both expansive `let` bindings, while

```
let val x = fn x => ref x in ... end
```

is a non-expansive binding. There are two typing rules for `let`: when the binding is expansive, then only the applicative type variables can be generalized; when the binding is non-expansive, then any variable can be generalized.

SML/NJ uses a scheme developed by Dave MacQueen, called *weak types*, to deal with imperative features. The basic idea is to assign a rank to each type variable, which is an approximation of the number of levels of abstraction protecting the variable. When the rank of a variable gets to zero, it must be instantiated to a monotype. Applicative type variables have a rank of infinity, and thus are not weak. It is conjectured, although not proven, that MacQueen's scheme is sound and strictly more polymorphic than Tofte's. Although **CML** inherits this typing scheme from **SML/NJ**, I use Tofte's scheme in this chapter, because it has a well-defined inference system and because it is the type system used in the definition of **SML** [MTH90].

Since channels and processes can be used to implement references (as shown in Section 7.5.2), it is clear that the typing problems of polymorphic references also exist for polymorphic channels. One might naïvely view the implementation of **CML** as a proof of the soundness of polymorphic channels, since it is written in **SML** (plus `callcc`) and it typechecks, but it has recently been discovered that the typing rules given for `callcc` in **SML/NJ** are not sound (Bob Harper, personal communication, July 1991). A simple counter-example (owed to Harper and Lillibridge) is the expression:

```
let val (a, b) = (callcc (fn k =>
          (fn x => x, fn f => throw k (f, fn f => ()))))
in
  print (a "hello");
  b (fn x => x+2)
end
```

The typing of `callcc` has been changed in **SML/NJ** to fix this problem (the correct typing is given in Section 2.3.3). The implementation of **CML**, however, uses the unsound typing,[1] which means that the soundness of polymorphic channels is a serious concern. The remainder of this chapter presents the type system for λ_{cv} and proves it sound.

8.1 Static semantics

The type terms of λ_{cv} are richer than those of λ_v. Let $\iota \in \textsc{TyCon} = \{\texttt{int}, \texttt{bool}, \ldots\}$ designate the *type constants*. Type variables are partitioned into two sets:

$$\begin{array}{rcll} u & \in & \textsc{ImpTyVar} & \text{imperative type variables} \\ t & \in & \textsc{AppTyVar} & \text{applicative type variables} \\ \alpha, \beta & \in & \textsc{TyVar} = \textsc{ImpTyVar} \cup \textsc{AppTyVar} & \text{type variables} \end{array}$$

The set of types, $\tau \in \textsc{Ty}$, is defined by

$$\begin{array}{rcll} \tau & ::= & \iota & \text{type constants} \\ & | & \alpha & \text{type variables} \\ & | & (\tau_1 \rightarrow \tau_2) & \text{function types} \\ & | & (\tau_1 \times \tau_2) & \text{pair types} \\ & | & \tau\ \texttt{chan} & \text{channel types} \\ & | & \tau\ \texttt{event} & \text{event types} \end{array}$$

and the set of type schemes, $\sigma \in \textsc{TyScheme}$, are defined by

$$\begin{array}{rcl} \sigma & ::= & \tau \\ & | & \forall \alpha . \sigma \end{array}$$

As with λ_v, we write $\forall \alpha_1 \cdots \alpha_n . \tau$ for the type scheme $\sigma = \forall \alpha_1 \cdots \forall \alpha_n . \tau$, and write $\mathrm{FTV}(\sigma)$ for the free type variables of σ. We define the set of *imperative types* by

$$\theta \in \textsc{ImpTy} = \{\tau \mid \mathrm{FTV}(\tau) \subset \textsc{ImpTyVar}\}$$

Note that all of the free type variables in an imperative type are imperative.

As with λ_v, type environments assign type schemes to variables in terms. Since we are interested in assigning types to intermediate stages of evaluation, channel names also need

[1] The reasons for this are discussed in Section 10.1.1.

to be assigned types. Therefore, a *typing environment* is a pair of finite maps: a *variable typing* and a *channel typing*:

$$\begin{array}{rcl} \mathrm{VT} & \in & \textsc{VarTy} = \textsc{Var} \xrightarrow{\text{fin}} \textsc{TyScheme} \\ \mathrm{CT} & \in & \textsc{ChanTy} = \textsc{Ch} \xrightarrow{\text{fin}} \textsc{ImpTy} \\ \mathrm{TE} = (\mathrm{VT}, \mathrm{CT}) & \in & \textsc{TyEnv} = (\textsc{VarTy} \times \textsc{ChanTy}) \end{array}$$

We use FTV(VT) and FTV(CT) to denote the sets of free type variables of variable and channel typings, and

$$\mathrm{FTV}(\mathrm{TE}) = \mathrm{FTV}(\mathrm{VT}) \cup \mathrm{FTV}(\mathrm{CT})$$

where $\mathrm{TE} = (\mathrm{VT}, \mathrm{CT})$. Note that there are no bound type variables in a channel typing, and that $\mathrm{FTV}(\mathrm{CT}) \subset \textsc{ImpTyVar}$. The following shorthand is useful for type environment modification:

$$\begin{array}{rcl} \mathrm{TE} \pm \{x \mapsto \sigma\} & \equiv_{def} & (\mathrm{VT} \pm \{x \mapsto \sigma\}, \mathrm{CT}) \\ \mathrm{TE} \pm \{\kappa \mapsto \theta\} & \equiv_{def} & (\mathrm{VT}, \mathrm{CT} \pm \{\kappa \mapsto \theta\}) \end{array}$$

where $x \in \textsc{Var}$, $\kappa \in \textsc{Ch}$, and $\mathrm{TE} = (\mathrm{VT}, \mathrm{CT})$.

Because of the need to preserve imperative types, we require that substitutions map imperative type variables to imperative types. As before, we allow substitutions to be applied to types and type environments.

Definition 8.1 A type τ' is an *instance* of a type scheme $\sigma = \forall \alpha_1 \cdots \alpha_n.\tau$, written $\sigma \succ \tau'$, if there exists a finite substitution, S, with $\mathrm{dom}(S) = \{\alpha_1, \ldots, \alpha_n\}$ and $S\tau = \tau'$. If $\sigma \succ \tau'$, then we say that σ is a *generalization* of τ'. We say that $\sigma \succ \sigma'$ if whenever $\sigma' \succ \tau$, then $\sigma \succ \tau$.

Definition 8.2 The *closure* of a type τ with respect to a type environment TE is defined as: $\textsc{Clos}_{\mathrm{TE}}(\tau) = \forall \alpha_1 \cdots \alpha_n.\tau$, where

$$\{\alpha_1, \ldots, \alpha_n\} = \mathrm{FTV}(\tau) \setminus \mathrm{FTV}(\mathrm{TE})$$

And the *applicative closure* of τ is defined as: $\textsc{AppClos}_{\mathrm{TE}}(\tau) = \forall \alpha_1 \cdots \alpha_n.\tau$, where

$$\{\alpha_1, \ldots, \alpha_n\} = (\mathrm{FTV}(\tau) \setminus \mathrm{FTV}(\mathrm{TE})) \cap \textsc{AppTyVar}$$

The following important facts about type closure and generalization are used later:

Lemma 8.1 The following two properties hold for any TE, σ, σ', τ, and x:

- If $\sigma \succ \sigma'$, then $\textsc{Clos}_{\mathrm{TE}\pm\{x\mapsto\sigma\}}(\tau) \succ \textsc{Clos}_{\mathrm{TE}\pm\{x\mapsto\sigma'\}}(\tau)$.

- If $x \notin \mathrm{dom}(\mathrm{TE})$, then $\textsc{Clos}_{\mathrm{TE}}(\tau) \succ \textsc{Clos}_{\mathrm{TE}\pm\{x\mapsto\sigma\}}(\tau)$.

Proof. These both follow from the observation that if $\mathrm{FTV}(\mathrm{TE}) \subseteq \mathrm{FTV}(\mathrm{TE}')$ then $\textsc{Clos}_{\mathrm{TE}}(\tau) \succ \textsc{Clos}_{\mathrm{TE}'}(\tau)$ ■

8.1.1 Expression typing rules

As before, the function TypeOf assigns types to the constants. For the concurrency related constants, TypeOf assigns the following type schemes:

$$\begin{array}{lcl}
\texttt{never} & : & \forall\alpha.(\texttt{unit} \to \alpha\ \texttt{event}) \\
\texttt{receive} & : & \forall\alpha.(\alpha\ \texttt{chan} \to \alpha\ \texttt{event}) \\
\texttt{transmit} & : & \forall\alpha.((\alpha\ \texttt{chan} \times \alpha) \to \texttt{unit event}) \\
\texttt{wrap} & : & \forall\alpha\beta.((\alpha\ \texttt{event} \times (\alpha \to \beta)) \to \beta\ \texttt{event}) \\
\texttt{choose} & : & \forall\alpha.((\alpha\ \texttt{event} \times \alpha\ \texttt{event}) \to \alpha\ \texttt{event}) \\
\texttt{guard} & : & \forall\alpha.((\texttt{unit} \to \alpha\ \texttt{event}) \to \alpha\ \texttt{event}) \\
\texttt{wrapAbort} & : & \forall\alpha.((\alpha\ \texttt{event} \times (\texttt{unit} \to \texttt{unit})) \to \alpha\ \texttt{event})
\end{array}$$

We also assume that there are no event-valued constants. More formally, we require that there does not exist any b such that $\mathrm{TypeOf}(b) = \tau\ \texttt{event}$, for some type τ.

The typing rules for λ_{cv} are divided into two groups. The core rules are given in Figure 8.1. These are a modification of the rules in Figure 6.1. There are two rules for `let`: the rule (**τ-app-let**) applies in the non-expansive case (in the syntax of λ_{cv}, this is when the bound expression is in $\textsc{Val}$); the rule (**τ-imp-let**) applies when the expression is expansive (not a value). There are also rules for typing channel names, and pair expressions. The rule (**τ-chan**) restricts the type of the introduced channel to be imperative. In addition to these core typing rules, there are rules for the other syntactic forms (see Figure 8.2). Given the appropriate environment, these rules can be derived from rule (**τ-app**) (rule (**τ-const**) in the case of Λ). It is useful, however, to include them explicitly. As before, it is worth noting that the syntactic form of a term uniquely determines which typing rule applies.

In order that the typing of constants be sensible, we impose a typability restriction on the definitions of δ and TypeOf. If $\mathrm{TypeOf}(b) \succ (\tau' \to \tau)$ and $\mathrm{TE} \vdash v : \tau'$, then $\delta(b, v)$ is defined and $\mathrm{TE} \vdash \delta(b, v) : \tau$. It is worth noting that the δ rules we defined for the concurrency constants respect this restriction.

The following lemma defines a derived typing rule for the sequencing syntax:

Lemma 8.2 The typing rule for the sequencing is

$$\frac{\mathrm{TE} \vdash e_1 : \tau_1 \quad \mathrm{TE} \vdash e_2 : \tau_2}{\mathrm{TE} \vdash \texttt{(}e_1\texttt{;}\ e_2\texttt{)} : \tau_2}$$

$$\frac{\mathrm{TypeOf}(b) \succ \tau}{\mathrm{TE} \vdash b : \tau} \qquad (\tau\text{-}\mathbf{const})$$

$$\frac{x \in \mathrm{dom}(\mathrm{VT}) \quad \mathrm{VT}(x) \succ \tau}{(\mathrm{VT}, \mathrm{CT}) \vdash x : \tau} \qquad (\tau\text{-}\mathbf{var})$$

$$\frac{\mathrm{CT}(\kappa) = \theta}{(\mathrm{VT}, \mathrm{CT}) \vdash \kappa : \theta} \qquad (\tau\text{-}\mathbf{chvar})$$

$$\frac{\mathrm{TE} \vdash e_1 : (\tau' \to \tau) \quad \mathrm{TE} \vdash e_2 : \tau'}{\mathrm{TE} \vdash e_1\ e_2 : \tau} \qquad (\tau\text{-}\mathbf{app})$$

$$\frac{\mathrm{TE} \pm \{x \mapsto \tau\} \vdash e : \tau'}{\mathrm{TE} \vdash \lambda x\,(e)\ : (\tau \to \tau')} \qquad (\tau\text{-}\mathbf{abs})$$

$$\frac{\mathrm{TE} \vdash e_1 : \tau_1 \quad \mathrm{TE} \vdash e_2 : \tau_2}{\mathrm{TE} \vdash (e_1 \,.\, e_2) : (\tau_1 \times \tau_2)} \qquad (\tau\text{-}\mathbf{pair})$$

$$\frac{\mathrm{TE} \vdash v : \tau' \quad \mathrm{TE} \pm \{x \mapsto \mathrm{C{\scriptsize LOS}}_{\mathrm{TE}}(\tau')\} \vdash e : \tau}{\mathrm{TE} \vdash \mathtt{let}\ x = v\ \mathtt{in}\ e : \tau} \qquad (\tau\text{-}\mathbf{app\text{-}let})$$

$$\frac{\mathrm{TE} \vdash e_1 : \tau' \quad \mathrm{TE} \pm \{x \mapsto \mathrm{A{\scriptsize PP}C{\scriptsize LOS}}_{\mathrm{TE}}(\tau')\} \vdash e_2 : \tau}{\mathrm{TE} \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : \tau} \qquad (\tau\text{-}\mathbf{imp\text{-}let})$$

$$\frac{\mathrm{TE} \pm \{x \mapsto \theta\ \mathtt{chan}\} \vdash e : \tau}{\mathrm{TE} \vdash \mathtt{chan}\ x\ \mathtt{in}\ e : \tau} \qquad (\tau\text{-}\mathbf{chan})$$

Figure 8.1: Core type inference rules for λ_{cv}

$$\frac{\mathrm{TE} \vdash e : (\mathtt{unit} \rightarrow \tau)}{\mathrm{TE} \vdash \mathtt{spawn}\ e : \mathtt{unit}} \qquad (\tau\textbf{-spawn})$$

$$\frac{\mathrm{TE} \vdash e : \tau\ \mathtt{event}}{\mathrm{TE} \vdash \mathtt{sync}\ e : \tau} \qquad (\tau\textbf{-sync})$$

$$\frac{\mathrm{TE} \vdash e : \tau\ \mathtt{event}}{\mathrm{TE} \vdash (\mathbf{G}\ e) : \tau\ \mathtt{event}} \qquad (\tau\textbf{-guard})$$

$$\frac{\forall \alpha . \alpha\ \mathtt{event} \succ \tau}{\mathrm{TE} \vdash \Lambda : \tau} \qquad (\tau\textbf{-never})$$

$$\frac{\mathrm{TE} \vdash \kappa : \tau\ \mathtt{chan} \quad \mathrm{TE} \vdash v : \tau}{\mathrm{TE} \vdash \kappa\mathbf{!}v : \mathtt{unit\ event}} \qquad (\tau\textbf{-output})$$

$$\frac{\mathrm{TE} \vdash \kappa : \tau\ \mathtt{chan}}{\mathrm{TE} \vdash \kappa\mathbf{?} : \tau\ \mathtt{event}} \qquad (\tau\textbf{-input})$$

$$\frac{\mathrm{TE} \vdash ev : \tau'\ \mathtt{event} \quad \mathrm{TE} \vdash e : (\tau' \rightarrow \tau)}{\mathrm{TE} \vdash (ev \Rightarrow e) : \tau\ \mathtt{event}} \qquad (\tau\textbf{-wrap})$$

$$\frac{\mathrm{TE} \vdash ev_1 : \tau\ \mathtt{event} \quad \mathrm{TE} \vdash ev_2 : \tau\ \mathtt{event}}{\mathrm{TE} \vdash (ev_1 \oplus ev_2) : \tau\ \mathtt{event}} \qquad (\tau\textbf{-choice})$$

$$\frac{\mathrm{TE} \vdash ev : \tau\ \mathtt{event} \quad \mathrm{TE} \vdash v : (\mathtt{unit} \rightarrow \mathtt{unit})}{\mathrm{TE} \vdash (ev \mid v) : \tau\ \mathtt{event}} \qquad (\tau\textbf{-abort})$$

Figure 8.2: Other type inference rules for λ_{cv}

Proof. This follows from the definition of sequencing and the type rules above:

$$\dfrac{\dfrac{\text{TypeOf}(\texttt{snd}) \succ ((\tau_1 \times \tau_2) \rightarrow \tau_2)}{\text{TE} \vdash \texttt{snd} : ((\tau_1 \times \tau_2) \rightarrow \tau_2)} \quad \dfrac{\text{TE} \vdash e_1 : \tau_1 \quad \text{TE} \vdash e_2 : \tau_2}{\text{TE} \vdash \texttt{(}e_1\,\texttt{.}\,e_2\texttt{)} : (\tau_1 \times \tau_2)}}{\text{TE} \vdash \texttt{snd}\ \texttt{(}e_1\,\texttt{.}\,e_2\texttt{)} : \tau_2}$$

■

8.1.2 Process typings

A *process typing* is a finite map from process identifiers to types:

$$\text{PT} \in \text{ProcTy} = \text{ProcId} \xrightarrow{\text{fin}} \text{Ty}$$

Typing judgements are extended to process configurations by the following definition.

Definition 8.3 A well-formed configuration $\mathcal{K}, \mathcal{P}$ has type PT under a channel typing CT, written

$$\text{CT} \vdash \mathcal{K}, \mathcal{P} : \text{PT}$$

if the following hold:

- $\mathcal{K} \subseteq \text{dom}(\text{CT})$,
- $\text{dom}(\mathcal{P}) \subseteq \text{dom}(\text{PT})$, and
- for every $\langle \pi; e \rangle \in \mathcal{P}$, $(\{\}, \text{CT}) \vdash e : \text{PT}(\pi)$.

For **CML**, where spawn requires a $(\texttt{unit} \rightarrow \texttt{unit})$ argument, the process typing is $\text{PT}(\pi) = \texttt{unit}$ for all $\pi \in \text{dom}(\mathcal{P})$.

8.2 Type soundness

This section presents a proof of the soundness of the type system given in Section 8.1 with respect to the dynamic semantics of Section 7.2. As discussed in Section 6.2.4, I use the approach of [WF91b]. The basic idea is to show that evaluation preserves types (also called *subject reduction*); then characterize run-time type errors (called "*stuck states*") and show that stuck states are untypable. This allows us to conclude that well-typed programs cannot go wrong.

8.2.1 The Substitution and Replacement lemmas

Before we can prove the main results, we need several important lemmas. The following lemma states that any variable or channel name in the domain of a typing environment, which is not free in an expression e, can be ignored in the typing of e.

Lemma 8.3 If $x \notin \mathrm{FV}(e)$, then $\mathrm{TE} \vdash e : \tau$ *iff* $\mathrm{TE} \pm \{x \mapsto \sigma\} \vdash e : \tau$. Likewise, if $\kappa \notin \mathrm{FCN}(e)$, then $\mathrm{TE} \vdash e : \tau$ *iff* $\mathrm{TE} \pm \{\kappa \mapsto \theta\} \vdash e : \tau$.

Proof. The proof is a straightforward induction on the height of the typing deduction. ∎

Note that the variable convention insures that $x \notin \mathrm{FV}(e)$ whenever this lemma applies.

The following lemma is very important; it allows us to replace a subexpression with another expression of the same type, without affecting the type of the whole term.

Lemma 8.4 (Replacement) Let $C[]$ be a single-hole context. If the following hold:

1. $\mathcal{D}$ is a type deduction concluding $\mathrm{TE} \vdash C[e_1] : \tau$,
2. $\mathcal{D}_1$ is a subdeduction of $\mathcal{D}$, which concludes $\mathrm{TE}' \vdash e_1 : \tau'$,
3. $\mathcal{D}_1$ occurs in $\mathcal{D}$ in the position corresponding to the hole in C, and
4. $\mathrm{TE}' \vdash e_2 : \tau'$,

then $\mathrm{TE} \vdash C[e_2] : \tau$.

Proof. The basic idea is that the term $C[e_1]$ and type deduction $\mathcal{D}$ have isomorphic structure, thus the replacement of e_1 by e_2 is paralleled by a replacement of the deduction of $\mathrm{TE}' \vdash e_1 : \tau'$ by the deduction of $\mathrm{TE}' \vdash e_2 : \tau$, giving the deduction of $\mathrm{TE} \vdash C[e_2] : \tau$. This is proven by induction on the structure of the deduction. See [HS86] or [WF91b] for detailed proofs. ∎

The following lemma essentially says that β-reduction preserves types.

Lemma 8.5 (Substitution) If $x \notin \mathrm{FV}(v)$, $\mathrm{TE} \vdash v : \tau$, and

$$\mathrm{TE} \pm \{x \mapsto \forall \alpha_1 \cdots \alpha_n . \tau\} \vdash e : \tau'$$

with $\{\alpha_1, \ldots, \alpha_n\} \cap \mathrm{FTV}(\mathrm{TE}) = \emptyset$, then $\mathrm{TE} \vdash e[x \mapsto v] : \tau'$.

Proof. The proof is by induction on the height of the typing deduction; the detailed proof is given in the appendix. ∎

The following lemma is useful in showing that spawn preserves types.

Lemma 8.6 If $(\mathrm{VT}, \mathrm{CT}) \vdash e : \tau$ and $\mathrm{FV}(e) = \emptyset$, then $(\{\}, \mathrm{CT}) \vdash e : \tau$.

Proof. This is a more specific version of Lemma 8.3 and follows immediately. ∎

8.2.2 Subject reduction

We are now ready to state and prove the first subject reduction theorem, which says that sequential evaluation preserves types.

Theorem 8.7 (Sequential type preservation) For any type environment TE, expression e_1 and type τ, such that $\mathrm{TE} \vdash e_1 : \tau$, if $e_1 \longmapsto e_2$ then $\mathrm{TE} \vdash e_2 : \tau$.

Proof. Let $E[e] = e_1$ and $E[e'] = e_2$, and assume that $\mathrm{TE}' \vdash e : \tau'$ with $\mathrm{TE}' = (\mathrm{VT}', \mathrm{CT})$. Then, by the Replacement Lemma (8.4), it is sufficient to show that $\mathrm{TE}' \vdash e' : \tau'$. This is done by case analysis of the definition of $\longmapsto$ (i.e., the structure of e).

Case $E[b\ v] \longmapsto E[\delta(b, v)]$**.**
Rules (τ-**app**) and (τ-**const**) apply:

$$\frac{\dfrac{\mathrm{TypeOf}(b) \succ (\tau \to \tau')}{\mathrm{TE}' \vdash b : (\tau \to \tau')} \qquad \mathrm{TE}' \vdash v : \tau}{\mathrm{TE}' \vdash b\ v : \tau'}$$

Thus, by the typability restriction on δ, we have $\mathrm{TE}' \vdash \delta(b, v) : \tau'$.

Case $E[\lambda x\,(e)\ v] \longmapsto E[e[x \mapsto v]]$**.**
Rules (τ-**app**) and (τ-**abs**) apply:

$$\frac{\dfrac{\mathrm{TE}' \pm \{x \mapsto \tau''\} \vdash e : \tau'}{\mathrm{TE}' \vdash \lambda x\,(e) : (\tau'' \to \tau')} \quad \mathrm{TE}' \vdash v : \tau''}{\mathrm{TE}' \vdash \lambda x\,(e)\ v : \tau'}$$

Applying the Substitution Lemma (8.5), gives us

$$\mathrm{TE}' \vdash e[x \mapsto v] : \tau'$$

Case $E[\mathtt{let}\ x = v\ \mathtt{in}\ e] \longmapsto E[e[x \mapsto v]]$**.**
Rule (τ-**app-let**) applies:

$$\frac{\mathrm{TE}' \vdash v : \tau'' \quad \mathrm{TE}' \pm \{x \mapsto \mathrm{CLOS}_{\mathrm{TE}'}(\tau'')\} \vdash e : \tau'}{\mathrm{TE}' \vdash \mathtt{let}\ x = v\ \mathtt{in}\ e : \tau'}$$

Let $\text{Clos}_{\text{TE}'}(\tau'') = \forall\alpha_1 \cdots \alpha_n.\tau''$, then, by definition,

$$\{\alpha_1, \ldots, \alpha_n\} \cap \text{FTV}(\text{TE}') = \emptyset$$

Then, by the Substitution Lemma (8.5), we get

$$\text{TE}' \vdash e[x \mapsto v] : \tau'$$

Case $E[\texttt{sync}\ (\mathbf{G}\, e)] \longmapsto E[\texttt{sync}\ e]$**.**

Rules (**τ-sync**) and (**τ-guard**) apply:

$$\cfrac{\cfrac{\text{TE}' \vdash e : \tau'\ \texttt{event}}{\text{TE}' \vdash (\mathbf{G}\, e) : \tau'\ \texttt{event}}}{\text{TE}' \vdash \texttt{sync}\ (\mathbf{G}\, e) : \tau'}$$

Hence, by rule (**τ-sync**),

$$\frac{\text{TE}' \vdash e : \tau'\ \texttt{event}}{\text{TE}' \vdash \texttt{sync}\ e : \tau'}$$

■

Lemma 8.8 If $ev_1 \overset{\mathcal{K}}{\supset} ev_2$ with (e_1, e_2) and $\text{TE} \vdash ev_i : \tau_i\ \texttt{event}$, then $\text{TE} \vdash e_i : \tau_i$ (for $i \in \{1, 2\}$).

Proof. This is proved by induction on the definition of event matching; the details are given in the appendix. ■

We are now ready to prove the second subject reduction theorem, which says that concurrent evaluation preserves process typing.

Theorem 8.9 (Concurrent type preservation) If a configuration $\mathcal{K}, \mathcal{P}$ is well-formed with

$$\mathcal{K}, \mathcal{P} \Longrightarrow \mathcal{K}', \mathcal{P}'$$

and, for some channel typing CT,

$$\text{CT} \vdash \mathcal{K}, \mathcal{P} : \text{PT}$$

Then there is a channel typing CT′ and a process typing PT′, such that the following hold:

- $\text{CT} \subseteq \text{CT}'$,
- $\text{PT} \subseteq \text{PT}'$, and

- $\mathrm{CT}' \vdash \mathcal{K}', \mathcal{P}' : \mathrm{PT}'$.

- $\mathrm{CT}' \vdash \mathcal{K}, \mathcal{P} : \mathrm{PT}'$.

Proof. The fourth property follows from the others; the proof of the first three properties proceeds by case analysis of the left hand side of the $\Longrightarrow$ relation.

Case $\mathrm{CT} \vdash \mathcal{K}, \mathcal{P}{+}\langle \pi;\ e \rangle : \mathrm{PT}$.

If $e \longmapsto e'$, then, by sequential type preservation (Theorem 8.7), we have

$$(\{\}, \mathrm{CT}) \vdash e' : \mathrm{PT}(\pi)$$

and hence

$$\mathrm{CT} \vdash \mathcal{K}, \mathcal{P}{+}\langle \pi;\ e' \rangle : \mathrm{PT}$$

Letting $\mathrm{CT}' = \mathrm{CT}$ and $\mathrm{PT}' = \mathrm{PT}$ satisfies the theorem.

Case $\mathrm{CT} \vdash \mathcal{K}, \mathcal{P}{+}\langle \pi;\ E[\mathtt{chan}\ x\ \mathtt{in}\ e] \rangle : \mathrm{PT}$.

Then there is a type environment $\mathrm{TE} = (\mathrm{VT}, \mathrm{CT})$ and types τ and θ (with $\theta \in$ ImpTy), such that

$$\cfrac{\begin{array}{c}\vdots\\ \mathrm{TE} \pm \{x \mapsto \theta\ \mathtt{chan}\} \vdash e : \tau\end{array}}{\begin{array}{c}\mathrm{TE} \vdash \mathtt{chan}\ x\ \mathtt{in}\ e : \tau\\ \vdots\\ (\{\}, \mathrm{CT}) \vdash E[\mathtt{chan}\ x\ \mathtt{in}\ e] : \mathrm{PT}(\pi)\end{array}}$$

Let κ be the name of the new channel (hence $\kappa \notin \mathcal{K}$) and define $\mathrm{CT}' = \mathrm{CT} \pm \{\kappa \mapsto \theta\ \mathtt{chan}\}$ (obviously $\mathrm{CT} \subseteq \mathrm{CT}'$). Then, by Lemma 8.3,

$$(\{\}, \mathrm{CT}') \vdash E[\mathtt{chan}\ x\ \mathtt{in}\ e] : \mathrm{PT}(\pi)$$

Thus, by the Replacement and Substitution lemmas,

$$(\{\}, \mathrm{CT}') \vdash E[e[x \mapsto \kappa]] : \mathrm{PT}(\pi)$$

and, therefore, $\mathrm{CT}' \vdash \mathcal{K}{+}\kappa, \langle \pi;\ E[e[x \mapsto \kappa]] \rangle : \mathrm{PT}$. Letting $\mathrm{PT}' = \mathrm{PT}$ satisfies the theorem.

Case $\mathrm{CT} \vdash \mathcal{K}, \mathcal{P}{+}\langle \pi;\ E[\mathtt{spawn}\ v] \rangle : \mathrm{PT}$.

Then there is a variable typing VT and a type τ, such that

$$\cfrac{\begin{array}{c}\vdots\\ (\mathrm{VT}, \mathrm{CT}) \vdash v : (\mathtt{unit} \rightarrow \tau)\end{array}}{\begin{array}{c}(\mathrm{VT}, \mathrm{CT}) \vdash \mathtt{spawn}\ v : \mathtt{unit}\\ \vdots\\ (\{\}, \mathrm{CT}) \vdash E[\mathtt{spawn}\ v] : \mathrm{PT}(\pi)\end{array}}$$

By Lemma 7.1, we know that $\mathrm{FV}(v) = \emptyset$, and thus, by Lemma 8.6,

$$(\{\}, \mathrm{CT}) \vdash v : (\mathtt{unit} \to \tau)$$

Applying rule (**τ-app**), we get

$$(\{\}, \mathrm{CT}) \vdash (v\ ()) : \tau$$

Let π' be the process identifier of the new process (hence $\pi' \notin \mathrm{dom}(\mathcal{P})$), then

$$\mathrm{CT} \vdash \mathcal{K}, \mathcal{P} + \langle \pi;\ E[()] \rangle + \langle \pi';\ v\ () \rangle : \mathrm{PT} \pm \{\pi' \mapsto \tau\}$$

Letting $\mathrm{PT}' = \mathrm{PT} \pm \{\pi' \mapsto \tau\}$ and $\mathrm{CT}' = \mathrm{CT}$ satisfies the theorem.

Case $\mathrm{CT} \vdash \mathcal{K}, \mathcal{P} + \langle \pi_1;\ E_1[\mathtt{sync}\ ev_1] \rangle + \langle \pi_2;\ E_2[\mathtt{sync}\ ev_2] \rangle : \mathrm{PT}$.

Then, for $i \in \{1, 2\}$, there is a type environment TE_i and a type τ_i, such that

$$\begin{array}{c} \vdots \\ \dfrac{\mathrm{TE}_i \vdash ev_i : \tau_i\ \mathtt{event}}{\mathrm{TE}_i \vdash \mathtt{sync}\ ev_i : \tau_i} \\ \vdots \\ (\{\}, \mathrm{CT}) \vdash E_i[\mathtt{sync}\ ev_i] : \mathrm{PT}(\pi_i) \end{array}$$

If $ev_1 \overset{\mathcal{K}}{\supset} ev_2$ with (e_1, e_2), then, by Lemma 8.8,

$$\mathrm{TE}_i \vdash e_i : \tau_i$$

Thus, by the Replacement Lemma (8.4), we have

$$(\{\}, \mathrm{CT}) \vdash E_i[e_i] : \mathrm{PT}(\pi_i)$$

hence,

$$\mathrm{CT} \vdash \mathcal{K}, \mathcal{P} + \langle \pi_1;\ E_1[e_1] \rangle + \langle \pi_2;\ E_2[e_2] \rangle : \mathrm{PT}$$

Letting $\mathrm{CT}' = \mathrm{CT}$ and $\mathrm{PT}' = \mathrm{PT}$ satisfies the theorem.

■

This theorem leads immediately to the following fact about traces:

Corollary 8.10 Let $\langle\!\langle \mathcal{K}_1, \mathcal{P}_1;\ \ldots;\ \mathcal{K}_n, \mathcal{P}_n \rangle\!\rangle$ be a finite trace, with

$$\mathrm{CT} \vdash \mathcal{K}_1, \mathcal{P}_1 : \mathrm{PT}$$

Then there is a channel typing CT' and process typing PT', such that:

- $\mathrm{CT} \subseteq \mathrm{CT}'$,
- $\mathrm{PT} \subseteq \mathrm{PT}'$, and
- for $i \in \{1, \ldots, n\}$, $\mathrm{CT}' \vdash \mathcal{K}_i, \mathcal{P}_i : \mathrm{PT}'$.

Proof. This follows by a simple induction on n. ■

8.2.3 Stuck expressions

In order to show that well-typed programs do not have run-time type errors, we first need to characterize such errors.

Definition 8.4 A process $p = \langle \pi;\, e \rangle$ is *stuck* if e is not a value and there do not exist well-formed configurations $\mathcal{K}, \mathcal{P}{+}p$ and $\mathcal{K}', \mathcal{P}'$ such that $\mathcal{K}, \mathcal{P}{+}p \Longrightarrow \mathcal{K}', \mathcal{P}'$, with π a selected process. A well-formed configuration is *stuck* if one or more or its processes are stuck.

The notion of being stuck is a semantic one; in Section 6.2.4 and [WF91b], this is conservatively approximated by the syntactic notion of *faulty* expressions. For λ_{cv}, I take a somewhat different approach that focuses more on stuck expressions.

Lemma 8.11 (Uniform evaluation) Let e be a program, $T \in \mathrm{Comp}(e)$, and $\pi \in \mathrm{Procs}(T)$, then either $\pi{\Uparrow}_T$, $\pi{\Downarrow}_T v$, or $\mathcal{P}_i(\pi)$ is stuck for some $\mathcal{K}_i, \mathcal{P}_i \in T$.

Proof. This follows immediately from the definitions. ∎

It remains to show that stuck expressions are untypable.

Lemma 8.12 (Untypability of stuck configurations) If π is stuck in a well-formed configuration $\mathcal{K}, \mathcal{P}$, then there do not exist $\mathrm{CT} \in \textsc{ChanTy}$ and $\mathrm{PT} \in \textsc{ProcTy}$, such that

$$(\{\}, \mathrm{CT}) \vdash \mathcal{P}(\pi) : \mathrm{PT}(\pi)$$

In other words, $\mathcal{K}, \mathcal{P}$ is untypable.

Proof. The proof is given in the appendix. ∎

8.2.4 Soundness

We are now in a position to state the main result of this chapter: that well-typed programs do not go wrong. This result is stated in terms of the computations of a program. (recall from Section 7.3 that a computation is a maximal trace).

Theorem 8.13 (Syntactic soundness) Let e be a program, with $\vdash e : \tau$. Then, for any $T \in \mathrm{Comp}(e)$, $\pi \in \mathrm{Procs}(T)$, with $\mathcal{K}_i, \mathcal{P}_i$ the first occurrence of π in T, there exists a CT and PT, such that

$$\mathrm{CT} \vdash \mathcal{K}_i, \mathcal{P}_i : \mathrm{PT}$$

and $\mathrm{PT}(\pi_0) = \tau$. And either

- $\pi \Uparrow_T$, or
- $\pi \Downarrow_T v$ and there exists an extension CT$'$ of CT with $(\{\}, \mathrm{CT}') \vdash v : \mathrm{PT}(\pi)$.

Proof. The existence of CT and PT follows from Concurrent Type Preservation (Theorem 8.9). By Uniform evaluation (Lemma 8.11), we know that either $\pi \Uparrow_T$, $\pi \Downarrow_T v$, or $\mathcal{P}_j(\pi)$ is stuck for some $\mathcal{K}_j, \mathcal{P}_j \in T$.

Assume that π is stuck in $\mathcal{K}_j, \mathcal{P}_j$. By Lemma 7.2, $\mathcal{K}_j, \mathcal{P}_j$ is well-formed and, by Lemma 8.12, it must be untypable. But, since the configuration $\{\}, \{\langle \pi_0; e \rangle\}$ is typable, by Concurrent Type Preservation (Theorem 8.9), there is a $\mathrm{CT}' \in$ ChanTy and $\mathrm{PT}' \in$ ProcTy such that $\mathrm{CT}' \vdash \mathcal{K}_j, \mathcal{P}_j : \mathrm{PT}'$. Which means that $(\{\}, \mathrm{CT}') \vdash \mathcal{P}_j(\pi) : \mathrm{PT}'(\pi)$, hence π cannot be stuck and either $\pi \Uparrow_T$ or $\pi \Downarrow_T v$.

If $\pi \Uparrow_T$ then we are done.

Assume that $\pi \Downarrow_T v$ and let $\mathcal{K}_j, \mathcal{P}_j \in T$ such that $\mathcal{P}_j(\pi) = v$. Concurrent Type Preservation means that there exists an extension CT$'$ of CT and an extension PT$'$ of PT such that $\mathrm{CT}' \vdash \mathcal{P}_j : \mathrm{PT}'$. Since PT$'$ is an extension of PT, $\mathrm{PT}'(\pi) = \mathrm{PT}(\pi)$, and hence $(\{\}, \mathrm{CT}') \vdash v : \mathrm{PT}(\pi)$. ■

To state more traditional soundness results, we first need to define a notion of evaluation that distinguishes those processes that have run-time type errors.

Definition 8.5 For a computation T, define the *evaluation* of a process π in T as

$$\mathrm{eval}_T(\pi) = \begin{cases} \mathrm{WRONG} & \text{if } \mathcal{P}_i(\pi) \text{ is stuck for some } \mathcal{K}_i, \mathcal{P}_i \in T \\ v & \text{if } \pi \Downarrow_T v \end{cases}$$

Note that for sequential programs, this is essentially the same as the definition on page 71. Using this definition we can now state weak and strong soundness results for λ_{cv}.

Theorem 8.14 (Soundness) If e is a program with $\vdash e : \tau$, then for any $T \in \mathrm{Comp}(e)$ and any $\pi \in \mathrm{Procs}(T)$, the following hold:

(Strong soundness) If $\mathrm{eval}_T(\pi) = v$, and $\mathcal{K}_i, \mathcal{P}_i$ is the first occurrence of π in T, then for any CT and PT, such that $\mathrm{CT} \vdash \mathcal{K}_i, \mathcal{P}_i : \mathrm{PT}$ and $\mathrm{PT}(\pi_0) = \tau$, there is an extension CT$'$ of CT, such that $(\{\}, \mathrm{CT}') \vdash v : \mathrm{PT}(\pi)$.

(Weak soundness) $\mathrm{eval}_T(\pi) \neq \mathrm{WRONG}$

Proof. This follows immediately from Syntactic soundness (Theorem 8.13) and the definition of eval. ■

In other words, a well-typed CML program can never have a run-time type error. It is also worth noting that for the sequential subset of λ_{cv}, Theorem 8.14 reduces to the Soundness theorem of Section 6.2.4 (Theorem 6.4).

Part IV

Practice

Chapter 9

Applications

This part of this dissertation addresses the question of the usefulness and practicality of the proposed language mechanisms. While Chapter 5 describes a number of abstractions that can be implemented using **CML**, it does not fully address the question of how useful **CML** is for real applications and whether it can be efficiently implemented.

To address these questions, I have implemented **CML** on top of **SML/NJ**. This implementation has been used by a number of people, including myself, for various different applications. This practical experience demonstrates the validity and usefulness of my design as well as the efficiency of my implementation. In this chapter, I describe some of these applications. I describe the implementation in Chapter 10, and its performance in Chapter 11. The final chapter of this part (Chapter 12) describes further research related to the implementation and use of **CML** on multiprocessors.

9.1 eXene: A multi-threaded X window system toolkit

As argued in Section 1.1, concurrency is a useful tool for structuring interactive applications. To this end, Emden Gansner of AT&T Bell Laboratories and I have been developing a multi-threaded **X** window system toolkit [SG86], called **eXene** [GR91], which is implemented using **CML**. This implementation serves two roles: it provides a strenuous test of the performance of **CML** in a real-world setting, and it serves as a platform for interactive applications (discussed in Section 9.2). Because the **X** window system is a distributed system, the implementation of **eXene** also involves distributed systems programming (discussed in Section 9.1.4). This section describes the architecture of **eXene** and gives a couple of examples of the use of **CML** primitives in its implementation.

9.1.1 An overview of eXene

EXene provides a similar level of function as **Xlib** [Nye90b], but with a substantially different model of user interaction. Windows in **eXene** have an *environment*, consisting of three streams of input from the window's parent (mouse, keyboard and control), and one output stream for requesting services from the window's parent. For each child of the window, there are corresponding output streams and an input stream. The input streams are represented by event values and the output streams by event valued functions. A window is responsible for routing messages to its children, but this can almost always be done using a generic router function provided by **eXene**. Typically, each window has a separate thread for each input stream as well as a thread, or two, for managing state and coordinating the other threads. By breaking the code up this way, each individual thread is quite simple. This event-handling model is similar to those of [Pik89] and [Haa90].

There are other differences between **eXene** and more traditional **X** toolkits. For example, **eXene** uses *immutable pens* to specify the semantics of drawing operations, instead of the mutable *graphics contexts* provided by the **X**-protocol. Since pens are immutable, concurrency control issues are avoided when two threads share the same pen.

9.1.2 An X window system overview

The **X** window system is a distributed system with the application clients communicating with the **X** server process. The core **X**-protocol consists of 211 different messages, divided into 119 request messages, of which 42 have replies, 33 event messages and 17 error messages [Nye90a]. Each request to the server has an implicit sequence number (i.e., the first message sent is number 1, etc.). Messages from the server to the client are tagged with the sequence number of the last request processed by the server; this is used to match replies with requests.

9.1.3 The architecture of eXene

Unlike some non-**C** language bindings for **X**, **eXene** is implemented directly on top of the **X**-protocol. The only non-**CML** code involved is the run-time system's support for socket communication. This implementation approach has the advantage of avoiding the **C** language bias of **Xlib**. Furthermore, it provides a demonstration that **CML** can be used to implement low-level systems programs without significant loss of performance.

A connection to an **X**-server is called a *display*. In **eXene** a display consists of seven threads; Figure 9.1 gives the message-passing architecture of these threads. The *input* and *output* threads provide buffering of the communication with the server. The *sequencer*

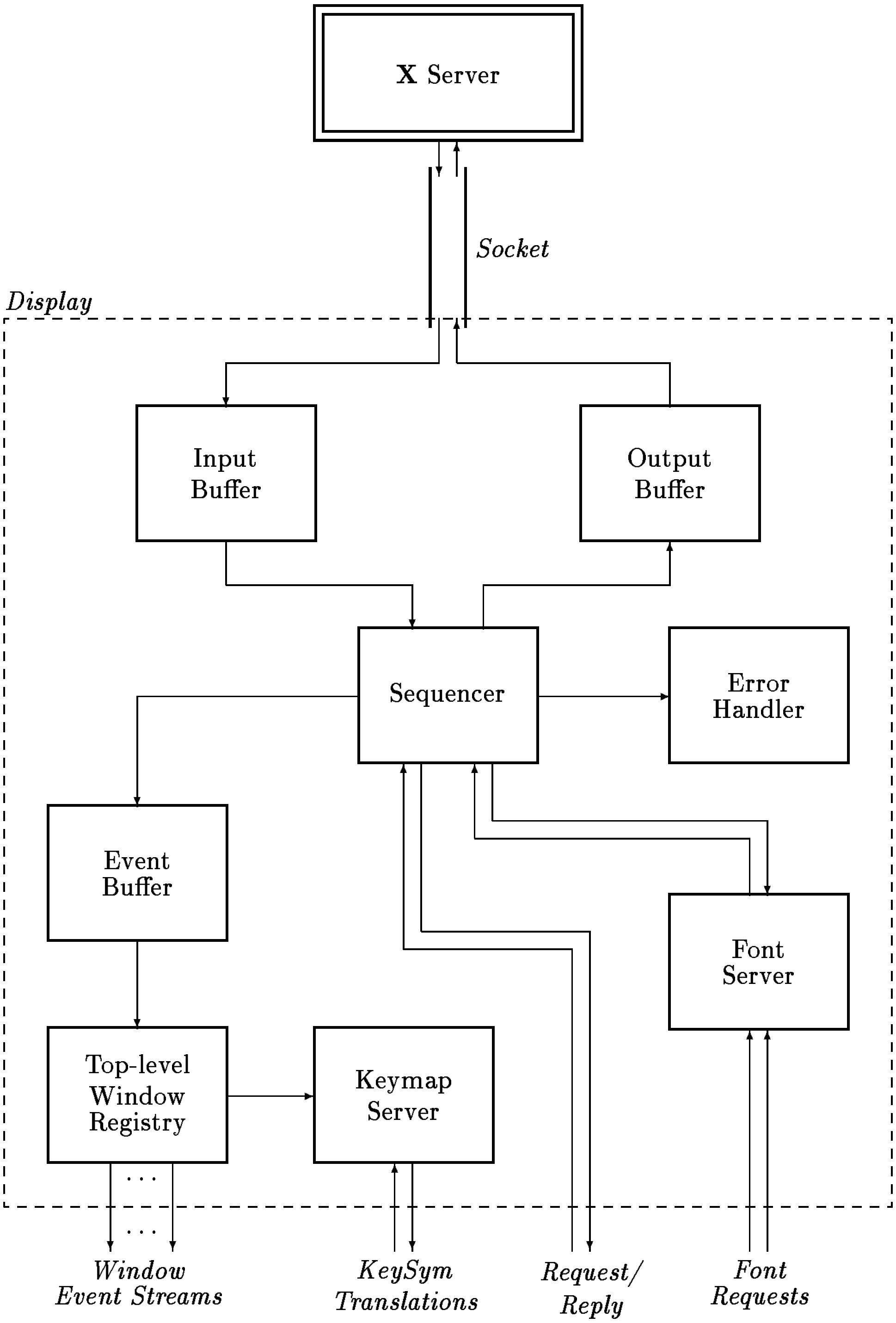


Figure 9.1: The display message-passing architecture

thread generates sequence numbers and matches replies with requests. All error messages are logged with the *error handler*; in addition, errors on requests that expect a reply are forwarded to the requesting thread. The sequencer sends **X**-events to the *event buffer*, which decodes and buffers them. The *top-level window registry* is a thread that keeps track of the top-level windows in the application and their descendants. It manages a stream of events for each top-level window in the application. The other two display threads manage global resources: the *keymap server* provides translations from *keycodes* to *keysyms*; the *font server* keeps track of the open fonts used by the application.

A display has one or more *screens*, each of which can support different *visuals* and *depths* (e.g., black and white or 8-bit color). Each visual and depth combination of a screen is supported by two threads; Figure 9.2 shows the message architecture for these. The *draw*

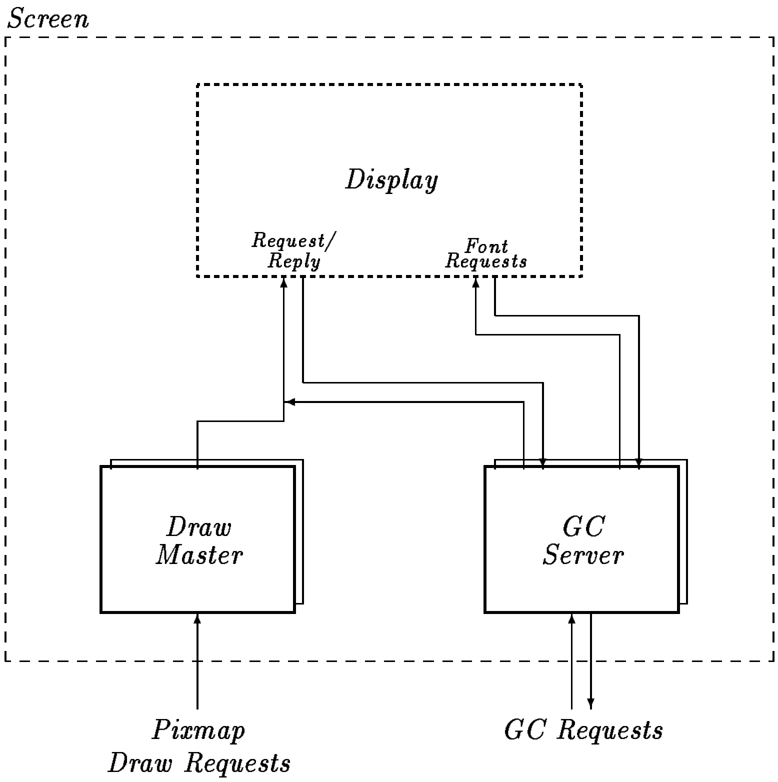


Figure 9.2: The screen message-passing architecture

master is a thread that encodes and batches drawing requests for a particular visual and depth combination; the draw masters at the screen level are used for operations on *pixmaps* (off screen rectangles of pixels). The *GC server* handles the mapping of **eXene**'s immutable pens to **X**'s mutable graphics contexts.[1]

Windows are displayed with a particular visual and depth on a screen. Internally, windows are organized into a tree hierarchy with a top-level window at the root. Figure 9.3 gives the message-passing architecture for the top-level window threads. As described above,

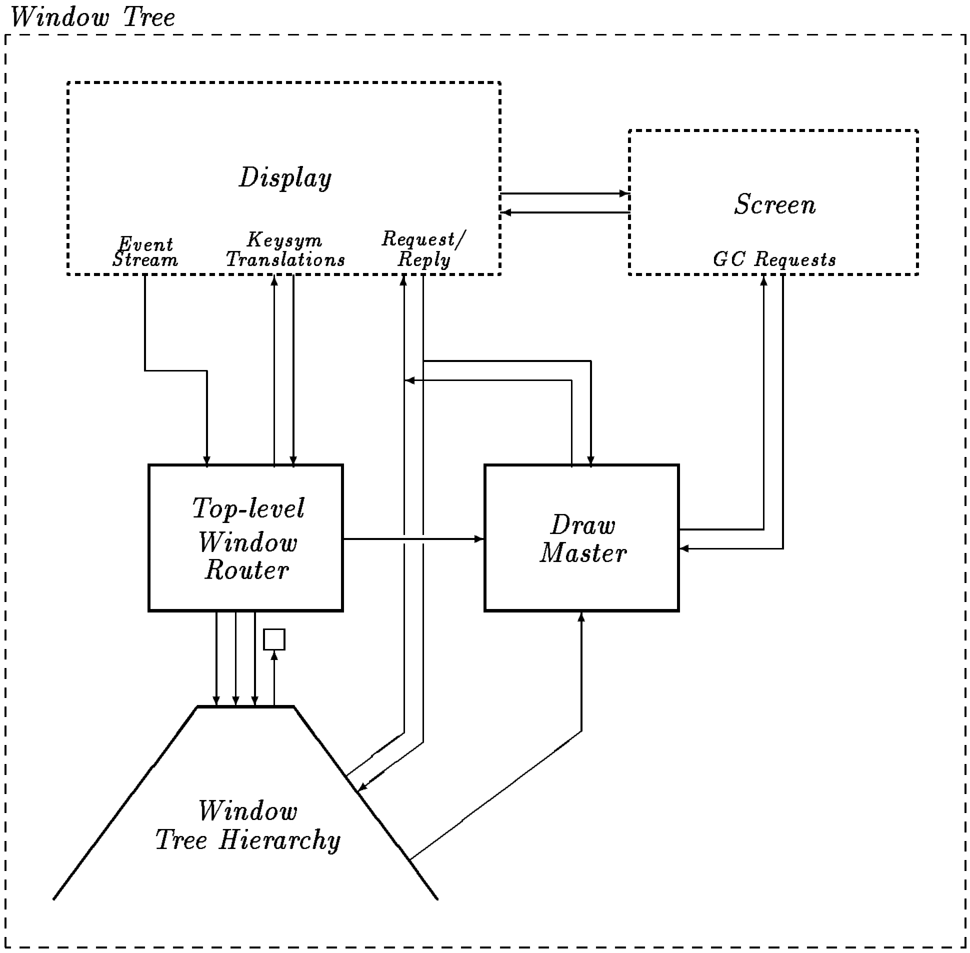


Figure 9.3: The top-level window message-passing architecture

each top-level window in an application has a dedicated stream of **X**-events from the display. This stream is monitored by the *top-level window router* thread. This thread provides the transition from the **X** view of events to the **eXene** view (i.e., a window environment). There is a draw master thread for each window tree as well.

[1] It is an unpleasant artifact of **X** that pixmaps and graphics contexts must be associated with a particular screen, visual and depth.

9.1.4 Promises in eXene

The `CopyArea` operation in the **X11** protocol can be used to copy a rectangle of pixels from one place on the screen to another. A complication arises if a portion of the source rectangle is obscured by another window. For example, Figure 9.4 shows a use of `CopyArea` to translate a rectangle on the screen; here the cross-hatched region of the destination corresponds to the obscured region of the source. While some window system maintain a

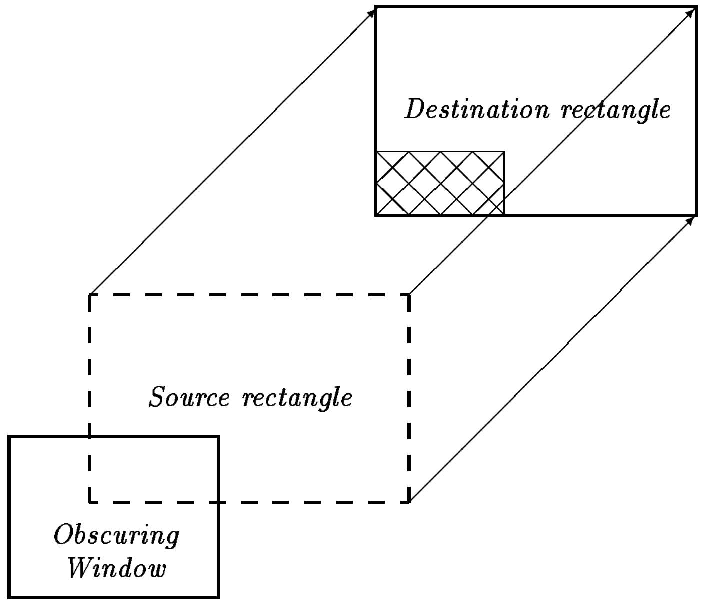
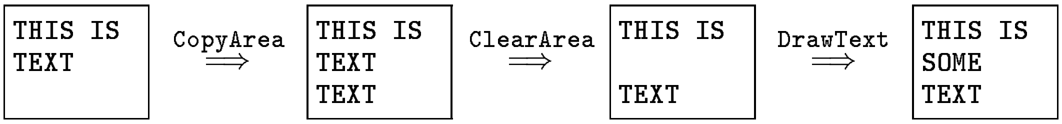


Figure 9.4: The `CopyArea` operation

backing store (or *virtual bitmap*) to handle these situations, the standard **X** policy is to notify the client that the `CopyArea` operation was not able to completely fill in the destination.[2] This policy is called *damage control*, since it is up to the client to repair the damage.

A typical use of `CopyArea` is in inserting a line of text. In this case the client thread might issue the following sequence of operations: a `CopyArea` to create space for the new text, followed by a `ClearArea` to erase the old text and lastly a `DrawText` to insert the new line. The following picture illustrates these steps:

[2] Some **X** servers do support backing store as an option, but applications must be designed to function correctly when it is not available.



It is important that the user of the system see this sequence as a single smooth transition. This has implications for the implementation of operations using `CopyArea`.

If `CopyArea` is treated as a normal **X** RPC, which returns a list of damaged rectangles, then the user is subjected to screen flicker. To understand the reasons for this examine Figure 9.5, which shows the timing information for the client doing the text scrolling, the thread handling the buffering of communication with the server,[3] and the **X**-server. Because

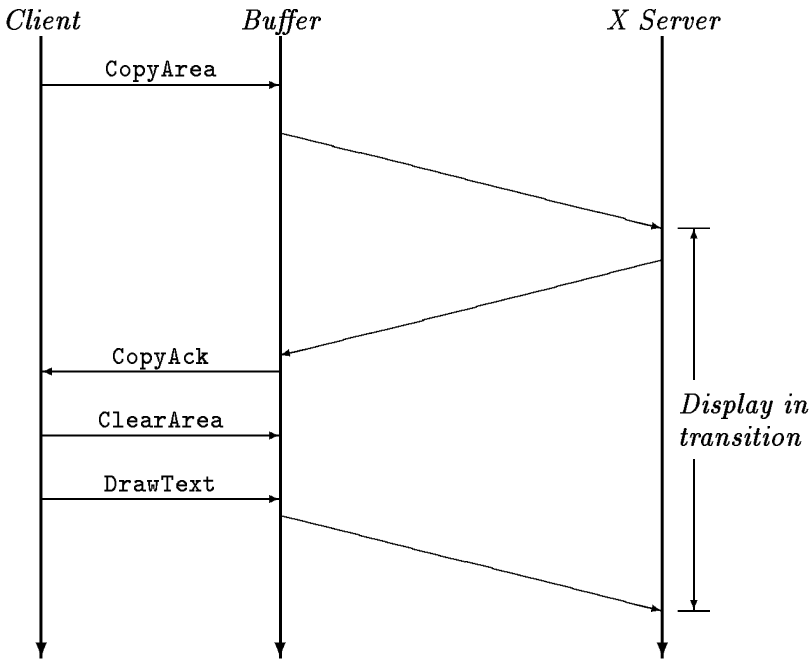


Figure 9.5: Synchronous text scrolling

the other drawing operations are postponed until an acknowledgement of the `copyArea` is received, the period of time the display is in transition can be quite lengthy.

Because of these performance concerns, the **X** protocol does not use the standard reply mechanism for `CopyArea`. Instead there are two special **X**-events, `GraphicsExpose` and `NoExpose`, which are used to notify the client of the result of a `CopyArea` request.[4] For single-

[3]For purposes of this discussion, I have collapsed the buffer and sequencer threads into a single thread.

[4]Things are a little more complicated, since multiple `GraphicsExpose` events can be generated for a single

threaded **C** clients (which make up the vast majority of **X** clients), this means that the code using the `CopyArea` operation must also scan the event stream for the acknowledgement. In **eXene**, where we have concurrency and events, we can handle this operation in a much more elegant way. Our solution is to implement `CopyArea` as an asynchronous RPC operation, also known as a *promise* [LS88]. **EXene** provides an event-valued function with the type

```
val copyArea : arg-type -> rect_t list event
```

where *arg-type* is the type of the arguments that specify the actual operation. The event that is returned is the promise of the result. Figure 9.6 gives the implementation sketch of this operation, where `request` sends the operation to the buffer thread and `flush` tells the buffer thread to flush any buffered messages to the server. The guard is optimized to first check to

```
fun copyArea arg = let
      val replyCh = channel()
      in
        spawn (fn () => request (COPY_AREA(reply_ch, arg)));
        guard (fn () => (
          case (poll (receive replyCh))
           of (SOME rects) => always rects
            | NONE => (flush(); receive replyCh)
          (* end case *))
      end
```

Figure 9.6: The implementation of `copyArea`

see if the acknowledgement is already available. The buffer code is more complicated, since it must match the acknowledgements with outstanding `CopyArea` requests. The advantage of this approach can be seen by comparing its timing diagram, given in Figure 9.7, with Figure 9.5.

9.1.5 Menus

Another example of the way concurrency is used in **eXene** is in the way that popup menus are attached to windows. This is done by interposing a thread on the window's mouse stream. When the thread sees a down transition on the appropriate mouse button, it creates the menu window and starts tracking the mouse (the **X** semantics cause all mouse events until the up transition to be directed at the window). Other mouse events are passed through without action. This is a form of *delegation*, and the window wrapped by the menu thread can be viewed as a "sub-class" of the window.

`CopyArea` request.

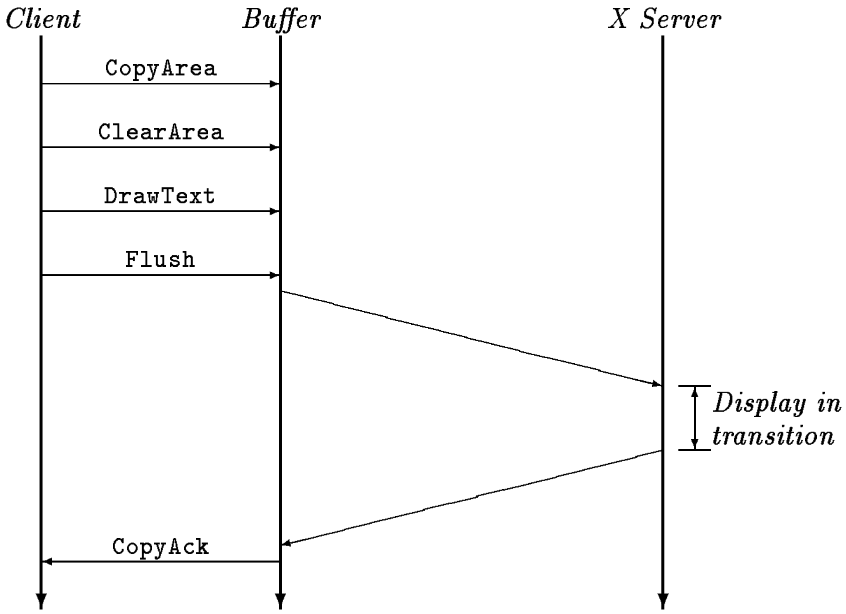


Figure 9.7: Asynchronous text scrolling

9.2 Interactive applications

The combination of **eXene** and **CML** provides a foundation for building interactive applications in the spirit of **Pegasus** [RG86, GR92]. In this section, I describe an application of **eXene** and how it uses the features of **CML**.

Currently, the most sophisticated application built on top of **eXene** is **Graph-o-matica**, which is an interactive tool for viewing and analyzing directed graphs.[5] **Graph-o-matica** was originally implemented on top of **Pegasus** by Emden Gansner and Steve North at AT&T Bell Laboratories; Emden Gansner ported it to **eXene**.

Graph-o-matica provides the user with two kinds of windows: command windows, which provide a terminal-style, language interface to a command shell, and viewers, which provide a view on a 2D layout of a graph. At any time a user can have multiple command windows and multiple viewers. Each viewer is associated with a particular layout of a particular abstract graph. Different graphs can have different layouts, and each layout can have multiple views. Figure 9.8 is a screen dump from a sample session with Graph-o-matica. The bottom window is a command window; the two windows above provide two views of a single layout of a graph. A viewer allows the user to pan and zoom (using menus

[5]Huimin Lin at the University of Sussex has built an interactive theorem prover on top of **eXene**, but I do not know the details of its implementation.

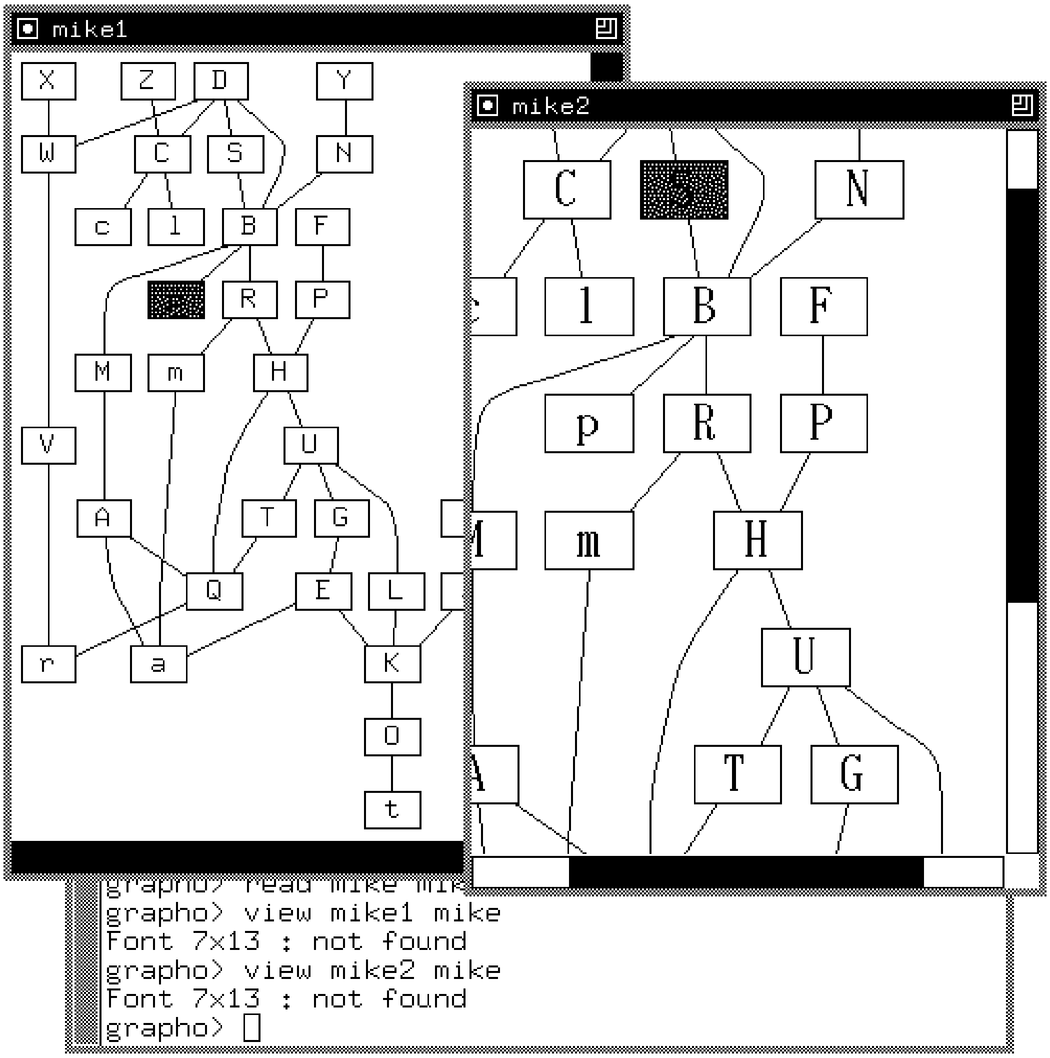


Figure 9.8: Graph-o-matica screen dump

and the scrollbars) over the particular layout. The user can manually change a layout using editing operations such as moving a node, or elision of a subgraph.

The implementation of **Graph-o-matica** exploits the features of **CML** in several ways. If a graph is edited, this information needs to be propagated to the layouts and views of the graph. We use the multicast channel abstraction (described in Section 5.2) to manage the propagation of update notifications to the layouts and from the layouts to the views. This simplifies the implementation of the graph object, since it does not need to know anything about multiple layouts. The layout objects, if they decide a given change affects them, can query the graph object for more detailed information.

The command shell is a thread that communicates with a *virtual terminal* (vtty) widget. The vtty widget is a good example of the need for both communication abstraction and selective communication. At any time, the vtty must be able to handle both input from the user and output from its client (the command shell). **EXene** provides an abstract interface to the input stream, but since it is event-valued, it can still be used in selective communication.

Concurrency is also used in the structuring of the application code. Layout algorithms, for example, run as separate threads, thus allowing the user to continue other activities while waiting for a new layout.

9.3 Distributed systems programming

Many distributed programming languages have concurrent languages at their core (e.g., **SR** [AOCE88]), and distributed programming toolkits often include thread packages (e.g., **Isis** [BCJ$^+$90]). This is because threads provide a needed flexibility for dealing with the asynchronous nature of distributed systems.

The flexibility provided by **CML** is a good base for distributed programming. Its support for low-level I/O is sufficient to build a structured synchronous interface to network communication (as was done in **eXene**). Higher-level linguistic support for distributed programming, such as the promise mechanism described in Section 9.1.4, can be built using events to define the new abstractions.

Another example is Chet Murthy's reimplementation of the **Nuprl** environment [Con86] using **CML**. His implementation is structured as a collection of "proof servers" running on different workstations. When an expensive operation on a proof tree is required, it can be decomposed and run in parallel on several different workstations. This system uses **CML** to manage the interactions between the different workstations.

9.3.1 Distributed ML

Another project involving **CML** is the development of a distributed programming toolkit for **ML** that is being done at Cornell University [Kru91]. This work builds on the mechanisms prototyped in Murthy's distributed **Nuprl** and on the protocols developed for **Isis** [BCJ$^+$90]. A new abstraction, called a *port group* has been developed to model distributed communication. The communication operations provided by port groups are represented by event-value constructors. For details see [Kru91].

9.4 Other applications of CML

CML has been used by various people for a number of other purposes. Andrew Appel has used it to teach concurrent programming to undergraduates at Princeton University (Appel, personal communication, January 1991). Gary Lindstrom and Lal George have used it to experiment with functional control of imperative programs for parallel programming [GL91]. And Clément Pellerin has implemented a compiler from a concurrent constraint language to **CML**.

Chapter 10

Implementation

There have been several implementations of first-class synchronous operations. I wrote the first implementation in **C** as part of the **Pegasus/PML** run-time system [Rep88]. I later implemented the concurrency mechanisms of **PML** on top of **SML/NJ** in a coroutine[1] library [Rep89], and Norman Ramsey has implemented a similar system at Princeton [Ram90]. More recently, I implemented **CML** on top of **SML/NJ** [Rep91a]. Of these implementations, **CML** provides the richest programming notation and the best performance. It is written entirely in **SML**, using two non-standard extensions provided by **SML/NJ**, *first-class continuations* [DHM91] and *asynchronous signals* [Rep90a], and one minor compiler modification. This chapter describes the implementation of **CML** in some detail (a brief sketch was given in [Rep91a]), and discusses implementation techniques that might further improve performance. Some specific performance measurements of this implementation are reported in the next chapter. This implementation runs on single processor computers; the issues related to a multiprocessor implementation of **CML** are discussed in the Chapter 12.

10.1 The implementation of SML/NJ

SML/NJ is a high-performance implementation of **SML** [AM87, AM91]; it uses a combination of sophisticated compiler techniques and clever run-time system support to provide a level of performance that is competitive with **C** on large examples. In this section, I describe the aspects of **SML** that have a direct bearing on the implementation of **CML**.

[1]By "coroutine," I mean that this system does not use preemptive thread scheduling.

10.1.1 First-class continuations

As discussed in Section 2.3.3, **SML/NJ** provides continuations as first-class values. Until recently the type of `callcc` was fully polymorphic; i.e.,

```
val callcc : ('a cont -> 'a) -> 'a
```

As discussed in Chapter 8 (p. 92), it has been discovered that this typing of first-class continuations is unsound. The type of `callcc` in **SML/NJ** is now:

```
val callcc : ('1a cont -> '1a) -> '1a
```

which corrects the soundness problem [WF91a]. Unfortunately, using this weakly polymorphic type has the effect of reducing the polymorphism of the **CML** primitives. For example, the type of `sync` is

```
val sync : '1a event -> '1a
```

using the weakly polymorphic version of `callcc`. For this reason, I use the unsafe, fully polymorphic, version of `callcc` in my implementation. The typing of the resulting primitives, however, is proven sound in Chapter 8.

10.1.2 The compiler

The **SML/NJ** compiler is a multi-pass compiler. The front-end is fairly conventional (scanning, parsing, type-checking, etc.); it is the back-end (optimization and code generation) that interests us. The back-end uses a representation called *continuation-passing style*, or CPS for short [Ste78, KKR$^+$86, AJ89, App92]. The CPS representation is a specialized form of λ-calculus that has a uniform representation for all transfers of control (conditional branches, loops, function calls and function returns). This representation is a "goto with arguments," better known as a tail-recursive call. Since a function return is represented as a tail-recursive call, functions must be parameterized by the *return continuation*. It is from this explicit passing of continuations that CPS gets its name. The advantage of this approach is that the compiler can concentrate on making function calls as fast as possible, which is one of the keys to good performance for languages like **SML** and **Scheme**. Unlike other continuation-passing style compilers, such as [Ste78] and [KKR$^+$86], the code generated by the **SML/NJ** compiler does not use a run-time stack; instead, return continuations are heap allocated. This means that the code generated for `callcc` and `throw` is essentially the same as that for function calls. The only difference is that the current continuation created by `callcc` must restore the current enclosing exception handler. Unfortunately, this means that `callcc` breaks tail-recursion (in the same way that exception handlers do), which has implications for the implementation of `sync` (see Section 10.4).

10.1.3 The run-time system

The **SML/NJ** run-time system provides automatic memory management, an interface to the underlying operating system (UNIX), and a mechanism for building stand-alone **ML** worlds. An older version of the run-time system is described in [App90], but I and others have revised it several times since then.

The run-time system is logically divided into two coroutines: the **ML** program and the **C** program that provides run-time support. The actual implementation uses procedure call and return to implement the coroutine switches, with a global **C** `struct` used to hold the **ML** state in the run-time system.[2] Two assembly routines, `restoreregs` and `saveregs`, are used to (respectively) call and return from the **ML** program. When the **ML** program needs a service from the run-time system, it loads the global variable `request` with name of the needed service and jumps to `saveregs`.

Memory management

Each object has a one-word descriptor at its beginning that contains the object's length and a 4-bit tag. There are four kinds of objects: tuples, arrays, strings and bytearrays (mutable strings). Code objects in the heap are represented by strings. A single code object is used to hold all of the code for a compilation unit (e.g., **ML** structure or functor), which requires a mechanism for supporting pointers into the middle of strings. This is accomplished by an *embedded-string* descriptor, which is used to mark substrings of a code object; preceding the embedded-string descriptor is a *back-pointer* descriptor, which tells the garbage collector how to find the beginning of the code object. For more details on run-time representations see [App90].

Memory allocation is the dominating cost of **ML** execution, thus it must be as cheap as possible. **SML/NJ** uses inline allocation with minimal overhead (allocation of a tuple requires 3 or 4 instructions over the cost of object initialization). The run-time system uses two dedicated registers to support allocation: the *allocation pointer*, which points to the start of the next object to be allocated; and the *heap-limit pointer*, which is used to test for the need for garbage collection. Instead of testing the heap-limit on every allocation, the compiler tests only at the beginning of an extended basic block.[3] The compiler computes

[2]For releases of **SML/NJ** since 0.70, the **ML** state is no longer global. Instead, each run-time routine takes it as an argument, which allows multiple **ML** states to exist (e.g., on a multiprocessor).

[3]An *extended basic block* is an acyclic graph of basic blocks with a single entry-point, but multiple exit-

the maximum possible allocation in an extended basic block and generates a heap-limit test at the root, which will insure that the execution of the block does not run out of allocation space (allocation of dynamic sized objects, such as arrays, is done by hand-coded assembly routines in the run-time system). To simplify the test, the heap limit is set at 4096 bytes below the actual top of the allocation space, so that the test for any block that allocates less than 4096 bytes only involves a pointer comparison. On many machines, this can be cleverly coded using one register-register instruction; for blocks that might allocate more than 4096 bytes, a more expensive test involving pointer arithmetic is required. When a heap-limit overflow is detected a trap is generated, which the operating system maps to a UNIX *signal* [UNI86] that is caught by a handler in the run-time system. The signal handler saves the program counter of the **ML** program, replaces it with the address of `saveregs`, and returns to the operating system, which causes program execution to resume in `saveregs`. This technique of using a UNIX signal handler to vector to an assembly routine that saves the register state is owed to Cormack [Cor88]. The compiler generates an embedded-string descriptor prior to the entry-point of each extended basic block, thus the program counter at the heap-limit test is treated like a normal **ML** value by the garbage collector.

Signals

SML/NJ provides an asynchronous signal mechanism [Rep90a], which has semantics similar to that of UNIX signals [UNI86]. When a signal occurs, the current continuation is grabbed and passed to the appropriate **ML** signal handler. The signal handler executes atomically with respect to signals; it returns a continuation that is used to resume execution. Signal handlers provide a natural mechanism for implementing preemptive thread scheduling (see below).

The actual translation of a UNIX signal to an **ML** signal is more complicated than described. The principal difficulty is that constructing a continuation to pass to the signal handler at any arbitrary point in the execution is not feasible. The solution to this problem is to delay capturing the continuation to a safe point where the state of execution can be easily captured. The heap-limit checks used to trigger garbage collection conveniently provide such safe points. Thus, when a UNIX signal occurs, the UNIX signal handler in the run-time system records it and modifies the heap-limit pointer to insure that the next heap-limit check will trigger a garbage collection. The garbage collector then recognizes that the request for garbage collection is actually a pending signal, builds a continuation closure out of the **ML** state,[4] and passes the signal and continuation to an **ML** routine that

points [Ros81].

[4]Recall from above that the heap-limit check is preceded by a descriptor, thus the garbage collector will be able to deal with the code address of the continuation built by the run-time system.

dispatches the appropriate **ML** signal handler. The use of heap-limit checks as safe points is similar to the preemption technique used in **Argus** [LCJS87]. For a complete description of the **SML/NJ** signal mechanism, see [Rep90a].

10.2 Implementing threads

The implementation of threads exploits the fact that first-class continuations are exactly the thread state that needs to be saved and restored on context switches [Wan80]. This section describes the implementation of threads and preemptive scheduling.

10.2.1 Threads

Internally, a thread is represented by two pieces of information: a *thread ID* and a continuation. The thread ID serves as a unique identifier for the thread, as well as providing a handle for implementing the `threadWait` operation, while the continuation represents the suspended state of the thread's computation. Threads are either *ready* (able to execute) or *blocked* (waiting to synchronize on some event). At any time, one of the ready threads is designated as the currently *running* thread; the IDs and current continuations of the other ready threads are kept in the *ready queue*. The global variable `runningThreadId` is used to refer to the currently running thread's ID.[5] Switching thread contexts involves putting the current thread's continuation and ID into the ready queue and dispatching the next thread in the queue. The following code illustrates the mechanics of a context switch:

```
fun contextSwitch runningK = let
      val _ = rdyQInsert (!runningThreadId, runningK)
      val (newId, newK) = rdyQRemove ()
      in
        runningThreadId := newId;
        throw newK ()
      end)
```

where `runningK` is bound to the running thread's current continuation. Variations on this scheme are used throughout the implementation.

Although the **SML/NJ** compiler knows nothing about threads and concurrency, the fact that `callcc` and `throw` are used to implement threads means that the implementation gets many of the benefits of specialized compiler support for free. In particular, the compiler knows exactly which registers are live at the point of a context switch, thus only the minimum amount of thread state required is actually saved and restored. Furthermore,

[5] In the most recent version of **CML** (0.9.6), I switched to using the `varptr` register to refer to the current thread's ID. This register is a dedicated per-processor register provided by the compiler.

the fact that continuations are heap allocated means that thread creation is a very fast, constant time, operation.[6] For the MIPS processor, a thread context switch is about 190 instructions, and thread creation is about 490 instructions; there is some hope that these numbers can be substantially reduced (see Section 11.1.2 for more details).

10.2.2 Preemptive scheduling

In order to prevent a thread that is executing a long (or infinite) computation from monopolizing the processor, **CML** uses preemptive thread scheduling. This is done in a straightforward manner using the UNIX interval timer [UNI86] and the signal mechanism described above. The interval timer is set to generate a `SIGALRM` every n milliseconds (n is typically in the range from 10 to 50), and a signal handler that forces a context switch is installed for `SIGALRM`. The only complication is the possible interference between the running thread and signal handler. To avoid this problem, a global flag is used to mark when execution is in a critical region:

```
datatype atomic_state = NonAtomic | Atomic | SignalPending
val atomicState : atomic_state ref

val atomicBegin : unit -> unit
val atomicEnd   : unit -> unit
```

The function `atomicBegin`, which sets the flag to `Atomic`, is called just prior to entering a critical region, and the function `atomicEnd`, which resets the flag to `NonAtomic`, is called on exit. If a signal occurs while `atomic_state` is `Atomic`, then the signal handler does not force a context switch. Instead it sets `atomic_state` to `SignalPending` and returns. The function `atomicEnd` checks the flag before resetting it; if it is `SignalPending`, then a context switch is performed.[7] Note that this mechanism is internal to the implementation of **CML**; user programmers have no access to these operations.

10.3 Implementing channels

Channels are represented by a pair of queues; one for threads waiting for input and one for threads waiting for output (see Figure 10.1). Each item in a channel queue is a triple,

[6] While constant time `callcc` is possible in stack based implementations (e.g. [HDB90]), it is not clear that these techniques are fast enough to implement true light-weight threads on today's hardware. For example, Haahr reports that a number of **Scheme** implementations were unsatisfactory for anything more than prototypes of his multi-threaded window system [Haa90].

[7] The reader may recognize that there is a potential race when exiting a critical region between the time of the test for a pending signal and the resetting of the flag. The delaying of signals to heap-limit check points, however, means that this race can not occur in practice, since the test and resetting of the flag is done without any intervening heap-limit checks.

```
type 'a chanq = (bool ref * thread_id * 'a) queue
datatype 'a chan = CHAN of {
    inq  : 'a cont chanq,
    outq : ('a * unit cont) chanq
  }
```

Figure 10.1: The representation of channels

consisting of a *dirty flag* (described below), a thread ID, and an offered communication. In the input queue (inq), an offered communication is represented by a continuation that will accept a message; in the output queue (outq), an offered communication consists of the message being sent and the continuation to resume the thread when the message is accepted. As an example, the implementation of send is given in Figure 10.2. This code

```
fun send (CHAN{inq, outq}, msg) = callcc (fn send_k => (
      atomicBegin();
      case (cleanAndRemove inq)
       of SOME(rid, rkont) => (
            rdyQInsert (!runningThreadId, send_k);
            runningThreadId := rid;
            atomicEnd();
            throw rkont msg)
        | NONE => (
            insert(outq, (ref false, getTid(), (msg, send_k)));
            atomicDispatch())
     (* end case *)))
```

Figure 10.2: The implementation of send

works by first capturing the rendezvous point continuation using callcc. Since the channel queue and ready queue are going to be manipulated, atomicBegin is called to mark the start of a critical region. The call to cleanAndRemove returns the ID and communication of the first "clean" item from the input queue[8] if one is available, otherwise it returns NONE. If there is an offered communication available (i.e., matching accept or receive), then the sending thread is added to the ready queue and the message is thrown to the receiving thread's continuation. If no matching communication is available, then the sender is added to the output waiting queue, and another thread is dispatched (atomicDispatch dispatches a thread while exiting the critical region). The implementation of accept is essentially a mirror image of send.

[8]The notion of cleanliness is related to the dirty flag and is explained below.

10.4 Implementing events

The implementation of events is moderately complex, so it is useful to first consider a very simple subset of events without choice, guards or abort actions. In particular, consider the **P** operation on binary semaphores, which is one of the simplest synchronous operation. An implementation of binary semaphores (ignoring issues of atomic regions and thread IDs) is quite simple:

```
datatype semaphore = SEMAPHORE of {
    flg : bool ref,
    waitq : unit cont list ref
  }

fun V (SEMAPHORE{flg, waitq}) = (case !waitq
       of [] => flg := true
        | (k::r) => (waitq := r; enqueue k))

fun P (SEMAPHORE{flg, waitq}) = let
      fun Pbody resumek = if (!flg)
            then (flg := false)
            else (waitq := !waitq @ [resumek]; dispatch())
      in
        callcc Pbody
      end
```

where the body of the **P** operation is factored out for pedagogical reasons. The thing to notice about this code is that the resumption continuation of the calling process is a free variable in the body of the operation. This observation, which holds for all synchronous operations, is the key to the implementation of events. In this simple setting, it means that event values can be represented as

```
type 'a event = 'a cont -> 'a
```

Using this representation, the event-valued implementation of **P** is:

```
fun P (SEMAPHORE{flg, waitq}) = let
      fun Pbody resumek = if (!flg)
            then (flg := false)
            else (waitq := !waitq @ [resumek]; dispatch())
      in
        Pbody
      end
```

It follows that sync is implemented directly by `callcc`. The implementation of wrap must feed the value produced by synchronizing on its first argument to its second argument, which is done as follows:

```
fun wrap (evt, f) = fn k => (throw k (f (callcc evt)))
```

The continuation that applies `f` to its argument is passed to the event value being wrapped; the result of evaluating `f` is then thrown to the continuation that is the argument to the event value constructed by `wrap`. The astute reader will recognize this as a convoluted form of function composition.

10.4.1 Event value representation

Unfortunately, this simple representation of events is unable to support choice, guards or abort actions. In the more general setting of **CML** events, there are five distinct aspects to a thread synchronizing on an event values:

Forcing. If the event is a guard event, then it must be *forced* (i.e., the guard function is applied).

Polling. For a non-guard event, the first step is to poll the base events to see if any of them are immediately satisfiable.

Selection. If one or more of the base events is immediately satisfiable, then one of these is selected and executed.

Logging. If there are no immediately satisfiable events, then the synchronizing thread must be added to the waiting queues of the base events.

Unlogging. Once one of the base events is satisfied, the thread must be removed from the other base events' waiting queues and their abort actions (if any) must be spawned.

Figure 10.3 gives the representation of event values, which reflects the five aspects described above. An event value is either a guard function, or a list of base-event descriptors. A base-event descriptor is a record of four fields: the function `pollfn` is used to test if the base event is immediately satisfied; the function `dofn` is used to execute the base event if it is selected; the function `blockfn` is used to log the base-event value; and the field `abortfn` is either `NO_ABORT` or the abort action. In [Rep88], the informal semantics of **PML** events is defined in terms of a rewriting system that converts events to a "canonical" form; this form is essentially the above representation and the rewrite rules are the implementation of the various combinators.

10.4.2 Synchronization

As described above, there are five aspects to applying `sync` to an event value. These can be divided into two phases. The first phase is forcing guards, and corresponds to the

```
datatype abort_fn = NO_ABORT | ABORT of (unit -> unit)

datatype 'a base_evt = BASE_EVT of {
    pollfn  : unit -> bool,
    dofn    : abort_fn -> 'a,
    blockfn : (bool ref * abort_fn * (unit -> unit)) -> 'a,
    abortfn : abort_fn
  }

datatype 'a event
  = EVT 'a base_evt list
  | GUARD of (unit -> 'a event)
```

Figure 10.3: The representation of event values

sequential evaluation rule for "sync (**G** e)" in Chapter 7. The second phase corresponds to the notion of event matching (Definition 7.3), and consists of polling and either selection or logging (unlogging is done as part of the selection step). This involves accessing shared data structures and so must be done inside an atomic region. The fact that the second phase is done as an atomic operation greatly simplifies the implementation of `sync` (cf., Section 12.2.2). The actual implementation of the second phase is tuned for various common special cases, such as singleton events and events without any abort actions, but, to simplify the discussion, I describe the general case.

Forcing guards

The recursive forcing of guards is done by the following function:

```
fun forceGuard (GUARD g) = forceGuard (g ())
  | forceGuard (EVT evts) = evts
```

Once the guards (if any) have been forced, `forceGuard` returns a list of base-event descriptors (`evts`).

Polling

Polling the base events involves traversing the base-event list and calling the `pollfn` for each element, while extracting the abort action. The polling step produces a status value for each base event in the list. A base event's status is either *ready* or *blocked*, and either with or without an abort action (see Figure 10.4). If one or more of the base events is ready, then the blocked base events are irrelevant. Thus, once the polling loop sees a ready base

```
type 'a block_fn = (bool ref * abort_fn * (unit -> unit)) -> 'a
datatype 'a bevt_status
  = BLK of 'a block_fn
  | BLK_ABORT of ('a block_fn * (unit -> unit))
  | RDY of (abort_fn -> 'a)
  | RDY_ABORT of ((abort_fn -> 'a) * (unit -> unit))
```

Figure 10.4: The representation of event status

event, it can discard the status of any blocked base events. The polling of an individual base event's status is done by the following function:

```
fun pollBaseEvt (BASE_EVT{pollfn, dofn, blockfn, abortfn}) = (
      case (pollfn(), abortfn)
       of (false, NO_ABORT) => BLK blockfn
        | (false, ABORT a)  => BLK_ABORT(blockfn, a)
        | (true,  NO_ABORT) => RDY dofn
        | (true,  ABORT a)  => RDY_ABORT(dofn, a))
```

Selection

If the resulting list of base-event statuses includes one or more ready base events, then one of these is selected. The implementation uses a pseudo-random selection policy that gives probabilistic guarantee of fairness. A global counter is maintained; its value modulo the number of ready events is used to select one of the events. The counter is incremented after each selection and by the preemptive scheduler; the latter introduces a random element that helps avoid any kind of resonance in the selection patterns. Once a ready base event is selected, the abort actions of the other base events must be spawned and the `dofn` of the selected base event must be executed. The order in which this is done is tricky, since the `dofn` must be executed before leaving the atomic region and there is no guarantee that it will ever return (e.g., if a tail-recursive wrapper is involved). The solution is to pass the abort actions as an argument to the `dofn`, which invokes them immediately after leaving the atomic region. The actual argument is a single abort action that spawns all of the required abort actions. Section 10.4.3 describes the internals of the `dofns` of several base-event constructors.

Logging

If no base event is ready then the base events must be logged. Logging a base event requires capturing a continuation that, when thrown to, will spawn the abort actions of the other base events and apply the base event's wrapper functions. The base event, thread ID, and continuation together constitute a *base-event instance*. In order to understand the logging process it is also necessary to see how the blocking function works. Figure 10.5 gives the code for the logging loop and the skeleton of a typical blocking function. The logging loop

```
(* Logging loop *)
  System.Unsafe.capture (fn k => let
      val escape = System.Unsafe.escape k
      val dirtyFlg = ref false
      fun log ([], _) = atomicDispatch ()
        | log ((BLK bfn) :: r, i) = escape (
            bfn (dirtyFlg, allAborts, fn () => (log(r, i); error "[log]")))
        | log ((BLK_ABORT(bfn, _)) :: r, i) = escape (
            bfn (dirtyFlg, mkAbortFn i, fn () => (log(r, i+1); error "[log]")))
        | log _ = error "[log]"
      in
        log (sts, 0)
      end)

...

(* A typical blocking function *)
  fun blockFn (dirty, abort, next) = let
        fun block k = (
              add the thread to the waiting list;
              next())
        in
          case abort
           of NO_ABORT => (callcc block)
            | (ABORT a) => ((callcc block) before (a ()))
        end
```

Figure 10.5: Event logging

is implemented in continuation-passing style; the third argument to a block function, called `next` in the skeleton version, is a function that continues the logging loop. The function `error` reports an internal error by raising an exception; its principal purpose is to make the types work out. The functions `capture` and `escape` are unsafe versions of `callcc` that do not save or restore the exception handler continuation. They are required here in order that the logging of base events not break tail recursion, which is important since the wrapper functions often contain tail-recursive calls (e.g., the buffered channel in Section 5.1).

Unlogging

Once a particular base-event instance of an event is selected, the other base-event instances of that event must be unlogged. To support unlogging there is a boolean reference, called the *dirty flag*, for each event instance that is shared by its base-event instances. When one of the base-event instances is chosen, the flag is set to `true`, which marks all of the instances as being dirty. For the channel operations, the marking of the flag is done by the functions that remove items from the waiting queues (e.g., `cleanAndRemove`).

This trick of using a shared reference to mark the dirty instances was invented by Norman Ramsey [Ram90]. The reason for using this technique, instead of an explicit unlogging loop, is that it is simple and is constant time (since the cleaning of dirty items can be charged to their insertion operation). Unfortunately, there are certain situations in which this trick can result in unbounded heap growth. For example, if a thread is continuously selecting between communication on two channels, where one of the channels is never used, then the unused channel's waiting queue will be filled with dirty requests that are never removed. To avoid this problem, the channel must be cleaned when items are inserted, which destroys the constant-time bound on cleaning overhead. The dirty-flag technique still has the advantage of simplicity.

10.4.3 Base-event constructors

The simplest example of a base-event constructor is `always`, which builds an event that is always ready for synchronization. Its implementation is given in Figure 10.6. As expected,

```
fun always x = let
      fun doFn abortAct = (
            atomicEnd();
            case abortAct of (ABORT a) => a() | _ => ();
            x)
      in
        EVT[BASE_EVT{
            pollfn  = (fn () => true),
            dofn    = doFn,
            blockfn = (fn _ => error "[always]"),
            abortfn = NO_ABORT
          }]
      end
```

Figure 10.6: The implementation of `always`

the `pollfn` always returns true. The `dofn` is minimal; it leaves the atomic region, spawns the abort action (if any), and returns the argument with which the event value was created.

Since pollfn always returns true, blockfn is never called, and since this is a base event, there is no abort action.

A more complicated base-event constructor is transmit; the code for this is given in Figure 10.7. The implementation of transmit should be compared to the implementation

```
fun transmit (CHAN{inq, outq}, msg) = let
      fun pollFn () = (clean inq; isEmpty inq)
      fun doFn abortfn = let
            val (rid, rkont) = remove inq
            fun doit k = (
                  rdyQInsert (!runningThreadId, k);
                  runningThreadId := rid;
                  atomicEnd();
                  throw rkont msg)
            in
              case abortfn
               of NO_ABORT => callcc doit
                | (ABORT f) => (callcc doit; f())
            end
      fun blockFn (flg, abortfn, next) = let
            fun block k = (
                  insert(outq, flg, (getTid(), msg, k));
                  next(); error "[transmit]")
            in
              case abortfn
               of NO_ABORT => (callcc block)
                | (ABORT f) => (callcc block; f())
            end
      in
        EVT[BASE_EVT{
            pollfn = pollFn,
            dofn = doFn,
            blockfn = blockFn,
            abortfn = NO_ABORT
          }]
      end
```

Figure 10.7: The implementation of transmit

of send in Figure 10.2. The pollfn plays the role of the case in send; it cleans an the head of the channel's inq and returns true if there is an outstanding input request. The dofn corresponds to the case in send where there is a clean item in the queue; the sending thread is enqueued in the ready queue, a request is removed from the queue (note that remove takes care of marking the dirty flag) and the message is thrown to the accepter. If there are abort actions, then they are spawned by the sending thread's continuation. The blockfn corresponds to the case in send where there are no pending input requests. The blockfn inserts its continuation (which embodies any wrappers) in to the channel's outq and continues the logging loop. As with the dofn, any abort actions are spawned by the sender's continuation.

10.4.4 Event combinators

In terms of implementation, the simplest combinator is guard, which has the implementation:

```
val guard = GUARD
```

The implementations of the various other event combinators must deal with guarded event values. As discussed in Sections 4.5.1 and 7.2, the guard function is essentially a delay operation. When an event combinator is applied to a guard event, the guard is lifted to the top level. For example, the implementation of the wrap combinator handles the GUARD case as follows:

```
fun wrap (GUARD g, f) = GUARD(fn () => wrap (g(), f))
  | wrap ...
```

When the guarded wrapper is forced, g() will be evaluated to an event value that will be wrapped by f. The implementations of wrapHandler and wrapAbort handle GUARD in a similar fashion. The implementation of choose is a little more complicated and is discussed below.

The actual implementation of the wrap combinator is semantically similar to that described previously, but the implementation details are quite different. The wrapper function must be composed with the dofn and blockfn fields of each base-event descriptor. This is done by mapping the following function across the list of base-event descriptors:

```
fun wrapBaseEvt (BASE_EVT{pollfn, dofn, blockfn, abortfn}) =
      BASE_EVT{
          pollfn = pollfn,
          dofn = (f o dofn),
          blockfn = (f o blockfn),
          abortfn = abortfn
        }
```

where f is the wrapper function.

The wrapHandler combinator must also compose its wrapper with the dofn and blockfn fields of each base event, but the composition involves interjecting an exception handler. The function for wrapping a handler is:

```
fun wrapHBaseEvt (BASE_EVT{pollfn, dofn, blockfn, abortfn}) =
      BASE_EVT{
          pollfn = pollfn,
          dofn = fn x => ((dofn x) handle e => h e),
          blockfn = fn x => ((blockfn x) handle e => h e),
          abortfn = abortfn
        }
```

where h is the handler function being wrapped.

The choose combinator is fairly straightforward to implement. It essentially takes a list of lists and flattens them into a single list. If any one of the events in the argument list passed to choose is a guard event, then the guard is lifted to the top-level. Also, care must be taken to preserve the left-to-right order of evaluation of guards.

```
fun choose l = let
      fun f ([], el, []) = EVT el
        | f ([], el, gl) = let
            val applyGuards = revmap (fn g => (g ()))
            in
              GUARD(fn () =>
                choose ((EVT el) :: (applyGuards gl)))
            end
        | f ((EVT el') :: r, el, gl) = f (r, el' @ el, gl)
        | f ((GUARD g) :: r, el, gl) = f (r, el, g::gl)
      in
        f (l, [], [])
      end
```

The implementation of wrapAbort is the most interesting of the combinators. When wrapAbort is applied to a singleton event (i.e., an event consisting of exactly one base event), the implementation simply adds the abort action to the base event, which is done by the following function:

```
fun addAbortFn (BASE_EVT{pollfn, dofn, blockfn, abortfn}, a) =
      BASE_EVT{
          pollfn = pollfn,
          dofn = dofn,
          blockfn = blockfn,
          abortfn = (case abortfn
             of NO_ABORT => a
              | (ABORT a') => fn () => (spawn a'; a()))
        }
```

The more complicated case is when wrapAbort is applied to an event value consisting of n base events, where $n > 1$. The semantics of wrapAbort require that the abort action be spawned only in the case that none of the base events is chosen. This must be implemented in terms of the individual base-event abort actions, which are spawned if their base event is not chosen. In other words, the abort functions of the base events must coordinate to implement the abort action of the wrapped event. The way this works is that the abort actions are partitioned into a single `leader` action and $n - 1$ *follower* actions. A special channel is allocated for these threads to communicate by. Each follower sends an "I am here" message to the leader; the leader attempts to read $n - 1$ messages and then executes the abort action. If any one of the base-event actions is not spawned, i.e., because the corresponding base event is the selected one, then the abort action does not get executed. The channel used by these threads must be allocated anew for each synchronization attempt, so the creation of the abort actions is protected by a guard. The actual implementation of the resulting event value is:

```
GUARD(fn () => let
  val ackCh = channel()
  fun addFollowerAbortFn b =
        addAbortFn (b, fn () => send(ackCh, ()))
  val n = length followers
  fun leaderAbort 0 = abort()
    | leaderAbort i = (accept ackCh; leaderAbort(i-1))
  in
    EVT(
      (addAbortFn (leader, fn () => (leaderAbort n)))
         :: (map addFollowerAbortFn followers))
  end)
```

where `leader` and `followers` are, respectively, the base-event descriptors of the leader and follower abort actions.

10.5 Implementing I/O

I/O operations pose two problems for concurrent programming systems: first, the I/O devices (file descriptors in UNIX systems) are a form of shared state, and thus require concurrency control; and, second, input operations (and in some cases output operations) have the potential to block. As described in Sections 4.4 and 4.6, the implementation of **CML** supports I/O at two levels: synchronization on UNIX file descriptor conditions and a concurrent version of the **SML** stream I/O interface.

10.5.1 Low-level I/O support

The low-level I/O base-event constructors (e.g., `syncOnInput`) provide a mechanism similar to that of the UNIX system call `select` [UNI86], and, in fact, are implemented using this system call.

A global I/O waiting list is maintained by the implementation, with each entry corresponding to a particular instance of an I/O base-event value. Each time the preemptive scheduler is called, it dispatches a continuation that checks the status of the file descriptors in the I/O waiting list. This is done by projecting out the file descriptors of the non-dirty event instances in the waiting list and building the corresponding file descriptor sets. A call to the `select` system call is made to poll the file descriptors, which returns the set of ready descriptors. The threads waiting on the ready descriptors are then added to the ready queue.

The only complication to this scheme is handling I/O errors; e.g., if one of the files has been closed. In such a case, the system call `select` returns an error code, but no specification of which file descriptor is the source of the error. Since this situation is relatively rare, a moderately expensive, but simple, linear search for the bad file descriptors is used. Each file descriptor is tested by a call to the `ftype` system call, which returns an error if, and only if, the file descriptor is the source of the error. This error is then mapped back to the blocked thread by raising an exception in its context. This requires saving two continuations in the block function; one for successful synchronization and one for error condition. To make this all concrete, the I/O waiting list data structure and the `synchOnInput` event constructor's block function are given in Figure 10.8. Each `io_item` in the waiting list corresponds to a pending I/O base-event instance, and contains the file descriptor, type of operation, the waiting thread ID and the two possible continuations.

10.5.2 Stream I/O

CML includes a structure `CIO`, which implements a concurrent version of **SML** I/O streams (see [Rep90b] for a complete description). There are two types of I/O streams, *in-streams* and *out-streams*, which provide buffered input and output operations. Each open stream is represented by a thread, which implements the buffering. For out-streams, the protocol is straightforward and uninteresting.[9] The in-stream protocols, however, are an example of the advanced use of events to build complex communication abstractions.

Because input operations might block indefinitely (e.g., while waiting for the user to enter a line of text), it is necessary to provide an event-valued interface to in-streams. The `CIO` structure includes the following operations:

```
val inputEvt     : instream * int -> string event
val inputLineEvt : instream -> string event
val lookaheadEvt : instream -> string event
```

[9] The implementation makes the simplifying assumption that write operations are non-blocking.

```
datatype io_operation_t = IO_RD | IO_WR
type io_item = {
    fd       : int,             (* the file descriptor *)
    io_op    : io_operation_t, (* the kind of operation *)
    id       : thread_id,       (* the waiting thread's id *)
    kont     : unit cont,       (* the successful continuation *)
    err_kont : unit cont,       (* the error continuation *)
    dirty    : bool ref         (* the dirty bit *)
  }
val ioWaitList = ref ([] : io_item list)
  ...
fun inputBlockFn (flg, abort, next) = (
      callcc (fn okay_k => (
        callcc (fn err_k => (
          ioWaitList := {
              fd=fd, io_op=IO_RD, dirty=flg, kont=okay_k,
              err_kont=err_k, id=getTid()
            } :: !ioWaitList;
          next()));
      (* continue here on an error *)
        applyAbortFn abort;
        raise (InvalidFileDesc fd)));
    (* continue here on success *)
      applyAbortFn abort)
```

Figure 10.8: Low-level I/O support

The function `inputEvt` builds an event value for reading a specified number of characters, `inputLineEvt` builds an event value for reading a line of input, and `lookaheadEvt` builds an event value for examining the next character to be read from the buffer. As an illustration, the following function either reads a line of input or times out:

```
fun getAnswer t = select [
        wrap (inputLineEvt std_in, SOME),
        wrap (timeout t, fn () => NONE)
      ]
```

In order for code of this sort to work properly, the implementation of the in-stream event constructors must satisfy the following two requirements:

(1) The commit point of the event must correspond to the availability of input that satisfies the request.

(2) Input must never be lost or discarded.

The implementation uses a request-reply protocol (a simple version of this is described in

Section 4.5). In order to meet requirement (1), the commit point must be the server's reply, which means that the request must be generated by a guard. Meeting requirement (2) means that the server thread must be informed that the request has been aborted. This is the scenario discussed in Section 4.5.2, and the client-side code is similar to that of `clientCallEvt5`. For example, the client-side implementation of `inputLineEvt` is:[10]

```
fun inputLineEvt (INSTRM{req_ch, ...}) = guard (fn () => let
      val abortCh = channel() and replyCh = channel()
      in
        spawn (fn () =>
          send (req_ch, INPUT_LN(receive abortCh, replyCh)))
        wrapAbort (receive replyCh, fn () => send(abortCh, ()))
      end)
```

The event value constructed by this function is a guard that sends a request to the server consisting of the operation, an abort event and a reply channel. The commit point of this event is receiving a reply from the server. If the client synchronizes on some other event, then the abort action sends an abort message to the server. On the server side, when a request comes in the server attempts to satisfy it — either from the input buffer or by requesting input from the operating system. Once the server can satisfy the request, it synchronizes on the choice of sending the input as a reply and receiving an abort message on the abort channel. In the latter case, the input is reinserted into the buffer.

10.6 Implementation improvements

The current implementation of **CML** uses practically no specialized run-time or compiler support. There are a number of techniques that could be used in the run-time system and compiler that would improve performance.

A very simple modification, which requires little work, is to introduce a dedicated register for referring to the thread ready queue. This would have two benefits: it would reduce memory traffic, and it would eliminate the store-list allocations associated with ready queue updates. Using the `varptr` register to refer to the current thread ID, instead of a global `ref` variable, improved the speed of thread context switching by over 10%. This suggests that thread context switching might improve by as much as 20% by use of a dedicated ready queue register. And as the clock speed of RISC processors increases, the potential savings become larger.

In the **PML** compiler, the event-value constructors and channel operations are explicitly represented by primops. The **PML** compiler does very aggressive intermodule inlining, thus

[10] The actual implementation uses a condition variable for the abort message (see Section 5.3).

it is able to recognize applications of `sync` to static event values, which can be optimized into more efficient operations. For example, `sync (receive ch)` can be replaced by `accept ch`, which is about 40% faster. A more modest improvement (about 5%) is achieved by inline expansion of `sync`, `transmit` and `receive`.

It is also possible for compilers to recognize more complex communication patterns. A common example is the use of a thread to encapsulate state, with an RPC interface. In this case, each time the server thread is dispatched to handle some operation, there is a client thread that is suspended, waiting for the reply. A more efficient implementation of this pattern is to use a monitor (see Section 3.2.2) to encapsulate the state.[11] When there is no contention, monitors avoid the necessity of context switches on entry and exit; a comparison of the costs in the context of **Ada** can be found in [EHP80]. Instead of providing monitors at the language level, it may be possible for the compiler to detect when a monitor is suitable and translate the RPC operations to monitor calls; this has been done for **Ada** rendezvous [HN80]. Unlike **Ada**, **CML** does not have the syntactic signposts that mark RPC-style interactions, since they are derived operations. Thus, automatically recognizing that an RPC operation can be single-threaded is a very difficult problem. Other communication patterns that might be recognized by the compiler include channels that are used exactly once and channels that are used for point-to-point communication. For example, consider an RPC protocol in which the reply channel is dynamically allocated for each request (e.g., the implementation of input streams described in Section 10.5.2). If the reply channel is used exactly once, then it can be replaced with a condition variable, which reduces communication overhead by 30%.

[11] This may not be the case on non-uniform memory access multiprocessors, where the client and encapsulated data are on different processors (see Chapter 12).

Chapter 11

Performance

In the previous chapter, I described the implementation of **CML** in detail; in this chapter, I report various performance measurements that I have made of this implementation. The measurements include timing results for a collection of small benchmarks on three different workstations, and instruction counts for these benchmarks on the MIPS R3000 processor. In addition, I compare the performance of **CML** with the μ**System**, a **C**-based thread package. These measurements show that **CML** provides a high-level notation at a competitive price.

11.1 The benchmarks

I have conducted a series of benchmarks on three different machines, representing three different processor architectures. Table 11.1 summarizes the features of these computers. The benchmarks measure the cost of low-level concurrency operations, such as sending a

Table 11.1: Benchmark machines

	NeXT	**SPARC 2**	**DEC 5000**
Full name	NeXT Cube	SPARCstation 2	DECstation 5000/120
Processor	25MHz 68040	40MHz SPARC	20MHz R3000
Memory	24Mb	64Mb	16Mb
Operating System	NeXTstep 2.1	SunOS 4.1.1	ULTRIX 4.2

message, so to get accurate numbers I measured the time to perform 100,000 operations.[1] For each benchmark, I measured the CPU time spent executing the program (both user and system) and the time spent in the garbage collector. All times are in micro-seconds. The benchmarks were run using version 0.75 of **SML/NJ** (released November 11, 1991), and version 0.9.6 of **CML** (released October 11, 1991).

[1] 10,000 iterations of a loop of 10 operations.

The benchmarks are logically divided into two groups; the first measure the basic concurrency primitives:

Thread switch. This measures the cost of an explicit context switch.

Thread spawn/exit. This measures the time it takes to spawn and run a null thread. It includes the cost of two context switches: the `spawn` operation switches control to the newly spawned thread and a terminating thread must dispatch a new thread.

Rendezvous. This measures the cost of a `send`/`accept` rendezvous between two threads.

Event rendezvous. This is an implementation of the rendezvous benchmark using `sync` composed with `transmit` and `receive`, instead of `send` and `accept`.

The second group measures the cost of several different versions of an RPC implementation of a simple service. The service is essentially a memory cell; a transaction sets a new value and returns the old value.

RPC. This uses `send` and `accept` to implement the protocol. The client-side code is:

```
fun call x = (send (reqCh, x); accept replyCh)
```

Event RPC. This implements the protocol as an event value. The client-side code is:

```
fun call x = sync (
      wrap (transmit(reqCh, x), fn () => accept replyCh))
```

Fast RPC. This uses a *condition variable* (see Section 5.3) to implement a fast, asynchronous reply. The client-side code for a call is:

```
fun call x = let val replyVar = condVar()
      in
        send (reqCh, (x, replyVar));
        readVar replyCh
      end
```

11.1.1 Timing results

The measured times for all of the benchmarks are given in Table 11.2. Each entry has the form $t + g$, where t is the combined user and system time for the operation and g is the amortized garbage collection overhead. Each entry is the average of five test runs, and there was little deviation between runs. Real-time measurements were only slightly higher than than the CPU time measurements.

Table 11.2: **CML** benchmarks

	Time in μS / Operation (program + garbage collection)		
Operation	**NeXT**	**SPARC 2**	**DEC 5000**
Thread switch	23+4	18+4	15+7
Thread spawn/exit	47+7	46+7	42+13
Rendezvous	54+10	50+7	49+20
Event rendezvous	110+12	90+9	88+23
RPC	105+20	95+15	90+38
Event RPC	171+21	134+15	125+39
Fast RPC	79+11	68+9	65+23

11.1.2 Instruction counts

Andrew Appel modified the MIPS code generator to generate instrumentation that counts the number of executed instructions. Using this mechanism, I measured the instruction counts for the various benchmarks. Table 11.3 gives the results of these measurements. These numbers do not include the loop or garbage collection overhead.

Table 11.3: MIPS instruction counts

Operation	**Instructions / Operation**
Thread switch	197
Thread spawn/exit	483
Rendezvous	558
Event rendezvous	934
RPC	1,008
Event RPC	1,346
Fast RPC	711

These numbers are higher than one might expect (particularly for the thread switch and creation operations). In [Rep91a], I reported that a thread switch required around 100 instructions on the SPARC processor, which is about half of what I measured for the MIPS. Since the MIPS and SPARC are both RISC processors, the difference is not one of instruction sets, but rather is because of changes in the **SML/NJ** compiler (Appel, personal communication, December 1991). There is some hope that future improvements in the compiler will reduce the instruction counts to more reasonable values (for example, a thread context switch should require no more than 75 instructions).

11.2 Analysis

The measurements show that the penalty for using abstract interfaces (i.e., hiding channel communication in event values) is acceptable. Table 11.4 gives the ratio between the non-GC time of the event version and the non-event version of the two communication protocols I benchmarked. For a simple rendezvous, the performance cost of using events is about 80%;

Table 11.4: Cost of abstraction

	Protocol			
	Rendezvous		RPC	
Machine	cost (μS)	ratio	cost (μS)	ratio
SPARC 2	40	1.8	39	1.4
DEC 5000	39	1.8	35	1.4

for the RPC it is 40%. The reason for the lower impact on the RPC protocol cost is that only one of the two communications is being represented by an event value. In general, only the communication that is the commit point needs to be implemented using an event value; communications in the guard and wrapper can be implemented using `send` and `accept`.

11.2.1 Garbage collection overhead

The high garbage collection overhead in these benchmarks is mostly a result of the way the current **SML/NJ** collector, which is a simple generational collector, keeps track of inter-generational references [App89]. Each time a mutable object is updated, a record of that update is added to the *store list*. This store list is examined for potential roots at the beginning of each garbage collection. The implementation of **CML** uses a small number of very frequently updated objects: the thread ready queue, current thread pointer and channel waiting queues. This "hot-spot" behavior is the worst-case scenario for **SML/NJ**'s collector, destroying the $O(|\text{LIVE}|)$ normally expected from copying collection. The collector also suffers from the problem of poor "real-time" responsiveness.

11.3 Comparison with the μSystem

To put these measurements into perspective, I implemented a similar set of benchmarks in version 4.4 of the μ**System**, which is a **C** light-weight process library [BS90]. The μ**System** provides threads and a request/reply communication primitive (it also has shared-memory primitives), but it does not have selective communication. It runs on the SPARCstation and DECstation, but not on the NeXT. Table 11.5 reports the results for the SPARCstation

Table 11.5: μ**System** benchmarks

Operation	**SPARC 2** Time (μS)	Ratio	**DEC 5000** Time (μS)	Ratio
Task switch	62	2.8	13	0.6
Task create	161	3.0	59	1.1
Send/receive	127	2.2	42	0.6
Send/receive/reply	128	1.7	43	0.5

and DECstation. As before, the times are given in micro-seconds and represent the sum of the user and system CPU times; obviously there is no garbage collection overhead. The column labeled "ratio" gives the ratio of the μ**System** and **CML** times (including garbage collection overhead); a ratio greater than 1.0 means that **CML** is faster.

On the SPARCstation, **CML** is uniformly faster than the μ**System**. The principal reason for this is that **SML/NJ** does not use the SPARC's register windows, and thus does not have to flush them on a thread switch. The comparison for the DECstation is not as favorable, but **CML** is still competitive, even though the μ**System** provides a lower-level concurrency model (no selective communication, for example). This shows that we can have the advantages of the higher-level language without sacrificing performance.

Chapter 12

Multiprocessors

While the main thrust of this dissertation is the study of concurrency as a tool for structuring programs, it is worth considering the issues associated with a possible multiprocessor implementation of **CML**. In this chapter, I survey various parallel language features and parallel programming techniques and discuss how they might apply to parallel programming in **CML**. I also discuss the implementation issues that must be addressed in a multiprocessor implementation of **CML**, and finally summarize the prospects for multiprocessor **CML**.

For purposes of this chapter, I assume a multiple-instruction multiple-data (MIMD) machine with a shared address space. There are many experimental and commercial examples of these machines, and it is reasonable to expect that they will appear on desks in the near future. Although these machines provide a shared-memory model, they usually also have non-uniform memory access (NUMA); e.g., each processor may have a local memory or at least a cache. Because of NUMA, maintaining locality is important for good performance, and as the number of processors increases NUMA effects become more pronounced. Programs written in a message-passing language typically have good locality, and can out-perform the shared-memory versions [LS90].

There are several benefits to be obtained from a multiprocessor implementation of **CML**. Although most existing applications, such as **eXene**, have fairly limited amounts of parallelism (typically only a few ready threads at a time), a multiprocessor implementation should result in noticeable performance improvements for many existing applications. Owicki reports improvements for these kinds of applications written in **Modula-2+** running on a Firefly multiprocessor [Owi89]. In particular, she mentions that the Trestle window system exploits the multiprocessor by pipelining graphics operations. **EXene** is designed with some pipelining, so that running on a multiprocessor should improve its performance. And, of course, all programs, including highly sequential ones, will benefit from parallel garbage collection.

With a multiprocessor implementation available, it becomes reasonable to implement parallel algorithms. Some examples are parallel attribute grammar evaluation [Zar90], parallel theorem provers [BCLM89], and parallel graphics algorithms [Gre91].

12.1 Parallel programming in CML

This section examines the use of **CML** as a parallel programming language. There are two parts to this discussion: first, I discuss several parallel programming techniques and how they might apply to **CML**; and second, I describe possible extensions to provide better support for parallel programming. The proposed extensions do not represent changes in the semantics of **CML**; rather, my approach is to define new communication operations that can be derived from the **CML** primitives, but that are also amenable to efficient implementation on multiprocessors.

12.1.1 Pipelining and data-flow

One class of parallel programs naturally expressed in **CML** are those programs that can be structured as *data-flow* networks [KM77]. A data-flow network consists of a graph of computation nodes, where the edges are communication links (Landin's *streams* [Lan65] are a precursor to this). A data-flow graph does not have to be static; a computation node can be replaced by subgraph having the same I/O interface.

Data-flow graphs provide parallelism in two ways. If two computation nodes do not depend on each other for data, then they can compute in parallel. Even if there is a dependency, if they operate on a sequence of values then they form a *pipeline*. In order for a data-flow network to be efficient, the granularity of the operations at individual nodes must be large enough to compensate for the communication overhead.

Using threads for the computation nodes and channels for the edges, a data-flow network can be directly implemented in **CML**. For example, **eXene**'s thread network is essentially a data-flow network (see Section 9.1.3). Another example is given in [Rep90b], where I describe a pipelined implementation of the *Sieve of Eratosthenes* using **CML**.

A nice illustration of the use of data-flow networks is given by McIlroy in [McI90], where he describes the use of processes and channels to compute power series (this was actually suggested in [KM77], but without implementation details). McIlroy represents a power series as a stream of rational coefficients. For example, the power series for the exponential function

$$e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!} = 1 + x + \frac{1}{2}x^2 + \frac{1}{6}x^3 + \cdots$$

can be implemented in **CML** as follows:

```
fun e () = let
      val ch = channel()
      fun loop (i, ifact) = (
            send(ch, {num=1, denom=ifact});
            loop(i+1, (i+1)*ifact))
      in
        spawn (fn () => loop(0, 1));
        ch
      end
```

Using this representation, operations such as addition and multiplication of power series can be coded up as networks of threads (see [McI90] for details).

12.1.2 Controlling parallelism

One of the key problems in writing parallel programs is avoiding excessive parallelism. A basic technique in many parallel programs is to divide a problem into two or more pieces and to spawn a thread for each piece. If there are many more pieces than processors, then this technique leads to excessive parallelism and the cost of thread management can dominate the execution time. Premature limiting of parallelism, however, can result in *starvation*; i.e., idle processors without any work.

Work crews

One approach to limiting excessive parallelism in concurrent programs is the *work crew* abstraction [RV89]. In this scheme, a fixed set of threads, called *workers*, execute jobs taken from a queue,[1] where a job is a piece of computation. When a worker gets a job that can be computed in n parallel pieces, it chooses one piece for itself and generates $n - 1$ *help requests* for the remainder. When the worker finishes the its piece of work, it then checks to see if the help requests have been answered. If not, then the worker computes the next piece of the job, and so on until the job is completed. After completing the job, the worker looks for help requests from other workers; if it finds one, that becomes its next job. Figure 12.1 sketches the code for a job consisting of three pieces. In addition to providing a mechanism for limiting parallelism, work crews also have the important property of *breadth-first* parallel decomposition, which results in coarse-grain parallelism. And, since jobs are

[1]In [RV89], the term *tasks* is used to refer to jobs.

```
fun job () = (
      requestHelp (job₂);
      requestHelp (job₃);
      do job₁;
      if noHelp(job₂) then  do job₂ else ();
      if noHelp(job₃) then  do job₃ else ())
```

Figure 12.1: Work crew job decomposition

decomposed eagerly, worker starvation is avoided. When no extra workers are available for a job, execution reduces to the standard sequential order.

The structuring of job decomposition in a breadth-first manner is probably the most important benefit that work crews would provide **CML**. Since thread creation and space overhead are low, a **CML** implementation of work crews could use a thread for each job, but only enable a subset of threads to run at any time. A "token" mechanism could be used for this, where each job (i.e., thread) would wait for a token before executing, and would pass the token to the next job when it completed. In this scheme, a token holder corresponds to a worker.

Futures

The semantics of futures in concurrent **Lisp** systems provide another opportunity for limiting parallelism. Consider the general form of a future creation:

K `(future` e`)`

where K is the context of the future call. The keyword `future` can be viewed as an annotation, which tells the compiler that e is a good candidate for parallel evaluation. The actual evaluation of e, however, can be immediate (called *inline evaluation*), in parallel with K, or when K demands its value. One approach is to choose dynamically between inline and parallel evaluation of e based on the current load; this is called *load-based inlining* [KH88, MKH91]. A problem with this approach is that the rate of thread creation in a program may not be uniform, so a decision to inline a future at one point may lead to starvation later. Furthermore, load-based inlining can introduce deadlock [MKH91].

An alternative to load-based inlining is lazy task creation, which is is a scheme that always inlines the evaluation of e, but saves enough information to spawn a thread to evaluate K in parallel if the number of ready tasks falls below the number of processors [MKH91]. This scheme is quite similar to work crews in its effect, but requires less effort by the programmer. Futures with lazy task creation can be implemented in **CML** fairly easily. Assuming that we have a global channel

```
val continue : unit chan
```

for sparking new threads, then the `future` operation is implemented as follows:[2]

```
fun future f x = let
      val resVar = condVar()
      in
        spawn (fn () => writeVar (resVar, f x));
        select [
            receive continue,
            wrap (readVarEvt resVar, fn _ => ())
          ];
        readVarEvt resVar
      end
```

The idea is that an idle processor sends a message on the `continue` channel to wake up some waiting thread. Since channel communication is FIFO, this results in the desired breadth-first problem decomposition. Of course, it would be much more efficient to directly implement futures and lazy task creation using `callcc` and the techniques of [MKH91]. For **CML**, a principal advantage of lazy task creation is that it doesn't introduce deadlock; even in the case when the body of a future attempts to synchronize with the future's parent.

12.1.3 Speculative parallelism

Certain classes of parallel programs, such as parallel search, use speculative parallelism to improve performance [Osb89]. For example, consider the problem of finding an item in a balanced binary tree; by searching subtrees in parallel, the running time of the search is reduced from $O(n)$ to $O(\log n)$, given a sufficient supply of processors. Although this example is trivial, it is illustrative of problems arising in applications such as theorem provers. In addition to the problem of controlling excessive parallelism (discussed in Section 12.1.2), there is the problem of terminating unnecessary computations (e.g., once one thread has found the item, there is no reason for the others to keep searching). **CML** does not currently support the asynchronous termination of threads, thus it would be necessary to add `kill` operation on thread IDs. The other aspect of thread termination is recognizing which threads need to be killed. It is also important to note that the speculative threads must be referentially transparent, otherwise killing them changes the semantics of the program.

[2]Note that this version is simplified by ignoring the issue of exceptions (cf., Section 5.5).

One attractive, but tricky, approach is to garbage collect those threads that are able to run, but are irrelevant to the future execution of the program.[3] One way to do this is to give the garbage collector special knowledge about channel and thread objects, which allows it to trace thread interconnections [BH77, KNW90]. The problem with using this technique for **CML** is that it does not interact well with the implementation of threads as ordinary **SML** values. Another strategy, which might be more suitable for **CML**, is judicious use of *weak pointers* in the representations of some of the concurrency objects [ME89]. Weak pointers, which are already supported by **SML/NJ**, are a way to hold a reference to an object while allowing the garbage collector the option of collecting it. If, during a garbage collection, the only references to an object are weak pointers, then the garbage collector collects the object and nullifies the weak pointers that refer to it. Using weak pointers, a future mechanism can be implemented that gives the parent thread a strong pointer to the future object and gives the child thread a weak pointer to it. If the parent discards its reference to the future object, then the child's weak pointer is nullified, and, using object finalization [Rov85], the child thread is collected. A similar scheme is described in [ME89].

A simple technique, which has similar utility to the weak pointer scheme above, is to exploit the `guard` and `wrapAbort` combinators to implement a speculative fork operation:

```
fun fork f x = guard (fn () => let
      val cv = condVar()
      val id = spawn (fn () => writeVar(cv, f x))
      in
        wrapAbort (readVarEvt cv, fn () => kill id)
      end)
```

Using multiple instances of this in a `select` implements *or-parallelism*. The first thread to finish provides the answer and triggers the abort actions of the other choices, which kill the other threads. For example, the following function sorts a list while testing to see if it is already sorted in parallel:

```
fun fastSort l = select [
        fork (fn () => sort l),
        fork (fn () => if (isOrdered l) then l else exit())
      ]
```

where `sort` is some sorting function and `isOrdered` tests a list to see if it is sorted.

[3]The **CML** implementation collects unreachable blocked threads, but not threads that are ready (i.e., threads that are reachable via the ready queue).

12.1.4 I-structures

The parallel programming functional language **Id** provides a form a mutable state called *I-structures* [ANP89, Nik91]. I-structures come in various flavors, including aggregate structures such as arrays.

Condition variables are essentially the value-return mechanism of a future. Futures have been promoted as a useful mechanism for parallel programming by the **Lisp** community (e.g., [Hal85] and [KH88]). Although, as discussed in Section 5.5, futures can be implemented using channels, an implementation based on condition variables has the significant advantage of avoiding a context switch each time the value of the future is read.[4]

Condition variables are an example of what are called *I-structures* in the parallel language **Id** [ANP89, Nik91]. **Id** provides I-structures in various flavors, including aggregate structures such as arrays. A discussion of the use of I-structures in parallel programs and some small example programs can be found in [ANP89].

12.1.5 M-structures

Another form of state supported by **Id** is the *M-structure* [BNA91]. Like I-structures, M-structures are either empty or full. There are two basic operations on M-structures: *put*, which initializes a cell and, like I-structures, raises an exception if the cell already has a value; and *take*, which removes and returns the contents of a cell (making it empty). The take operation forces synchronization, since a thread may have to wait for another thread to put a value into the cell. This is similar to the "I/O ports with memory" described in [KS79]. M-structures can also be viewed as finitely buffered asynchronous channel with only one slot. In **CML**, M-structured variables have the following interface:

```
type 'a mstruct

val mstruct : unit -> '1a mstruct

val put : ('a mstruct * 'a) -> unit
exception Put

val takeEvt : 'a mstruct -> 'a event
val take    : 'a mstruct -> 'a
```

Since M-structures are mutable, the allocation function is weakly polymorphic. As mentioned above, the `take` operation involves synchronization, so an event-valued form is also provided. If a `put` is attempted on a full cell, the exception `Put` is raised.

[4] Condition variables have proved quite useful in **CML** programs when "single-shot" communication is required (e.g., for abort messages, see Section 10.5.2).

M-structures can be updated atomically if threads use the following update protocol:

put (m, f (take m))

where f computes the new value of m from the previous value. The reason why this update is atomic is that the take operation locks the variable against take operations by other threads (this is similar to the safety of shared request-reply channels discussed in Section 4.2). Paul Barth at MIT has developed a number of parallel algorithms in **Id** using M-structures (Barth, personal communication, August 1991); some examples can be found in [BNA91].

M-structures can be defined as a derived feature in **CML**; Figure 12.2 gives an implementation of the above interface. In a parallel implementation of **CML**, M-structures might

```
datatype 'a mstruct = M of {
    full_ch : unit chan,
    take_ch : 'a chan,
    put_ch : 'a chan
  }

fun mstruct () = let
      val fullCh = channel()
      val takeCh = channel() and putCh = channel()
      fun undefined () = defined (accept putCh)
      and defined v = select [
              wrap (transmit(takeCh, v), fn () => undefined()),
              wrap (transmit(fullCh, ()), fn () => defined v)
            ]
      in
        spawn undefined;
        M{full_ch = fullCh, take_ch = takeCh, put_ch = putCh}
      end

fun takeEvt (M{take_ch, ...}) = receive take_ch
fun take (M{take_ch, ...}) = accept take_ch

exception Put
fun put (M{full_ch, put_ch, ...}, x) = select [
        wrap (receive full_ch, fn () => raise Put),
        transmit (put_ch, x)
      ]
```

Figure 12.2: **CML** implementation of M-structure variables

be implemented directly on top of the low-level shared-memory primitives, making them very efficient. Other operations that **Id** supports on M-structures, such as a *non-destructive read* operation, could also be directly supported.

12.2 Multiprocessor implementation

Designing and building a high-performance multiprocessor **CML** implementation is a major research project in its own right, and I leave it for future work. It is possible, however, to identify and discuss some of the major implementation issues.

12.2.1 Concurrency control

The uniprocessor implementation described in Section 10.4 relies on a single *global* mutex lock for guaranteeing atomic access to the channel and thread data structures (see Section 3.2.1 for a description of mutex locks). On a uniprocessor, using a global lock is the most efficient approach, since it reduces locking overhead and does not cause any loss of parallelism. For multiprocessors, however, a single global lock is likely to cause contention and idle processors. For example, on a four processor machine ($P_{\{1,2,3,4\}}$), a thread on P_1 should be able to communicate with a thread on P_2 in parallel with communication between threads on P_3 and P_4. This means that channels must be locked independently.

Different multiprocessors provide different kinds of support for locking. A common mechanism is the *test-and-set* instruction, which atomically applies the following function to a word:

```
fun testAndSet w = if !w then true else (w := true; false)
```

This operation can be used to implement a *spin-lock*, which is busy-waiting mutex lock:

```
fun aquireSpinLock w = if (testAndSet w)
      then (aquireSpinLock w)
      else ()
fun releaseSpinLock w = (w := false)
```

A more sophisticated implementation might use *exponential back-off* or other techniques to improve performance (see [And89] and [CS91] for a comparison of locking techniques). Some multiprocessors do not provide hardware support for locking, but Lamport has developed an algorithm for these cases, which is optimal in the number of memory reads and writes [Lam87]. Other machines only provide test-and-set on a limited number of memory locations, in which case software locks must be implemented on top of the hardware supported spin-locks.

12.2.2 Generalized selective communication

With the introduction of a separate lock on each channel's data structure, the implementation of sync applied to a choice of multiple communications becomes significantly more complicated. For example, a naïve implementation of `select` on a list of communications is to first grab the locks of all the channels and then do the operation. This fails in the following situation. Assume there are two threads t_1 and t_2, running on different processors, with t_1 attempting

select [receive c_1, transmit (c_2, v)]

while simultaneously t_2 attempts

select [receive c_2, receive c_1]

This can result in a situation in which t_1 holds a lock on c_1 and needs a lock on c_2, while t_2 holds a lock on c_2 and needs a lock on c_1 — i.e., *deadlock*. There are various known algorithms for this problem (e.g., see [BS83], [Bor86], or [Bag89]). The basic strategy is to first make *tentative* offers of communication; when two tentative offers match, one thread must freeze its state until the other thread either commits or rejects the communication. The choice of which thread will fix its state is based the order of the threads' IDs; this avoids the possibility of cyclic dependencies and deadlock. Greg Morrisett has implemented a protocol similar to [Bor86] on top of **ML-threads** (Greg Morrisett, personal communication, July 1991).

12.2.3 Thread scheduling

The techniques and data structures used for thread scheduling can have a significant impact on multiprocessor performance, because of contention for thread queues and cache consistency effects.

A single global scheduling queue would be a significant source of contention. Furthermore, a single queue does not provide any mechanism for keeping a thread on a single processor, which is important for preserving cache consistency. Accordingly, as a first cut, it is clear that each processor should have its own queue of ready threads. Some policy is needed to balance out the load. One possibility is to balance the scheduling queues at garbage collection time. Since typical memory allocation rates in **ML** programs are high (on the order of 5 to 10 megabytes per second on a SPARCstation-2), a processor that runs out of work would not have to wait long for load balancing. A fall-back would be to allow an idle processor to force a garbage collection if it has been idle for more than a few milliseconds. This scheme has the advantage of insuring that the scheduling queue is only accessed by its processor (except during load balancing), which means that a light-weight locking mechanism, such as that used in the single processor implementation (see Section 10.2.2), can be used to protect the queue. Since the scheduling queue is the single most heavily accessed shared data structure, this scheme might provide good performance. The question that needs to be answered by empirical tests is how often do processors run out of threads to schedule?

The implementation of **Mul-T** uses two thread queues per processor; one for threads that have never run, called the *new thread queue*, and one for threads that have been suspended, called the *suspended thread queue*. When selecting a new thread to dispatch, the processor's scheduler first looks in its own suspended thread queue, then in its own new thread queue, then in other processors' new thread queue and lastly, if it has not found a thread, it looks in the other processors' suspended thread queues. By selecting new threads over suspended threads when migrating threads, the impact on cache consistency of thread migration is reduced.

12.2.4 Memory management

Implementations of heap-based languages, such as **SML** or **CML**, live or die by the performance of their memory allocation and garbage collection techniques. An efficient multiprocessor implementation of **CML** must address several memory management issues. The most important of these is avoiding contention during memory allocation. The standard scheme to address this problem is to divide the allocation space into multiple chunks and to give each processor its own allocation chunk (e.g., [AEL88], [KH88], [ME89] and [Mar91]). When a processor fills its allocation chunk, it grabs another from the global list of free chunks. The only source of allocation contention are the accesses to the global chunk list, which are relatively rare.

When the allocation chunks are exhausted, it is necessary to perform a garbage collection. For a "stop-the-world" collector, this first requires synchronizing the processors, so that they are all in *collection state*. One possible technique to force synchronization is to have the processor that notes the need for garbage collection use a UNIX signal to notify the other processors [KH88]. Another approach is to wait for the other processors to exhaust their allocation chunks [Mar91]. To avoid problems in the unlikely case of an infinite, non-allocating, computation, a global flag is set that is checked by the `SIGINT` signal handler. Since allocation rates in **SML/NJ** are very high (typically one 4-byte word per 5-10 instructions), the idle-time of the processor that initiated the garbage collection might be less costly than the overhead of using signals to interrupt the other processors.

Once all of the processors are in collection state, the garbage collection can begin. The simplest technique is to run a standard collection algorithm on a single processor. This has the clear disadvantage, however, of increasing the cost of garbage collection relative to the rest of the program. A sequential collector is a performance bottleneck; it is much more desirable to garbage collect in parallel. There are a number of systems using parallel garbage collection (e.g., [KH88], [ME89] and [Mar91]). The techniques of [Mar91] seem to fit the **SML/NJ** memory management system fairly well. In this scheme, each processor has its own to-space. When a collector process encounters a reference to a from-space object while sweeping its to-space, it examines the object's descriptor. If the descriptor is a forward pointer, then the collector process updates the reference in its to-space. If the object has not been forwarded, then the collector process locks the descriptor word, allocates space in its to-space, sets the forward pointer, unlocks the descriptor, and then copies the object. For machines, like the SGI 4D/380, which have a limited number of hardware locks, a hashing scheme on the object's address can be used to multiplex the hardware locks. Since the lock on the object's descriptor is only held for a few instructions, contention should be rare.

These techniques still suffer from the problem that they stop the world during garbage collection. Although the use of generational techniques reduces the frequency of noticeable pauses [Ung84], providing uniform responsiveness for real-time applications, such as user interfaces, requires interleaving garbage collection activities with mutator computation. On a multiprocessor, the most obvious approach is to dedicate one or more processors to the task of garbage collection. The principal technical problem with interleaving mutator and garbage collection activity is synchronization. If synchronization overhead is high, then any performance benefits will be lost. In lieu of special purpose hardware, the virtual memory system can be used to implement synchronization [AEL88].

12.3 The outlook for multiprocessor CML

This chapter has described a number of issues related to the implementation and use of a multiprocessor version of **CML**. Some work has already been done towards supporting **CML** on multiprocessors. Greg Morrisett has implemented a low-level library of multiprocessing primitives, such as spin-locks, for **SML/NJ** on the SGI 4D/380 [Mor]. This should provide a suitable base for implementing a multiprocessor version of **CML**.

Once a multiprocessor implementation exists, it will be possible to experiment with different styles of parallel programming. The flexibility provided by first-class synchronous operations means that **CML** can accommodate different parallel programming paradigms without serious disruption or incompatibilities with existing code. Condition variables and some form of M-structures should provide the right primitives for programming parallel algorithms, while being semantically consistent with **CML**'s other primitives, and the techniques of work crews and lazy futures should provide reasonable mechanisms for controlling parallelism.

Part V

Conclusion

Chapter 13

Future Work

Although this dissertation is a comprehensive treatment of the design, semantics, application and implementation of a concurrent language, there is still room for additional research and implementation. Following the structure of this dissertation, the topics for future research are divided into design, theory and practice.

13.1 Design

The design of **CML** has evolved for a number of years based on practical experience and is now fairly mature. Given the amount of practical experience with the mechanisms, it is unlikely that **CML** will change in any radical way. There are, however, a couple of areas for exploration.

One of the attractive aspects of **CML** is that it supports a wide range of concurrency mechanisms using a small set of core primitives. A possible area of exploration is to take a reductionist approach in the choice of primitives. **CML** uses synchronous message passing as the basic synchronous operation, but perhaps there are other, more primitive, choices. For example, some variation on low-level shared-memory primitives might be possible. This would factor out the communication from the primitive synchronous operations. While this exercise would be intellectually interesting, I suspect that the resulting language design would be too low-level. Synchronous message passing seems to provide a happy medium between low-level performance and high-level abstraction.

As I discussed in Section 5.4, choosing synchronous message passing as the primitive synchronization mechanism limits rendezvous to two threads. This limitation interferes with a potentially useful class of abstractions — the use of threads to implement *active channels*. An example of an active channel is a channel that logs all message traffic for debugging purposes. One approach to supporting such abstractions is to add a multiway rendezvous primitive [Cha87]. The implementation details remain to be worked out, but solutions to this problem are discussed in Chapter 14 of [CM88], where it is called the *committee coordination* problem.

The original prototype of first-class synchronous operations was implemented in **C** and included support for using events in **C** programs. The lack of closures, however, limited the usefulness of events in **C**. Adding events to a language such as **Modula-3** might prove more satisfactory, since *objects* can be used to provide a closure-like mechanism[1] [Nel91].

13.2 Theory

In Section 7.5, I described a number of ways to enrich the λ_{cv} calculus to more fully model **CML**. It remains to prove type soundness results for the extended calculus. In particular, the combination of exceptions and channels, both potential sources of type system loopholes, should be shown to be sound.

The operational semantics that I presented in Chapter 7 could be used as the basis for a "theory" of first-class synchronous operations. There are a number of transformations on event values that should be shown to be semantics preserving. For example, the representation of event values in the implementation can be described by a rewriting system of event values. Showing that the rewriting of an event value preserves its meaning would be a significant step toward showing that the implementation is correct. In Section 10.6, a number of optimizations are suggested (e.g., replacing use-once channels with condition variables). A theory of events would provide a framework for showing that these optimizations are "safe."

Proving such results requires a notion of event value equivalence: the obvious definition is that two event values are equivalent if they are indistinguishable in all contexts. This definition requires, in turn, some notion of process equivalence. This is an active area of theoretical research (e.g., [Blo89]) and there are many different notions of what it means for two processes to be equivalent. For various reasons, I think that a modified notion of *testing equivalence* [Hen88] is the most suitable for developing these results.

13.3 Practice

As we gain more experience with **CML**, certain common abstractions may emerge. By supporting these abstractions directly as primitives, performance can be improved substantially. The condition variables discussed in Section 5.3 is an example of this; using them for

[1] In fact, a closure in **Modula-3** is an object type with an `apply` method.

replys in an RPC abstraction reduces overhead by about 30% (see Chapter 11). Another possible candidate is buffered channels, which are often used in interactions with external processes (e.g., the **X**-server). It is important to note that adding these new primitive operations does not change the semantics of **CML**, since semantically they are still derived operations.

The most glaring weakness of **CML** is the lack of debugging facilities. A short term solution is to provide a version of the **CML** primitives that allows monitoring of communication, thread scheduling, etc. A more ambitious scheme is to provide an interactive debugger. Andrew Tolmach, who is responsible for the **SML/NJ** debugger [TA90], is working on a concurrent version for a "safe" version of **ML-threads** [TA91]. It is likely that his work can be adapted to **CML**; in fact, **CML** may be a better target than **ML-threads**, since the shared state is more clearly defined.

Chapter 12 discussed many of the issues relating to the implementation and use of **CML** on multiprocessors. I view this as the most important direction for future implementation work. Multiprocessor server machines are already common, and that technology is likely to trickle down to single-user workstations in the next few years. **CML** provides a natural migration path for **SML** applications to benefit from the parallel processing capabilities of multiprocessor workstation.

There are a number of active ongoing projects that are using **CML**. Emden Gansner and I are continuing to develop **eXene** and plan use it as part of a foundation for interactive programming environments [RG86, GR92]. The **DML** project at Cornell University is exploring issues in distributed systems, using **CML** as starting point [Kru91]. These applications and others will help to guide future evolution of **CML** and its implementation.

Chapter 14

Conclusion

Concurrent programming is an area of growing importance, but there has been little recent progress in the design of concurrent languages. For example, **Modula-3** encompasses many recent ideas in sequential language design, but uses concurrency features that date back to the 1970s [Nel91]. In this dissertation, I have presented a new approach to concurrent language design that supports a higher level of concurrent programming. The key new idea is to treat synchronous operations as first-class values that can be composed into new synchronous operations. This allows many different styles of communication to be supported in the same linguistic framework. I call this new style of programming "*higher-order concurrent programming*," as an analogy with higher-order programming in languages such as **ML**. This dissertation is a broad look at this new approach to concurrent language design, exploring the design, theory and practice of first-class synchronous operations.

The ideas of this thesis are presented in the context of the language **CML**, which is an extension of **SML** that supports first-class synchronous operations. I use **CML** to illustrate the usefulness and practicality of my approach. In Chapter 5, I show how a number of synchronization and communication abstractions found in other languages can be implemented in **CML** as first-class citizens. This demonstrates that **CML** can support different concurrent paradigms in a single linguistic framework.

I have also developed the formal underpinnings of first-class synchronous operations. In Chapter 7, I give the operational semantics of a simple untyped language, called λ_{cv}, that has first-class synchronous operations. This language includes most of the concurrency features of **CML**, and is a substantial step toward a formal definition of **CML**. In Chapter 8, I define a polymorphic type discipline for λ_{cv} that is in the tradition of **ML** type systems, and I prove that this type system is sound with respect to the operational semantics of λ_{cv}. To my knowledge, this is the first proof of type soundness for a polymorphic concurrent language.

CML has been implemented and has been used to build several non-trivial applications. The most significant of these is **eXene**, which is a multi-threaded **X** window system toolkit. **EXene** and some other applications of **CML** are described in Chapter 9. **CML** has also been publically distributed since November of 1991, and is being used by a number of other researchers. The implementation of **CML** is described in Chapter 10 and performance measurements are reported in Chapter 11. The use of **CML** to implement substantial applications, as well as the performance of the implementation (which is competitive with lower-level concurrency packages), demonstrates that **CML** is a useful and practical language for systems programming. In many respects, the **CML** system is the most important result of this research, and I expect that it will provide a solid basis for other research and development for years to come. In the future, I plan to implement **CML** on a shared-memory multiprocessor (Chapter 12 discusses issues related to this).

Bibliography

[AB80] Arvind and J. D. Brock. Streams and managers. In *Operating Systems Engineering; Proceedings of the 14th IBM Computer Science Symposium*, vol. 143 of *Lecture Notes in Computer Science.* Springer-Verlag, October 1980, pp. 452–465.

[AB86] Abramsky, S. and R. Bornat. Pascal-m: A language for loosely coupled distributed systems. In Y. Paker and J.-P. Verjus (eds.), *Distributed Computing Systems*, pp. 163–189. Academic Press, New York, N.Y., 1986.

[AEL88] Appel, A. W., J. R. Ellis, and K. Li. Real-time concurrent collection on stock multiprocessors. In *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, June 1988, pp. 11–20.

[Agh86] Agha, G. *Actors: A Model of Concurrent Computation in Distributed Systems.* The MIT Press, Cambridge, Mass., 1986.

[AJ89] Appel, A. W. and T. Jim. Continuation-passing, closure-passing style. In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*, January 1989, pp. 293–302.

[AM87] Appel, A. W. and D. B. MacQueen. A Standard ML compiler. In *Functional Programming Languages and Computer Architecture*, vol. 274 of *Lecture Notes in Computer Science.* Springer-Verlag, September 1987, pp. 301–324.

[AM91] Appel, A. W. and D. B. MacQueen. Standard ML of New Jersey. In *Programming Language Implementation and Logic Programming*, vol. 528 of *Lecture Notes in Computer Science.* Springer-Verlag, August 1991, pp. 1–26.

[And89] Anderson, T. E. The performance of spin lock alternatives for shared-memory multiprocessors. *Technical Report 89-04-03*, Department of Computer Science, University of Washington, August 1989.

[And91] Andrews, G. R. *Concurrent Programming: Principles and Practice.* Benjamin/Cummings, Redwood City, California, 1991.

[ANP89] Arvind, R. S. Nikhil, and K. K. Pingali. I-structures: Data structures for parallel computing. *ACM Transactions on Programming Languages and Systems*, **11**(4), October 1989, pp. 598–632.

[AOCE88] Andrews, G. R., R. A. Olsson, M. Coffin, and I. Elshoff. An overview of the SR language and implementation. *ACM Transactions on Programming Languages and Systems*, **10**(7), January 1988, pp. 51–86.

[App89] Appel, A. W. Simple generational garbage collection and fast allocation. *Software – Practice and Experience*, **19**(2), February 1989, pp. 275–279.

[App90] Appel, A. W. A runtime system. *Lisp and Symbolic Computation*, **4**(3), November 1990, pp. 343–380.

[App92] Appel, A. W. *Compiling with Continuations.* Cambridge University Press, New York, N.Y., 1992.

[AS83] Andrews., G. R. and F. B. Schneider. Concepts and notations for concurrent programming. *ACM Computing Surveys*, **15**(1), March 1983, pp. 3–43.

[Bag89] Bagrodia, R. Synchronization of asynchronous processes in CSP. *ACM Transactions on Programming Languages and Systems*, **11**(4), October 1989, pp. 585–597.

[Bar84] Barendregt, H. P. *The Lambda Calculus*, vol. 103 of *Studies in Logic and the Foundations of Mathematics.* North-Holland, revised edition, 1984.

[BB90] Berry, G. and G. Boudol. The chemical abstract machine. In *Conference Record of the 17th Annual ACM Symposium on Principles of Programming Languages*, January 1990, pp. 81–94.

[BCJ+90] Birman, K., R. Cooper, T. A. Joseph, K. Marzullo, M. Makpangou, K. Kane, F. Schmuck, and M. Wood. *The ISIS system manual, version 2.0.* Computer Science Department, Cornell University, Ithaca, N.Y., March 1990.

[BCLM89] Bose, S., E. M. Clarke, D. E. Long, and S. Michaylov. Parthenon: A parallel theorem prover for non-horn clauses. In *Proceedings of the 4th Annual Symposium on Logic in Computer Science*, June 1989, pp. 80–89.

[BD80] Bryant, R. and J. B. Dennis. Concurrent programming. In *Operating Systems Engineering; Proceedings of the 14th IBM Computer Science Symposium*, vol. 143 of *Lecture Notes in Computer Science.* Springer-Verlag, October 1980, pp. 426–451.

[BH77] Baker, Jr., H. G. and C. Hewitt. The incremental garbage collection of processes. In *Proceedings of the Symposium on Artificial Intelligence and Programming Languages*, August 1977, pp. 55–59.

[Blo89] Bloom, B. *Ready Simulation, Bisimulation, and the Semantics of CCS-like Languages.* Ph.D. dissertation, Massachusetts Institute Technology, Laboratory for Computer Science, October 1989. Available as MIT/LCS/TR-491.

[BMT92] Berry, D., R. Milner, and D. N. Turner. A semantics for ML concurrency primitives. In *Conference Record of the 19th Annual ACM Symposium on Principles of Programming Languages*, January 1992. *To appear.*

[BNA91] Barth, P., R. S. Nikhil, and Arvind. M-structures: Extending a parallel, non-strict, functional language with state. In *Functional Programming Languages and Computer Architecture*, vol. 523 of *Lecture Notes in Computer Science*. Springer-Verlag, August 1991, pp. 538–568.

[Bor86] Bornat, R. A protocol for generalized occam. *Software – Practice and Experience*, **16**(9), September 1986, pp. 783–799.

[Bri77] Brinch Hansen, P. *The Architecture of Concurrent Programs*. Prentice-Hall, Englewood Cliffs, N.J., 1977.

[Bri89] Brinch Hansen, P. The Joyce language report. *Software – Practice and Experience*, **19**(6), June 1989, pp. 553–578.

[BS83] Buckley, G. N. and A. Silberschatz. An effective implementation for the generalized input-output construct of CSP. *ACM Transactions on Programming Languages and Systems*, **5**(2), April 1983, pp. 223–235.

[BS90] Buhr, P. A. and R. A. Stroobosscher. The μSystem: Providing light-weight concurrency on shared-memory multiprocessor computers running UNIX. *Software – Practice and Experience*, **20**(9), September 1990, pp. 929–963.

[Bur88] Burns, A. *Programming in occam 2*. Addison-Wesley, Reading, Mass., 1988.

[Car86] Cardelli, L. Amber. In *Combinators and Functional Programming Languages*, vol. 242 of *Lecture Notes in Computer Science*. Springer-Verlag, July 1986, pp. 21–47.

[Car89] Cardelli, L. Typeful programming. *Technical Report 45*, DEC Systems Research Center, May 1989.

[CD88] Cooper, E. C. and R. P. Draves. C threads. *Technical Report CMU-CS-88-54*, School of Computer Science, Carnegie Mellon University, February 1988.

[CDDK86] Clément, D., J. Despeyroux, T. Despeyroux, and G. Kahn. A simple applicative language: Mini-ML. In *Conference record of the 1986 ACM Conference on Lisp and Functional Programming*, August 1986, pp. 13–27.

[Cha87] Charlesworth, A. The multiway rendezvous. *ACM Transactions on Programming Languages and Systems*, **9**(2), July 1987, pp. 350–366.

[CHL91] Cooper, E. C., R. Harper, and P. Lee. The Fox project: Advanced development of system software. *Technical Report CMU-CS-91-178*, School of Computer Science, Carnegie Mellon University, August 1991.

[CM88] Chandy, K. M. and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, Reading, Mass., 1988.

[CM90] Cooper, E. C. and J. G. Morrisett. Adding threads to Standard ML. *Technical Report CMU-CS-90-186*, School of Computer Science, Carnegie Mellon University, December 1990.

[Con86] Constable, R. *et al. Implementing Mathematics with The Nuprl Development System.* Prentice-Hall, Englewood Cliffs, N.J., 1986.

[Cor88] Cormack, G. V. A micro-kernel for concurrency in C. *Software – Practice and Experience*, **18**(5), May 1988, pp. 485–491. Short Communication.

[CS91] Crummey, J. M. M. and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, **9**(1), February 1991, pp. 21–65.

[Dam85] Damas, L. M. M. *Type assignment in programming languages.* Ph.D. dissertation, Department of Computer Science, University of Edinburgh, April 1985.

[DH89] Dybvig, R. K. and R. Hieb. Engines from continuations. *Computing Languages*, **14**(2), 1989, pp. 109–123.

[DHM91] Duba, B., R. Harper, and D. MacQueen. Type-checking first-class continuations. In *Conference Record of the 18th Annual ACM Symposium on Principles of Programming Languages*, January 1991, pp. 163–173.

[Dij75] Dijkstra, E. W. Guarded commands, nondeterminacy, and formal derivation of programs. *Communications of the ACM*, **18**(8), August 1975, pp. 453–457.

[DM82] Damas, L. and R. Milner. Principal types for functional programs. In *Conference Record of the 9th Annual ACM Symposium on Principles of Programming Languages*, January 1982, pp. 207–212.

[DoD83] *Reference Manual for the Ada Programming Language*, January 1983.

[EHP80] Eventoff, W., D. Harvey, and R. J. Price. The rendezvous and monitor concepts: Is there an efficiency difference? In *Proceedings of the ACM-SIGPLAN Symposium on the Ada Programming Language*, December 1980, pp. 156–165.

[Fel87a] Felleisen, M. *The Calculi of Lambda-v-CS Conversion in Imperative Higher-order programming languages.* Ph.D. dissertation, Computer Science Department, Indiana University, 1987. Available as Technical Report Nr. 226.

[Fel87b] Felleisen, M. Reflections on Landin's J-operator: A partly historical note. *Computer Languages*, **12**(3/4), 1987, pp. 197–207.

[FF86] Felleisen, M. and D. P. Friedman. Control operators, the SECD-machine, and the λ-calculus. In M. Wirsing (ed.), *Formal Description of Programming Concepts – III*, pp. 193–219. North-Holland, New York, N.Y., 1986.

[FLP85] Fischer, M. J., N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, **32**(2), April 1985, pp. 374–382.

[FY85] Francez, N. and S. A. Yemini. Symmetric intertask communication. *ACM Transactions on Programming Languages and Systems*, **7**(4), October 1985, pp. 622–636.

[GC84] Gehani, N. H. and T. A. Cargill. Concurrent programming in the Ada language: The polling bias. *Software – Practice and Experience*, **14**(5), May 1984, pp. 413–427.

[GL91] George, L. and G. Lindstrom. Using a functional language and graph reduction to program multiprocessor machines. *Technical Report UUCS-91-020*, Department of Computer Science, University of Utah, October 1991.

[GM88] Gehani, N. H. and A. D. McGettrick (eds.). *Concurrent Programming*. Addison-Wesley, Reading, Mass., 1988.

[GMP89] Giacalone, A., P. Mishra, and S. Prasad. Facile: A symemetric integration of concurrent and functional programming. In *TAPSOFT'89 (vol. 2)*, vol. 352 of *Lecture Notes in Computer Science*. Springer-Verlag, March 1989, pp. 184–209.

[GMW79] Gordon, M. J., R. Milner, and C. P. Wadsworth. *Edinburgh LCF*, vol. 72 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, N.Y., 1979.

[Gor79] Gordon, M. J. C. *The Denotational Description of Programming Languages*. Springer-Verlag, New York, N.Y., 1979.

[GR86] Gehani, N. H. and W. D. Roome. Concurrent C. *Software – Practice and Experience*, **16**(9), September 1986, pp. 821–844.

[GR91] Gansner, E. R. and J. H. Reppy. eXene. In *Third International Workshop on Standard ML*, Carnegie Mellon University, September 1991.

[GR92] Gansner, E. R. and J. H. Reppy. A foundation for user interface construction. In B. A. Myers (ed.), *Languages for Developing User Interfaces*, pp. 239–260. Jones & Bartlett, Boston, Mass., 1992.

[Gre91] Green, S. *Parallel Processing for Computer Graphics*. The MIT Press, Cambridge, Mass, 1991.

[Haa90] Haahr, D. Montage: Breaking windows into small pieces. In *USENIX Summer Conference*, June 1990, pp. 289–297.

[Hal85] Halstead, Jr., R. H. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, **7**(4), October 1985, pp. 501–538.

[Har86] Harper, R. Introduction to Standard ML. *Technical Report ECS-LFCS-86-14*, Laboratory for Foundations of Computer Science, Computer Science Department, Edinburgh University, August 1986.

[HDB90] Hieb, R., R. K. Dybvig, and C. Bruggeman. Representing control in the presence of first-class continuations. In *Proceedings of the SIGPLAN'90 Conference on Programming Language Design and Implementation*, June 1990, pp. 66–77.

[Hen88] Hennessy, M. *Algebraic Theory of Processes*. The MIT Press, Cambridge, Mass., 1988.

[HN80] Habermann, A. N. and I. R. Nassi. Efficient implementation of Ada tasks. *Technical Report CMU-CS-80-103*, Computer Science Department, Carnegie Mellon University, January 1980.

[Hoa74] Hoare, C. A. R. Monitors: An operating system concept. *Communications of the ACM*, **17**(10), October 1974, pp. 549–557.

[Hoa78] Hoare, C. A. R. Communicating sequential processes. *Communications of the ACM*, **21**(8), August 1978, pp. 666–677.

[Hoa85] Hoare, C. A. R. *Communicating Sequential Processes.* Prentice-Hall, Englewood Cliffs, N.J., 1985.

[Hol83a] Holmström, S. PFL: A functional language for parallel programming. In *Declarative programming workshop*, April 1983, pp. 114–139.

[Hol83b] Holt, R. C. *Concurrent Euclid, the UNIX System, and Tunis.* Addison-Wesley, Reading, Mass., 1983.

[HS86] Hindley, R. J. and J. P. Seldin. *Introduction to Combinators and the λ-calculus.* Cambridge University Press, New York, N.Y., 1986.

[INM84] INMOS Limited. *Occam Programming Manual.* Prentice-Hall, Englewood Cliffs, N.J., 1984.

[KH88] Kranz, D. A. and R. H. Halstead, Jr. Mul-T: A high-performance parallel Lisp. In *Proceedings of the SIGPLAN'89 Conference on Programming Language Design and Implementation*, June 1988, pp. 81–90.

[KKR+86] Kranz, D., R. Kelsey, J. Rees, P. Hudak, J. Philbin, and N. Adams. Orbit: An optimizing compiler for Scheme. In *Proceedings of the SIGPLAN'86 Symposium on Compiler Construction*, July 1986, pp. 219–233.

[KM77] Kahn, G. and D. B. MacQueen. Coroutines and networks of parallel processes. In *Information Processing 77*, August 1977, pp. 993–998.

[KNW90] Kafura, D., J. Nelson, and D. Washabaugh. Garbage colection of actors. In *OOPSLA/ECOOP'90 Proceedings*, October 1990, pp. 126–134.

[Kru91] Krumvieda, C. D. DML: Packaging high-level distributed abstractions in SML. In *Proceedings of the 1991 CMU Workshop on Standard ML*, September 1991.

[KS79] Kieburtz, R. B. and A. Silberschatz. Comments on 'Communicating sequential processes'. *ACM Transactions on Programming Languages and Systems*, **1**(2), April 1979, pp. 218–225.

[Kwi89] Kwiatkowska, M. Z. Survey of fairness notions. *Information and Software Technology*, **31**(7), September 1989, pp. 371–386.

[Lam87] Lamport, L. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, **5**(1), February 1987, pp. 1–11.

[Lan65] Landin, P. J. A correspondence between Algol 60 and Churh's lambda notation: Part I. *Communications of the ACM*, **8**(2), February 1965, pp. 89–101.

[LCJS87] Liskov, B., D. Curtis, P. Johnson, and R. Scheifler. Implementation of Argus. In *Proceedings of the 11th ACM Symposium on Operating System Principles*, November 1987, pp. 111–122.

[LHG86] Liskov, B., M. Herlihy, and L. Gilbert. Limitations of synchronous communication with static process structure in langauges for distributed programming. In *Conference Record of the 13th Annual ACM Symposium on Principles of Programming Languages*, January 1986, pp. 150–159.

[LR80] Lampson, B. W. and D. D. Redell. Experience with processes and monitors in Mesa. *Communications of the ACM*, **23**(2), February 1980, pp. 105–116.

[LS83] Liskov, B. and R. Scheifler. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Transactions on Programming Languages and Systems*, **5**(3), July 1983, pp. 381–404.

[LS88] Liskov, B. and L. Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, June 1988, pp. 260–267.

[LS90] Lin, C. and L. Snyder. A comparison of programming models for shared memory multiprocessors. In *1990 International Conference on Parallel Processing*, vol. 2, 1990, pp. 163–170.

[Mar91] Maranget, L. GAML: A parallel implementation of lazy ML. In *Functional Programming Languages and Computer Architecture*, vol. 523 of *Lecture Notes in Computer Science*. Springer-Verlag, August 1991, pp. 102–123.

[Mat89] Matthews, D. C. J. Processes for Poly and ML. In *Papers on Poly/ML, Technical Report 161*. University of Cambridge, February 1989.

[McI90] McIlroy, M. D. Squinting at power series. *Software – Practice and Experience*, **20**(7), July 1990, pp. 661–683.

[ME89] Miller, J. S. and B. S. Epstein. Garbage collection in multischeme (preliminary version). In *Parallel Lisp: Languages and Systems*, vol. 441 of *Lecture Notes in Computer Science*. Springer-Verlag, June 1989, pp. 138–160.

[MKH91] Mohr, E., D. A. Kranz, and R. H. Halstead, Jr. Lazy task creation: A technique for increasing the granularity of parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, **2**(3), July 1991, pp. 264–280. A longer version is available as DEC CRL report 90/7, November 1990.

[MMS79] Mitchell, J. G., W. Maybury, and R. Sweet. *Mesa Language Manual (Version 5.0)*. Xerox PARC, April 1979.

[Mor] Morrisett, J. G. A multi-processor interface for SML. CMU technical report (*in preparation*).

[MT91] Milner, R. and M. Tofte. *Commentary on Standard ML.* The MIT Press, Cambridge, Mass, 1991.

[MTH90] Milner, R., M. Tofte, and R. Harper. *The Definition of Standard ML.* The MIT Press, Cambridge, Mass, 1990.

[Nel81] Nelson, B. J. *Remote Procedure Call.* Ph.D. dissertation, Computer Science Department, Carnegie Mellon University, May 1981. Available as Xerox PARC Report CSL-81-9.

[Nel91] Nelson, G. (ed.). *Systems Programming with Modula-3.* Prentice-Hall, Englewood Cliffs, N.J., 1991.

[Nik91] Nikhil, R. S. *ID Language Reference Manual.* Laboratory for Computer Science, MIT, Cambridge, Mass., July 1991.

[Nye90a] Nye, A. *X Protocol Reference Manual*, vol. 0. O'Reilly & Associates, Inc., 1990.

[Nye90b] Nye, A. *Xlib Programming Manual*, vol. 1. O'Reilly & Associates, Inc., 1990.

[Osb89] Osborne, R. B. Speculative computation in Multilisp. In *Parallel Lisp: Languages and Systems*, vol. 441 of *Lecture Notes in Computer Science.* Springer-Verlag, June 1989, pp. 101–137.

[Owi89] Owicki, S. Experience with the firefly multiprocessor workstation. *Technical Report 51*, DEC Systems Research Center, September 1989.

[Pau91] Paulson, L. C. *ML for the Working Programmer.* Cambridge University Press, New York, N.Y., 1991.

[Pik89] Pike, R. A concurrent window system. *Computing Systems*, **2**(2), 1989, pp. 133–153.

[Plo75] Plotkin, G. D. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, **1**, 1975, pp. 125–159.

[Ram90] Ramsey, N. Concurrent programming in ML. *Technical Report CS-TR-262-90*, Department of Computer Science, Princeton University, April 1990.

[RC86] Rees, J. and W. Clinger (Eds.). The revised^3 report on the algorithmic language Scheme. *SIGPLAN Notices*, **21**(12), December 1986, pp. 37–43.

[Rep88] Reppy, J. H. Synchronous operations as first-class values. In *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, June 1988, pp. 250–259.

[Rep89] Reppy, J. H. First-class synchronous operations in Standard ML. *Technical Report TR 89-1068*, Computer Science Department, Cornell University, December 1989.

[Rep90a] Reppy, J. H. Asynchronous signals in Standard ML. *Technical Report TR 90-1144*, Computer Science Department, Cornell University, August 1990.

[Rep90b] Reppy, J. H. *Concurrent programming with events – The Concurrent ML manual.* Computer Science Department, Cornell University, Ithaca, N.Y., November 1990. (Last revised October 1991).

[Rep91a] Reppy, J. H. CML: A higher-order concurrent language. In *Proceedings of the SIGPLAN'91 Conference on Programming Language Design and Implementation*, June 1991, pp. 293–305.

[Rep91b] Reppy, J. H. An operational semantics of first-class synchronous operations. *Technical Report TR 91-1232*, Computer Science Department, Cornell University, August 1991.

[RG86] Reppy, J. H. and E. R. Gansner. A foundation for programming environments. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, December 1986, pp. 218–227.

[RLW85] Rovner, P., R. Levin, and J. Wick. On extending Modula-2 for building large, integrated systems. *Technical Report 3*, DEC Systems Research Center, January 1985.

[Ros81] Rosen, B. K. Degrees of availability as an introduction to the general theory of data flow analysis. In S. S. Muchnick and N. D. Jones (eds.), *Program Flow Analysis: Theory and Applications*, pp. 55–76. Prentice-Hall, Englewood Cliffs, N.J., 1981.

[Rov85] Rovner, P. On adding garbage collection and runtime types to a strongly-typed, statically-check, concurrent language. *Technical Report CSL-84-7*, Xerox PARC, July 1985.

[RV89] Roberts, E. S. and M. T. Vandevoorde. WorkCrews: An abstraction for controlling parallelism. *Technical Report 42*, DEC Systems Research Center, April 1989.

[SG86] Scheifler, R. W. and J. Gettys. The X window system. *ACM Transactions on Graphics*, **5**(2), April 1986, pp. 79–109.

[SR90] Saraswat, V. A. and M. Rinard. Concurrent constraint programming. In *Conference Record of the 17th Annual ACM Symposium on Principles of Programming Languages*, January 1990, pp. 232–245.

[Ste78] Steele Jr., G. L. Rabbit: A compiler for Scheme. Master's dissertation, MIT, May 1978.

[TA90] Tolmach, A. P. and A. W. Appel. Debugging Standard ML without reverse engineering. In *Conference record of the 1990 ACM Conference on Lisp and Functional Programming*, June 1990, pp. 1–12.

[TA91] Tolmach, A. P. and A. W. Appel. Debuggable concurrency extensions for Standard ML. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, May 1991, pp. 120–131.

[Tof88] Tofte, M. *Operational semantics and polymorphic type inference.* Ph.D. dissertation, Department of Computer Science, University of Edinburgh, May 1988.

[Tof90] Tofte, M. Type inference for polymorphic references. *Information and Computation*, **89**, 1990, pp. 1–34.

[Ung84] Ungar, D. Generation scavenging: A non-disruptive high-performance storage reclamation algorithm. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, April 1984, pp. 157–167.

[UNI86] University of California, Berkeley. *UNIX Programmer's Reference Manual (4.3bsd)*, 1986.

[Wan80] Wand, M. Continuation-based multiprocessing. In *Conference Record of the 1980 Lisp Conference*, August 1980, pp. 19–28.

[WF91a] Wright, A. and M. Felleisen. Corrigendum to "A syntactic approach to type soundness", July 1991.

[WF91b] Wright, A. and M. Felleisen. A syntactic approach to type soundness. *Technical Report TR91-160*, Department of Computer Science, Rice University, April 1991.

[WS83] Wegner, P. and S. A. Smolka. Processes, tasks and monitors: A comparative study of concurrent programming primitives. *IEEE Transactions on Software Engineering*, **9**(4), July 1983, pp. 446–462.

[Zar90] Zaring, A. K. *Parallel Evaluation in Attribute Grammar-based Systems.* Ph.D. dissertation, Computer Science Department, Cornell University, August 1990. Available as Technical Report 90-1149.

Appendix

Appendix A

Proofs from Chapter 8

This appendix contains the detailed proofs of some of the lemmas in Chapter 8. It also includes some additional definitions and lemmas needed for these proofs.

Proof of Lemma 8.5

Before proving the Substitution Lemma, we need a couple of minor lemmas. The following lemma extends substitution to type judgements.

Lemma A.1 If S is a substitution and $\mathrm{TE} \vdash e : \tau$, then $S(\mathrm{TE}) \vdash e : S\tau$.

Proof. Proofs of this for a similar system can be found in [Tof88] (Lemma 5.2, p. 48) and in [Tof90] (Lemma 4.2, p. 18). ■

The following lemma says that the typing assumptions (i.e., the type environment) of a type derivation can be generalized without affecting the result.

Lemma A.2 If $\mathrm{TE} \pm \{x \mapsto \sigma\} \vdash e : \tau$ and $\sigma' \succ \sigma$, then $\mathrm{TE} \pm \{x \mapsto \sigma'\} \vdash e : \tau$.

Proof. The proof is by induction on the height of the deduction of

$$\mathrm{TE} \pm \{x \mapsto \sigma\} \vdash e : \tau$$

and by case analysis of the last step (i.e., analysis of the shape of e). The interesting cases are those involving the variable typing component of TE. Recall that the variable convention means that x is not bound in e.

Case $e = x'$.

If $x \neq x'$, then Lemma 8.3 means that $\mathrm{TE} \vdash x : \tau$. Applying Lemma 8.3 again, we get $\mathrm{TE} \pm \{x \mapsto \sigma'\} \vdash x' : \tau$.

If $x = x'$, then $\sigma \succ \tau$, and since $\sigma' \succ \sigma$, $\sigma' \succ \tau$. Then $\mathrm{TE} \pm \{x \mapsto \sigma'\} \vdash x : \tau$, by rule ($\tau$**-var**).

Case $e = \lambda x'(e')$**.**

Rule (τ**-abs**) applies:

$$\frac{\mathrm{TE} \pm \{x \mapsto \sigma,\, x' \mapsto \tau'\} \vdash e' : \tau}{\mathrm{TE} \pm \{x \mapsto \sigma\} \vdash \lambda x'(e') : (\tau' \to \tau)}$$

So, by the induction hypothesis,

$$\mathrm{TE} \pm \{x \mapsto \sigma',\, x' \mapsto \tau'\} \vdash e' : \tau$$

And, thus, applying rule (τ**-abs**), we get

$$\mathrm{TE} \pm \{x \mapsto \sigma'\} \vdash \lambda x'(e') : (\tau' \to \tau)$$

Case $e = \mathtt{let}\ x' = v\ \mathtt{in}\ e'$**.**

Rule (τ**-app-let**) applies:

$$\frac{\mathrm{TE} \pm \{x \mapsto \sigma\} \vdash v : \tau' \quad \mathrm{TE} \pm \{x \mapsto \sigma,\, x' \mapsto \textsc{Clos}_{\mathrm{TE} \pm \{x \mapsto \sigma\}}(\tau')\} \vdash e : \tau}{\mathrm{TE} \pm \{x \mapsto \sigma\} \vdash \mathtt{let}\ x' = v\ \mathtt{in}\ e : \tau}$$

Then, by the induction hypothesis,

$$\mathrm{TE} \pm \{x \mapsto \sigma'\} \vdash v : \tau'$$

and

$$\mathrm{TE} \pm \{x \mapsto \sigma',\, x' \mapsto \textsc{Clos}_{\mathrm{TE} \pm \{x \mapsto \sigma\}}(\tau')\} \vdash e : \tau$$

Since $\sigma' \succ \sigma$, Lemma 8.1 gives us

$$\textsc{Clos}_{\mathrm{TE} \pm \{x \mapsto \sigma'\}}(\tau') \succ \textsc{Clos}_{\mathrm{TE} \pm \{x \mapsto \sigma\}}(\tau')$$

We can then apply the induction hypothesis again to get

$$\mathrm{TE} \pm \{x \mapsto \sigma',\, x' \mapsto \textsc{Clos}_{\mathrm{TE} \pm \{x \mapsto \sigma'\}}(\tau')\} \vdash e : \tau$$

And then, by rule (τ**-app-let**), we get

$$\mathrm{TE} \pm \{x \mapsto \sigma'\} \vdash \mathtt{let}\ x' = v\ \mathtt{in}\ e : \tau$$

Case $e = \mathtt{let}\ x' = e_1\ \mathtt{in}\ e_2$**.**

This follows the argument of the previous case.

Case $e = \textbf{chan}\ x'\ \textbf{in}\ e'$**.**

This case is similar to the case $e = \lambda x'(e')$ above.

■

Lemma 8.5 (Substitution) If $x \notin \mathrm{FV}(v)$, $\mathrm{TE} \vdash v : \tau$, and

$$\mathrm{TE} \pm \{x \mapsto \forall\alpha_1 \cdots \alpha_n.\tau\} \vdash e : \tau'$$

with $\{\alpha_1, \ldots, \alpha_n\} \cap \mathrm{FTV}(\mathrm{TE}) = \emptyset$, then $\mathrm{TE} \vdash e[x \mapsto v] : \tau'$.

Proof. The proof is mostly from [WF91b], and proceeds by induction on the height of the deduction of

$$\mathrm{TE} \pm \{x \mapsto \forall\alpha_1 \cdots \alpha_n.\tau\} \vdash e : \tau'$$

and by case analysis of the last step. Let $\mathrm{TE} = (\mathrm{VT}, \mathrm{CT})$, $\mathrm{VT}' = \mathrm{VT} \pm \{x \mapsto \forall\alpha_1 \cdots \alpha_n.\tau\}$, and $\mathrm{TE}' = (\mathrm{VT}', \mathrm{CT})$ in the following discussion. We skip the cases for the terms covered by the rules in Figure 8.2, since these cases follow those for (τ-**app**) and (τ-**const**). As before, recall that the variable convention means that x is not bound in e.

Case $e = b$**.**

The last step is rule (τ-**const**), so $\mathrm{TypeOf}(b) \succ \tau'$. Applying rule ($\tau$-**const**), we get $\mathrm{TE} \vdash b : \tau'$. Since $b[x \mapsto v] = b$, we are done.

Case $e = x'$**.**

If $x' \neq x$, then, by rule (τ-**var**), $\mathrm{VT}'(x') \succ \tau'$. Since $x'[x \mapsto v] = x'$ and $\mathrm{VT}(x') \succ \tau'$, $\mathrm{TE} \vdash x' : \tau'$.

If $x' = x$, then $\mathrm{VT}'(x) = \forall\alpha_1 \cdots \alpha_n.\tau$. By rule ($\tau$-**var**), $\forall\alpha_1 \cdots \alpha_n.\tau \succ \tau'$, which means that there is a substitution S, such that $\mathrm{dom}(S) = \{\alpha_1, \ldots, \alpha_n\}$ and $S\tau = \tau'$. Lemma A.1 gives us $S(\mathrm{TE}) \vdash v : S\tau$, which implies that $S(\mathrm{TE}) \vdash v : \tau'$. Since $\mathrm{dom}(S) \cap \mathrm{FTV}(\mathrm{TE}) = \emptyset$, we have $S(\mathrm{TE}) = \mathrm{TE}$; thus, $\mathrm{TE} \vdash v : \tau'$.

Case $e = \kappa$**.**

Rule (τ-**chvar**) applies, thus $\mathrm{CT}(\kappa) = \tau'$. Since $\kappa[x \mapsto v] = \kappa$, we can apply ($\tau$-**chvar**) to get $\mathrm{TE} \vdash \kappa[x \mapsto v] : \tau'$.

Case $e = e_1\ e_2$**.**

Rule (τ-**app**) applies, so we have

$$\frac{\mathrm{TE}' \vdash e_1 : (\tau'' \to \tau') \quad \mathrm{TE}' \vdash e_2 : \tau''}{\mathrm{TE}' \vdash e_1\ e_2 : \tau'}$$

By the induction hypothesis and rule (τ-**app**), we have

$$\frac{\mathrm{TE} \vdash e_1[x \mapsto v] : (\tau'' \to \tau') \quad \mathrm{TE} \vdash e_2[x \mapsto v] : \tau''}{\mathrm{TE} \vdash e_1[x \mapsto v]\ e_2[x \mapsto v] : \tau'}$$

Therefore, $\mathrm{TE} \vdash e_1\ e_2[x \mapsto v] : \tau'$.

Case $e = (e_1 \,.\, e_2)$**.**

This case is very similar to the previous case.

Case $e = \lambda x'(e')$**.**

Rule (τ-**abs**) applies:

$$\frac{\mathrm{TE}' \pm \{x' \mapsto \tau_1\} \vdash e' : \tau_2}{\mathrm{TE}' \vdash \lambda x'(e') : \tau'}$$

with $\tau' = (\tau_1 \to \tau_2)$. Let S be the substitution

$$\{\alpha_1 \mapsto \beta_1, \ldots, \alpha_n \mapsto \beta_n\}$$

where the α_i, β_i and FTV(TE) are all distinct. Then, by Lemma A.1,

$$S(\mathrm{TE}') \pm \{x' \mapsto S\tau_1\} \vdash e' : S\tau_2$$

Note that $S(\mathrm{TE}') = \mathrm{TE}'$, since $\mathrm{dom}(S) \cap \mathrm{FTV}(\mathrm{TE}') = \emptyset$, hence

$$\mathrm{TE}' \pm \{x' \mapsto S\tau_1\} \vdash e' : S\tau_2$$

The variable convention insures $x' \notin \mathrm{FV}(v)$, so Lemma 8.3 gives us

$$\mathrm{TE} \pm \{x' \mapsto S\tau_1\} \vdash v : \tau$$

And the choice of S means that

$$\mathrm{FTV}(\mathrm{TE} \pm \{x' \mapsto S\tau_1\}) \cap \{\alpha_1, \ldots, \alpha_n\} = \emptyset$$

These facts, coupled with the induction hypothesis gives us

$$\mathrm{TE} \pm \{x' \mapsto S\tau_1\} \vdash e'[x \mapsto v] : S\tau_2$$

The substitution S is a bijection, so S^{-1} exists; hence, by Lemma A.1,

$$S^{-1}(\mathrm{TE} \pm \{x' \mapsto S\tau_1\}) \vdash e'[x \mapsto v] : S^{-1}S\tau_2$$

simplifying, we get

$$\mathrm{TE} \pm \{x' \mapsto \tau_1\}) \vdash e'[x \mapsto v] : \tau_2$$

thus, applying (τ-**abs**), we get

$$\frac{\mathrm{TE} \pm \{x' \mapsto \tau_1\} \vdash e'[x \mapsto v] : \tau_2}{\mathrm{TE} \vdash \lambda x'(e'[x \mapsto v]) : (\tau_1 \to \tau_2)}$$

and therefore, $\mathrm{TE} \vdash (\lambda x'(e'))[x \mapsto v] : (\tau_1 \to \tau_2)$.

Case $e = \texttt{let}\ x' \texttt{=} v'\ \texttt{in}\ e'$**.**

This is the case of a non-expansive `let`, so the first step of the type deduction must be rule (τ**-app-let**):

$$\frac{\mathrm{TE}' \vdash v' : \tau'' \quad \mathrm{TE}' \pm \{x' \mapsto \textsc{Clos}_{\mathrm{TE}'}(\tau'')\} \vdash e' : \tau'}{\mathrm{TE}' \vdash \texttt{let}\ x' \texttt{=} v'\ \texttt{in}\ e' : \tau'}$$

Since $\mathrm{TE} \vdash v : \tau$, Lemma 8.3 gives us

$$\mathrm{TE} \pm \{x' \mapsto \textsc{Clos}_{\mathrm{TE}'}(\tau'')\} \vdash v : \tau \qquad (*)$$

Recall that $\{\alpha_1, \ldots, \alpha_n\} \cap \mathrm{FTV}(\mathrm{TE}) = \emptyset$ in the following:

$$\begin{array}{rl} & \{\alpha_1, \ldots, \alpha_n\} \cap \mathrm{FTV}(\mathrm{TE} \pm \{x' \mapsto \textsc{Clos}_{\mathrm{TE}'}(\tau'')\}) \\ = & \{\alpha_1, \ldots, \alpha_n\} \cap (\mathrm{FTV}(\mathrm{TE}) \cup \mathrm{FTV}(\textsc{Clos}_{\mathrm{TE}'}(\tau''))) \\ = & \{\alpha_1, \ldots, \alpha_n\} \cap \mathrm{FTV}(\textsc{Clos}_{\mathrm{TE}'}(\tau'')) \\ = & \{\alpha_1, \ldots, \alpha_n\} \cap (\mathrm{FTV}(\tau'') \setminus (\mathrm{FTV}(\tau'') \setminus \mathrm{FTV}(\mathrm{TE}'))) \\ = & \{\alpha_1, \ldots, \alpha_n\} \cap \mathrm{FTV}(\tau'') \cap \mathrm{FTV}(\mathrm{TE}') \\ = & \{\alpha_1, \ldots, \alpha_n\} \cap \mathrm{FTV}(\tau'') \cap (\mathrm{FTV}(\mathrm{TE}) \cup \mathrm{FTV}(\forall \alpha_1 \cdots \alpha_n . \tau)) \\ = & \{\alpha_1, \ldots, \alpha_n\} \cap \mathrm{FTV}(\tau'') \cap \mathrm{FTV}(\forall \alpha_1 \cdots \alpha_n . \tau) \\ = & \{\alpha_1, \ldots, \alpha_n\} \cap \mathrm{FTV}(\tau'') \cap (\mathrm{FTV}(\tau) \setminus \{\alpha_1, \ldots, \alpha_n\}) \\ = & \emptyset \end{array}$$

The second premise of (τ**-app-let**) and $(*)$ with the induction hypothesis give us:

$$\mathrm{TE} \pm \{x' \mapsto \textsc{Clos}_{\mathrm{TE}'}(\tau'')\} \vdash e'[x \mapsto v] : \tau'$$

Note that $\textsc{Clos}_{\mathrm{TE}}(\tau'') \succ \textsc{Clos}_{\mathrm{TE}'}(\tau'')$, so we can apply Lemma A.2 to get:

$$\mathrm{TE} \pm \{x' \mapsto \textsc{Clos}_{\mathrm{TE}}(\tau'')\} \vdash e'[x \mapsto v] : \tau'$$

Thus, by the induction hypothesis and (τ**-app-let**), we have:

$$\frac{\mathrm{TE} \vdash v'[x \mapsto v] : \tau'' \quad \mathrm{TE} \pm \{x' \mapsto \textsc{Clos}_{\mathrm{TE}}(\tau'')\} \vdash e'[x \mapsto v] : \tau'}{\mathrm{TE} \vdash \texttt{let}\ x' \texttt{=} v'[x \mapsto v]\ \texttt{in}\ e'[x \mapsto v] : \tau'}$$

and, therefore, $\mathrm{TE} \vdash \texttt{(let}\ x' \texttt{=} v'\ \texttt{in}\ e'\texttt{)}[x \mapsto v] : \tau'$.

Case $e = \texttt{let}\ x' \texttt{=} e_1\ \texttt{in}\ e_2$**.**

This is the case of an expansive `let`, so rule (τ**-imp-let**) applies:

$$\frac{\mathrm{TE}' \vdash e_1 : \tau'' \quad \mathrm{TE}' \pm \{x' \mapsto \textsc{AppClos}_{\mathrm{TE}'}(\tau'')\} \vdash e_2 : \tau'}{\mathrm{TE}' \vdash \texttt{let}\ x' \texttt{=} e_1\ \texttt{in}\ e_2 : \tau'}$$

Choose a substitution

$$S : (\{\alpha_1, \ldots, \alpha_n\} \cap \textsc{ImpTyVar}) \to \{u_1, \ldots, u_m\}$$

such that $u_1, \ldots, u_m$ are distinct imperative type variables, S is a bijection, and

$$\{u_1, \ldots, u_m\} \cap (\mathrm{FTV}(\mathrm{TE}) \cup \mathrm{FTV}(\tau) \cup \{\alpha_1, \ldots, \alpha_n\}) = \emptyset$$

Then, Lemma A.1 tells us that

$$S(\mathrm{TE}' \pm \{x' \mapsto \textsc{AppClos}_{\mathrm{TE}'}(\tau'')\}) \vdash e_2 : S\tau'$$

Since $\mathrm{dom}(S) \cap \mathrm{FTV}(\mathrm{TE}') = \emptyset$ and $\mathrm{dom}(S) \subset \textsc{ImpTy}$, we have

$$\mathrm{TE}' \pm \{x' \mapsto \textsc{AppClos}_{\mathrm{TE}'}(S\tau'')\} \vdash e_2 : S\tau' \qquad (*)$$

Since $x' \notin \mathrm{FV}(v)$ and since $\mathrm{TE} \vdash v : \tau$, we have

$$\mathrm{TE} \pm \{x' \mapsto \textsc{AppClos}_{\mathrm{TE}'}(S\tau'')\} \vdash v : \tau \qquad (**)$$

Let $\textsc{AppClos}_{\mathrm{TE}'}(S\tau'') = \forall t_1, \cdots, t_l.S\tau''$; i.e.,

$$\{t_1, \ldots, t_l\} = (\mathrm{FTV}(S\tau'') \setminus \mathrm{FTV}(\mathrm{TE}')) \cap \textsc{AppTyVar}$$

then

$$\begin{array}{rl} & \{\alpha_1, \ldots, \alpha_n\} \cap \mathrm{FTV}(\mathrm{TE} \pm \{x' \mapsto \forall t_1, \cdots, t_l.S\tau''\}) \\ = & \{\alpha_1, \ldots, \alpha_n\} \cap (\mathrm{FTV}(\mathrm{TE}) \cup \mathrm{FTV}(\forall t_1, \cdots, t_l.S\tau'')) \\ = & \{\alpha_1, \ldots, \alpha_n\} \cap (\mathrm{FTV}(S\tau'') \setminus \{t_1, \ldots, t_l\}) \\ = & \emptyset \end{array}$$

By the inductive hypothesis with $(*)$ and $(**)$, we have

$$\mathrm{TE} \pm \{x' \mapsto \textsc{AppClos}_{\mathrm{TE}'}(S\tau'')\} \vdash e_2[x \mapsto v] : S\tau'$$

Since S was chosen to be a bijection, S^{-1} exists, so by Lemma A.1, we have

$$S^{-1}(\mathrm{TE} \pm \{x' \mapsto \textsc{AppClos}_{\mathrm{TE}'}(S\tau'')\}) \vdash e_2[x \mapsto v] : S^{-1}(S\tau')$$

simplifying, we get

$$\mathrm{TE} \pm \{x' \mapsto \textsc{AppClos}_{\mathrm{TE}'}(\tau'')\} \vdash e_2[x \mapsto v] : \tau'$$

Since $\textsc{AppClos}_{\mathrm{TE}}(\tau'') \succ \textsc{AppClos}_{\mathrm{TE}'}(\tau'')$, Lemma A.2 applies:

$$\mathrm{TE} \pm \{x' \mapsto \textsc{AppClos}_{\mathrm{TE}}(\tau'')\} \vdash e_2[x \mapsto v] : \tau'$$

By the induction hypothesis, we have

$$\mathrm{TE} \vdash e_1[x \mapsto v] : \tau''$$

and, thus, we can apply ($\boldsymbol{\tau}$**-app-let**):

$$\frac{\mathrm{TE} \vdash e_1[x \mapsto v] : \tau'' \quad \mathrm{TE} \pm \{x' \mapsto \textsc{AppClos}_{\mathrm{TE}'}(\tau'')\} \vdash e_2[x \mapsto v] : \tau'}{\mathrm{TE} \vdash \texttt{let}\ x' = e_1[x \mapsto v]\ \texttt{in}\ e_2[x \mapsto v] : \tau'}$$

and, therefore, $\mathrm{TE} \vdash (\texttt{let}\ x' = e_1\ \texttt{in}\ e_2)[x \mapsto v] : \tau'$.

Case $e = \texttt{chan}\ x'\ \texttt{in}\ e'$**.**

Rule (τ**-chan**) applies:

$$\frac{\mathrm{TE}' \pm \{x' \mapsto \theta\ \texttt{chan}\} \vdash e' : \tau'}{\mathrm{TE}' \vdash \texttt{chan}\ x'\ \texttt{in}\ e' : \tau'}$$

By the variable convention, $x' \notin \mathrm{FV}(v)$, so Lemma 8.3 gives us

$$\mathrm{TE} \pm \{x' \mapsto \theta\ \texttt{chan}\} \vdash v : \tau$$

Thus, by the induction hypothesis and rule (τ**-chan**)

$$\frac{\mathrm{TE} \pm \{x' \mapsto \theta\ \texttt{chan}\} \vdash e'[x \mapsto v] : \tau'}{\mathrm{TE} \vdash \texttt{chan}\ x'\ \texttt{in}\ e'[x \mapsto v] : \tau'}$$

and therefore, $(\mathrm{VT}, \mathrm{CT}) \vdash \texttt{chan}\ x'\ \texttt{in}\ e'[x \mapsto v] : \tau'$.

■

Proof of Lemma 8.8

In this section, I show that the matching of event values preserves the parameter type of the events. This requires the following fact about abort actions:

Lemma A.3 If $\mathrm{TE} \vdash ev : \tau\ \texttt{event}$, then $\mathrm{TE} \vdash \mathrm{AbortAct}(ev) : \texttt{unit}$.

Proof. The proof is by induction on the structure of event values and the definition of AbortAct. ■

Lemma 8.8 If $ev_1 \overset{\kappa}{\frown} ev_2$ with (e_1, e_2) and $\mathrm{TE} \vdash ev_i : \tau_i\ \texttt{event}$, then $\mathrm{TE} \vdash e_i : \tau_i$ (for $i \in \{1, 2\}$).

Proof. This is proved by induction on the definition of event matching. Let $\mathrm{TE} = (\mathrm{VT}, \mathrm{CT})$ below.

Base case: $\kappa\texttt{!}v \overset{\kappa}{\frown} \kappa\texttt{?}$ **with** $(\texttt{()}, v)$**.** For $i = 1$, the claim follows immediately from the type of $\texttt{()}$ and rule (τ**-output**). For $i = 2$, we must examine the type of κ. We have the following judgements:

$$\begin{array}{ll} \mathrm{TE} \vdash \kappa\texttt{!}v : \texttt{unit event} & (1) \\ \mathrm{TE} \vdash \kappa\texttt{?} : \tau\ \texttt{event} & (2) \end{array}$$

By rule (τ**-input**) and (2), we have $\mathrm{TE} \vdash \kappa : \tau\ \texttt{chan}$, thus, the deduction of (1) by rule (τ**-output**) requires that $\mathrm{TE} \vdash v : \tau$.

Inductive cases. For the inductive cases, the $i = 1$ case follows immediately from the induction hypothesis. The $i = 2$ case is proven by case analysis:

Case $ev_2 \overset{\kappa}{\supset} ev_1$ **with** (e_2, e_1)**.**
This case follows immediately.

Case $ev_1 \overset{\kappa}{\supset} (ev' \Rightarrow v)$ **with** $(e_1, v\ e')$**.**
Rule (τ**-wrap**) applies:

$$\frac{\mathrm{TE} \vdash ev' : \tau' \ \texttt{event} \quad \mathrm{TE} \vdash v : (\tau' \to \tau_2)}{\mathrm{TE} \vdash (ev' \Rightarrow v) : \tau_2 \ \texttt{event}}$$

Thus, applying the induction hypothesis and rule (τ**-app**) we get:

$$\frac{\mathrm{TE} \vdash e' : \tau' \quad \mathrm{TE} \vdash v : (\tau' \to \tau_2)}{\mathrm{TE} \vdash v\ e' : \tau_2}$$

Case $ev_1 \overset{\kappa}{\supset} (ev_2 \oplus ev_3)$ **with** $(e_1, \texttt{(}\mathrm{AbortAct}(ev_3)\texttt{;}\ e_2\texttt{)})$**.**
Rule (τ**-choice**) applies:

$$\frac{\mathrm{TE} \vdash ev_2 : \tau_2 \ \texttt{event} \quad \mathrm{TE} \vdash ev_3 : \tau_2 \ \texttt{event}}{\mathrm{TE} \vdash (ev_2 \oplus ev_3) : \tau_2 \ \texttt{event}}$$

Then, by the induction hypothesis, and Lemmas 8.2 and A.3, we get

$$\frac{\mathrm{TE} \vdash \mathrm{AbortAct}(ev_3) : \texttt{unit} \quad \mathrm{TE} \vdash e_2 : \tau_2}{\mathrm{TE} \vdash \texttt{(}\mathrm{AbortAct}(ev_3)\texttt{;}\ e_2\texttt{)} : \tau_2}$$

Case $ev_1 \overset{\kappa}{\supset} (ev_3 \oplus ev_2)$ **with** $(e_1, \texttt{(}\mathrm{AbortAct}(ev_3)\texttt{;}\ e_2\texttt{)})$**.**
This is the same as the previous case.

Case $ev_1 \overset{\kappa}{\supset} (ev_2 \mid v)$ **with** (e_1, e_2)**.**
This case follows immediately from the induction hypothesis.

■

Proof of Lemma 8.12

In this section, I show that stuck expressions are untypable. First, we need to characterize the syntactic form of stuck expressions.

Definition A.1 The set of *acceptable arguments* to sync is defined as

$$\textsc{SyncArg} = \textsc{Event} \cup \{(\mathbf{G}\, e) \mid e \in \textsc{Exp}\}$$

Lemma A.4 A process $\langle \pi;\, e\rangle$, with e closed, is stuck *iff* e has one of the following forms:

(1) $E[b\ v]$, such that $\delta(b, v)$ is undefined.

(2) $E[v\ v']$, where v has the form $(v_1 \,.\, v_2)$, κ, ev, or $(\mathbf{G}\, e')$.

(3) $E[\mathtt{sync}\ v]$, such that $v \notin$ SYNCARG.

Proof.

($\Rightarrow$) Let $E[e'] = e$, then this direction proceeds by case analysis of the possible forms of e'.

Case $e' = v$**.**

Then $E[e'] = [v]$, thus π it is not stuck.

Case $e' = v\ v'$**.**

This case proceeds by analysis of the form of v:

Case $v = b$**.**

If $\delta(b, v)$ is defined, then π is not stuck, otherwise it is stuck and has form 1.

Case $v = x$**.**

Then e is not closed, which is a contradiction.

Case $v = \lambda x\,(e'')$**.**

In this case, π is not stuck.

Otherwise.

In the other cases, e is stuck and has form 2.

Case $e' = \mathtt{let}\ x = v\ \mathtt{in}\ e''$**.**

In this case, π is not stuck.

Case $e' = \mathtt{sync}\ v$**.**

This case proceeds by analysis of the form of v:

Case $v = ev$**.**

π is not stuck.

Case $v = (\mathbf{G}\, e'')$**.**

π is not stuck.

Otherwise.

In the other cases, e is stuck and has form 3.

Case $e' = \mathtt{spawn}\ v$**.**

In this case, π is not stuck.

Case $e' = \mathtt{chan}\ x\ \mathtt{in}\ e''$**.**

In this case, π is not stuck.

Thus, for each possible form of e', either π is not stuck (a contradiction), or the lemma holds.

($\Leftarrow$) This direction follows immediately from the definitions.

$\blacksquare$

Before we can prove that stuck configurations are untypable, we need a lemma that characterizes the values that have event types.

Lemma A.5 If $\mathrm{TE} \vdash v : \tau$, for $v \notin \textsc{Var}$, then $\tau = \tau'$ $\mathtt{event}$, for some τ', *iff* $v \in \textsc{SyncArg}$.

Proof.

($\Rightarrow$) This direction proceeds by examination of the terms in the set

$$\textsc{Val} \setminus (\textsc{Var} \cup \textsc{SyncArg})$$

None of these terms has an inference rule that can derive a judgement of the form $\mathrm{TE} \vdash ev : \tau'\ \mathtt{event}$. Thus, since $v \notin \textsc{Var}$, $v \in \textsc{SyncArg}$.

($\Leftarrow$) This direction is by examination of the terms in $\textsc{SyncArg}$. The inference rules for these terms are (τ-**never**), (τ-**output**), (τ-**input**), (τ-**wrap**), (τ-**choice**), (τ-**abort**), and (τ-**guard**), all of which derive judgements of the form

$$\mathrm{TE} \vdash ev : \tau'\ \mathtt{event}$$

Thus, $\tau = \tau'$ $\mathtt{event}$, for some τ'.

$\blacksquare$

Finally, we are ready to the main proof.

Lemma 8.12 (Untypability of stuck configurations) If π is stuck in a well-formed configuration $\mathcal{K}, \mathcal{P}$, then there do not exist $\mathrm{CT} \in \textsc{ChanTy}$ and $\mathrm{PT} \in \textsc{ProcTy}$, such that

$$(\{\}, \mathrm{CT}) \vdash \mathcal{P}(\pi) : \mathrm{PT}(\pi)$$

In other words, $\mathcal{K}, \mathcal{P}$ is untypable.

Proof. Let π be stuck in $\mathcal{K}, \mathcal{P}$, with $\mathcal{P}(\pi) = E[e']$, and assume that there exist $\mathrm{CT} \in \textsc{ChanTy}$ and $\mathrm{PT} \in \textsc{ProcTy}$, such that $(\{\}, \mathrm{CT}) \vdash \mathcal{P}(\pi) : \mathrm{PT}(\pi)$. It suffices to show that e' is untypable, which is a contradiction. Let τ be the type of e'; i.e., $\mathrm{TE}' \vdash e' : \tau$, for some TE'. Note that since $\mathcal{K}, \mathcal{P}$ is well-formed, e' is closed; and thus Lemma A.4 gives the possible forms of e'. The proof proceeds by case analysis of e', showing that e' is untypable in each case.

Case $e' = v\ v'$**.** Rule (**τ-app**) applies:

$$\frac{\mathrm{TE}' \vdash v : (\tau' \to \tau) \quad \mathrm{TE}' \vdash v' : \tau'}{\mathrm{TE}' \vdash v\ v' : \tau} \tag{$*$}$$

There are five subcases, depending on the structure of v.

Case $v = b$, with $\delta(b, v')$ undefined**.**

By the δ–typability restriction, $\delta(b, v')$ is defined, which contradicts e' being stuck.

Case $v = (v_2 \,.\, v_3)$**.**

Rule (**τ-pair**) requires that

$$\mathrm{TE}' \vdash (v_1 \,.\, v_2) : (\tau_1 \times \tau_2)$$

where $\mathrm{TE}' \vdash v_i : \tau_i$, which contradicts the first premise of $(*)$, thus e' is untypable.

Case $v = ev$**.**

By Lemma A.5,

$$\mathrm{TE}' \vdash ev : \tau_1 \ \mathtt{event}$$

but this contradicts the first premise of $(*)$, thus e' is untypable.

Case $v = \kappa$**.**

Rule (**τ-chvar**) requires that κ have the type τ_1 **chan**, for some τ_1, but this contradicts the first premise of $(*)$, thus e' is untypable.

Case $v = (\mathbf{G}\, e'')$**.**

By Lemma A.5,

$$\mathrm{TE}' \vdash (\mathbf{G}\, e'') : \tau_1 \ \mathtt{event}$$

but this contradicts the first premise of $(*)$, thus e' is untypable.

Case $e' = \mathtt{sync}\ v$, with $v \notin \textsc{SyncArg}$**.**

Rule (**τ-sync**) applies:

$$\frac{\mathrm{TE}' \vdash v : \tau' \ \mathtt{event}}{\mathrm{TE}' \vdash \mathtt{sync}\ v : \tau'}$$

but, by Lemma A.5, $v \in \textsc{SyncArg}$, which is a contradiction.

■