Higher-Order Conditional Synchronization

by

Niki Afshartous

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
Courant Institute of Mathematical Sciences
New York University

January, 1999

For Jenny and Brian

Acknowledgements

I would first, and foremost, like to thank my wife and parents for their support and love during the many years it took to fulfill my education.

I would also like to thank my advisor, Ben Goldberg, and my dissertation committee, who's participation and review was critical throughout. John Reppy and Jon Riecke carefully reviewed earlier drafts of the dissertation. The presentation of the semantics Chapters benefited greatly from their suggestions and feedback. I also thank them both for their hospitality during several visits to Bell Labs - Lucent. Suresh Jaganthan carefully reviewed the dissertation draft and provided insightful comments and suggestions. I also benefited from the participation of Edmond Schonberg and Vijay Karemcheti.

I thank Allen Leung for many hours of discussions and for comments on earlier drafts. I would also like to thank Malcolm Harrison who headed the Griffin project at NYU. It was my involvement in Griffin which gave me experience in concurrency and programming language design and implementation. This experience sparked the idea for the the translation technique developed in this dissertation.

In addition, Anina Karmen assisted throughout by helping to interpret the academic bureaucracy. Finally I thank the LyX team for a nice writing package.

Preface

Conditional synchronization - a mechanism that conditionally blocks a thread based on the value of a boolean expression currently exists in several programming languages in second-class form. In this thesis we propose promoting conditional synchronization to first-class status. To demonstrate our idea we extend Concurrent ML and present several examples illustrating the expressiveness of first-class conditional synchronization (FCS). The end result facilitates abstraction and adds flexibility in writing concurrent programs. To minimize re-evaluation of synchronization conditions we propose a static analysis and translation that identifies expressions for the run-time system that could affect the value of a synchronization condition. The static analysis (which is based on an effect type system) therefore precludes excessive run-time system polling of synchronization conditions. This dissertation makes the following contributions:

- We show how FCS can be seamlessly integrated into an existing language (Concurrent ML)
- We illustrate the usefulness of FCS by using it to construct several abstractions (barrier synchronization and discrete event simulation in Section 2.1.2).
- We define source and target level semantics for FCS and prove that a simulation relation holds between consistent configurations of the source and target.
- We perform static analysis and optimization in conjunction with a a type-directed translation from source to target.
- We prove type soundness for the target language

The dissertation is organized into two parts. Part I (Introduction) surveys a representative selection of languages from the three categories of concurrent programming languages: concurrent object-oriented languages, concurrent constraint languages, and concurrent functional languages. Part II (Extending Concurrent ML) begins the design of first-class conditional synchronization as an extension to Concurrent ML. We chose Concurrent ML since the language has an existing framework for first-class synchronous operations. The subsequent chapters present examples, the semantics, translation scheme, and implementation. Finally we conclude and identify some topics for future research.

Contents

List of Figures

Part 1

Introduction

CHAPTER 1

Introduction

This dissertation consists of the design and implementation of the first higher-order synchronization mechanism based on using boolean conditions to control synchronization. We refer to this notion as first-class conditional synchronization (FCS). FCS has broadcast semantics making it appropriate for applications where many threads of control need to be unblocked when a `false` condition becomes `true`. Some examples of applications where this is required are synchronization barriers [**Jor78**] and discrete-event simulation (illustrated in subsection 2.1.2). Our idea is demonstrated as an extension to Concurrent ML [**Rep92**], since this language has first-class synchronous operations in order to facilitate abstraction in concurrent programming. This dissertation makes the following contributions:

- We show how FCS can be seamlessly integrated into an existing language (Concurrent ML)
- We illustrate the usefulness of FCS by using it to construct several abstractions (barrier synchronization and discrete event simulation in Section 2.1.2).
- We define source and target level semantics for FCS and prove that a simulation relation holds between consistent configurations of the source and target.
- We perform static analysis and optimization in conjunction with a type-directed translation from source to target.
- We prove type soundness for the target language

```
empty := true
slot := nil

procedure put (item) {
  await empty
  slot := item
  empty := false
}

procedure get () {
  await not empty
  empty := true
  return slot
}
```

FIGURE 1.0.1. Bounded buffer using conditional critical region

One of the earliest forms of conditional synchronization is the conditional critical region where a thread of control must wait (if necessary) for a condition to become `true` before entering a critical section. This mechanism can be used to implement abstractions such as the canonical bounded buffer (illustrated by the pseudo-code in Figure 1.0.1).

Harland in his book Concurrency and Programming Languages [**Har86**] notes that a potential problem is that a condition may become briefly `true` and then `false` again without waiting tasks noticing due to the non-determinism inherent in scheduling. The limitation is that there is no guarantee that transient changes in the value of conditions will not be missed. While this limitation was known, the alternative (searching the system for waiting tasks every time a variable update occurs) was considered too expensive. Moreover, the transient limitation is only a problem for applications that have synchronization conditions that are non-monotonic. Consider a concurrent system consisting of cars and traffic lights shown in Figure 1.0.2.

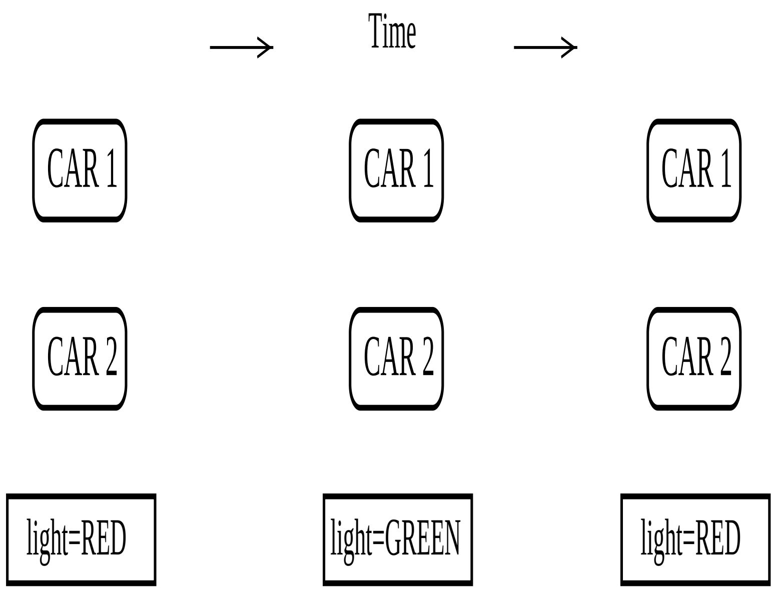

FIGURE 1.0.2. Cars waiting at a traffic light

In the example the two cars are initially waiting at a red light (synchronizing on the condition that `light=GREEN`). The traffic light then becomes briefly green and then red again. Without a semantics that guarantees transients are not missed the cars can miss a signal change!

As will be seen in Section 1.2 the conditional critical region has been integrated into Ada95 and Orca. Conditions are also used for synchronization in concurrent-constraint languages and we examine the strength and limitations of these more recent incarnations of conditional synchronization. Next, before examining some existing concurrent programming languages we discuss the history and motivation for concurrency.

1.1. Concurrency

Support for expressing concurrency (the potential to execute separate tasks in parallel) previously existed solely in the operating systems (OS) domain. At the

OS level concurrency was only accessible to the programmer via system calls like the Unix [**Bac86**] `fork` routine. Later, support for concurrency appeared at the programming language level. One of the earliest forms was the `cobegin` statement of Algol-68 [**iTCS96**]. Subsequently, Pascal Plus [**Per87**] incorporated monitors [**Hoa74**], Modula-2 [**Wir83**] supported co-routines, and Ada [**Bar89**] introduced the *task* as a unit of concurrency. There were several motivations driving this trend:

- improved performance - concurrent programs can exploit the inherent parallelism of multi-processors and distributed systems.
- improved response of interactive applications - it has been argued that designing concurrent graphical-user interfaces (GUI's) improves response time relative to sequential GUI's. [**GR93**]. This is in contrast to the traditional GUI which is based on a sequential event model.
- expressiveness - certain application's (i.e. client/server and simulation) have inherent concurrent structure and thus are more easily implemented as concurrent programs.

Another factor in the trend to support concurrency has been the increasing pervasiveness of multiprocessors and network clusters. In addition to languages supporting concurrency, there are a plethora of distributed systems that allow programmers to exploit the inherent parallelism of networks. Most provide libraries on top of existing languages (Linda [**Gel85**], and PVM [**BDG$^+$91**]).

Writing and debugging concurrent programs however is not a trivial endeavor. The concurrent programmer must deal with potential problems such as deadlock and race conditions that are not present in sequential programming. Moreover the programming language can help by providing a framework, which facilitates the construction of higher-level abstractions that hide the low-level details of communication and synchronization protocols. To address this difficulty, several mechanisms to facilitate the use of concurrency and synchronization have been incorporated into programming languages. Synchronization is provided by some variation of either message-passing (synchronous or asynchronous), monitors, or the rendezvous. Some languages also provide a select construct to allow a non-deterministic choice among several synchronous operations [**Hoa78**]. These synchronization mechanisms are described as follows.

- message-passing (synchronous) - After both a sender and a receiver have arrived at their respective synchronization points a message is transmitted from the sender to the receiver and then both processes continue. Whichever process (sender or receiver) arrives first must wait for the other process to arrive at its corresponding synchronization point.
- message-passing (asynchronous) - The sender deposits a message in a buffer where it may later be picked up by the receiver. A receiver must wait for a message (block) if the buffer is empty.
- rendezvous - The Ada rendezvous is a request-reply protocol. Like synchronous message passing two processes (Ada tasks) arrive at their respective synchronization points. Task A executes the rendezvous body (using input from task B) and then sends a reply back to task B.
- monitor - a set of procedures which collectively define a critical section. Only a single thread of control may be executing inside the monitor at any time.

Several advantages can be gained by supporting concurrency at the programming language level. First, portability, since it is no longer necessary to make external system calls. And second, when a concept becomes part of a language, the compiler can more thoroughly analyze a program's semantics. In addition improved performance may be achieved by having the compiler perform optimization. For instance, it has been discovered that in the case of integrating transactions and persistence into a language, greater concurrency can be achieved with user defined type-specific locking as opposed to traditional read/write locking [**HW87**]. Thus, it becomes clear that when new language concepts are well integrated into a programming language there are many benefits. We propose the following as criteria to help evaluate whether or not a language extension has merit:

- does incorporating the extension into the language create opportunities for improved performance via optimizing transformations ?
- can guarantees of correctness properties be made by analyzing and verifying program semantics ?
- does the language extension facilitate abstraction ?

In the remainder of this Chapter we survey existing forms of conditional synchronization in the context of concurrent object-oriented languages and concurrent constraint languages. Since FCS extends Concurrent ML (a concurrent functional language) we survey a representative selection of concurrent functional languages to identify and characterize existing support for concurrency and synchronization. Part II (Extending Concurrent ML) begins the design of a new synchronization mechanism (FCS - first-class conditional synchronization) as an extension to Concurrent ML. FCS uses conditions as a basis for synchronization and has broadcast semantics. The subsequent Chapters present examples, the formal semantics, and translation scheme.

We conclude by evaluating our language extension based on the above criteria and then suggest some topics for future research.

1.2. Concurrent object-oriented languages

During the same time that languages supporting concurrency and distribution have evolved, languages based on the object-oriented paradigm [**Mey88**] have been gaining popularity. The benefits that are commonly cited are:

- data abstraction
- facilitation of code re-use via inheritance
- flexibility resulting from subtype polymorphism and dynamic binding

It seems natural then that the concepts surrounding concurrency and object-orientation should meet and interact giving rise to Concurrent Object-Oriented Languages (COOL's). Various approaches have been undertaken to integrate the two areas. See [**Pap89**] for a classification scheme. In the following Subsubsections, our focus is on COOL's which have a form of conditional synchronization.

1.2.1. Ada95.

The main additions to Ada introduced by the Ada95 language definition are support for object-oriented programming and protected objects. Ada95 has two kinds of objects. Associated with every *protected object* is a monitor so that concurrent access by competing *tasks* controlled. Conditional synchronization may be used in conjunction with protected objects. The other kind of object called a *tagged* object is associated with polymorphism and inheritance.

Concurrency in Ada is provided by *tasks.* A task consists of a specification and a body. The specification lists the entries which may be called by other tasks in order to rendezvous. A rendezvous may proceed when task A invokes the `entry` of task

```
task Buffer is
  entry Get (X: out item);
  entry Put (X: in item);
end Buffer;

task body Buffer is
  V: item;
begin
  loop
    accept Get (X: out item) do
      X := V;
    end Get;

    accept Put (X: in item) do
      V := X;
    end Put;
  end loop;
end Buffer;
```

FIGURE 1.2.1. A one-slot buffer task in Ada95

B and task B has reached the `accept` statement corresponding to the entry. This is illustrated by an example from [**Int95**] in Figure 1.2.1. The task Buffer encapsulates a single value while supporting read and write operations as entries. Other tasks may invoke task Buffer's entries by calling `Buffer.Put` or `Buffer.Get`. Entries may have both `in` and `out` parameters and hence the rendezvous is a synchronous request-reply protocol.

A protected object type definition may consist of data (declared in the private part) and three kinds of subprograms: functions, procedures, and entries. Functions may only read (not write) the data of a protected object and hence an implementation is free to execute function calls concurrently. Since procedures and entries may write to a protected object's data, mutual exclusion is implicitly applied. The distinction between procedures and entries is that entries must specify boolean conditions (called barriers) which control access into the entry body (conditional synchronization). An `entry` may contain parameters but there is an inhibiting restriction that the formal parameters of an `entry` may not be referenced in the boolean expression.

Tasks invoking entries with `false` barriers are queued until the barrier condition becomes `true` and then only one queued task may proceed due to the mutual exclusion requirement of the protected object. If there are tasks queued on barriers, then the conditions are re-evaluated every time a task exits a procedure or entry. To illustrate protected objects we examine several examples from [**Bar96**]. Figure 1.2.2 is a solution to the readers/writers problem. Since the read operation has no side-effects, it is written as a function while the write operation is a procedure. The next example of a producer/consumer protected object in Figure 1.2.3 illustrates the use of entries and the use of associated barriers for conditional synchronization. The `put` operation may proceed when the buffer is not full (`count < N`) and conversely the get operation may proceed when the buffer is not empty (`count > 0`).[1] Since the entries occur within a protected type, the entry provides a more modern form of the conditional critical region discussed earlier.

1.2.2. Orca. Orca [**BKT88**] supports concurrent programming by processes that communicate via shared objects. The language goals were to make it simple, expressive, and efficient with clean semantics. Logically shared memory is supported via a reliable broadcast protocol. The salient issue addressed by ORCA is how data can be shared among distributed processors in an efficient way. Access to shared

[1]The automatic wrapping (mod N arithmetic) of the variables `I` and `J` is a property of the type `Index`.

```
protected Variable is
  procedure Read(X: out item);
  procedure Write(X: in item);
private
  V: Item := initial value;
end Variable;

protected body Variable is

  procedure Read(X: out item) is
  begin
    X := V;
  end Read;

  procedure Write(X: in item) is
    V := X;
  end Write;

end Variable;
```

FIGURE 1.2.2. Readers/writers in Ada95

data structures is performed through higher level operations instead of using low level instructions for reading, writing, and locking data. Operations may contain guarded branches which are used for conditional synchronization.

Orca intentionally omits pointers because pointers are invalid across node address spaces. Instead Orca provides language constructs that allow the run-time system to keep track of the data structure that a dynamic node is associated with. This yields dynamic data structures that are first-class and type-secure. Orca has hierchical objects (objects can contain other objects) but no class inheritance.

A shared data-object is an instance of an abstract data type. An abstract data type definition in Orca consists of two parts: *specification* and *implementation*. The specification part defines the operations allowed on the corresponding type. The

```
N: constant := 8;
type Index is mod N;
type Item_array is array (Index of Item;

protected type Buffer is
  entry Get(X: out item);
  entry Put(X: in item);
private
  A: Item_Array;
  I,J: Index := 0;
  Count: Integer range 0..N := 0;
end Buffer;

protected body Buffer is

  entry Get(X: out item) when Count > 0 is
  begin
    X := A(J);
    J := J + 1;  Count := Count - 1;
  end Read;

  entry Write(X: in item);
    A(I) := X;
    I := I + 1;  Count := Count + 1;
  end Write;

end Buffer;
```

FIGURE 1.2.3. Readers/writers in Ada95

implementation part consists of the data and code used to implement the operations described in the specification. Objects are passive entities in Orca. Concurrency in Orca is expressed by explicit creation of sequential processes. Initially a program consists of a single process but new processes can be created via the `fork` statement:

```
fork name (actual-parameters) [ on (cpu-number) ]
```

The on keyword allows the option of locating the new process on a specified physical processor. If the **on** part is absent then the new process resides on the same processor as its parent.

There are two kinds of actual parameters. The first is value or input parameters. Copies of a value parameter are passed to a process. The second is shared objects. In this case the data object is shared between the parent and child and can be used for communication.

For objects that are shared among multiple processes the issue of synchronization arises. Orca provides two types of synchronization: mutual exclusion and conditional synchronization.

Mutual exclusion is performed implicitly by executing all object operations atomically. Conceptually an operation locks the entire object so that other invocations of operations cannot interfere. More precisely, *serializability*[**EGLT76**] of operations is guaranteed. An implementation of the serializability model is not restricted to executing each operation one by one. To allow for greater concurrency, operations which do not conflict may be run simultaneously. An operation can wait (block) by using a condition synchronization. In Orca the conditional synchronization is integrated with operations as illustrated below.

```
operation op(formal-parameters):  ResultType;
begin
  guard condition1 do statements1 od;
  ..
  guard conditionN do statementsN od;
end;
```

```
object specification BoundedBuffer;
  operation Put (item: T);
  operation Get (): T;
end generic;
```

FIGURE 1.2.4. Orca specification of bounded buffer

The conditions are boolean expressions called *guards*. An operation initially blocks until one of the guards evaluates to `true` (conditional synchronization). One of the `true` branches is then selected non-deterministically and its statement sequence is executed. Part of the Orca version of the bounded buffer is shown in figures 1.2.4 and 1.2.5 and uses guards.

A guard condition may contain operation parameters, and local variables of the object and operation. A guard that initially fails may later become `true`, so it may be necessary to evaluate a guard more than once. In this model an operation only blocks initially which allows Orca to guarantee serializability. If an operation could block at anytime serializability could not be provided.

1.3. Concurrent constraint languages

Vijay Saraswat's dissertation research [**Sar93**] was the design of concurrent-constraint programming CCP, which unified the areas of concurrent logic programming [**Sha86**] [**Sha87**] and constraint logic languages [**BC93**] [**JL87**]. In CCP, communication and control between concurrently executing agents is mediated through operations on constraints. What makes this model of computation unique is that there is a global store of constraints. This store has the monotonic property that a constraint `c` can only be added if `c` does not conflict with existing constraints.

```
generic
object implementation BoundedBuffer
 buffer: array [1..N] of integer;
 getIndex, putIndex: integer;

 operation Put(item: T)
  begin
   guard not full do
    buffer[putIndex] := item;
    if putIndex = N then
     putIndex := 1;
    else
     putIndex := putIndex + 1;
    fi;
    if putIndex = getIndex then
     full := true;
    fi;
    empty := false;
   od;
  end;

 operation Get(): T
  item: T;
  begin
    ...
  end;
 begin
  putIndex := 1; getIndex := 1;
  empty := true; full := false;
 end generic;
```

FIGURE 1.2.5. Orca implementation of bounded buffer

Hence once a constraint c becomes satisfied in a particular store, c will continue to hold for all successive store configurations.

The fundamental operations in CCP are `ask` and `tell`. An agent may either atomically add ("tell") a constraint on the value of variables, or verify ("ask") that

```
max(X,Y,Z) :: ask(X<=Y) : tell(Y=Z) --> stop.

max(X,Y,Z) :: ask(Y<=X) : tell(X=Z) --> stop.
```

FIGURE 1.3.1. Max of two numbers

the store entails a given constraint. The ask operation can be viewed as a form of conditional synchronization. When an agent performs an `ask` to establish whether the store entails a constraint `c` there are three possible results:

1. The information in the store agrees with `c` and the agent proceeds
2. The information in the store rejects `c` and the agent is aborted
3. At the current time there is not enough information in the store to either reject or confirm `c`. The agent is then blocked indefinitely until other agents add information to the store that either confirms or rejects `c`.

To illustrate this framework we examine several of the examples in [**Sar93**]. First a simple program which calculates the maximum of two numbers. Invoking the goal `max(X,Y,Z)` (Figure 1.3.1) suspends until the values of `X` and `Y` become fixed in the current store. The colon : is used to compose the `ask` and `tell` into an atomic unit that must be satisfied before committing to a branch with the `-->`. If more than one goal branch may be committed to, then one is non-deterministically chosen and the commit point is returned to via backtracking if the chosen branch results in goal failure. Once the `-->` is reached the other possible branches that may satisfy a goal are aborted. Operationally, $c_1 : c_2$ succeeds only if the constraint c_1 and c_2 hold in the current store. If c_1 suspends then $c_1 : c_2$ suspends and if c_2 suspends then $c_1 : c_2$ suspends. Finally after determining the maximum number to be `X` or `Y` the agent is terminated with `stop`.

The next example is the admissible pairs problem. The goal is to verify that a list of pairs L satisfies two properties:

1. if $(x, y) \in L$ then $y = 2 * x$
2. if (x_1, y_1) and (x_2, y_2) are successive elements of L then $x_2 = 3 * y_1$

The approach taken in [**Sar93**] is to spawn two agents to verify each of the two properties. To verify the first property we have predicate `double` shown in Figure 1.3.4. The notation `(X, Y, L1)^` introduces the three variables `X,Y,` and `L1` into the current scope. The goal `double(L)` succeeds if either `L` is the empty list (`L=<>`), or `L` is of the form `<pair(X, Y) | L1>` where `Y=2*X` and the goal `double(L1)` also succeeds. The goal `triple(L)` succeeds if `L` is of the form `<pair(X, Y) | L1>` and either `L1` is empty or `L1` is of the form `<pair(3*Y, Z) | L2>` and `triple(L2)` succeeds. Now with the predicates `double` and `triple` defined the complete solution is shown in Figure 1.3.2. The curly braces have the effect of introducing a local procedure into the current scope. In Figure 1.3.2 `double` and `triple` are introduced as local procedures before invoking them. Invoking the goal `admissible(L)` spawns three agents. The first agent invokes the goal `double(L)` which then suspends until it is determined that `L` is either the empty list or of the form `<pair(X,Y)|L1>` . The second agent invokes the goal `triple(L)` which first establishes that L is of the form `<pair(X,Y)|L1>` and then chooses one of the two `OR` branches. Thus the first agent calculates the doubles which are input to the second agent which calculates the triples which in turn triggers the first agent. The third agent generates the initial structure of the list and partially specifies the first pair. The trace in Figure 1.3.3below shows the intermediate steps involved in constructing the list with labels to identify the agent responsible for each step.

```
admissible(L)::(S, U)^
  {double(L)::(X, Y, L1)^
    ask(L = <>) --> stop
   OR
    ask(X, Y, L1) L = <pair(X, Y) | L1> --> (Y = 2 * X), double(L1).
   double(L)},

  {triple(L)::(X, Y, L1, L2, Z)^
    L = <pair(X, Y) | L1>,
    (tell(L = <>) --> stop
      OR
     tell(L = <pair(3 * Y, Z) | L2>) --> triple(L2)).
   triple(L)},
   L = <pair(1,S) | U>.
```

FIGURE 1.3.2. Complete admissible pairs solution

$< \texttt{pair}(1, Z_1) \mid U_1 >$	`agent 3`
$< \texttt{pair}(1, 2) \mid U_1 >$	`agent 1`
$< \texttt{pair}(1, 2), \texttt{pair}(6, Z_2) \mid U_2 >$	`agent 2`
$< \texttt{pair}(1, 2), \texttt{pair}(6, 12) \mid U_2 >$	`agent 1`
$< \texttt{pair}(1, 2), \texttt{pair}(6, 12), \texttt{pair}(36, Z_3) \mid U_3 >$	`agent 2`

FIGURE 1.3.3. Trace of admissible program.

1.4. Discussion

Both Ada95 and Orca have introduced conditional synchronization (a mechanism that uses boolean conditions to control synchronization). The event of a condition becoming `true` enables waiting processes to proceed. However, there are drawbacks. In both cases the boolean expression is "hard-wired" in the syntax of the language

```
double(L)::(X, Y, L1)^
  ask(L = <>) --> stop
 OR
  ask(X, Y, L1) L = <pair(X, Y) | L1> --> (Y = 2 * X), double(L1).

triple(L)::(X, Y, L1, L2, Z)^
  L = <pair(X, Y) | L1>,
  (tell(L = <>) --> stop
    OR
   tell(L = <pair(3 * Y, Z) | L2>) --> triple(L2)).
```

FIGURE 1.3.4. Verifying admissible properties

and the synchronization point is not a first-class value. Also, no guarantee is provided that transient changes are not missed. That is, a synchronization condition may becomes briefly `true` and then `false` again without unblocking any waiting tasks.

In particular to Ada95, referencing variables non-local to a protected type inside a synchronization condition yields unpredictable results. This is because an update to a non-local variables will not trigger re-evaluation of a barrier condition. Finally, it is not apparent from a specification that an Ada95 entry or an Orca operation may block. Thus an important property, the fact that operations are synchronous, is lost in constructing abstractions [**Rep92**].

CCP presents a radically different model of concurrent computation based on viewing the store as a set of monotonically increasing constraints. The `ask` operation is a form of conditional synchronization that involves suspending an agent until such time that the monotonic store can either accept or reject a constraint. Like the forms of conditional synchronization in Ada95 and Orca, the CCP `ask` operation is a second-class conditional synchronization. The issue of missing transient changes does not apply to CCP due to the monotonic store. That is, a constraint that holds at a given point in time continues to hold subsequently.

1.5. Concurrent functional languages

There is considerable variety in the characteristics of functional languages (FL's), but the basis is Church's λ-calculus [**Bar84**]. Functional languages typically extend the calculus with higher-level constructs such as pattern-matching and facilitate the use of recursion by performing optimizations. Garbage collection [**JL96**] is employed to relieve the programmer from the burden of dynamic memory management and variables are bound to values as opposed to state. The variations pertain to the following design points:

- static versus dynamic typing - some FL's have statically typed expressions and hence can perform type-checking prior to run-time. In the dynamically typed languages the types of expressions is not fixed and consequently type-checking is performed at run-time.
- imperative features - some FL's support mutable references while *pure* FL's do not. However the pure FL's can mimic imperative features through the use of monads [**Wad95**].
- strict - the semantics of the language specify the exact order in which the actual arguments are evaluated.
- non-strict - the semantics of the language do not specify the order of evaluation. One has to be careful so that different evaluations do not result in different execution results. Non-strict language may be *lazy* meaning that arguments are not evaluated until their values are needed inside the function body. One way to implement non-strict evaluation is by associating a

```
(let
  ((x (future exp)))
  bodyExp)
```

FIGURE 1.5.1. Futures in Multi-lisp

parameterless function (called a *thunk* [**iTCS96**]) with each argument which is evaluated for each use of the corresponding formal parameter inside the function body.

As with COOL's, concurrent functional languages integrate a notion of process and some variation of the mechanisms for synchronization/communication. As conditional synchronization has yet to be explored in the context of a CFL, the purpose of the CFL survey is to examine the other existing mechanisms for communication and synchronization.

1.5.1. Multi-lisp. Multi-lisp [**RHH85**]is one of the earliest CFL's. A test-and-set like mechanism is provided for communication and synchronization. In addition, Multi-lisp (as well as other concurrent dialects of Lisp and Scheme [**KJ88**]) provides *futures.* The `let` expression in Figure 1.5.1 spawns a child thread to evaluate *exp*. Referencing variable `x` suspends the parent thread until the child thread has finished evaluating *exp* with the resulting value stored in variable x. Futures are a one-shot synchronization mechanism and are intended for use in parallel programming as opposed building higher-level communication and synchronization abstractions.

1.5.2. Concurrent ML. An important advance towards the goal of integrating concurrency into programming languages is the concept of first-class synchronous operations, which first appeared in PML [**Rep88**] and was then further developed as Concurrent ML (CML) ([**Rep92**],[**Rep95**]).

The salient feature of CML is the `event` type constructor, which is the type of first-class synchronous operations. An event `value` represents a *delayed* synchronous operation. The delayed synchronous operation may later be *forced* applying the `sync` function to an `event` value. Below are the specifications of the `sendEvt, recvEvt`, and `sync` functions, which are used for synchronous message-passing:

```
val sendEvt: 'a chan * 'a -> unit event

val recvEvt: 'a chan -> 'a even

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

Both `sendEvt` and `recvEvt` are non-blocking operations. That is, they both create delayed synchronous operations that may later be *forced* by applying `sync` to an `event` value. The `sendEvt` function takes two arguments: a channel of type `'a chan` and a value of type `'a` and then returns a `unit event` value. The `unit event` value represents the delayed synchronous operation of attempting synchronization by sending the value of type `'a` across the channel argument to `sendEvt`. Applying the `sync` function to the `event` value forces the synchronization by actually attempting to send the value across the channel. The `recvEvt` function takes a single channel argument and returns an event value corresponding to the delayed operation of synchronizing by receiving on the channel argument to `recvEvt.` Applying `sync` to this `event value` forces the synchronization by attempting to receive a value from the channel.

Thus in order for a message to be transmitted there must be two threads which have applied `sync` to two events corresponding to the act of sending and receiving on the same channel. When the synchronization takes place a message is sent from the sending thread to the receiving thread. The core operations of CML are shown in Figure 1.5.2.

```
val sendEvt:  'a chan * 'a -> unit event
val recvEvt:  'a chan -> 'a event

val recv:     'a chan -> 'a
val send:     'a chan * 'a -> unit

val spawn:    (unit -> unit) -> thread_id
val channel:  unit -> 'a chan
val wrap:     ('a event * (a -> b)) -> 'b event

val choose:   'a event list -> 'a event
val select:   'a event list -> 'a

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

FIGURE 1.5.2. The core CML operations

When `sync` is directly applied to an `event` created with `sendEvt` or `recvEvt`, the shorthands `send` or `recv` may be used. `send` is `sync` composed with `sendEvt`, and `recv` is `sync` composed with `recvEvt`. The `wrap` combinator binds an action, represented as a function, to an `event` value. And the `choose` combinator composes a list of events into a new `event` that represents the non-deterministic selection of one of the events in the list. `select` is `sync` composed with `choose`. The advantage of this approach to concurrency is that it allows the programmer to construct first-class synchronization and communication abstractions. For example, as shown in [**Rep92**], one could define an event-valued function that implements the client side of the RPC protocol:

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

Applying it `ClientCallEvt` to a value `v` does not actually send a request to the server, rather it returns an event value that can be used later to send `v` to the server

```
type 'a mvar
val mVar      : unit -> 'a mvar
val mVarInit : 'a -> 'a mvar
val mPut      : ('a mvar * 'a) -> unit
val mTake     : 'a mvar -> 'a
val mTakeEvt : 'a mvar -> 'a event
val mGet      : 'a mvar -> 'a
val mGetEvt  : 'a mvar -> 'a event
```

FIGURE 1.5.3. CML M-variable operations

and then accept the server's reply. Furthermore, since `ClientCallEvt` returns an event value, we can place a call to `ClientCallEvt` in the list argument to `select`.

The key advantage to first-class synchronous values is that different kinds of synchronization mechanisms may be integrated into the framework. In addition to the events corresponding to channel communication, CML also has Id-style [**AP89**] synchronous variables called M-variables whose operations are shown in Figure 1.5.3. An M-variable may be created by calling `mVar` or by calling `mVarInit` to store an initial value in the M-variable. The `mPut` operation updates the M-variable location with a value while `mTake` operation consumes the value in the M-variable. Calling `mPut` blocks if the M-variable is full while calling `mTake` blocks if the M-variable is empty. The `mGet` operation is similar to `mTake` except that `mGet` returns the encapsulated value without emptying the M-variable.

CML also has I-variables which are similar to M-variables except that I-variables encapsulate write-once memory locations. An exception is raised on successive attempts to put a value in the I-variable.

```
datatype 'a buffer_chan = BC of {
                            inCh : 'a chan,
                            outCh : 'a chan
                   }

fun buffer () = let
  val  inCh = channel()
      and
       outCh = channel()
  fun loop ([],[]) = loop([accept inCh], [])
    | loop (front as (x::r), rear) =
        select [wrap(recvEvt inCh,
                      fn y => loop(front, y::rear)),
                wrap(sendEvt(outCh x),
                      fn y => 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,...}, x) = recvEvt outCh
```

FIGURE 1.5.4. A buffered channel abstraction

The example in Figure 1.5.4 from [**Rep92**] is a CML implementation of a buffered channel with an asynchronous send operation (`bufferSend`). Since the receive operation is blocking, `bufferReceive` returns an event value. Calling function `buffer` spawns a thread which selectively waits on `inCh` and `outCh`. The argument to function `loop` consists of a pair of lists. The first list is used to hold outgoing messages while the second holds newly received messages. When the first list is empty, the contents of the second list are reversed and transferred to the first list via a recursive call. Reppy makes the following observations:

- buffered channels are a new first-class communication abstraction. In particular, `bufferedReceive` returns an event value allowing calls to `buffevredReceive` to be used in contexts where an event is required (i.e. the argument list to `select`).
- the buffered channel implementation illustrates the need for generalized selective communication since the call to `select` inside function `loop` waits to both send and receive on different channels
- the buffered channel relies on the fact that unreachable threads and channels are cleaned up by the garbage collector. Thus a client does not have to be concerned about a server thread which runs in an infinite loop.

In addition to CML there have been several other designs which integrate message-passing and ML [**Mat89**] [**Ram90**] [**KPT96**].

1.5.3. Synchrons. The *synchron*[**Tur96**] is a first-class barrier synchronization mechanism. With the *synchron* the number of threads participating in the barrier is not fixed as additional threads may subsequently join the barrier group. A rendezvous takes place when all threads holding a *synchron* arrive at the barrier by *waiting* on the synchron. Synchrons are powerful but more complex to manage and are not intended for use by the "casual programmer" but are intended to be packaged inside higher-level abstractions. The motivation for *synchrons* was the design of space-efficient algorithms as the modular composition of aggregate data operations.

1.5.4. Concurrent Haskell. Concurrent Haskell [**JGF96**] is a lazy (non-strict) purely functional language in which IO and mutable state operations are based on monads [**Wad95**]. Monadic operations are viewed as sate transformations which transform the current state of the world into a new state. This idea is reflected in the following type definition:

```
type IO a = World -> (a, World)
```

where an operation of type `IO a` takes a world state as input and transforms the state into a new state with a value of type `a`. Processes are created by calling `forkIO`:

```
forkIO :: IO () -> IO ()
```

which takes an action as its argument and performs the action in a new concurrent process. Synchronization between concurrent processes is based on the `MVar` which integrates previous work on mutable structures [**LJ96**] and the I-structures and M-structures of the dataflow language Id [**AP89**]. Mvar's are similar to the CML M-variable. The `MVar` operations are:

- `newMVar :: IO (MVar a)` - creates a new `MVar`.
- `takeMVar :: MVar a -> IO a` - returns the value of type `a` encapsulated inside the `MVar` (blocking if necessary until the `MVar` becomes full).
- `putMVar :: MVar a -> IO ()` - stores a value of type `a` inside the `MVar`. This operation results in an error if the `MVar` is already full.

The design philosophy was to provide only the `MVar` as a low-level synchronization mechanism around which higher-level abstraction could be built. Using `MVar` it is shown in [**JGF96**] how to construct a buffered channel, skip channel, and quantity semaphore.

The example in Figure 1.5.5 from [**JGF96**] uses the `MVar` to implement a one-slot bounded buffer. The `CVar` abstraction encapsulates two `MVar's.` The first is used to hold the data and the second is used for the consumer to communicate to

```
type CVar a = (MVar a,
               MVar ())

newCar :: IO (CVar a)
newCVar =
  newMVar >>= \ data_var ->
  newMVar >>= \ ack_var ->
  putMVar ack_var ()
  return (data_var, ack_var)

putCVar :: CVar a -> a -> IO ()
putCVar (data_var, ack_var) val =
  takeMVar ack_var >>
  putMVar data_Var val

getCVar :: CVar a ->  IO a
getCVar (data_var, ack_var) val =
  takeMVar data_var  >>= \ val ->
  putMVar ack_var ()    >>
  return val
```

FIGURE 1.5.5. One-slot bounded buffer in Concurrent Haskell

the producer that the `CVar` is empty. Invoking `newCVar` creates two `MVar's` and `putMVar` is applied to the second before returning the pair. The syntax `>>= \var` is used to bind the result of an operation to a variable. and the `::` indicates the types of the `CVar` operations. The `putCVar` operation checks the `ack_var` first to determine that the `CVar` is empty and subsequently calls `putMVar` on `data_var` to produce a value. `getCVar` first calls `takeMVar` on `data_var` (blocking if necessary) and then changes the `CVar` state to empty by calling `putMVar` on `ack_var`.

We note that if `putMVar` on a full `MVar` simply blocked the calling process (as opposed to generating an error) then one `MVar` would suffice as a solution to the one-slot bounded buffer since the second `MVar` would be unnecessary.

Concurrent Clean [**AP95**] is another purely functional language which is also based on monadic operations. Concurrent Clean has processes which communicate via either synchronous or asynchronous message-passing.

1.5.5. Erlang. The concurrent functional language Erlang [**AWWV96**] [**Wik96**] was designed in the late 1980's at Ericsson's Computer Science Lab for application development within Ericsson. Erlang is a dynamically typed language although recently a type system has been designed for Erlang [**MW97**]. Multiprocessor and distributed implementations of Erlang are also available [**Wik86**] . Erlang is distinct among functional languages in that it has been used commercially in large scale applications.

Concurrency is supported by processes and message-passing for communication and synchronization. The function `spawn` creates a new concurrent process to perform an action. The message-passing operations are send (infix !) and `receive` (which uses pattern-matching). Channels are not provided since the sender specifies the process ID of the receiver. We use the example in Figures 1.5.6 from [**AWWV96**] to illustrate the concurrent facilities of Erlang. In the example the a process executing the function `loop` suspends until receiving a message of the form:

- `increment` - which causes the process to increment the counter
- `{from}` - which sends the message `{self(), val}` back to the requesting process. Calling `self()` returns the Id of the current process.
- `stop` - which causes the process to terminate.
- Other - which is used to ignore unrecognized messages.

```
-module(counter).
-export([loop/1]).

loop(val) ->
  receive
    increment ->  loop(val+1);

    {from} -> from!{self(), val}, loop(val);

    stop -> true;

    Other -> loop(val)
end.
```

FIGURE 1.5.6. An Erlang process encapsulating a counter

```
...
  pid = spawn(counter,loop,[0]),

  pid!increment,
  pid!increment,

  pid!{self()}
  receive {pid,value} -> value
  end,
...
```

FIGURE 1.5.7. Spawning the counter process

The program fragment in 1.5.7 spawns a process to execute `loop` with an initial value of `0`. Then two increment message are sent before requesting that the current value be sent back to the parent process.

Synchronization/Communication	Languages
message-passing (sync)	CML, Concurrent Clean, Erlang
message-passing (async)	CML, Concurrent Clean
monitor	none
rendezvous	none
conditional synchronization	none
M-Variable	CML, Concurrent Haskell

TABLE 1. Communication/synchronization in concurrent functional languages

1.5.6. Discussion. By incorporating first-class synchronization mechanisms such as CML's `event` and Concurrent Haskell's `MVar`, functional programming languages are facilitating abstraction in concurrent programs. Moreover, as shown in Table 1, conditional synchronization has yet to be explored in the context of a concurrent function language.

Part 2

Extending Concurrent ML

CHAPTER 2

Design

To address the restrictions of existing forms of conditional synchronization (CS), we propose promoting CS to first-class status (FCS). Moreover, our static analysis and translation helps the the run-time system in deciding when to re-evaluate pending synchronization conditions. In this and the subsequent Chapters:

- We show how FCS can be seamlessly integrated into an existing language (Concurrent ML)
- We illustrate the usefulness of FCS with examples (barrier synchronization and discrete event simulation in Section 2.1.2)
- We define source and target level semantics for FCS and prove that a simulation relation holds between consistent configurations of the source and target.
- We define a type-directed translation from source to target.
- We prove type soundness for the target language.

2.1. Synchronizing with first-class conditions

We chose to introduce FCS as an extension of Concurrent ML (CML) [**Rep92**] since it already has an existing framework for first-class synchronous operations. As described in Subsection 1.5.2 CML is a concurrent extension of Standard ML of New Jersey.

2.1.1. Extension to CML. At the language level, our extension to CML consists of adding the function `condEvt`:

```
condEvt: (unit -> bool) -> unit event
```

in conjunction with special reference variables with the associated operations:

```
new : 'a -> 'a sync_ref
get : 'a sync_ref -> 'a
set : 'a * 'a -> unit
```

We do not use the SML reference variables for synchronization in order to avoid introducing overhead in programs which use references but not for synchronization. This will become clear when the translation is presented in Chapter 5 since the translation adds extra annotations for synchronization references to the target program.

The function argument to `condEvt` encapsulates a boolean expression corresponding to a synchronization point. We call the result returned by `condEvt` a *conditional event*. Operationally, a thread attempting to apply `sync` on a conditional event will block until the boolean expression becomes `true`. Therefore it may be necessary to re-evaluate the boolean expression multiple times during the synchronization.

The CML `event` type allows us to abstract away from the details behind synchronization. Moreover, with the addition of the conditional event, one cannot perceive if the synchronization behind an `event` value is associated with a rendezvous or a conditional synchronization. Thus synchronous values of the same type can actually represent different kinds of synchronization [**Rep92**].

2.1.2. Examples. We now illustrate the the conditional event mechanism through several examples. The first example in Figure 2.1.1 illustrates a thread waiting for a change to a shared reference variable `x` by blocking on a conditional event. The

```
let
  val x = new 5
  val xEvent = condEvt(let
                         val oldx = get x
                       in
                         fn () => get x <> oldx
                       end)
  fun foo () = set(x,10)
in
  spawn foo;
  sync(wrap (xEvent, fn () => print "x has changed value"))
end
```

FIGURE 2.1.1. A thread blocking on a shared variable

```
let
  val x = new 5
  val oldx = get x
  val ch = channel()
  fun foo () = set(x, 10)
  fun poll () = if (get x) <> set(oldx,10) then
                  send(ch,())
                else
                  poll()
  val xEvent = recvEvt ch
in
  spawn foo;
  spawn poll;
  sync (wrap (xEvent, fn () => print "x has changed value"))
end
```

FIGURE 2.1.2. A thread blocking on a shared variable by busy-waiting

event `xEvent` encapsulates the initial value of `x` and becomes enabled for synchronization when `x`'s value changes. While not of practical use, the example illustrates one use of a conditional event that would otherwise require the programmer to write

```
fun makeBarrier n =
  let
    val counter = new 0
    val incCh = channel()
    fun inc () = (recv incCh;
                  set(counter,get counter + 1);
                  inc())
    val barrierEvt = condEvt (fn () => get counter = n)

  in
    spawn inc;
    wrap(sendEvt(incCh,()),
         fn ()=> sync barrierEvt)
  end
```

FIGURE 2.1.3. Barrier synchronization

a busy-waiting loop as shown in Figure 2.1.2. However, there are two advantages to using `condEvt`. First it is not necessary to spawn an auxiliary thread to do the busy-waiting. And second, the semantics of `condEvt` provides a guarantee that no transient changes are missed where a condition becomes `true` briefly and then `false` before being detected. This kind of guarantee cannot be programmed since it is dependant upon the thread scheduler.

Next, we use `condEvt` to write the function `makeBarrier` that returns an `event` to be subsequently used by a group of threads participating in barrier synchronization. Barriers have been applied in parallel applications such as global atmospheric circulation, the n-body gravitational problem, and parallel sorting [**GA89**]. The argument `n` to function `makeBarrier` in Figure 2.1.3 corresponds to the number of threads participating. Each thread signals that it has reached the barrier by applying `sync` to the event returned by `makeBarrier`. An auxiliary thread is spawned to coordinate increments to the `counter` reference variable.

The second example (Figure 2.1.4) illustrates how FCS can be applied in implementing a discrete event simulator. The system consists of actor threads and a scheduler thread. The scheduler is parameterized by the number of cycles to run the system and the number of actors present. During each cycle the scheduler:

- advances the clock one tick
- waits for actors scheduled to wake up at the current time by calling `processTimeQueue`
- waits for all actors to sleep before starting the next cycle

Actors may call either `timeWait` or `eventWait` to sleep until a specified time or event respectively. Calling `timeWait` returns immediately if the clock has already advanced past the time argument `t`. Otherwise a message is sent on the `timeSleep` channel to inform the scheduler that an actor is waiting for time `t`. Then a wrapped event is returned for which synchronization will succeed when the clock is at time `t`, while the wrapper function informs the scheduler that the actor is awake. The broadcast semantics of `condEvt` are appropriate since all actors waiting for a given time can be simultaneously unblocked. The function `eventWait` is similar to `timeWait` except that synchronization is based on a boolean expression which is passed inside the argument `e`, a `unit -> bool` function.

The scheduler maintains an internal hash table in order to keep track of how many actors are waiting for a given time step. In the scheduler, `incTimeStep` (body not shown) increments the counter for a given time step. The scheduler also spawns an auxiliary thread to execute function `trackAll` which listens for messages from the actors in order to maintain the `awakeActors` count. In addition, the main scheduler thread sends a message on the `startTimeCh` at the beginning of each time step so that the auxiliary thread can wait for all actors slated for the current time step to wake up.

The `condEvt` function is used to implement the blocking and queuing of the actors. Without using `condEvt`, additional coding and data structures would be required in the scheduler to manage the blocking and queuing of the actors. By applying `condEvt` the queuing machinery is hidden in the run-time system internals. For example, advancing the clock directly wakes up all actors waiting for the next time step. Thus the dependency between the clock reference variable and the suspended actors is captured internally in the RTS.

FCS is ideal for this kind of application where there are several queues of blocked threads and a queue is flushed when a condition is established.

The final example in Figures 2.1.6 and 2.1.7 uses `condEvt` to synchronize cars against a set of traffic lights. Here broadcast semantics are appropriate since the event of a light changing to green should be broadcast to all waiting cars. In addition the property of not missing transient changes in the store is critical since it is not desirable for cars to miss signal changes! This is in contrast to the barrier and discrete event simulation examples where the synchronization conditions are monotonic.

In the code, a traffic light consists of three components, each represented as a record field in the `light` type constructor. The status field maintains the green/red status of the light while the `delayEvt` field is used to time the delay between signal changes. Thus each traffic light may have a different delay. Then for each traffic light a thread is spawned to execute function `manageLight` which toggles the status field of a light repeatedly while inserting delays by synchronizing on the `delayEvt` of the light. Four cars are created by spawning threads to execute functions `route1`

```
local
  val clock = new 0
  val timeSleepCh = channel()
  val eventSleepCh = channel()
  val timeAwakeCh = channel()
  val eventAwakeCh = channel()
in
  fun timeWait t =
    if get clock >= t then ()
    else (
      send(timeSleepCh,t);
      wrap(condEvt(fn()=> get clock = t),
           fn()=> send(timeAwakeCh,())))

  fun eventWait e =
    if e() then ()
    else (
      send(eventSleepCh,());
      wrap(condEvt e, fn()=> send(eventAwakeCh,())))

  fun scheduler ... (* see next Figure *)
end
```

FIGURE 2.1.4. A discrete event simulator

and `route2` which synchronize on the traffic lights in different orders to represent different traffic routes.

We also envision that `condEvt` would be useful for other kinds of applications. Imagine an animated object in motion which sets off multiple actions when it's position satisfies a constraint based on the object's position. This kind of synchronization could be programming easily with `condEvt` by creating a thread for each suspended action which is synchronizing on the following event:

```
condEvt(fn () => get x < 10 and get y < 20)
```

```
fun scheduler (cycles, numActors) = let
  val awakeActors = ref numActors
  val startTimeCh = channel()
  val finishTimeCh = channel()
  fun trackAll () = (
    select [wrap(recvEvt timeSleepCh,
                 fn (t)=> (set(awakeActors,get awakeActors - 1);
                           incTimeStep t)),
            wrap(recvEvt eventSleepCh,
                 fn ()=> set(awakeActors,get awakeActors - 1)),
            wrap(recvEvt eventAwakeCh,
                 fn () => set(awakeActors,get awakeActors + 1)),
            wrap(recvEvt startTimeCh,
                 fn()=>
                   let
                     val wakeUps = getTimeStep(get clock)
                     fun loop () =
                       if (get awakeActors) = wakeUps then ()
                       else (
                         recv timeAwakeCh;
                         set(awakeActors,get awakeActors + 1);
                         loop())
                   in
                     loop();
                     send(finishTimeCh,())
                   end)];
     trackAll())
  fun run cycles = if cycles = 0 then ()
                   else (
                     set(clock,get clock + 1);
                     send(startTimeCh,());
                     recv finishTimeCh;
                     sync(condEvt(fn () => get awakeActors = 0));
                     run(cycles-1))
in
  spawn trackAll;
  sync(condEvt(fn () => get awakeActors = 0));
  run cycles
end
```

FIGURE 2.1.5. Scheduler for discrete event simulator

```
datatype light_status = GREEN | RED

datatype light = LIGHT of {status    : light_status ref,
                           delayEvt  : unit event,
                           signalEvt : unit event}

fun makeLight (s,d) =
    LIGHT{status     = statusFlag,
          delayEvt   = timeOutEvt TIME{sec=d,usec=0},
          signalEvt  = condEvt(fn () => get statusFlag = GREEN)
   }

fun manageLight (LIGHT{status,delayEvt,...},n) =
  let
    fun loop () = (
      set(status,OFF);
      sync(delayEvt);
      set(status,ON);
      loop()
    )
  in
    loop n
  end

 fun getSignalEvt (LIGHT{signalEvt,...}) = signalEvt
```

FIGURE 2.1.6. Definitions for a traffic light control system

Again, the property of not missing transient changes in the store is key in case the object moves in and then back out of the region which satisfies the synchronization condition.

```
let
  val light1 = makeLight(new RED,3)
  val light2 = makeLight(new RED,4)
  val light3 = makeLight(new RED,5)

  fun route1 () = (
    sync (getSignalEvt light1);
    sync (getSignalEvt light3);
    sync (getSignalEvt light2);
    route1()
  )

  fun route2 () = (
    sync (getSignalEvt light1);
    sync (getSignalEvt light2);
    sync (getSignalEvt light3);
    route1()
  )

in
  spawn(fn () => manageLight light1);
  spawn(fn () => manageLight light2);
  spawn(fn () => manageLight light3);

  spawn route1;   (* create cars *)
  spawn route1;
  spawn route2;
  spawn route2
end
```

FIGURE 2.1.7. Body for a traffic light control system

CHAPTER 3

Source Language

To express the semantics of first-class conditional synchronization (FCS) we define λ^s_{cv}, which extends the PML subset of λ_{cv} [**Rep92**] with support for FCS and imperative features. The dynamic semantics consist of sequential $\longmapsto$ and concurrent $\Longrightarrow$ evaluation relations and an event matching relation $\leadsto$ that describes when rendezvous is possible.

3.1. Notation

The following notation is used throughout. A *finite* map is a map with finite domain. For sets A and B, the set of finite maps from A to B is denoted $A \overset{\text{fin}}{\to} B$. The *domain* and *range* of finite map f are denoted $\text{Dom}(f)$ and $\text{Range}(f)$ respectively. For finite maps f and g, $f \pm g$ is the finite map whose domain is $\text{Dom}(f) \cup \text{Dom}(g)$ and whose value is $g(x)$ when $x \in \text{Dom}(g)$ and $f(x)$ otherwise.

3.2. Dynamic semantics of source

From λ_{cv}, the ground terms are variables, base constants, function constants, and channel names. The set $FConst$ includes the following event-valued combinators and constructors: `choose`, `recvEvt`, `sendEvt`, and `wrap`. We add to this list `condEvt` and the store operations for synchronization references: assignment, dereference, and allocation (`set`,`get`, and `new`). The style of the dynamic semantics is based on Felleisen and Friedman's notion of an evaluation context [**FF86**] which uses

$$
\begin{array}{lcll}
e & ::= & v & \text{value} \\
  & \mid & e_1\, e_2 & \text{application} \\
  & \mid & (e_1, e_2) & \text{pair} \\
  & \mid & \texttt{let}\, x = e_1 \,\texttt{in}\, e_2 & \text{let} \\
  &&& \\
v & ::= & c & \text{constant} \\
  & \mid & x & \text{variable} \\
  & \mid & l & \text{locations} \\
  & \mid & (v_1, v_2) & \text{pair value} \\
  & \mid & \lambda x.e & \text{abstraction} \\
  & \mid & \kappa & \text{channel name} \\
  & \mid & ev & \text{event value} \\
  &&& \\
ev & ::= & \kappa!v & \text{channel output} \\
  & \mid & \kappa? & \text{channel input} \\
  & \mid & ev \Rightarrow v & \text{wrapper} \\
  & \mid & ev_1 \oplus ev_2 & \text{choice} \\
  & \mid & \beta_e & \text{conditional}
\end{array}
$$

FIGURE 3.2.1. Source language grammar

term-rewriting and small-step reductions. This method has been shown to simplify type soundness proofs and provides a framework amenable to language extensions [**WF92**]. The source language grammar is shown in Figure 3.2.1.

In addition to the syntactic class of expressions, $e \in \text{Exp}$, and values, $v \in \text{Val}$, there is a syntactic class of *event values*, $ev \in \text{Event} \subset \text{Val}$. In addition store locations $l \in \text{Loc} \subset \text{Val}$ are also values. We extend the syntactic class of event values with a value β_e that represents an event value whose synchronization is performed by evaluating the boolean expression e.

3.2.1. Sequential evaluation. A *sequential configuration* consists of a *store* θ and *evaluation context* E . The store θ is a finite map from locations to values:

$$\theta \in \text{Loc} \overset{\text{fin}}{\to} \text{Val}$$

The evaluation context E is an expression with one subexpression replaced by a hole, denoted $[]$. The expression $E[e]$ is formed by placing the expression e in the hole of E. The evaluation context ensures that expressions are evaluated in a leftmost-outermost fashion via the grammar:

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

The *free variables* of an expression e (denoted $\text{FV}(e)$) are defined inductively:

$$\begin{aligned}
\text{FV}(x) &= \{x\} \\
\text{FV}(e_1\ e_2) &= \text{FV}(e_1) \cup \text{FV}(e_2) \\
\text{FV}(\texttt{let}\, x = e_1\, \texttt{in}\, e_2) &= \text{FV}(e_1) \cup (\text{FV}(e_2) \backslash \{x\}) \\
\text{FV}(\lambda x.e) &= \text{FV}(e) \backslash \{x\}
\end{aligned}$$

A expression e is *closed* if $\text{FV}(e) = \emptyset$.

Definition 3.2.1. A sequential configuration θ, e is *well-formed* if $\text{FV}(e) \subseteq \text{Dom}(\theta)$ The sequential semantics are based upon the following reductions:

$$\begin{array}{llll}
\theta \vdash E[c\ v] & \longmapsto E[\delta(c, v)] & \text{if } \delta(c, v) \text{ is defined} & \text{(function constant)} \\
\theta \vdash E[(\lambda x.e)\ v] & \longmapsto E[e[x \mapsto v]] & & (\beta - \text{reduction}) \\
\theta \vdash E[\texttt{let}\ x = v \text{ in e}] & \longmapsto E[e[x \mapsto v]] & & \text{(let)}
\end{array}$$

where the δ function is used to define the meaning of the built-in function constants. Application and `let` expressions are reduced by substituting v for free occurrences of x in e. Renaming is also used to avoid capture in the store operations. The `get` operation dereferences x returning the value v in the store binding:

$$\theta \pm \{x \mapsto v\} \vdash E[\texttt{get}\ x] \longmapsto E[v] \quad \text{(get)}$$

The other store operations (`new` and `set`) are defined as concurrent reductions and hence sequential evaluation has no side effects with respect to the store θ. This guarantees that evaluating synchronization conditions is *pure,* since the transitive reflexive closure $\overset{*}{\longmapsto}$ is used.

In this semantic framework the partial function δ abstracts the set of constants:

$$\delta : \ \text{Const} \times \text{ClosedVal} \rightharpoonup \text{ClosedVal}$$

and the $\delta-$meanings of the event constructors and combinators are:

$$\begin{aligned}
\delta(\texttt{sendEvt},\ (\kappa, v)) &= \kappa!v \\
\delta(\texttt{recvEvt},\ \kappa) &= \kappa? \\
\delta(\texttt{wrap},\ (ev, v))) &= ev \Rightarrow v \\
\delta(\texttt{choose},\ (ev_1, ev_2)) &= (ev_1 \oplus ev_2) \\
\delta(\texttt{condEvt},\ \lambda x.e) &= \beta_e
\end{aligned}$$

3.2.2. Concurrent evaluation . As in semantics of λ_{cv}, concurrent evaluation is defined as a transition system between finite sets of process states. This is based on the style of the “Chemical Abstract Machine” [**BB90**]. The concurrent evaluation relation $\Longrightarrow$ extends $\longmapsto$ to processes, and adds additional rules for process creation, channel creation and communication.

$$\begin{array}{lll} \pi & \in \text{ProcID} & \\ p = \langle \pi; e\rangle & \in \text{Process} & = \langle \text{ProcID} \times \text{Exp}\rangle \\ \mathcal{P} & \in \text{ProcID} \overset{\text{fin}}{\to} & \text{Exp} \end{array}$$

The $\overset{*}{\Longrightarrow}$ relation is the transitive reflexive closure of $\Longrightarrow$. In addition since the store is used for synchronization we introduce a global store as part of concurrent configurations. The store must be global since synchronization references may be communicated across channels. Processes may then contain references to the store.

Definition 3.2.2. (**Concurrent Configuration** - adapted from [**Rep92**]) If θ is a finite map from locations to values, $\mathcal{P}_v \cup \mathcal{P}_r$ is a finite set of process states, and $\mathcal{K}$ a finite set of channel names then $\theta, \mathcal{K}, \mathcal{P}_v, \mathcal{P}_r$ is a *concurrent configuration.* The processes in $\mathcal{P}_r$ are *ready* while processes in $\mathcal{P}_v$ are attempting to rendezvous.

We use $\mathcal{P}$ to refer to $\mathcal{P}_v \cup \mathcal{P}_r$ when it is not necessary to distinguish between the ready and rendezvous processes.

Definition 3.2.3. A concurrent configuration $\theta, \mathcal{K}, \mathcal{P}$ is *well-formed* if $\mathcal{P}$ is well-formed with respect to the store θ and channel set $\mathcal{K}$. A process set P is well-formed with respect to store θ if for all $\langle \pi\, ;\, e\rangle \in P$ it is true that $\text{FV}(e) \subseteq \text{Dom}(\theta)$. Also a process set P is well-formed with respect to channel set $\mathcal{K}$ if for all $\langle \pi\, ;\, e\rangle \in P$ it is true that $\text{FCN}(e) \subseteq \text{Dom}(\mathcal{K})$ where $\text{FCN}(e)$ denotes the set of free channel names in e.

The rule 3.2.1 extends the sequential evaluation relation to concurrent configurations.

(3.2.1)
$$\frac{\theta, e \longmapsto \theta, e'}{\theta, \mathcal{K}, \mathcal{P}_v, \mathcal{P}_r + \langle \pi\, ;\, e \rangle \Longrightarrow \theta, \mathcal{K}, \mathcal{P}_v, \mathcal{P}_r + \langle \pi\, ;\, e' \rangle}$$

The rule for `new` (3.2.2) creates a new store binding which associates the variable x with the value v.

(3.2.2)
$$\theta, \mathcal{K}, \mathcal{P}_v, \mathcal{P}_r + \langle \pi\, ;\, E[\texttt{new}\ v] \rangle \Longrightarrow \theta + \{x \mapsto v\}, \mathcal{K}, \mathcal{P}_v, \mathcal{P}_r + \langle \pi\, ;\, E[x] \rangle \quad x \text{ fresh}$$

Figure 3.2.2 identifies the reductions that are applied to evaluate a sample expression.

The rule for dynamic channel creation consists of picking a new channel name κ and substituting κ for x in the body expression e .

(3.2.3)
$$\frac{\kappa \notin \mathcal{K}}{\theta, \mathcal{K}, \mathcal{P}_v, \mathcal{P}_r + \langle \pi\, ;\, E[\texttt{chan}\ x\ \texttt{in}\ e] \rangle \Longrightarrow \theta, \mathcal{K} + \kappa, \mathcal{P} + \langle \pi\, ;\, E[e[x \mapsto \kappa]] \rangle}$$

Dynamic process creation consists off picking a new process id π' which then executes the result of applying the spawn argument $\lambda x.e$ to the `unit` value.

(3.2.4)
$$\frac{\pi' \notin \mathrm{Dom}(\mathcal{P}) + \{\pi\}}{\theta, \mathcal{K}, \mathcal{P}_v, \mathcal{P}_r + \langle \pi\, ;\, E[\texttt{spawn}\ \lambda x.e] \rangle \Longrightarrow \theta, \mathcal{K}, \mathcal{P}_v, \mathcal{P}_r + \langle \pi\, ;\, E[()] \rangle + \langle \pi'\, ;\, \lambda x.e\ () \rangle}$$

A rendezvous is based upon the revised notion of *event matching* in [**Rep99**]. Informally k events $ev_1, ..., ev_k$ match if the events satisfy the conditions for a k-way

$$\begin{array}{ll}
\{\}, \texttt{let}\ x = \texttt{new}\,4\ \texttt{in}\ \texttt{let}\ y = \texttt{new}\,5\ \texttt{in}\ (\lambda z.\texttt{get}(x) + \texttt{get}(y))(\texttt{set}(x, 10)) & \\
\{a \mapsto 4\}, \texttt{let}\ x = a\ \texttt{in}\ \texttt{let}\, y = \texttt{new}\,5\ \texttt{in}\ (\lambda z.\texttt{get}(x) + \texttt{get}(y))(\texttt{set}(x, 10)) & \text{new} \\
\{a \mapsto 4\}, \texttt{let}\ y = \texttt{new}\,5\ \texttt{in}\ (\lambda z.\texttt{get}(a) + \texttt{get}(y))(\texttt{set}(a, 10)) & \text{let} \\
\{a \mapsto 4, b \mapsto 5\}, \texttt{let}\ y = b\ \texttt{in}\ (\lambda z.\texttt{get}(a) + \texttt{get}(y))(\texttt{set}(a, 10)) & \text{new} \\
\{a \mapsto 4, b \mapsto 5\}, (\lambda z.\texttt{get}(a) + \texttt{get}(b))(\texttt{set}(a, 10)) & \text{let} \\
\{a \mapsto 10, b \mapsto 5\}, \texttt{get}(a) + \texttt{get}(b) & \text{set}, \beta \\
\{a \mapsto 10, b \mapsto 5\}, 10 + 5 & \text{get}
\end{array}$$

FIGURE 3.2.2. Example applying store reductions

rendezvous:

$$\theta \vdash (ev_1, ..., ev_k) \rightsquigarrow_k (e_1, ..., e_k)$$

The above may be read as given the store θ, the events $ev_1, ..., ev_k$ satisfy the conditions for a k-way rendezvous and the processes synchronizing on events $ev_1, ..., ev_k$ will continue after the rendezvous by evaluating expressions $e_1, ..., e_k$ respectively. For instance, channel communication may occur when two processes are attempting to send and receive on the same channel κ:

$$\theta \vdash (\kappa?,\ \kappa!v)\ \rightsquigarrow_2\ (v, ())$$

Matching may also involve events constructed by `wrap` and `choose`:

$$\frac{\theta \vdash (ev_1, ..., ev_i, ...ev_k)\ \rightsquigarrow_k\ (e_1, ...e_i, ..., e_k)\ \ i \in \{1..k\}}{\theta \vdash (ev_1, ...,\ ev_i \Rightarrow v, ...ev_k)\ \rightsquigarrow_k\ (e_1, ..., e_i\ v, ...e_k)}$$

$$\frac{\theta \vdash (ev_1, ...ev_i, ..., ev_k,) \ \leadsto_k \ (e_1, ...e_i, ..., e_k) \ \ i \in \{1..k\}}{\theta \vdash (ev_1, ..., ev_i \oplus ev', ...ev_k) \ \leadsto_k \ (e_1, ...e_i, ..., e_k)}$$

$$\frac{\theta \vdash (ev_1, ...ev_i, ..., ev_k,) \ \leadsto_k \ (e_1, ...e_i, ..., e_k) \ \ i \in \{1..k\}}{\theta \vdash (ev_1, ..., ev' \oplus ev_i, ...ev_k) \ \leadsto_k \ (e_1, ...e_i, ..., e_k)}$$

The following rule indicates that the order of events in the matching is not significant.

$$\frac{\theta \vdash (ev_1, ...ev_{i-1}, ev_i, ..., ev_k) \ \leadsto_k \ (e_1, ...e_{i-1}, e_i, ..., e_k) \ \text{ for } i \in \{2..k\}}{\theta \vdash (ev_1, ...ev_i, ev_{i-1}, ..., ev_k) \ \leadsto_k \ (e_1, ...e_i, e_{i-1}, ..., e_k)}$$

In the first example below the value 20 is sent across the channel. In the second example a wrapper function is applied to the received value.

$$\theta \vdash (\kappa?, \ \kappa!20) \ \leadsto_2 \ (20, ())$$

$$\theta \vdash (\kappa? \Rightarrow \lambda x.x + 1, \ \kappa!20) \ \leadsto_2 \ (\lambda x.x + 1 \ 20, ())$$

An event value may be in the form of the non-deterministic choice $\oplus$ and therefore event matching is non-deterministic. Hence both of the following matches are valid:

$$\theta \vdash (\kappa!10 \oplus \kappa!20, \ \kappa?) \ \leadsto_2 \ ((), 10)$$

$$\theta \vdash (\kappa!10 \oplus \kappa!20, \ \kappa?) \ \leadsto_2 \ ((), 20)$$

Synchronization conditions have no side effects, that is the store bindings, channel and process sets are unaffected. As will be seen in Section 3.3, these restrictions are statically enforced by the type system. Synchronizing on a conditional event is a 1-way rendezvous where matching succeeds if the boolean expression e evaluates to

$\texttt{true}$ with respect to θ.

$$\frac{\theta \vdash e \overset{*}{\longmapsto} \texttt{true}}{\theta \vdash \beta_e \ \rightsquigarrow_1 \ ()} \tag{3.2.5}$$

Definition 3.2.4. **(Synchronization Object [Rep99])** The synchronization objects of an event value are defined as:

$$\begin{array}{rcl}
\text{SyncObj}(\kappa!v) & = & \{\kappa\} \\
\text{SyncObj}(\kappa?) & = & \{\kappa\} \\
\text{SyncObj}(\beta_e) & = & \beta_e \\
\text{SyncObj}(ev_1 \oplus ev_2) & = & \text{SyncObj}(ev_1) \cup \text{SyncObj}(ev_2) \\
\text{SyncObj}(ev \Rightarrow v) & = & \text{SyncObj}(ev)
\end{array}$$

Definition 3.2.5. **(Enabled [Rep99]**) A synchronization object ψ is enabled in configuration $\theta, \mathcal{K}, \mathcal{P}_v, \mathcal{P}_r$ if there is a process $\langle \pi_1 \,;\, E_1[\texttt{sync}\, ev_1] \rangle \in \mathcal{P}_v$ with $\psi \in \text{SyncObj}(ev_1)$, and there are also $k-1$ processes in the configuration such that:

$$\theta \vdash (ev_1, ..., ev_k) \rightsquigarrow_k (e_1, ..., e_k)$$

and $\langle \pi_i \,;\, E_i[\texttt{sync}\, ev_i] \rangle \in \mathcal{P}_v$ for $1 \leq i \leq k$.

Processes synchronizing on matching events may simultaneously reduce `sync` expressions to the corresponding expression specified on the right-hand side of the matching relation.

(3.2.6)
$$\frac{\theta \vdash (ev_1, ..., ev_k) \rightsquigarrow_k (e_1, ..., e_k)}{\begin{array}{l}\theta, \mathcal{K}, \mathcal{P}_v + \langle \pi_1 \,;\, E_1[\texttt{sync}\ ev_1] \rangle + ... + \langle \pi_k \,;\, E_k[\texttt{sync}\ ev_k] \rangle, \mathcal{P}_r \Longrightarrow \\ \theta, \mathcal{K}, \mathcal{P}_v, \mathcal{P}_r + \langle \pi_1 \,;\, E_1[e_1] \rangle + ... + \langle \pi_k \,;\, E_k[e_k] \rangle\end{array}}$$

Ready processes synchronizing on conditional events that are enabled may proceed without being moved into $\mathcal{P}_v$.

(3.2.7)
$$\frac{\theta \vdash ev \rightsquigarrow_1 ()}{\theta, \mathcal{K}, \mathcal{P}_v, \mathcal{P}_r + \langle \pi \,;\, E[\texttt{sync}\ ev] \rangle \Longrightarrow \theta, \mathcal{K}, \mathcal{P}_v, \mathcal{P}_r + \langle \pi \,;\, E[()] \rangle}$$

In 3.2.8, if all the synchronization conditions of an event value evaluate to `false` in θ, then processes evaluating `a sync` expression are moved from $\mathcal{P}_r$ to $\mathcal{P}_v$ by the following rule to indicate arrival at a rendezvous point.

(3.2.8)
$$\frac{\theta \vdash e \overset{*}{\longmapsto} \texttt{false} \text{ and } \beta_e \in \text{SyncObj}(ev)}{\theta, \mathcal{K}, \mathcal{P}_v, \mathcal{P}_r + \langle \pi \,;\, E[\texttt{sync}\ ev] \rangle \Longrightarrow \theta, \mathcal{K}, \mathcal{P}_v + \langle \pi \,;\, E[\texttt{sync}\ ev] \rangle, \mathcal{P}_r}$$

After the first attempt, subsequent evaluations of synchronization conditions are triggered by store updates. The subscript i is used since the update may unblock multiple processes synchronizing on different conditional events whose conditions are `true` in the resultant store.

(3.2.9)
$$\frac{\begin{array}{l} M = \{\langle \pi \,;\, E[\texttt{sync}\ ev] \rangle \mid \langle \pi \,;\, E[\texttt{sync}\ ev] \rangle \in \mathcal{P}_v \text{ and } \theta \pm \{x \mapsto v_2\} \vdash ev \rightsquigarrow_1 ()\} \\ N = \{\langle \pi \,;\, E[()] \rangle \mid \langle \pi \,;\, E[\texttt{sync}\ ev] \rangle \in N\}\end{array}}{\begin{array}{l}\theta \pm \{x \mapsto v_1\}, \mathcal{K}, \mathcal{P}_v, \mathcal{P}_r + \langle \pi' \,;\, E[\texttt{set}\ (x, v_2)] \rangle \Longrightarrow \\ \theta \pm \{x \mapsto v_2\}, \mathcal{K}, \mathcal{P}_v - N, \mathcal{P}_r + N + \langle \pi' \,;\, E[()] \rangle\end{array}}$$

The $\Longrightarrow$ relation has the following properties.

LEMMA 3.2.6. *(Adapted from* [**Rep92**]*) If* $\theta, \mathcal{K}, \mathcal{P}$ *is well-formed and* $\theta, \mathcal{K}, \mathcal{P} \Longrightarrow \theta', \mathcal{K}', \mathcal{P}'$ *then the following hold:*

1. $\theta', \mathcal{K}', \mathcal{P}'$ is well-formed
2. $\mathrm{Dom}(\theta) \subseteq \mathrm{Dom}(\theta')$
3. $\mathcal{K} \subseteq \mathcal{K}'$
4. $\mathrm{Dom}(\mathcal{P}) \subseteq \mathrm{Dom}(\mathcal{P}')$

PROOF. By examination of the rules for $\Longrightarrow$. □

The following definitions are used in the theorem that guarantees transient store configurations are not missed by waiting processes.

DEFINITION 3.2.7. **(Trace -** adapted from [**Rep92**]**)** A *trace* is a (possibly infinite) sequence of well-formed configurations

$$T = \langle\langle \theta_0, \mathcal{K}_0, \mathcal{P}_0;\ \theta_1, \mathcal{K}_1, \mathcal{P}_1; ...\rangle\rangle$$

such that $\theta_i, \mathcal{K}_i, \mathcal{P}_i \Longrightarrow \theta_{i+1}, \mathcal{K}_{i+1}, \mathcal{P}_{i+1}$, for all $i, i+1$ in the sequence. The *head* of T is $\theta_0, \mathcal{K}_0, \mathcal{P}_0$.

DEFINITION 3.2.8. **(Blocked)** A process π is blocked in configuration $\theta, \mathcal{K}, \mathcal{P}_v, \mathcal{P}_r$ if $\langle \pi\ ;\ E[\mathtt{sync}\, ev]\rangle \in \mathcal{P}_v$ and there does not exist $k-1$ other processes in the configuration such that:

$$\theta \vdash (ev_1, ..., ev_{k-1}, ev) \ \rightsquigarrow\ _k(e_1, ..., e_{k-1}, e)$$

The semantics of conditional synchronization guarantees that no transient store configurations are missed. In other words a blocked process waiting for a condition to become `true` is guaranteed to become ready before the condition subsequently becomes `false`.

Theorem 3.2.9. ***(Transient Property)*** *If* $\langle \pi\,;\, E[\texttt{sync}\, ev]\rangle \in \mathcal{P}_{v_x}$ *of configurations* $c_x \in T$ *(*$x \in \{i..j-1\}$*) and for all* $\beta_e \in \mathrm{SyncObj}(ev)$

$$\theta_x \vdash e \overset{*}{\longmapsto} \texttt{false}$$

and subsequently in configuration $c_j \in T$ *there exists a* $\beta_e \in \mathrm{SyncObj}(ev)$ *such that*

$$\theta_j \vdash e \overset{*}{\longmapsto} \texttt{true}$$

then $\langle \pi\,;\, E[()]\rangle \in \mathcal{P}_{r_j}$.

Proof. Since the store changed during the transition $c_{j-1} \Longrightarrow c_j$ reduction 3.2.9 must have been applied and this forces re-evaluation of *all* pending synchronization conditions e. Using

$$\theta_j \vdash e \overset{*}{\longmapsto} \texttt{true}$$

as a premise to 3.2.5 we get

$$\theta_j \vdash \beta_e \rightsquigarrow_1 \; ()$$

and therefore by 3.2.9

$$\langle \pi\,;\, E[()]\rangle \in \mathcal{P}_{r_j}$$

□

```
val Y = 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
```

FIGURE 3.2.3. Coding Y_v using processes and channels

3.2.3. Recursion. We provide no built-in mechanism for recursion since λ_{cv} has none due to the fact that the call-by-value Y_v combinator can be implemented using processes and channels. The CML example in Figure 3.2.3 by Reppy was adopted from [**GMP89**] and uses only what is provided by λ_{cv}. The unrolling normally associated with the Y_v combinator is done by sending a copy of the function across a channel. In Figure 3.2.3 a copy of the function `g` is sent across channel `a` for the next iteration. When discussing recursive functions we write:

$$\texttt{letrec}\ f(x) = e_1\ \texttt{in}\ \texttt{e}_2$$

as syntactic sugar for

$$\texttt{let}\ f = Y_v(\lambda f.\lambda x.e_1)\ \texttt{in}\ e_2$$

3.3. Static semantics of the source language

In order to enforce certain restrictions within synchronization conditions we incorporate *effects* into the source static semantics. Effects come out of research done at the MIT Laboratory for Computer Science in the late 1980's [**Luc87**] [**JG91**] [**LG91**]. One product of this research was the The FX language [**GJLS87**] which applies its effect system to perform parallel code generation and stack allocation of temporary data structures. Talpin and Jouvelot subsequently designed an effect type system for the core of ML [**TJ92**] . They then applied their framework to make the type generalization associated with let polymorphism more precise than previous methods [**TJ97**]. Communication effects have also been used to optimize CML programs [**NN94**]. This dissertation can be viewed as an application of effects to conditional synchronization. The source semantics applies effects in order to enforce restrictions and in addition the target semantics apply effects to preclude excessive re-evaluation of synchronization conditions.

Before presenting the static semantics of the target language we first give an overview of effects. An effect φ is defined as:

$$\begin{array}{rcl} \varphi & ::= & \emptyset \mid \text{init} \mid \text{read} \mid \text{write} \mid \epsilon \mid \varphi \cup \varphi \\ & & \text{channel} \mid \text{block} \mid \text{spawn} \end{array}$$

Null effects are represented by $\emptyset$. The effect init corresponds to the initialization of a synchronization reference. *Read* and *write* effects to synchronization references are denoted by read and write respectively. Effect variables are represented by ϵ. Discussion of the remaining effects (channel, block, and spawn) is deferred to Section 3.3.1. The source types $\tau \in$ Type are shown in Figure 3.3.1. Types τ can be either `unit`, type variable α, synchronization reference value of type τ in the region

ρ, and function type $\tau_1 \overset{\varphi}{\to} \tau_2$. A salient point is that the effects φ of a function body are captured in the function type. This allows the inference of effects incurred by function application. The event type $\tau\,\texttt{event}_\varphi$ has an associated effect φ since synchronizing on an event value may evaluate a synchronization condition or the body of a wrapper function.

A substitution S is a pair of type and effect substitutions. Type substitutions map type variables to types where S_t ranges over all type substitutions. Effect substitutions map effect variables to effects where S_e ranges over all effect substitutions. S then ranges over all substitutions. The type variables α are used for type generalization (polymorphism):

$$\alpha \in \text{TyVar}$$

We use $\text{FTV}(\tau)$ to denote the free type variables of τ. Type schemes $\sigma \in \text{TyScheme}$ [**DM82**] may have quantified type variables:

$$\begin{array}{rcl} \sigma & ::= & \tau \\ & | & \forall \alpha_1, ..., \alpha_n.\tau \end{array}$$

The variables $\alpha_1, ..., \alpha_n$ are *bound* in σ and are denoted by $\text{BV}(\sigma)$. Substitutions map type variables to types where S ranges over all substitutions. A type τ' is an *instance* of a type scheme $\sigma ::= \forall \alpha_1, ..., \alpha_n.\tau$, denoted $\sigma \succ \tau'$, if there exists a substitution S where $\text{Dom}(S) = \text{bv}(\sigma)$ and $S(\sigma) = \tau'$ with a renaming of the bound variables when necessary to avoid capture. Type environments $TE \in \text{TyEnv}$ are a

$$
\begin{array}{rcll}
\tau & ::= & \texttt{unit} & \\
& | & \texttt{bool} & \\
& | & \alpha & \text{type variables} \\
& | & \tau_1 \xrightarrow{\varphi} \tau_2 & \text{function types} \\
& | & \tau_1 \times \tau_2 & \text{pair types} \\
& | & \tau\ \texttt{chan} & \text{channel types} \\
& | & \tau\ \texttt{event}_\varphi & \text{event types} \\
& | & \tau\ \texttt{sync\_ref} & \text{reference types}
\end{array}
$$

FIGURE 3.3.1. Source language types

triple of finite maps: a *variable typing*, *channel typing*, and *store typing*:

$$
\begin{array}{lll}
VT & \in \text{VarTy} & = \text{Var} \overset{\text{fin}}{\to} \text{TyScheme} \\
CT & \in \text{ChanTy} & = \text{Ch} \overset{\text{fin}}{\to} \text{Type} \\
ST & \in \text{StoreTy} & = \text{Loc} \overset{\text{fin}}{\to} \text{Type} \\
TE = (VT, CT, ST) & \in \text{TyEnv} & = (\text{VarTy} \times \text{ChanTy} \times \text{StoreTy})
\end{array}
$$

We use FTV(VT) and FTV(CT) for the set of free type variables of variable and channel typings respectively. The free type variables of a type environment defined as:

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

There are no bound variables in a channel typing or store typing.

In ML, polymorphism is associated with the `let` construct by allowing different type instantiations of let-bound variables within the scope of the let body expression. However, it is well known that generalizing type variables that appear in the types of stored values results in a unsound type system [**Tof90**]. Hence a variety of mechanisms have been used to restrict generalization including effects [**TJ97**]

[**Wri92**]. More recently the trend has been to use the simple approach of only generalizing the type when the right-hand side of a let-binding is a value (*value restriction*). Although more conservative than other methods the value restriction is easier for a programmer to understand as to why generalization succeeds or fails. The reason for this restriction is that when the binding expression is a value it is safe to generalize the type since the domain of the store will not be expanded by evaluating the right-hand side. Generalizing the type of an expansive right-hand side expression in a let-binding would lead to an unsound type system as illustrated by the example:

```
let
  val f = new (fn x => x)
in
  set(f, fn x => x + 1);
  (get f) true
end
```

Assigning `f` the type $\forall\alpha.\,\alpha \rightarrow \alpha$ allows the program to pass through the type-checker. However, a run-time type error will occur when executing the expression (`(get f) true`) since the type of the value stored at `f's` location is `int` $\xrightarrow{\varphi}$ `int`.

Following the approach of Tofte and Talpin [**TT97**], we incorporate the value restriction in by associating polymorphism only with the `letrec` construct as shown in Figure 3.3.2. Thus the `letrec` construct is used for both recursion and polymorphism.

$$\frac{VT(x) \succ \tau}{(VT, CT, ST) \vdash x \,:\, \tau, \emptyset}$$

$$\frac{CT(\kappa) = \tau}{(VT, CT, ST) \vdash \kappa \,:\, \tau, \emptyset}$$

$$\frac{ST(l) = \tau}{(VT, CT, ST) \vdash l \,:\, \tau, \emptyset}$$

$$\frac{\text{TypeOf}(c) \succ \tau}{TE \vdash c \,:\, \tau, \emptyset}$$

$$\frac{TE \pm \{x \mapsto \tau\} \vdash e : \tau', \varphi}{TE \pm \{x \mapsto \tau\} \vdash \lambda x.e : \tau \xrightarrow{\varphi} \tau', \emptyset}$$

$$\frac{TE \vdash e : \tau \xrightarrow{\varphi} \tau', \varphi' \quad TE \vdash e' : \tau, \varphi''}{TE \vdash e\ e' : \tau', \varphi \cup \varphi' \cup \varphi''}$$

$$\frac{TE \vdash e_1 : \tau_1, \varphi_1 \quad TE \pm \{x \mapsto \tau_1\} \vdash e_2 : \tau_2, \varphi_2}{TE \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2, \varphi_1 \cup \varphi_2}$$

$$\frac{\begin{array}{ll} TE \pm \{f \mapsto \tau_1\} \vdash \lambda x.e_1 : \tau_1, \emptyset & \{\alpha_1, ..., \alpha_n\} \cap \text{FTV}(TE) = \emptyset \\ TE \pm \{f \mapsto \forall \alpha_1, ..., \alpha_n.\tau_1\} \vdash e_2 : \tau_2, \varphi_2 & \end{array}}{TE \vdash \texttt{letrec}\ f(x) = e_1\ \texttt{in}\ e : \tau_2, \varphi_2}$$

FIGURE 3.3.2. Core type rules

The type rules for event values are shown in Figure 3.3.3. The specifications of the store operations and the specifications of the event constructors/combinators are in Figures 3.3.4 and 3.3.5.

3.3.1. Pure and impure effects. The semantics of FCS illustrates that the synchronization condition is evaluated an arbitrary number of times. Since the motivation for FCS is to allow a thread to synchronize with the store being in a certain state, it would be undesirable to allow synchronization conditions which have

$$\frac{TE \vdash \kappa : \tau\,\texttt{chan}, \emptyset \quad TE \vdash v : \tau, \emptyset}{TE \vdash \kappa!v : \texttt{unit}\,\texttt{event}, \emptyset}$$

$$\frac{TE \vdash \kappa : \tau\,\texttt{chan}, \emptyset}{TE \vdash \kappa? : \tau\,\texttt{event}, \emptyset}$$

$$\frac{TE \vdash ev : \tau\,\texttt{event}_{\varphi}, \emptyset \quad TE \vdash \lambda x.e : \tau \xrightarrow{\varphi'} \tau', \emptyset}{TE \vdash ev \Rightarrow \lambda x.e : \tau', \texttt{unit}\,\texttt{event}_{\varphi \cup \varphi'}, \emptyset}$$

$$\frac{TE \vdash ev_1 : \tau\,\texttt{event}_{\varphi}, \emptyset \quad TE \vdash ev_2 : \tau\,\texttt{event}_{\varphi'}, \emptyset}{TE \vdash ev_1 \oplus ev_2 : \tau\,\texttt{event}_{\varphi \cup \varphi'}, \emptyset}$$

$$\frac{TE \vdash e : \texttt{bool}, \varphi^{\text{pure}}}{TE \vdash \beta_e : \tau, \texttt{unit}\,\texttt{event}_{\varphi^{\text{pure}}}, \emptyset}$$

Figure 3.3.3. Type rules for event values

$$TE \vdash \texttt{new} : \forall \alpha.\alpha \xrightarrow{\text{init}} \alpha\ \texttt{sync\_ref}, \emptyset$$

$$TE \vdash \texttt{get} : \forall \alpha.\alpha\ \texttt{sync\_ref} \xrightarrow{\text{read}} \alpha, \emptyset$$

$$TE \vdash \texttt{set} : \forall \alpha.\alpha\ \texttt{sync\_ref} \xrightarrow{\emptyset} \alpha \xrightarrow{\text{write}} \texttt{unit}, \emptyset$$

Figure 3.3.4. Type specifications of store operations

$$TE \vdash \texttt{wrap} : \forall \epsilon\epsilon'\alpha\beta.\alpha\,\texttt{event}_{\epsilon} \xrightarrow{\emptyset} (\alpha \xrightarrow{\epsilon'} \beta) \xrightarrow{\emptyset} \beta\,\texttt{event}_{\epsilon \cup \epsilon'}, \emptyset$$

$$TE \vdash \texttt{sendEvt} : \forall \alpha\,\texttt{chan} \xrightarrow{\emptyset} \alpha \xrightarrow{\emptyset} \texttt{unit}\,\texttt{event}_{\emptyset}, \emptyset$$

$$TE \vdash \texttt{recvEvt} : \forall \alpha.\alpha\,\texttt{chan} \xrightarrow{\emptyset} \alpha\,\texttt{event}_{\emptyset}, \emptyset$$

$$TE \vdash \texttt{choose} : \forall \epsilon\epsilon'\alpha.\alpha\,\texttt{event}_{\epsilon} \times \alpha\,\texttt{event}_{\epsilon'} \xrightarrow{\emptyset} \alpha\,\texttt{event}_{\epsilon \cup \epsilon'}, \emptyset$$

Figure 3.3.5. Type specifications of event constructors and combinators

$$\texttt{sync} : \forall \epsilon \alpha . \alpha\, \texttt{event}_\epsilon \overset{\epsilon \cup \text{block}}{\rightarrow} \alpha$$

$$\texttt{spawn} : \forall \epsilon (\texttt{unit} \overset{\epsilon}{\rightarrow} \texttt{unit}) \overset{\epsilon \cup \text{spawn}}{\rightarrow} \texttt{thread\_id}$$

$$\texttt{channel} : \forall \alpha . \texttt{unit} \overset{\text{channel}}{\rightarrow} \alpha\, \texttt{chan}$$

FIGURE 3.3.6. Operations having effects

effects. To allow otherwise would permit the re-evaluation of the synchronization condition to unblock the thread attempting synchronization. In effect a thread could unblock itself with its own condition as illustrated by the code fragment:

```
condEvt(fn () => (set(x,get x + 1); get x > 4)
```

Clearly the semantics of such programs is nonsensical and dependant on process scheduling. In addition, blocking operations should be prevented during evaluation of the synchronization condition. To allow otherwise would result in ambiguous semantics. To enforce these restrictions we identify effects as being either pure or impure effects as in [**BF96**]. Di Blasio and Fisher define an impure effect as containing a write effect for the purpose of disallowing write effects in their synchronization guards. We take this definition a step further by incorporating *block* effects and any other effect which generates a new concurrent configuration in the dynamic semantics. Thus in addition to read, write, and init effects, we now introduce *channel, spawn,* and *block* effects.

DEFINITION 3.3.1. ***(Pure Effect)*** An effect is *pure* (denoted φ^{pure}) if the effect does not contain any of the following effects: *block, write, init, channel*, or *spawn.* Hence a pure effect φ^{pure} may only contain *read* effects.

Otherwise the effect is *impure*. In addition to `set` and `new`, impure effects are generated by the functions in Figure 4.2.5. Thus the desired restrictions are enforced by the type system as the type of `condEvt` is implicitly quantified over all pure effects.

$$\text{(3.3.1)} \qquad \frac{TE \vdash e : \texttt{unit} \overset{\varphi^{\text{pure}}}{\to} \texttt{bool}, \varphi}{TE \vdash \texttt{condEvt}\ e : \texttt{event}_{\varphi^{\text{pure}}}, \varphi}$$

CHAPTER 4

Target Language

The translation scheme described in Chapter 5 generates a target program which precludes continuous polling of synchronization conditions. To support our translation scheme, we incorporate into the target language (λ^t_{cv}) *effects* (as in the source) and in addition *regions* [**TJ92**] [**TJ97**]. Next in Section 4.1 we define the dynamic semantics of our target language λ^t_{cv} followed by the static semantics in Section 4.2.

4.1. Dynamic semantics of target

The salient distinction of the target language is the introduction of *region names.* Region names are run-time representations of regions and are values in the target as shown in Figure 4.1.1. Regions names are used to help distinguish store locations and are necessary due to aliasing. Hence region names are used to uniquely identify the memory locations associated with synchronization references. Thus every synchronization reference value is assigned its own region name at run-time. Other techniques such as abstract interpretation [**CC77**] have been applied to perform alias analysis. However, Talpin [**TJ92**] cites several advantages in the context of high-order functional programs in using the regions of type and effect inference over other techniques

As in the source language, the rules of the dynamic semantics are categorized as either sequential $\longmapsto$ or concurrent $\Longrightarrow$, and as before, the matching relation $\leadsto$ defines when a rendezvous is possible.

e	$::=$	v	value
	$\mid$	$e_1\, e_2$	application
	$\mid$	(e_1, e_2)	pair
	$\mid$	$\texttt{let}\, x = e_1\, \texttt{in}\, e_2$	let
	$\mid$	$\texttt{nil}$	empty list
	$\mid$	$\texttt{cons}(e_1, e_2)$	cons
	$\mid$	$\texttt{case}\ e_0\ \texttt{of}\ \texttt{nil}\ \rightarrow\ e_1\ \texttt{OR}\ \texttt{cons}(v_1, v_2)\ \rightarrow\ e_2$	case
v	$::=$	c	constant
	$\mid$	x	variable
	$\mid$	(v_1, v_2)	pair value
	$\mid$	$\lambda x.e$	abstraction
	$\mid$	κ	channel name
	$\mid$	ev	event value
	$\mid$	r	region names
	$\mid$	$\texttt{nil}$	empty list value
	$\mid$	$\texttt{cons}(v_1, v_2)$	list value
ev	$::=$	$\kappa!v$	channel output
	$\mid$	$\kappa?$	channel input
	$\mid$	$ev \Rightarrow v$	wrapper
	$\mid$	$ev_1 \oplus ev_2$	choice
	$\mid$	$\beta_{e\,[r_1, r_2, \ldots]}$	conditional

FIGURE 4.1.1. Target language grammar

4.1.1. Sequential evaluation. The store θ is as in the source semantics:

$$\theta \in \mathrm{Loc} \overset{\mathrm{fin}}{\rightarrow} \mathrm{Va}$$

While evaluation contexts E are extended for lists:

$$\begin{aligned} E \ ::= \ & [\,] \mid E\ e \mid v\ E \mid \texttt{let}\, x = E\ \texttt{in}\ e \mid (E, e) \mid (v, E) \mid \\ & \texttt{case}\ E\ \texttt{of}\ \texttt{nil}\ \rightarrow\ e_1\ \texttt{OR}\ \texttt{cons}(v_1, v_2)\ \rightarrow\ e_2 \end{aligned}$$

as are the sequential reductions:

$$
\begin{array}{lll}
\theta \vdash E[c\ v] & \longmapsto E[\delta(c, v)] & \text{(function constant)} \\
\theta \vdash E[(\lambda x.e)\ v] & \longmapsto E[e[x \mapsto v]] & (\beta - \text{reduction}) \\
\theta \vdash E[\mathtt{let}\ x = v \text{ in e}] & \longmapsto E[e[x \mapsto v]] & \text{(let)} \\
\theta \pm \{x \mapsto v\} \vdash E[\mathtt{get}\ x] & \longmapsto E[v] & \text{(get)} \\
\theta \vdash \mathtt{case\ nil\ of\ nil}\ \ \rightarrow\ \ e_1 & \longmapsto E[e_1] & \\
\qquad \mathtt{OR}\ \mathtt{cons}(x_1, x_2)\ \ \rightarrow\ \ e_2 & & \\
\theta \vdash \mathtt{case}\ \mathtt{cons}(v_1, v_2)\ \mathtt{of\ nil}\ \ \rightarrow\ \ e_1 & \longmapsto E[e_2[x_1 \mapsto v_1, x_2 \mapsto v_2]] & \\
\qquad \mathtt{OR}\ \mathtt{cons}(x_1, x_2)\ \ \rightarrow\ \ e_2 & &
\end{array}
$$

The list constructors `cons` and `nil` are utilized in the annotations introduced by the translation. Moreover, we write `[a,b,c]` as syntactic sugar for `cons(a,cons(b,cons(c,nil)))`.

The event constructor $\mathtt{condEvt}'$ creates a conditional event from a pair consisting of a lambda and a region name list. The event value returned $\beta_{e\,[r_1,...,r_n]}$ encapsulates not only the boolean expression e but also the region names $r_1, ..., r_n$ corresponding to the regions read by e. The region names $r_1, ..., r_n$ are automatically passed to $\mathtt{condEvt}'$ by the translation described in Chapter 5.

$$\delta(\mathtt{condEvt}',\ (\lambda x.e, [r_1, ..., r_n])) = \beta_{e\,[r_1,...,r_n]}$$

The function `newRegName` generates a new region name on each call. The translation inserts calls to `newRegName` in the target at points corresponding to the allocation of a synchronization reference value.

$$\delta(\mathtt{newRegName},\ ()) = r_i \quad i \in \{1..n\}$$

4.1.2. Concurrent evaluation. Concurrent configurations are as in the source semantics. The definition for the synchronization objects of and event value is modified to take into account the target form of a conditional event:

$$\mathrm{SyncObj}(\beta_{e\,[r_1,\ldots,r_n]}) \quad = \quad \beta_{e\,[r_1,\ldots,r_n]}$$

Also, the two rules below for extending sequential evaluation and `new` are identical to the source semantics.

$$\frac{\theta, e \longmapsto \theta, e'}{\theta, \mathcal{K}, \mathcal{P}_v, \mathcal{P}_r + \langle \pi\,;\, e\rangle \Longrightarrow \theta, \mathcal{K}, \mathcal{P}_v, \mathcal{P}_r + \langle \pi\,;\, e'\rangle} \tag{4.1.1}$$

(4.1.2)

$$\theta, \mathcal{K}, \mathcal{P}_v, \mathcal{P}_r + \langle \pi\,;\, E[\texttt{new}\ v]\rangle \Longrightarrow \theta + \{x \mapsto v\}, \mathcal{K}, \mathcal{P}_v, \mathcal{P}_r + \langle \pi\,;\, E[x]\rangle \quad x \text{ fresh}$$

As before, processes synchronizing on matching events may simultaneously reduce their `sync` expressions to the corresponding expression specified by the matching relation.

$$\frac{\theta \vdash (ev_1, \ldots, ev_k) \rightsquigarrow_k (e_1, \ldots, e_k)}{\begin{array}{l}\theta, \mathcal{K}, \mathcal{P}_v + \langle \pi_1\,;\, E_1[\texttt{sync}\ ev_1]\rangle + \ldots + \langle \pi_k\,;\, E_k[\texttt{sync}\ ev_k]\rangle, \mathcal{P}_r \Longrightarrow \\ \theta, \mathcal{K}, \mathcal{P}_v, \mathcal{P}_r + \langle \pi_1\,;\, E_1[e_1]\rangle + \ldots + \langle \pi_k\,;\, E_k[e_k]\rangle\end{array}} \tag{4.1.3}$$

Rule 4.1.4 illustrates that the initial attempt at evaluating the synchronization condition allows a process to synchronize on an event without being moved into the rendezvous set $\mathcal{P}_v$.

$$\frac{\theta \vdash ev \rightsquigarrow_1 ()}{\theta, \mathcal{K}, \mathcal{P}_v, \mathcal{P}_r + \langle \pi\,;\, E[\texttt{sync}\ ev]\rangle \Longrightarrow \theta, \mathcal{K}, \mathcal{P}_v, \mathcal{P}_r + \langle \pi\,;\, E[()]\rangle} \tag{4.1.4}$$

In 4.1.5, if all the synchronization conditions of an event value evaluate to `false` in θ then the corresponding processes are moved to the rendezvous set.

$$(4.1.5) \qquad \frac{\theta \vdash e \overset{*}{\longmapsto} \texttt{false} \;\text{ for all } \beta_{e\,[r_1,..,r_n]} \in \text{SyncObj}(ev_i)}{\theta, \mathcal{K}, \mathcal{P}_v, \mathcal{P}_r + \langle \pi_i\,;\, E[\texttt{sync}\ ev_i] \rangle \Longrightarrow \theta, \mathcal{K}, \mathcal{P}_v + \langle \pi_i\,;\, E[\texttt{sync}\ ev_i] \rangle, \mathcal{P}_r}$$

The rule for re-evaluating synchronization conditions refers to the following definition for the *Regions* of an event value.

Definition 4.1.1. **(Regions)** The Regions of an event value ev is the union of the regions in the conditional events:

$$\text{Regions}(ev) \;=\; \bigcup_{\beta_{e\,[r_1,\ldots,r_n]} \in \text{SyncObj}(ev)} \{r_1, \ldots, r_n\}$$

After the first attempt, subsequent evaluations of synchronization conditions are triggered by updates to regions r which intersect with the Regions of an event value. The subscript i is used in Rule 4.1.6, since the update may unblock multiple processes synchronizing on different conditional events reading th same region r.

$$\begin{aligned} \text{M} = \{\langle \pi\,;\, E[\texttt{sync}\ ev] \rangle \mid\ & \langle \pi\,;\, E[\texttt{sync}\ ev] \rangle \in \mathcal{P}_v \text{ and } r \in \text{Regions}(ev) \\ & \text{and } \theta \pm \{x \mapsto v_2\} \vdash ev \leadsto_1 ()\} \end{aligned}$$

$$(4.1.6) \qquad \frac{N = \{\langle \pi\,;\, E[()] \rangle \mid \langle \pi\,;\, E[\texttt{sync}\ ev] \rangle \in M\}}{\begin{aligned} & \theta \pm \{x \mapsto v_1\}, \mathcal{K}, \mathcal{P}_v, \mathcal{P}_r + \langle \pi\,;\, E[\texttt{set}'\ (x, v_2, r)] \rangle \;\Longrightarrow \\ & \theta \pm \{x \mapsto v_2\}, \mathcal{K}, \mathcal{P}_v - M, \mathcal{P}_r + N + \langle \pi\,;\, E[()] \rangle \end{aligned}}$$

where $\leadsto_1$ is defined as in the source as:

$$(4.1.7) \qquad \frac{\theta \vdash e \overset{*}{\longmapsto} \texttt{true}}{\theta \vdash \beta_e \;\leadsto_1\; ()}$$

The target semantics also guarantees the transient property.

THEOREM 4.1.2. ***(Transient Property)*** *If* $\langle \pi\,;\, E[\texttt{sync}\, ev] \rangle \in \mathcal{P}_{v_x}$ *of configurations* $c_x \in T$ $(x \in \{i..j-1\})$ *and for all* $\beta_{e\,[r_1,\ldots,r_n]} \in \mathrm{SyncObj}(ev)$

$$\theta_x \vdash e \overset{*}{\longmapsto} \texttt{false}$$

and in configuration $c_j \in T$ *there exists a* $\beta_{e\,[r_1,\ldots,r_n]} \in \mathrm{SyncObj}(ev)$ *such that*

$$\theta_j \vdash e \overset{*}{\longmapsto} \texttt{true}$$

then $\langle \pi\,;\, E[()] \rangle \in \mathcal{P}_{r_j}$.

PROOF. Since the store changed during the transition $c_{j-1} \Longrightarrow c_j$ reduction 4.1.6 must have been applied and this forces re-evaluation of conditions e reading the region r being updated. Using

$$\theta_j \vdash e \overset{*}{\longmapsto} \texttt{true}$$

as a premise to 4.1.7 we get

$$\theta_j \vdash \beta_{e\,[r_1,\ldots,r_n]} \rightsquigarrow_1 \;()$$

and therefore by 4.1.6

$$\langle \pi\,;\, E[()] \rangle \in \mathcal{P}_{r_j}$$

$\square$

Finally, the rules for channel and process creation are as in the source.

(4.1.8)
$$\frac{\kappa \notin \mathcal{K}}{\theta, \mathcal{K}, \mathcal{P}_v, \mathcal{P}_r + \langle \pi \,;\, E[\texttt{chan}\ x\ \texttt{in}\ e] \rangle \Longrightarrow \theta, \mathcal{K} + \kappa, \mathcal{P} + \langle \pi \,;\, E[e[x \mapsto \kappa]] \rangle}$$

(4.1.9)
$$\frac{\pi' \notin \mathrm{Dom}(\mathcal{P}) + \{\pi\}}{\theta, \mathcal{K}, \mathcal{P}_v, \mathcal{P}_r + \langle \pi \,;\, E[\texttt{spawn}\ \lambda x.e] \rangle \Longrightarrow \theta, \mathcal{K}, \mathcal{P}_v, \mathcal{P}_r + \langle \pi \,;\, E[()] \rangle + \langle \pi' \,;\, \lambda x.e\ () \rangle}$$

4.2. Static semantics of the target language

The static semantics have three kinds of semantic objects: effects, types, and regions as in [**TJ92**].

$$\begin{array}{lcll} \rho & ::= & \omega \mid \varrho & \text{regions} \\ \varphi & ::= & \emptyset \mid \mathrm{init}(\rho) \mid \mathrm{read}(\rho) \mid \mathrm{write}(\rho) \mid \epsilon \mid \varphi \cup \varphi & \text{effects} \\ & & \mathrm{spawn} \mid \mathrm{block} \mid \mathrm{channel} & \end{array}$$

Regions can be either region constants ω or region variables ϱ. Regions statically abstract the memory used to hold values in the store. Every store value is assigned to a region upon initialization. Since more than one memory location may correspond to the same region, regions approximate memory locations. By unifying region variables, a static analysis can make conservative approximations when it is not possible to statically infer whether two values in the program correspond to the same memory location. Note that the regions in our semantics are specifically for synchronization. If SML references were present in the semantics then there would be two kinds of regions: one kind for synchronization and the other for the SML references. The distinction is necessary since the motivation for the translation is to inform the run-time system regarding updates to the regions read by synchronization conditions.

Effects correspond to the kind of effects present in the source semantics except that read, write, and init effects are associated with some region ρ. Null effects are represented by $\emptyset$. The effect init(ρ) corresponds to the initialization of a value in the region ρ. *Read* and *write* effects to the region ρ are denoted by read(ρ) and write(ρ) respectively. Types τ (Figure 4.2.1) can be either `unit`, type variables α, reference values of type τ in the region ρ, and function types $\tau_1 \xrightarrow{\varphi} \tau_2$. A salient point is that the effects of a function body are captured in the function type. This allows the inference of effects incurred by function application.

A substitution S is a triple (S_t, S_r, S_e) of type, region and effect substitutions. Type substitutions map type variables to types where S_t ranges over all type substitutions. Region substitutions map region variables to region variables where S_r ranges over all region substitutions. Effect substitutions map effect variables to effects where S_e ranges over all effect substitutions. S then ranges over all substitutions. The target type schemes are either *simple* or *compound*:

$$\begin{array}{rcl} \sigma & ::= & \tau \\ & | & \forall \rho_1, ..., \rho_i, \alpha_1, ..., \alpha_j, ..., \epsilon_1, ..., \epsilon_k . \underline{\tau} \end{array}$$

where $i \geq 0$, $j \geq 0$, $k \geq 0$. This method was devised by Tofte and Talpin [**TT97**] to distinguish the types of *region polymorphic functions* from regular functions. Compound type schemes are used solely for region polymorphic functions. The

$$
\begin{array}{rcll}
\tau & ::= & \texttt{unit} & \\
 & \mid & \texttt{bool} & \\
 & \mid & \alpha & \text{type variables} \\
 & \mid & \tau_1 \xrightarrow{\varphi} \tau_2 & \text{function types} \\
 & \mid & \tau_1 \times \tau_2 & \text{pair types} \\
 & \mid & \tau\,\texttt{chan} & \text{channel types} \\
 & \mid & \tau\,\texttt{event}_{\varphi} & \text{event types} \\
 & \mid & \tau\ \texttt{sync\_ref}_{\rho} & \text{reference types} \\
 & \mid & \tau\,\texttt{list} & \text{lists}
\end{array}
$$

FIGURE 4.2.1. Target language types

underlining $\underline{\tau}$ helps to identify the compound type schemes. For compound type schemes, the bound variables bv(σ) are the variables in the set:

$$\{\rho_1, ..., \rho_n, \alpha_1, ..., \alpha_n, ..., \epsilon_1, ..., \epsilon_n\}$$

The vector notation $(\vec{\alpha}, \vec{\rho}, \vec{\epsilon})$ is used to refer to the bound variables in a type scheme. As before, a type is an instance of a type scheme $\sigma \succ \tau$ if there is a substitution S where $\mathrm{Dom}(S) = \mathrm{bv}(\sigma)$ and $S(\sigma) = \tau$ with a renaming of bound variables when necessary to avoid capture.

Figure 4.2.1 shows the result of incorporating effects and regions into the types of λ^t_{cv}. In Figure 4.2.1 an effect φ is attached to the `event` type since synchronization may evaluate synchronization conditions or calls to wrapper functions. Thus the effects φ is incurred when applying `sync`.

In the remaining discussion of this Subsection there are no salient distinctions from the source semantics. The inference rules of the static semantics associate a type environment TE and an expression e with its type τ and effects φ, noted $TE \vdash e : \tau, \varphi$. As in the source semantics, type environments are actually a triple of variable typing VT, channel typing CT and store typing ST.

$$\begin{array}{llll} VT & \in & \text{VarTy} & = \text{Var} \xrightarrow{\text{fin}} \text{TyScheme} \\ CT & \in & \text{ChanTy} & = \text{Ch} \xrightarrow{\text{fin}} \text{Type} \\ ST & \in & \text{StoreTy} & = \text{Loc} \xrightarrow{\text{fin}} \text{Type} \\ TE = (VT, CT, ST) & \in & \text{TyEnv} & = (\text{VarTy} \times \text{ChanTy} \times \text{StoreTy}) \end{array}$$

The core type rules are shown in Figure 4.2.2 and the store operations are in Figure 4.2.4. The type rules for event values are shown in Figure 4.2.6. Next, in order to describe the static semantics of `condEvt`′ we distinguish effects φ as being either *pure* or *impure* (as in the source).

DEFINITION 4.2.1. ***(Pure Effect)*** An effect is *pure* (denoted φ^{pure}) if the effect does not contain any of the following effects: *block, write, init, channel*, or *spawn.* Hence a pure effect φ^{pure} may only contain *read* effects.

Otherwise the effect is *impure.* In addition to `set`′ and `new,` impure effects are generated by functions in Figure 4.2.5.

Source calls to `condEvt` are translated `to condEvt`′ whose argument function has a pure effect φ^{pure}. Thus the desired restrictions are enforced by the type system as the type of `condEvt`′ is implicitly quantified over all pure effects.

$$\text{(4.2.1)} \qquad \frac{TE \vdash e : \texttt{unit} \overset{\varphi^{\text{pure}}}{\rightarrow} \texttt{bool}, \varphi \quad TE \vdash [r_1, ..., r_n] : \texttt{reg\_name list}, \varphi}{TE \vdash \texttt{condEvt}' \ (e, \ [r_1, ..., r_n]) : \texttt{unit event}_{\varphi^{\text{pure}}}}$$

Note that if the standard SML reference variable were present in the semantics then there would be another kind of region distinct from our regions which are intended for synchronization. In this case the effects pertaining to SML references would also be disallowed within pure effects. The specifications of the other event

$$\frac{VT(x) \succ \tau}{(VT, CT, ST) \vdash x \,:\, \tau, \emptyset}$$

$$\frac{CT(\kappa) = \tau}{(VT, CT, ST) \vdash \kappa \,:\, \tau, \emptyset}$$

$$\frac{ST(l) = \tau}{(VT, CT, ST) \vdash x \,:\, \tau, \emptyset}$$

$$\frac{\text{TypeOf}(c) \succ \tau}{TE \vdash c \,:\, \tau, \emptyset}$$

$$\frac{TE \pm \{x \mapsto \tau\} \vdash e : \tau', \varphi}{TE \pm \{x \mapsto \tau\} \vdash \lambda x.e : \tau \xrightarrow{\varphi} \tau', \emptyset}$$

$$\frac{TE \vdash e : \tau \xrightarrow{\varphi} \tau', \varphi' \quad TE \vdash e' : \tau, \varphi''}{TE \vdash e\ e' : \tau', \varphi \cup \varphi' \cup \varphi''}$$

$$\frac{TE \vdash e_1 : \tau_1, \varphi_1 \quad TE + \{x \mapsto \tau_1\} \vdash e_2 : \tau_2, \varphi_2}{TE \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e : \tau_2, \varphi_1 \cup \varphi_2}$$

$$\frac{\begin{array}{ll} TE \pm \{f \mapsto \tau_1\} \vdash \lambda x.e_1 \,:\, \tau_1, \emptyset & \{\alpha_1, ..., \alpha_n\} \cap \text{FTV}(TE) = \emptyset \\ TE \pm \{f \mapsto \forall \alpha_1, ..., \alpha_n.\tau_1\} \vdash e_2 \,:\, \tau_2, \varphi_2 & \end{array}}{TE \vdash \texttt{letrec}\ f(x) = e_1\ \texttt{in}\ e : \tau_2, \varphi_2}$$

$$\frac{TE \pm \{x \mapsto \tau_1\} \vdash e : \tau_2, \varphi}{TE \vdash \texttt{chan}\ x\ \texttt{in}\ e : \tau_2, \varphi}$$

FIGURE 4.2.2. Core type rules for target

constructors and combinators are shown in Figure 4.2.6. Finally, the types of the list constructors are shown in Figure .

$$\frac{TE \vdash \kappa : \tau\,\texttt{chan}, \emptyset \quad TE \vdash v : \tau, \emptyset}{TE \vdash \kappa!v : \texttt{unit}\,\texttt{event}, \emptyset}$$

$$\frac{TE \vdash \kappa : \tau\,\texttt{chan}, \emptyset}{TE \vdash \kappa? : \tau\,\texttt{event}, \emptyset}$$

$$\frac{TE \vdash ev : \tau\,\texttt{event}_{\varphi}, \emptyset \quad TE \vdash \lambda x.e : \tau \xrightarrow{\varphi'} \tau', \emptyset}{TE \vdash ev \Rightarrow \lambda x.e : \tau', \texttt{unit}\,\texttt{event}_{\varphi \cup \varphi'}, \emptyset}$$

$$\frac{TE \vdash ev_1 : \tau\,\texttt{event}_{\varphi}, \emptyset \quad TE \vdash ev_2 : \tau\,\texttt{event}_{\varphi'}, \emptyset}{TE \vdash ev_1 \oplus ev_2 : \tau\,\texttt{event}_{\varphi \cup \varphi'}, \emptyset}$$

$$\frac{TE \vdash e : \texttt{bool}, \varphi^{\text{pure}} \quad TE \vdash [r_1, ..., r_n] : \texttt{reg\_name list}, \emptyset}{TE \vdash \beta_{e\,[r_1, ..., r_n]} : \tau, \texttt{unit}\,\texttt{event}_{\varphi^{\text{pure}}}, \emptyset}$$

Figure 4.2.3. Type rules for event values

$$TE \vdash \texttt{new} : \forall \alpha.\alpha \xrightarrow{\text{init}(\rho)} \texttt{sync\_ref}_{\rho}(\alpha), \emptyset$$

$$TE \vdash \texttt{get} : \forall \alpha.\texttt{sync\_ref}_{\rho}(\alpha) \xrightarrow{\text{read}(\rho)} \alpha, \emptyset$$

$$TE \vdash \texttt{set}' : \forall \alpha.\texttt{sync\_ref}_{\rho}(\alpha) \xrightarrow{\emptyset} \alpha \xrightarrow{\emptyset} \texttt{reg\_name} \xrightarrow{\text{write}(\rho)} \texttt{unit}, \emptyset$$

Figure 4.2.4. Type specifications of store operations

$$\texttt{sync} : \forall \epsilon \alpha.\alpha\,\texttt{event}_{\epsilon} \xrightarrow{\epsilon \cup \text{block}} \alpha$$

$$\texttt{spawn} : \forall \epsilon(\texttt{unit} \xrightarrow{\epsilon} \texttt{unit}) \xrightarrow{\epsilon \cup \text{spawn}} \texttt{thread\_id}$$

$$\texttt{channel} : \forall \alpha.\texttt{unit} \xrightarrow{\text{channel}} \alpha\,\texttt{chan}$$

Figure 4.2.5. Operations having effects

$$TE \vdash \texttt{wrap} : \forall\alpha\beta.\alpha\,\texttt{event}_{\varphi} \xrightarrow{\varphi'} (\alpha \xrightarrow{\varphi''} \beta) \xrightarrow{\varphi'''} \beta\,\texttt{event}_{\varphi\cup\varphi''}, \emptyset$$

$$TE \vdash \texttt{sendEvt} : \forall\alpha\,\texttt{chan} \xrightarrow{\varphi} \alpha \xrightarrow{\varphi'} \texttt{unit}\,\texttt{event}_{\emptyset}, \emptyset$$

$$TE \vdash \texttt{recvEvt} : \forall\alpha.\alpha\,\texttt{chan} \xrightarrow{\varphi} \alpha\,\texttt{event}_{\emptyset}, \emptyset$$

$$TE \vdash \texttt{choose} : \forall\alpha.\alpha\,\texttt{event}_{\varphi} \times \alpha\,\texttt{event}_{\varphi'} \xrightarrow{\varphi''} \alpha\,\texttt{event}_{\varphi\cup\varphi'}, \emptyset$$

FIGURE 4.2.6. Specifications of event constructors and combinators

$$TE \vdash \texttt{nil} : \forall\alpha.\alpha\ \texttt{list}, \emptyset$$

$$TE \vdash \texttt{cons} : \forall\alpha.\alpha\ \times \alpha\ \texttt{list} \xrightarrow{\emptyset} \alpha\ \texttt{list}, \emptyset$$

FIGURE 4.2.7. Specifications of list constructors

4.3. Type soundness

The main result presented at the end of this Section is that well-typed λ_{cv}^{t} programs do not cause run-time type errors. We follow the approach of Wright and Felleisen in proving syntactic soundness for our semantics. This approach involves:

- proving subject reduction holds (i.e, that evaluation preserves types)
- characterizing answers and stuck expressions
- proving stuck expressions are untypable

The following sequence of lemmas and definitions are used in the subject reduction theorems.

LEMMA 4.3.1. ***(Replacement Lemma -* [HS86]*)*** *If:*

1. D is a deduction concluding $TE \vdash C[e_1]\ : \tau, \sigma$

2. D_1 is a subdeduction of D concluding $TE' \vdash e_1 : \tau', \sigma'$

3. D_1 occurs in D in the position corresponding to the hole ([]) in C, and

4. $TE' \vdash e_2 : \tau', \sigma'$

then $TE \vdash C[e_2] : \tau, \sigma$

The basic idea is to view type deductions as trees. In this case the tree for deduction D contains a sub-tree for D_1. Let D_2 be the tree for the deduction $TE' \vdash e_2 : \tau', \sigma'$. Then if we cut out D_1 and replace it with D_2 and also substitute occurrences of e_1 for e_2 then the resulting tree still satisfies deduction D. The detailed proof in [**HS86**] uses induction on the height of the tree.

LEMMA 4.3.2. ***(Substitution*** [**WF92**]*) If* $TE \pm \{x \mapsto \forall\alpha_1 ... \alpha_n.\tau\} \vdash e : \tau', \varphi$ *and* $x \notin \mathrm{Dom}(TE)$ *and* $TE \vdash v : \tau$ *and* $\{\alpha_1 ... \alpha_n\} \cap \mathrm{FTV}(TE) = \emptyset$ *then* $TE \vdash e[x \mapsto v] : \tau', \varphi$.

See [**WF92**] for the detailed proof.

Subject reduction illustrates that evaluation preserves types. We now demonstrate subject reduction for both sequential and concurrent configurations in the target language λ^t_{cv}.

DEFINITION 4.3.3. **(Well-formed Store)** A store θ is well-formed with respect to store typing ST (denoted $ST \vdash \theta : ST$), if for all $l \in \mathrm{Dom}(\theta)$ it holds that $ST \vdash \theta(l) : ST(l), \emptyset$.

DEFINITION 4.3.4. (**Well-formed Sequential Configuration**) A well-formed sequential configuration θ, e has type τ and effect φ under store typing ST denoted:

$$ST \vdash \theta, e : \tau, \varphi$$

if the following hold:

- $\text{Dom}(\theta) \subseteq \text{Dom}(ST)$
- $(\{\}, \{\}, ST) \vdash e : \tau, \varphi$
- $ST \vdash \theta : ST$

THEOREM 4.3.5. ***(Sequential Subject Reduction)*** *If θ, e is a well-formed sequential configuration where and $\theta, e_1 \longmapsto \theta, e_2$ and for store typing ST if $ST \vdash \theta, e : \tau, \varphi$, then $ST \vdash \theta, e' : \tau, \varphi$.*

PROOF. If $e_1 = E[e]$ and $e_2 = E[e']$ then by the Replacement Lemma it suffices to show that if $TE \vdash e : \tau', \varphi'$ then $TE \vdash e' : \tau', \varphi'$. The proof is a case analysis of the syntactic structure of e_1 and $\longmapsto$.

$\boxed{\text{Case } \theta, E[c\ v] \longmapsto \theta, E[\delta(c, v)]}$

By the typing of application (Figure 4.2.2) there exists a type environment TE such that

$$TE \vdash c\ v : \tau', \varphi'$$

Hence by the $\delta-$typability we get

$$TE \vdash \delta(c, v) : \tau', \varphi'$$

$\boxed{\text{Case } \theta, E[(\lambda x.e)\ v] \longmapsto \theta, E[e[x \mapsto v]]}$

By the typing of application (Figure 4.2.2) there exists a type environment TE such that

$$TE \vdash (\lambda x.e)\ v : \tau', \varphi'$$

and from the premises (Figure 4.2.2)

$$TE \vdash \lambda x.e : \tau \xrightarrow{\varphi''} \tau', \varphi'$$
$$TE \pm \{x \mapsto \tau\} \vdash e : \tau', \varphi''$$

Hence by the Substitution Lemma 4.3.2

$$TE \vdash e[x \mapsto v] : \tau', \varphi'$$

$\boxed{\text{Case } \theta, E[\texttt{let}\ x = v \text{ in e}] \longmapsto \theta, E[e[x \mapsto v]]}$

By the type of `let` (Figure 4.2.2) there exists a type environment TE such that

$$TE \vdash \texttt{let}\ x = v \text{ in } e : \tau', \varphi'$$

and from the premises

$$TE \vdash v : \tau'', \emptyset$$
$$TE + \{x \mapsto \text{Closure}(\tau'', TE) \vdash e : \tau', \varphi'$$

Hence by the Substitution Lemma 4.3.2

$$TE \vdash e[x \mapsto v] : \tau', \varphi'$$

$\boxed{\text{Case } \theta \pm \{x \mapsto v\}, E[\texttt{get}\ x] \longmapsto \theta \pm \{x \mapsto v\}, E[v]}$

By the type of `get` (Figure 4.2.4) and application (Figure 4.2.2) there exists a type environment TE such that

$$TE \vdash \texttt{get}\ x : \tau', \varphi'$$

and from the premises

$$TE \vdash x : \tau'\ \texttt{sync\_ref}_\rho, \emptyset$$

Hence by well-formed configurations 4.3.4

$$TE \vdash v : \tau', \emptyset$$

$\square$

Next we introduce process typings which allow the extension of typing judgments to concurrent configurations. A *process typing* is a finite map from process identifiers to types

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

In addition, the following definitions and lemmas are used in the concurrent subject reduction theorem.

Definition 4.3.6. (**Well-formed Concurrent Configuration** - adapted from [**Rep92**]) A well-formed concurrent configuration $\theta, \mathcal{K}, \mathcal{P}$ has type PT under channel typing CT and store typing ST denoted:

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

if the following hold:

- $\mathcal{K} \subseteq \mathrm{Dom}(CT)$
- $\mathrm{Dom}(\mathcal{P}) \subseteq \mathrm{Dom}(PT)$
- $\mathrm{Dom}(\theta) \subseteq \mathrm{Dom}(ST)$
- for every $\langle \pi\,;\, e\rangle \in \mathcal{P}$, $(\{\}, CT, ST) \vdash e : PT(\pi), \varphi$

Due to the fact that the `spawn` function requires an argument of type $\texttt{unit} \xrightarrow{\varphi} \texttt{unit}$ all CML processes have type $PT(\pi) = \texttt{unit}$ for all $\pi \in \mathrm{Dom}(\mathcal{P})$.

LEMMA 4.3.7. ***(Matching - adapted from* [Rep92]*)*** *If $\theta \vdash (ev_1, ..., ev_k) \rightsquigarrow_k (e_1, ..., e_k)$ and $TE \vdash ev_i : \tau_i\, \texttt{event}_{\varphi_i}, \emptyset$ ($i \in \{1..k\}$), then $TE \vdash e_i : \tau_i, \varphi_i$.*

The proof is given at the end of the Chapter.

LEMMA 4.3.8. **[Rep92]** *If $x \notin \mathrm{FV}(e)$, then $TE \vdash e : \tau, \varphi$ iff $TE \pm \{x \mapsto \sigma\} \vdash e : \tau, \varphi$. Similarly if $\kappa \notin \mathrm{FCN}(e)$ then $TE \vdash e : \tau, \varphi$ iff $TE \pm \{\kappa \mapsto \tau'\} \vdash e : \tau, \varphi$.*

PROOF. The proof is a simple induction on the height of the typing deduction.

□

THEOREM 4.3.9. ***(Concurrent Subject Reduction - adapted from* [Rep92]*)*** *If $\theta, \mathcal{K}, \mathcal{P}$ is a well-formed configuration where $\theta, \mathcal{K}, \mathcal{P} \Longrightarrow \theta', \mathcal{K}', \mathcal{P}'$ and for channel, store and process typings CT, ST, PT*

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

then there exists $CT,' ST', PT'$ *such that:*

- $CT \subseteq CT'$
- $PT \subseteq PT'$
- $ST \subseteq ST'$
- $CT',\, ST' \vdash \theta', \mathcal{K}', \mathcal{P}' : PT'$
- $CT',\, ST' \vdash \theta, \mathcal{K}, \mathcal{P} : PT'$

PROOF. The fifth property follows from the first four. The proof of the first four properties is a case analysis of the left hand side of the $\Longrightarrow$ relation.

$\boxed{\text{Case } \theta, \mathcal{K}, \mathcal{P}_v, \mathcal{P}_r + \langle \pi \,;\, E[\texttt{new } v] \rangle \Longrightarrow \theta + \{x \mapsto v\}, \mathcal{K}, \mathcal{P}_v, \mathcal{P}_r + \langle \pi \,;\, E[x] \rangle \quad x \text{ fresh}}$

By the typing of application application (Figure 4.2.2) and the type of `new` (Figure 4.2.4) there is a type environment TE such that

$$TE \vdash \texttt{new } v : \tau \ \texttt{sync\_ref}_\rho, \mathrm{init}(\rho)$$
$$(\{\}, CT, ST) \vdash E[\texttt{new } v] : PT(\pi), \varphi$$

and from the premises

$$TE \vdash v : \tau, \emptyset$$

Letting $ST' = ST \pm \{x \mapsto \tau \ \texttt{sync\_ref}_\rho\}$ by Lemma 4.3.8

$$(\{\}, \{\}, ST') \vdash v : \tau, \emptyset$$

Applying the Replacement Lemma to E we get

$$(\{\}, CT, ST') \vdash E[x] \,:\, PT(\pi), \varphi$$

and therefore

$$CT, ST' \vdash \theta \pm \{x \mapsto v\}, \mathcal{K}, \mathcal{P} + \langle \pi \,;\, E[x] \rangle \,:\, PT$$

letting $CT' = CT$ and $PT' = PT$ satisfies the theorem

$\boxed{\text{Case } \theta, \mathcal{K}, \mathcal{P}_v + \langle \pi_i \,;\, E_i[\texttt{sync}\ ev_i] \rangle, \mathcal{P}_r \Longrightarrow \theta, \mathcal{K}, \mathcal{P}_v, \mathcal{P}_r + \langle \pi_i \,;\, E_i[e_i] \rangle \;\; i \in \{1..k\}}$

By the typing of application (Figure 4.2.2) and the type of `sync` (Figure 4.2.5)

there is a type environment TE such that

$$\begin{aligned} &TE \vdash \texttt{sync}\ ev_i \,:\, \tau_i, \varphi_i \\ &(\{\}, CT, ST) \vdash E_i[\texttt{sync}\ ev_i] \,:\, PT(\pi_i), \varphi_i'' \end{aligned}$$

and from the premises

$$TE \vdash ev_i \,:\, \tau_i\, \texttt{event}_{\varphi_i}, \emptyset$$

From the premise of 4.1.3

$$\theta \vdash (ev_1, ..., ev_k) \leadsto_k (e_1, ..., e_k)$$

and hence by the Matching Lemma 4.3.7 we get

$$TE \vdash e_i \,:\, \tau_i, \varphi_i$$

Applying the Replacement Lemma to E we get

$$(\{\}, CT, ST) \vdash E_i[e_i] \, : \, PT(\pi_i), \varphi_i''$$

and therefore

$$CT, ST \vdash \theta, \mathcal{K}, \mathcal{P}_v, \mathcal{P}_r + \langle \pi_i \, ; \, E_i[e_i] \rangle \, : \, PT$$

letting $CT' = CT$, $ST' = ST$, and $PT' = PT$ satisfies the theorem

$\boxed{\text{Case } \theta, \mathcal{K}, \mathcal{P}_v, \mathcal{P}_r + \langle \pi \, ; \, E[\texttt{sync}\ ev] \rangle \Longrightarrow \theta, \mathcal{K}, \mathcal{P}_v, \mathcal{P}_r + \langle \pi \, ; \, E[()] \rangle}$

By the typing of application (Figure 4.2.2) and `sync` (Figure 4.2.5) there is a type environment TE such that

$$TE \vdash \texttt{sync}\ ev : \tau, \varphi$$
$$(\{\}, CT, ST) \vdash E[\texttt{sync}\ ev] \, : \, PT(\pi), \varphi$$

and from the premises

$$TE \vdash ev : \tau \, \texttt{event}_{\varphi}, \emptyset$$

From the premise of 4.1.4

$$\theta \vdash ev \leadsto_1 ()$$

and hence by the Matching Lemma 4.3.7

$$\tau = \texttt{unit}$$

Applying the Replacement Lemma to E we get

$$(\{\}, CT, ST) \vdash E[()] : PT(\pi), \varphi$$

and therefore

$$CT, ST \vdash \theta, \mathcal{K}, \mathcal{P}_v, \mathcal{P}_r + \langle \pi ; E[()] \rangle : PT$$

letting $CT' = CT$, $ST' = ST$, and $PT' = PT$ satisfies the theorem

$$\boxed{\text{Case} \begin{array}{l} \theta \pm \{x \mapsto v_1\}, \mathcal{K}, \mathcal{P}_v + \langle \pi_i ; E_i[\texttt{sync}\ ev_i] \rangle, \mathcal{P}_r + \langle \pi ; E[\texttt{set}'\ (x, v_2, r)] \rangle \implies \\ \theta \pm \{x \mapsto v_2\}, \mathcal{K}, \mathcal{P}_v, \mathcal{P}_r + \langle \pi_i ; E_i[()] \rangle + \langle \pi ; E[()] \rangle \end{array}}$$

By the typing of application (Figure 4.2.2) and the type of `sync` (Figure 4.2.5) and `set′` (Figure 4.2.4) there is a type environment TE such that

$$TE \vdash \texttt{sync}\ ev_i : \tau_i, \varphi_i$$
$$TE \vdash \texttt{set}'(x, v_2, r) : \texttt{unit}, \mathrm{write}(\rho)$$

$$(\{\}, CT, ST) \vdash E_i[\texttt{sync}\ ev_i] : PT(\pi_i), \varphi_1$$
$$(\{\}, CT, ST) \vdash E[\texttt{set}'(x, v_2, r)] : PT(\pi), \varphi_2$$

and from the premises

$$TE \vdash ev_i : \tau_i\ \texttt{event}_{\varphi_i}, \emptyset$$

From the premise of 4.1.6

$$\theta \vdash ev_i \rightsquigarrow_1 ()$$

and hence by the Matching Lemma 4.3.7

$$\tau_i = \texttt{unit}$$

Applying the Replacement Lemma to E_i and E we get

$$(\{\}, CT, ST) \vdash E_i[()] \,:\, PT(\pi_i), \varphi_1$$
$$(\{\}, CT, ST) \vdash E[()] \,:\, PT(\pi), \varphi_2$$

and therefore

$$CT, ST \vdash \theta \pm \{x \mapsto v_2\}, \mathcal{K}, \mathcal{P}_v, \mathcal{P}_r + \langle \pi_i \,;\, E_i[()]\rangle + \langle \pi \,;\, E[()]\rangle : PT$$

letting $CT' = CT$, $ST' = ST$, and $PT' = PT$ satisfies the theorem

$\boxed{\text{Case } \theta, \mathcal{K}, \mathcal{P}_v, \mathcal{P}_r + \langle \pi \,;\, E[\texttt{chan}\ x\ \texttt{in}\ e]\rangle \Longrightarrow \theta, \mathcal{K} + \kappa, \mathcal{P} + \langle \pi \,;\, E[e[x \mapsto \kappa]]\rangle}$

By the typing of channel binding (Figure 4.2.2) there is a type environment TE such that

$$TE \vdash \texttt{chan}\ x\ \texttt{in}\ e : \tau, \varphi$$

$$(\{\}, CT, ST) \vdash E[\texttt{chan}\ x\ \texttt{in}\ e] : PT(\pi), \varphi'$$

Letting $CT' = CT + \{\kappa \mapsto \tau'\}$ (where $\kappa \notin \mathcal{K}$) and by Lemma 4.3.8 we get

$$(\{\}, CT', ST) \vdash E[\texttt{chan}\ x\ \texttt{in}\ e] : PT(\pi), \varphi'$$

Applying the Replacement and Substitution Lemmas we get

$$(\{\}, CT', ST) \vdash E[e[x \mapsto \kappa]] : PT(\pi), \varphi'$$

and therefore

$$CT', ST \vdash \theta, \mathcal{K} + \{x \mapsto \kappa\}, \mathcal{P}_v, \mathcal{P}_r + \langle \pi \,;\, E[e[x \mapsto \kappa]] \rangle : PT$$

letting $ST' = ST$ and $PT' = PT$ satisfies the theorem

$$\boxed{\text{Case } \theta, \mathcal{K}, \mathcal{P}_v, \mathcal{P}_r + \langle \pi \,;\, E[\texttt{spawn}\ \lambda x.e] \rangle \Longrightarrow \theta, \mathcal{K}, \mathcal{P}_v, \mathcal{P}_r + \langle \pi \,;\, E[()] \rangle + \langle \pi' \,;\, \lambda x.e\ () \rangle}$$

By the typing of application (Figure 4.2.2) there is a type environment TE such that

$$TE \vdash \texttt{spawn}\ \lambda x.e : \texttt{unit}, \varphi$$

$$(\{\}, CT, ST) \vdash E[\texttt{spawn}\ \lambda x.e] : PT(\pi), \varphi'$$

and from the premises

$$TE \vdash \lambda x.e : \texttt{unit} \overset{\varphi}{\to} \texttt{unit}, \emptyset$$

and therefore

$$TE \vdash \lambda x.e\ () : \texttt{unit}, \varphi$$

By letting $PT' = PT \pm \{\pi' \mapsto \texttt{unit}\}$ (where $\pi' \notin \mathcal{P}$) and by Lemma 4.3.8 we get

$$(\{\}, CT, ST) \vdash \lambda x.e\ () : PT'(\pi'), \varphi$$

Applying the Replacement Lemma to E we get

$$(\{\}, CT, ST) \vdash E[()] : PT(\pi), \varphi'$$

and therefore

$$CT, ST \vdash \theta, \mathcal{K}, \mathcal{P}_v, \mathcal{P}_r + \langle \pi \,;\, E[()] \rangle + \langle \pi' \,;\, \lambda x.e\ () \rangle : PT'$$

letting $CT' = CT$ and $ST' = ST$ and satisfies the theorem $\square$

With the subject reduction theorems in hand we can now continue towards the overall goal of establishing syntactic soundness.

DEFINITION 4.3.10. **(Stuck -** adapted from **[Rep92])** A process $\langle \pi \,;\, e \rangle$ is *stuck* if e is not a value and there do not exist well-formed configurations $\theta, \mathcal{K}, \mathcal{P}$ and $\theta', \mathcal{K}', \mathcal{P}'$ such that $\theta, \mathcal{K}, \mathcal{P} \Longrightarrow \theta', \mathcal{K}', \mathcal{P}'$ with π a selected process. A well-formed configuration is *stuck* if one or more of its processes are stuck.

In **[Rep92]** the expressions that cause a process to become stuck are:

- $E[b\ v]$, such that $\delta(b,\ v)$ is undefined
- $E[v\ v']$, where v is not of the form $\lambda x.e$
- $E[\texttt{sync}\ v]$, such that $v \notin \text{Event}$.

To this list we add the cases:

- $E[\texttt{condEvt}'\ v]$, where v is not of the form $(\lambda x.e, [r_1, ..., r_n])$
- $E[\texttt{condEvt}'\ (\lambda x.e, [r_1, ..., r_n])]$, where evaluating e updates or expands the store, creates processes or channels, or applies `sync` to an event

In addition, function constants c of type $\texttt{unit} \overset{\epsilon}{\rightarrow} \texttt{bool}$ are not allowed in order to reject programs that pass c to `condEvt'` in the position $\lambda x.e$ above.

LEMMA 4.3.11. *(**Uniform Evaluation** - adapted from* **[WF92]***) Let e be a program, $T \in \text{Comp}(e)$ and $\pi \in \text{Procs}(T)$, then either $\pi \Uparrow_T$, $\pi \Downarrow_T v$ or $\mathcal{P}_\rangle(\pi)$ is stuck for some $\theta_i, \mathcal{K}_i, \mathcal{P}_i$.*

PROOF. This follows immediately from the definitions □

Next we show that stuck configurations are untypable.

LEMMA 4.3.12. ***(Untypability of Stuck Configurations** - adapted from* [**WF92**]*) If π is stuck in a well-formed configuration $\theta, \mathcal{K}, \mathcal{P}$ then there do not exist $CT \in \text{ChanTy}$, $ST \in \text{StoreTy}$ and $PT \in \text{ProcTy}$ such that*

$$(\{\}, CT, ST) \vdash \mathcal{P}(\pi) : PT(\pi), \varphi$$

PROOF. The idea is to show by case analysis that stuck expressions are inconsistent with type inference rules of the static semantics. See [**Rep92**] for the cases involving the concurrency operations and [**WF92**] for the cases involving the store operations. We do the case for `condEvt'` by contradiction. We assume that the stuck expression is $\mathcal{P}(\pi) = E[\texttt{condEvt}'\ v]$ and the configuration is well-typed. Hence by the typing of `condEvt'` (4.2.1) and the application type rule (Figure 4.2.2)

$$TE \vdash \texttt{condEvt}'\ v : \texttt{unit}\,\texttt{event}_{\varphi^{\text{pure}}}, \emptyset$$

and from the premises we get

$$TE \vdash v : \texttt{unit} \overset{\varphi^{\text{pure}}}{\to} \texttt{bool} \times \texttt{reg\_name list}, \emptyset$$

contrary to the assumption that v is not of the form $(\lambda x.e, [r_1, ..., r_n])$ or that e is not pure. □

The key result now follows which guarantees that well-typed programs do not become stuck. That is, a process π in a well-formed configuration either diverges or converges to a value that is consistent with π 's process typing. The important implication is that well-typed programs cannot cause run-time type errors.

THEOREM 4.3.13. ***(Syntactic Soundness - adapted from*** **[Rep92]***)* *Let e be a program with $\vdash e : \tau, \varphi$, then for any $T \in \mathrm{Comp}(e)$ where $\theta_i, \mathcal{K}_i, \mathcal{P}_i$ is the first occurrence of π in T, then there exists CT, ST, PT such that*

$$CT, ST \vdash \theta_i, \mathcal{K}_i, \mathcal{P}_i : PT$$
$$PT(\pi) = \tau$$

and either

- $\pi \Uparrow_T$
- $\pi \Downarrow_T v$ and there exists $CT' \supseteq CT$, $ST' \supseteq ST$ where $CT', ST' \vdash v : PT(\pi)$

PROOF. See **[Rep92]** for the proof which relies on uniform evaluation, subject reduction, untypability of stuck configurations, and well-formed configurations. $\square$

By defining an evaluation function that distinguishes programs that diverge from those that cause run-time type errors we can state strong and weak soundness theorems.

$$eval'(\pi) = \begin{cases} \text{WRONG} & \text{if } \pi_i \text{ is stuck for some } \theta_i, \mathcal{K}_i, \mathcal{P}_i \in T \\ v & \text{if } \pi \Downarrow_T \end{cases}$$

Then strong and weak soundness follow as corollaries to syntactic soundness.

THEOREM 4.3.14. *(**Strong Soundness -** adapted from* **[Rep92]***) if $eval'(\pi) = v$ and $\theta_i, \mathcal{K}_i, \mathcal{P}_i$ is the first occurrence of π in T, then for any CT, ST, PT such that $CT, ST \vdash \theta_i, \mathcal{K}_i, \mathcal{P}_i : PT$ and $PT(\pi) = \tau$ then there is a $CT' \supseteq CT$ and $ST' \supseteq ST$ such that $(\{\}, CT', ST') \vdash v : PT(\pi), \emptyset$*

THEOREM 4.3.15. ***(Weak Soundness* - [WF92]*)*** *eval*$'(\pi) \neq$ WRONG

Proof of Matching Lemma 4.3.7

LEMMA. ***(Matching - adapted from*** **[Rep92]*)*** *If* $\theta \vdash (ev_1, ..., ev_k) \rightsquigarrow_k (e_1, ..., e_k)$ *and* $TE \vdash ev_i : \tau_i\, \texttt{event}_{\varphi_i}, \emptyset$ *(*$i \in \{1..k\}$*), then* $TE \vdash e_i : \tau_i, \varphi_i$.

PROOF. The proof is by induction on the event matching relation

$\boxed{\text{Base case } \theta \vdash (\kappa!v, \kappa?) \rightsquigarrow_2 ((), v)}$

By the typing of channel output and input (Figure 4.2.3) there is a type environment TE such that

$$TE \vdash \kappa!v : \texttt{unit}\,\texttt{event}_{\emptyset}, \emptyset$$
$$TE \vdash \kappa? : \tau\,\texttt{event}_{\emptyset}, \emptyset$$

hence the result follows immediately for $\kappa!v$ and $()$
Then from the premises we get

$$TE \vdash v : \tau, \emptyset$$

$\boxed{\text{Base case } \theta \vdash \beta_{e\,[r_1,...,r_n]} \rightsquigarrow_1 ()}$

By the typing of conditional events (Figure 4.2.3) there is a type environment TE such that

$$TE \vdash \beta_{e\,[r_1,...,r_n]} : \texttt{unit}\,\texttt{event}_{\varphi^{\text{pure}}}, \emptyset$$

and hence the result follows immediately

Inductive cases: for the inductive cases the $i \in \{1..k-1\}$ cases follow immediately from the induction hypothesis. The $i = k$ case is proved by a case analysis.

$\boxed{\text{Case } \theta \vdash (ev_1, ...ev_k) \rightsquigarrow_k (e_1, ...e_k)}$

This case follows immediately

$\boxed{\text{Case } \theta \vdash (ev_1, ..., ev_{k-1}, ev \Rightarrow \lambda x.e) \rightsquigarrow_k (e_1, ..., e_{k-1}, \lambda x.e\ e')}$

By the typing of wrapped events (Figure 4.2.3) there is a type environment TE such that

$$TE \vdash ev \Rightarrow \lambda x.e : \tau\ \mathtt{event}_{\varphi \cup \varphi'}, \emptyset$$

and from the premises

$$TE \vdash ev : \tau'\ \mathtt{event}_{\varphi}, \emptyset$$
$$TE \vdash \lambda x.e : \tau' \xrightarrow{\varphi'} \tau, \emptyset$$

By applying the induction hypothesis we get

$$TE \vdash e' : \tau', \varphi$$

and therefore by the typing of application (Figure 4.2.2)

$$TE \vdash \lambda x.e\ e' : \tau, \varphi \cup \varphi'$$

$\boxed{\text{Case } \theta \vdash (ev_1, ..., ev_{k-1}, ev \oplus ev') \rightsquigarrow_k (e_i, ..., e_{k-1}, e)}$

By the typing of choice events (Figure 4.2.3) there exists a type environment TE such that

$$TE \vdash ev \oplus ev' : \tau\ \texttt{event}_{\varphi \cup \varphi'}, \emptyset$$

and from the premises

$$TE \vdash ev : \tau\ \texttt{event}_{\varphi}, \emptyset$$
$$TE \vdash ev' : \tau\ \texttt{event}_{\varphi'}, \emptyset$$

By applying the induction hypothesis we get

$$TE \vdash e' : \tau, \varphi \cup \varphi'$$

$\boxed{\text{Case } \theta \vdash (ev_1, ..., ev_{k-1}, ev' \oplus ev) \rightsquigarrow_k (e_i, ..., e_{k-1}, e)}$

Same as previous case $\square$

CHAPTER 5

Translation

5.1. Translation rules

By applying region and effect inference, the translation rules introduce annotations in the target to help the run-time system track synchronization references. The translation rules are of the form:

$$TE \vdash e \hookrightarrow e' : \tau, \varphi$$

read given the type environment TE, expression e translates to e' which has type τ and effect φ.

Rule 5.1.3 translates the `condEvt` expression as a call to `condEvt`$'$. The regions ρ_i are introduced in the target expression as an annotation. In addition, there may be effect variables ϵ_j contained in φ^{pure} and the function `getRegions` extracts the regions corresponding to the effects of ϵ_j. The operation of extracting the regions of an effect is based on the following definition.

Definition 5.1.1. **(Effect Regions)** The regions of an effect φ are the regions referenced in the effects of φ.

$$\begin{array}{lcl} \text{EffectRegions}(\text{read}(\rho) & = & \{\rho\} \\ \text{EffectRegions}(\text{write}(\rho)) & = & \{\rho\} \\ \text{EffectRegions}(\text{init}(\rho)) & = & \{\rho\} \\ \text{EffectRegions}(\epsilon_1 \cup \epsilon_2) & = & \text{EffectRegions}(\epsilon_1) \cup \text{EffectRegions}(\epsilon_2) \end{array}$$

The rule 5.1.7 translates calls to `set` to `set`$'$. As was shown in the dynamic semantics of the target (Subsection 4.1), `set`$'$ in addition to performing a store update, informs the run-time system that a sync-region is being updated. Rule 5.1.8 translates lambda expressions. The condition $\text{frv}(e') \subseteq \text{frv}(TE, \tau)$ ensures that the lambda is not region polymorphic. Rule 5.1.10 handles the translation of polymorphic functions and is based on a similar translation due to Tofte and Talpin [**TT97**]. The condition $\text{fv}(\vec{\rho_i}, \vec{\epsilon_j}, \vec{\alpha}) \cap \text{fv}(TE, \varphi) = \emptyset$ ensures that the bound variables do not occur free in neither the type environment nor in the effect of the region polymorphic function. The polymorphic function is arity-raised by the translation to add list parameters for the polymorphic regions $\vec{\rho_i}$ and effects $\vec{\epsilon_j}$. As mentioned in Subsection 3.2.3, we use `letrec` as synctactic sugar for the equivalent definition that uses the applicative Y_v. Then rule 5.1.11 instantiates the region and effect list arguments of polymorphic functions.

Rule 5.1.12 wraps a `let` around the scope of a sync-region and is similar to the `letregion` translation rule in [**TT97**]. Since the region ρ does not appear free in the type environment or in the result type τ of e, the translation can wrap e' inside a `let` which has a binding corresponding `to` ρ. The function `newRegName` will generate a value of type `reg_name` corresponding to the region name associated with ρ. It is these extra variable bindings that are referenced in the target expression of rule 5.1.3.

(5.1.1) $$TE \vdash c \hookrightarrow c :\ \tau, \emptyset$$

(5.1.2) $$TE \vdash v \hookrightarrow v :\ \tau, \emptyset$$

$$\frac{\begin{array}{l} TE \vdash e \hookrightarrow e' : \text{unit} \overset{\varphi^{\text{pure}}}{\to} \text{bool}, \varphi \quad \{\text{read}(\rho_i)\} \cup \{\epsilon_j\} = \varphi_{\text{pure}} \\ i, j \in \{0..n\} \end{array}}{TE \vdash \texttt{condEvt}\, e \hookrightarrow \texttt{condEvt}'\,(e', [\rho_i,\ \texttt{getRegions}(\epsilon_j)]) : \tau, \emptyset} \tag{5.1.3}$$

$$\frac{TE \vdash e \hookrightarrow e' : \tau, \varphi}{TE \vdash \texttt{chan}\ x\ \texttt{in}\ e \hookrightarrow \texttt{chan}\ x\ \texttt{in}\ e' : \tau, \varphi} \tag{5.1.4}$$

$$\frac{TE \vdash e_1 \hookrightarrow e_1' : \tau_1, \varphi_1 \qquad TE \vdash e_2 \hookrightarrow e_2' : \tau_2, \varphi_2}{TE \vdash (e_1, e_2) \hookrightarrow (e_1', e_2') : \tau_1 \times \tau_2, \varphi_1 \cup \varphi_2} \tag{5.1.5}$$

$$\frac{TE \vdash e_1 \hookrightarrow e_1' : \tau_1 \overset{\varphi}{\to} \tau_2, \varphi_1 \qquad TE \vdash e_2 \hookrightarrow e_2' : \tau_1, \varphi_2}{TE \vdash e_1\ e_2 \hookrightarrow e_1'\ e_2' : \tau_2, \varphi_1 \cup \varphi_2 \cup \varphi} \tag{5.1.6}$$

$$\frac{TE \vdash e_1 \hookrightarrow e_1' : \texttt{sync\_ref}_\rho(\tau), \varphi_1 \qquad TE \vdash e_2 \hookrightarrow e_2' : \tau, \varphi_2}{TE \vdash \texttt{set}(e_1, e_2) \hookrightarrow \texttt{set}'(e_1', e_2', \rho) : \texttt{unit}, \varphi_1 \cup \varphi_2 \cup \{\text{write}(\rho)\}} \tag{5.1.7}$$

$$\frac{\begin{array}{ll} TE \pm \{x \mapsto \tau_1\} \vdash e \hookrightarrow e' : \tau_2, \varphi & \tau = \tau_1 \overset{\varphi'}{\to} \tau_2 \\ \varphi \subseteq \varphi' & \text{frv}(e') \subseteq \text{frv}(TE, \tau) \end{array}}{TE \vdash \lambda x.e \hookrightarrow \lambda x.e' : \tau, \emptyset} \tag{5.1.8}$$

$$\frac{\begin{array}{l} TE \vdash e_1 \hookrightarrow e_1' : \tau_1, \varphi_1 \\ TE + \{x \mapsto \tau_1\} \vdash e_2 \hookrightarrow e_2' : \tau_2, \varphi_2 \end{array}}{\begin{array}{ll} TE \vdash & \texttt{let}\ x = e_1\ \texttt{in}\ e_2\ \texttt{end} \hookrightarrow \\ & \texttt{let}\ x = e_1'\ \texttt{in}\ e_2'\ \texttt{end} : \tau_2, \varphi_1 \cup \varphi_2 \end{array}} \tag{5.1.9}$$

$$\frac{\begin{array}{ll} TE + \{f \mapsto \forall\, \vec{\rho_i}, \vec{\epsilon_j}, \vec{\alpha}\, .\underline{\tau}\} \vdash \lambda x.e_1 \hookrightarrow \lambda x.e_1' : \tau, \emptyset & \tau = \tau_1 \xrightarrow{\varphi_1} \tau_2 \\ \mathrm{fv}(\vec{\rho_i}, \vec{\epsilon_j}, \vec{\alpha}) \cap \mathrm{fv}(TE, \varphi) = \emptyset & \\ TE + \{f \mapsto \forall\, \vec{\rho_i}, \vec{\epsilon_j}, \vec{\alpha}\, .\underline{\tau}\} \vdash e_2 \hookrightarrow e_2' : \tau, \varphi_2 & i, j \in \{0..n\} \end{array}}{\begin{array}{ll} TE \vdash & \texttt{letrec}\ f(x) = e_1\ \texttt{in}\ \mathtt{e}_2 \hookrightarrow \\ & \texttt{letrec}\ f[\vec{\rho_i}][\vec{\epsilon_j}]\ (x) = e_1'\ \texttt{in}\ \mathtt{e}_2' : \tau, \varphi_2 \end{array}} \tag{5.1.10}$$

$$\frac{\begin{array}{ll} TE(f) = \sigma & \sigma \succ \tau_2\ \mathrm{via}\, S \\ \sigma = \forall\, \vec{\rho_i}, \vec{\epsilon_j}, \vec{\alpha}\, .\underline{\tau_1} & i, j \in \{0..n\} \end{array}}{TE \vdash f \hookrightarrow f\,[S(\vec{\rho_i})][S(\vec{\epsilon_j})] : \tau_2, \varphi} \tag{5.1.11}$$

$$\frac{\begin{array}{l} TE \vdash e \hookrightarrow e' : \tau, \varphi \\ \rho \notin \mathrm{frv}(TE, \tau) \end{array}}{\begin{array}{llll} TE \vdash & e \hookrightarrow & \texttt{let} & \rho = \texttt{newRegName}() \\ & & \texttt{in} & \\ & & & e' \\ & & \texttt{end} : & \tau, \varphi \backslash \{\mathrm{read}(\rho), \mathrm{write}(\rho), \mathrm{init}(\rho)\} \end{array}} \tag{5.1.12}$$

5.2. Correctness of the translation

The main result of this Section is that consistency is preserved by concurrent reduction. That is, if source and target configurations c_s and c_t are consistent and they each perform a small-step reduction then the resulting configurations are also consistent. For now consistency is described informally in that consistent configurations are related by the translation rules. The approach of this Section is as follows. After stating several definitions we present translation rules for the intermediate values in the language. The new translation rules allow us to define consistency for configurations that contain intermediate values. Then following Lemma 5.2.4 we state and prove consistency for sequential and subsequently concurrent configurations. We also show that the consistency properties can be formulated as satisfying a similarity relation. The following definitions are from [**Mil89**].

DEFINITION 5.2.1. **(Simulation)** A relation $S \subseteq \mathit{Rel}$ is a *simulation* iff $a\,S\,b$ implies: whenever $a \rightarrow a'$ $\exists b'$ such that $b \rightarrow b'$ and $a'\,S\,b'$.

Let *similarity* $\lesssim \subseteq \mathit{Rel}$ be the greatest simulation

$$\lesssim = \bigcup\{S \mid S\,\mathrm{is\,a\,simulation}\}$$

We also define $[\;]$ as

$$[S] \overset{\mathrm{def}}{=} \{(a,b) \mid \mathrm{whenever\,a} \rightarrow a'\,\mathrm{there\,exists\,b'\,such\,that}\,b \rightarrow b'\,\mathrm{and\,a'}\,S\,b'\}$$

We are interested in similarity as opposed to bisimilarity since there is only a translation in one direction (although we conjecture that bisimilarity holds). Next we define consistency for sequential configurations.

Definition 5.2.2. **(Consistent Stores)** Two stores θ and θ' are consistent (denoted $\mathcal{C}(\theta, \theta')$ if $\mathrm{Dom}(\theta) \subseteq \mathrm{Dom}(\theta')$ and $TE \vdash \theta(x) \hookrightarrow \theta'(x) : \tau, \emptyset$ for all $x \in \mathrm{Dom}(\theta)$

Definition 5.2.3. **(Consistent Sequential Configurations)** Two well-formed sequential configurations $c_s = \theta, e$ and $c_t = \theta', e'$ are *consistent (denoted* $\mathcal{C}(c_s, c_t)$*) if* $TE \vdash e \hookrightarrow e' : \tau, \varphi$ and $C(\theta, \theta')$

Since a configuration may contain intermediate values which are not part of the concrete syntax we must define translation rules for the intermediate values. This will allow the translation of configurations containing both intermediate values and expressions. Channels are translated by the identity function (rule 5.2.1) and this is reflected in rules 5.2.2 and 5.2.3. A wrapper event is translated (rule 5.2.4) by combining the result of translating both the wrapped event and the wrapper function. Similarly for choice events (rule 5.2.5), the result is obtained by combining the results of translating component events. Conditional events (rule 5.2.6) are translated by translating the encapsulated expression e and also by adding the region name variables ρ_i which correspond to the read effects of e.

$$TE \vdash \kappa \hookrightarrow \kappa : \tau\ \texttt{chan}, \emptyset \tag{5.2.1}$$

$$\frac{TE \vdash v \hookrightarrow v' :\ \tau, \emptyset}{TE \vdash \kappa!v \hookrightarrow \kappa!v' : \texttt{unit event}_{\emptyset}} \tag{5.2.2}$$

$$TE \vdash \kappa? \hookrightarrow \kappa? : \texttt{unit event}_{\emptyset} \tag{5.2.3}$$

$$(5.2.4)\qquad \frac{TE \vdash ev \hookrightarrow ev' : \tau\ \mathtt{event}_{\varphi}, \emptyset \quad TE \vdash v \hookrightarrow v' : \tau \xrightarrow{\varphi} \tau', \emptyset}{TE \vdash ev \Rightarrow v \hookrightarrow ev' \Rightarrow v' : \tau'\ \mathtt{event}_{\varphi^{\text{pure}}}, \emptyset}$$

$$(5.2.5)\qquad \frac{TE \vdash ev_1 \hookrightarrow ev_1' : \tau\ \mathtt{event}_{\varphi_1^{\text{pure}}}, \emptyset \quad TE \vdash ev_2 \hookrightarrow ev_2' : \tau\ \mathtt{event}_{\varphi_2^{\text{pure}}}, \emptyset}{TE \vdash ev_1 \oplus ev_2 \hookrightarrow ev_1 \oplus ev_2 : \tau\ \mathtt{event}_{\varphi_1^{\text{pure}} \cup \varphi_2^{\text{pure}}}, \emptyset}$$

$$(5.2.6)\qquad \frac{\begin{array}{l} TE \vdash \beta_e : \mathtt{unit\ event}, \emptyset \qquad \{\text{read}(\rho_i)\} = \varphi^{\text{pure}} \\ TE \vdash e \hookrightarrow e' : \mathtt{bool}, \varphi^{\text{pure}} \end{array}}{TE \vdash \beta_e \hookrightarrow \beta_{e'\,[\rho_i]} : \mathtt{unit\ event}_{\varphi^{\text{pure}}}}$$

Next we introduce several lemmas which are used in the proofs of sequential and concurrent consistency.

LEMMA 5.2.4. *If* $TE \vdash E_s[e_s] \hookrightarrow E_t[e_t] : \tau, \varphi$ *and* $TE \vdash e_s' \hookrightarrow e_t', \tau', \varphi'$, *and* $TE \vdash e_s : \tau', \varphi''$ *then* $TE \vdash E_s[e_s'] \hookrightarrow E_t[e_t'] : \tau, \varphi'''$.

PROOF. The idea is to view the translation as transforming the syntax tree $E_s[e_s]$ into another tree $E_t[e_t]$. Then if we cut out the subtree e_s replacing it with e_s' (which translates into e_t') then the translation of $E_s[e_s']$ will generate $E_t[e_t']$. The only other requirement is that all expressions that can be placed in a given hole must have the same type. □

LEMMA 5.2.5. *If* $TE \vdash e_s \hookrightarrow e_t : \tau, \varphi$ *and* $TE \vdash v_s \hookrightarrow v_t : \tau', \emptyset$, *then* $TE \vdash e_s[x \mapsto v_s] \hookrightarrow e_t[x \mapsto v_t] : \tau, \varphi$.

LEMMA 5.2.6. *If* $TE \vdash ev_i \hookrightarrow ev_i' : \tau_i\ \mathtt{event}, \varphi_i$ *for* $i \in \{1..n\}$ *and* $TE \vdash (ev_1, ..., ev_n) \rightsquigarrow_n (e_1, ..., e_n)$ *then there exists* $e_1', .., e_n'$ *such that* $TE \vdash e_i \hookrightarrow e_i' : \tau_i, \varphi_i$ *and* $TE \vdash (ev_1', ..., ev_n') \rightsquigarrow_n (e_1', ..., e_n')$

LEMMA 5.2.7. ***(Sequential Consistency)*** *If* $c_s = \theta \vdash E[e]$ *and* $c_t = \theta' \vdash E'[e']$ *are well-formed sequential configurations where* $\mathcal{C}(c_s, c_t)$ *and* $c_s \longmapsto c'_s$ *then there exists a transition* $c_t \longmapsto c'_t$ *such that* $\mathcal{C}(c'_s, c'_t)$ *.*

In terms of similarity, the Lemma illustrates that if $S \subseteq Rel$ is $\{(c_s, c_t) \mid c_s$ and c_t are consistent sequential configurations $\mathcal{C}(c_s, c_t)\}$ then $S \subseteq [S \cup \lesssim]$.

PROOF. From the definition of consistency we have

$$TE \vdash E[e] \hookrightarrow E'[e'] : \tau, \varphi \tag{5.2.7}$$

$$TE \vdash \theta(x) \hookrightarrow \theta'(x) : \tau, \emptyset \quad \text{for all } x \in \mathrm{Dom}(\theta) \tag{5.2.8}$$

By Lemma 5.2.4 it suffices to show that in configurations c'_s, c'_t the expressions in the holes of E and E' are related by translation. Consistency is implicitly preserved with respect to the stores since sequential evaluation is pure. The proof proceeds as a case analysis on the structure of e and $\longmapsto$ in order to show that the resulting configurations are consistent.

$\boxed{\text{Case } e = c\ v}$ and the transition

$$\theta, E[c\ v] \longmapsto \theta, E[\delta(c, v)]$$

From the translation rule 5.1.1 for constants

$$TE \vdash c \hookrightarrow c : \tau, \varphi$$

The remaining part is a case analysis on c in order to determine the form of v

Subcase $e = +\ (v_1, v_2)$

By the typablility of δ we have

$$TE \vdash v_1 : \texttt{int}, \emptyset$$

$$TE \vdash v_2 : \texttt{int}, \emptyset$$

then by the translation for constants

$$TE \vdash v_1 \hookrightarrow v_1 : \texttt{int}, \emptyset$$

$$TE \vdash v_2 \hookrightarrow v_2 : \texttt{int}, \emptyset$$

hence the target expression is identical

$$e' = + \ (v_1, v_2)$$

Subcase $e_s = \texttt{first} \ (v_1, v_2)$ If

(5.2.9) $$TE \vdash v_1 \hookrightarrow v_1' : \tau_1, \emptyset$$

(5.2.10) $$TE \vdash v_2 \hookrightarrow v_2' : \tau_2, \emptyset$$

then the target expression is

$$e' = \texttt{first} \ (v_1', v_2')$$

hence

$$\theta, E[\texttt{first} \ (v_1, v_2)] \longmapsto \theta, E[v_1]$$

$$\theta', E'[\texttt{first} \ (v_1', v_2')] \longmapsto \theta', E'[v_1']$$

and therefore the resulting configurations are consistent.

$\boxed{\text{Case } e = \lambda x.e\ v}$ and the transition

$$\theta, E[\lambda x.e\ v] \longmapsto \theta, E[e[x \mapsto v]]$$

If by translation rules 5.1.8

$$TE \vdash \lambda x.e \hookrightarrow \lambda x.e' : \tau \xrightarrow{\varphi} \tau', \emptyset \tag{5.2.11}$$

$$TE \vdash v \hookrightarrow v' : \tau : \emptyset \tag{5.2.12}$$

and then from the premises

$$TE + \{x \mapsto \tau\} \vdash e \hookrightarrow e' : \tau', \varphi \tag{5.2.13}$$

then by 5.2.7, 5.2.11, and 5.2.12 the corresponding target configuration is

$$\theta', E[\lambda x.e'\ v']$$

with the target transition

$$\theta', E[\lambda x.e'\ v'] \longmapsto \theta', E'[e'[x \mapsto v']]$$

Finally by 5.2.13, Lemma 5.2.5, and 5.2.12 we get

$$TE \vdash e[x \mapsto v] \hookrightarrow e'[x \mapsto v'] : \tau', \varphi$$

and therefore the resulting configurations are consistent.

$\boxed{\text{Case } e = \texttt{let } x = e_1 \texttt{ in } e_2}$ and the transition

$$\theta, E[\texttt{let } x = v \texttt{ in } e] \longmapsto \theta, E[e[x \mapsto v]]$$

If by translation rules 5.1.8

$$TE \vdash e \hookrightarrow e' : \tau, \varphi \tag{5.2.14}$$

$$TE \vdash v \hookrightarrow v' : \tau :, \emptyset \tag{5.2.15}$$

then by consistency the corresponding target configuration is

$$\theta', E'[\texttt{let } x = v' \texttt{ in } e']$$

with the target transition

$$\theta', E'[\texttt{let } x = v' \texttt{ in } e'] \longmapsto \theta', E'[e'[x \mapsto v']]$$

Finally by 5.2.14, Lemma 5.2.5, and 5.2.15 we get

$$TE \vdash e[x \mapsto v] \hookrightarrow e'[x \mapsto v'] : \tau, \varphi$$

and therefore the resulting configurations are consistent.

$\boxed{\text{Case } c_s = \theta \pm \{x \mapsto v\}, E[\texttt{get } x]}$ and the transition

$$\theta \pm \{x \mapsto v\}, E[\texttt{get } x] \longmapsto \theta \pm \{x \mapsto v\}, E[v]$$

If by translation rule 5.1.6

$$TE \vdash E[\texttt{get}\ x] \hookrightarrow E'[\texttt{get}\ x] : \tau, \varphi \tag{5.2.16}$$

and

$$TE \vdash v \hookrightarrow v' : \tau', \emptyset \tag{5.2.17}$$

then by consistency the corresponding target configuration is

$$\theta' \pm \{x \mapsto v\}, E'[\texttt{get}\ x] \longmapsto \theta' \pm \{x \mapsto v\}, E'[v]$$

with the target transition

$$\theta' \pm \{x \mapsto v'\}, E'[\texttt{get}\ x] \longmapsto \theta' \pm \{x \mapsto v'\}, E'[v']$$

Finally by Lemma 5.2.4 and 5.2.17 we get

$$TE \vdash E[v] \hookrightarrow E'[v'] : \tau, \varphi$$

and therefore the resulting configurations are consistent. □

Definition 5.2.8. **(Consistent Processes)** Two process set $\mathcal{P}$ and $\mathcal{P}'$ are consistent (denoted $\mathcal{C}(\mathcal{P}, \mathcal{P}')$) if $\mathrm{Dom}(\mathcal{P}) \subseteq \mathrm{Dom}(\mathcal{P}')$ and $TE \vdash P(\pi) \hookrightarrow \mathcal{P}'(\pi), \tau, \varphi$ for all $\pi \in \mathrm{Dom}(\mathcal{P})$

Definition 5.2.9. **(Consistent Concurrent Configurations)** Two well-formed concurrent configurations $c_s = \theta, \mathcal{K}, \mathcal{P}$ and $c_t = \theta', \mathcal{K}, \mathcal{P}'$ are consistent (denoted $\mathcal{C}(c_s, c_t)$) if $\mathcal{C}(\theta, \theta')$ and $\mathcal{C}(\mathcal{P}, \mathcal{P}')$.

THEOREM 5.2.10. ***(Concurrent Consistency)*** *If* $c_s = \theta, \mathcal{K}, \mathcal{P}$ *and* $c_t = \theta, \mathcal{K}, \mathcal{P}$ *are well-formed concurrent configurations where* $\mathcal{C}(c_s, c_t)$ *and* $c_s \Longrightarrow c'_s$ *then there exists a transition* $c_t \Longrightarrow c'_t$ *such that* $\mathcal{C}(c'_s, c'_t)$ *.*

In terms of similarity, the Theorem illustrates that if $S \subseteq Rel$ is $\{(c_s, c_t) \mid c_s$ and c_t are consistent concurrent configurations $\mathcal{C}(c_s, c_t)\}$ then $S \subseteq [S \cup \lesssim]$.

PROOF. By consistency

$$\begin{array}{ll} TE \vdash P(\pi) \hookrightarrow \mathcal{P}'(\pi) : \tau, \varphi & \text{for all } \pi \in \mathrm{Dom}(\mathcal{P}) \\ TE \vdash \theta(x) \hookrightarrow \theta'(x) : \tau, \emptyset & \text{for all } x \in \mathrm{Dom}(\theta) \end{array}$$

The proof is a case analysis on the left hand side of $\Longrightarrow$ in order to show that the resulting configurations are consistent.

$\boxed{\text{Case } c_s = \theta, \mathcal{K}, \mathcal{P}_v, \mathcal{P}_r + \langle \pi\,;\, E[e] \rangle}$ and the transition

$$\theta, \mathcal{K}, \mathcal{P}_v, \mathcal{P}_r + \langle \pi\,;\, E[e] \rangle \Longrightarrow \theta, \mathcal{K}, \mathcal{P}_v, \mathcal{P}_r + \langle \pi\,;\, E[e'] \rangle$$

If by the translation rules

$$TE \vdash E[e] \hookrightarrow E'[e''] : \tau, \varphi$$

then by consistency the corresponding target configuration is

$$\theta', \mathcal{K}, \mathcal{P}'_v, \mathcal{P}'_r + \langle \pi\,;\, E[e''] \rangle$$

with the target transition

$$\theta', \mathcal{K}, \mathcal{P}_v, \mathcal{P}'_r + \langle \pi\,;\, E'[e''] \rangle \Longrightarrow \theta', \mathcal{K}, \mathcal{P}'_v, \mathcal{P}'_r + \langle \pi\,;\, E'[e'''] \rangle$$

Hence by the Sequential Consistency Lemma 5.2.7 the resulting configurations are consistent.

$\boxed{\text{Case } c_s = \theta, \mathcal{K}, \mathcal{P}_v, \mathcal{P}_r + \langle \pi\, ;\, E[\texttt{new}\ v] \rangle}$ and the transition

$$\theta, \mathcal{K}, \mathcal{P}_v, \mathcal{P}_r + \langle \pi\, ;\, E[\texttt{new}\ v] \rangle \Longrightarrow \theta + \{x \mapsto v\}, \mathcal{K}, \mathcal{P}_v, \mathcal{P}_r + \langle \pi\, ;\, E[x] \rangle$$

If by rule translation rule 5.1.6

$$TE \vdash E[\texttt{new}\ v] \hookrightarrow E'[\texttt{new}\ v'] : \tau, \varphi$$

and from the premises

$$TE \vdash v \hookrightarrow v' : \tau, \emptyset \tag{5.2.18}$$

then by consistency and 5.2.18 the corresponding target configuration is

$$\theta', \mathcal{K}, \mathcal{P}'_v, \mathcal{P}'_r + \langle \pi\, ;\, E[\texttt{new}\ v'] \rangle$$

with the target transition

$$\theta', \mathcal{K}, \mathcal{P}'_v, \mathcal{P}'_r + \langle \pi\, ;\, E'[\texttt{new}\ v'] \rangle \Longrightarrow \theta' \pm \{x \mapsto v'\}, \mathcal{K}, \mathcal{P}'_v, \mathcal{P}'_r + \langle \pi\, ;\, E'[x] \rangle$$

By Lemma 5.2.4 we get

$$TE \vdash E[x] \hookrightarrow E'[x] : \tau, \varphi$$

an by $\mathcal{C}(\theta, \theta')$ and 5.2.18 we get

$$\mathcal{C}(\theta \pm \{x \mapsto v\},\ \theta' \pm \{x \mapsto v'\})$$

and therefore the resulting configurations are consistent.

Case	$c_s = \theta \pm \{x \mapsto v_1\}, \mathcal{K}, \mathcal{P}_v + \langle \pi_i\,;\, E_i[\texttt{sync}\ ev_i] \rangle, \mathcal{P}_r + \langle \pi\,;\, E[\texttt{set}\ (x, v_2)] \rangle$ $i \in \{0..n\}$

and the transition

$$\begin{aligned}&\theta \pm \{x \mapsto v_1\}, \mathcal{K}, \mathcal{P}_v, \mathcal{P}_r + \langle \pi_i\,;\, E_i[\texttt{sync}\ ev_i] \rangle + \langle \pi\,;\, E[\texttt{set}\ (x, v_2)] \rangle \implies \\ &\theta \pm \{x \mapsto v_2\}, \mathcal{K}, \mathcal{P}_v, \mathcal{P}_r + \langle \pi_i\,;\, E_i[()] \rangle + \langle \pi\,;\, E[()] \rangle\end{aligned}$$

based on the premise

$$\theta \pm \{x \mapsto v_2\} \vdash ev_i \leadsto_1 ()$$

If by translation rules 5.1.6 and 5.1.7

$$\begin{aligned}&TE \vdash E[\texttt{set}\ (x, v_2)] \hookrightarrow E'[\texttt{set}'\ (x, v_2', r)] : \tau, \varphi \\ &TE \vdash E_i[\texttt{sync}\ ev_i] \hookrightarrow E_i'[\texttt{sync}\ ev_i'] : \tau_i, \varphi_i\end{aligned}$$

and from the premises

$$TE \vdash v_1 \hookrightarrow v_1' : \tau_1, \emptyset \tag{5.2.19}$$

$$TE \vdash v_2 \hookrightarrow v_2' : \tau_2, \emptyset \tag{5.2.20}$$

$$TE \vdash ev_i \hookrightarrow ev_i' : \tau_3, \emptyset \tag{5.2.21}$$

then by consistency the corresponding target configuration is

$$\theta' \pm \{x \mapsto v_1'\}, \mathcal{K}, \mathcal{P}_v' + \langle \pi_i\,;\, E_i'[\texttt{sync}\ ev_i'] \rangle, \mathcal{P}_r' + \langle \pi\,;\, E'[\texttt{set}'\,(x,\, v_2',\, r)] \rangle$$

and by Lemma 5.2.6 the target transition is

$$\theta' \pm \{x \mapsto v'_1\}, \mathcal{K}, \mathcal{P}'_v + \langle \pi_i \,;\, E'_i[\texttt{sync}\ ev'_i] \rangle\,, \mathcal{P}'_r + \langle \pi \,;\, E'[\texttt{set}'\,(x,\, v'_2,\, r)] \rangle \implies$$
$$\theta' \pm \{x \mapsto v'_2\}, \mathcal{K}, \mathcal{P}'_v, \mathcal{P}'_r + \langle \pi_i \,;\, E'_i[()] \rangle\,, \mathcal{P}'_r + \langle \pi \,;\, E'[())] \rangle$$

By Lemma 5.2.4 we get

$$TE \vdash E[()] \hookrightarrow E'[()] : \tau, \varphi$$

$$TE \vdash E_i[()] \hookrightarrow E'_i[()] : \tau_i, \varphi_i$$

and by 5.2.20 and $\mathcal{C}(\theta_s, \theta_t)$ the resultant stores are consistent

$$\pm(\theta \pm \{x \mapsto v_2\},\, \theta' \pm \{x \mapsto v'_2\})$$

therefore the resulting configurations are consistent.

$\boxed{\text{Case } c_s = \theta, \mathcal{K}, \mathcal{P}_v, \mathcal{P}_r + \langle \pi \,;\, E[\texttt{sync}\ ev] \rangle}$ and the transition

$$\theta, \mathcal{K}, \mathcal{P}_v, \mathcal{P}_r + \langle \pi \,;\, E[\texttt{sync}\ ev] \rangle \Longrightarrow \theta, \mathcal{K}, \mathcal{P}_v, \mathcal{P}_r + \langle \pi \,;\, E[()] \rangle$$

based on the premise

$$\theta \vdash ev \leadsto_1 ()$$

If by translation rule 5.1.6

$$TE \vdash E[\texttt{sync}\ ev] \hookrightarrow E'[\texttt{sync}\ ev'] : \tau, \varphi \qquad (5.2.22)$$

and from the premises

$$TE \vdash ev \hookrightarrow ev' : \tau', \varphi' \tag{5.2.23}$$

then by consistency the corresponding target configuration is

$$\theta', \mathcal{K}, \mathcal{P}'_v, \mathcal{P}'_r + \langle \pi \,;\, E'[\texttt{sync}\ ev'] \rangle$$

with the target transition

$$\theta', \mathcal{K}, \mathcal{P}'_v, \mathcal{P}'_r + \langle \pi \,;\, E'[\texttt{sync}\ ev'] \rangle \Longrightarrow \theta', \mathcal{K}, \mathcal{P}'_v, \mathcal{P}'_r + \langle \pi \,;\, E'[()] \rangle$$

By 5.2.22 and Lemma 5.2.4 the target configurations are consistent.

$\boxed{\text{Case } c_s = \theta, \mathcal{K}, \mathcal{P}_v + \langle \pi_1 \,;\, E_1[\texttt{sync}\ ev_1] \rangle + ... + \langle \pi_k \,;\, E_k[\texttt{sync}\ ev_k], \mathcal{P}_r \rangle}$ and the transition

$$\begin{aligned} &\theta, \mathcal{K}, \mathcal{P}_v + \langle \pi_1 \,;\, E_1[\texttt{sync}\ ev_1] \rangle + ... + \langle \pi_k \,;\, E_k[\texttt{sync}\ ev_k] \rangle, \mathcal{P}_r \quad \Longrightarrow \\ &\theta, \mathcal{K}, \mathcal{P}_v, \mathcal{P}_r + \langle \pi_1 \,;\, E_1[e_1] \rangle + ... + \langle \pi_k \,;\, E_k[e_k] \rangle \end{aligned}$$

based on the premise

$$\theta \vdash (ev_1, ..., ev_k) \leadsto_k (e_1, ..., e_k)$$

If by translation rule 5.1.6

$$TE \vdash E_i[\texttt{sync}\ ev_i] \hookrightarrow E'_i[\texttt{sync}\ ev'_i] : \tau_i, \varphi_i \qquad i \in \{1..k\} \tag{5.2.24}$$

$$TE \vdash e_i \hookrightarrow e'_i : \tau'_i, \varphi'_i \tag{5.2.25}$$

and from the premises

$$TE \vdash ev_i \hookrightarrow ev'_i : \tau'_i, \varphi'_i \tag{5.2.26}$$

then by consistency the corresponding target configuration is

$$\theta', \mathcal{K}, \mathcal{P}'_v + \langle \pi_1 \,;\, E'[\texttt{sync}\ ev'_1] \rangle + \ldots + \langle \pi_k \,;\, E'_k[\texttt{sync}\ ev'_k] \rangle\,, \mathcal{P}'_r$$

and by Lemma 5.2.6 the target transition is

$$\begin{aligned}&\theta', \mathcal{K}, \mathcal{P}'_v + \langle \pi_1 \,;\, E'[\texttt{sync}\ ev'_1] \rangle + \ldots + \langle \pi_k \,;\, E'_k[\texttt{sync}\ ev'_k] \rangle\,, \mathcal{P}'_r \Longrightarrow\\ &\theta', \mathcal{K}, \mathcal{P}'_v, \mathcal{P}'_r + \langle \pi_1 \,;\, E'[e'_1] \rangle + \ldots + \langle \pi_k \,;\, E'_k[e'_k] \rangle\end{aligned}$$

based on the premise

$$\theta \vdash (ev', \ldots, ev'_k) \leadsto_k (e'_1, \ldots, e'_k)$$

By 5.2.25 and Lemma 5.2.4 the target configurations are consistent.

$\boxed{\text{Case } c_s = \theta, \mathcal{K}, \mathcal{P}_v, \mathcal{P}_r + \langle \pi \,;\, E[\texttt{channel}\ x\ \texttt{in}\ e] \rangle}$ and the transition

$$\begin{aligned}&\theta, \mathcal{K}, \mathcal{P}_v, \mathcal{P}_r + \langle \pi \,;\, E[\texttt{channel}\ x\ \texttt{in}\ e] \rangle \;\;\Longrightarrow\\ &\theta_s, \mathcal{K} + \kappa, \mathcal{P}_v, \mathcal{P}_r + \langle \pi;\, E[e[x \mapsto \kappa]] \rangle\end{aligned}$$

If by translation rule 5.1.4

$$TE \vdash E[\texttt{channel}\ x\ \texttt{in}\ e] \hookrightarrow E'[\texttt{channel}\ x\ \texttt{in}\ e'] : \tau, \varphi \tag{5.2.27}$$

and from the premises

$$TE \pm \{x \mapsto \tau'\} \vdash e \hookrightarrow e' : \tau'', \varphi' \tag{5.2.28}$$

then by consistency the corresponding target configuration is

$$\theta, \mathcal{K}, \mathcal{P}_v, \mathcal{P}_r + \langle \pi \, ; \, E[\texttt{channel}\ x\ \texttt{in}\ e'] \rangle$$

with the target transition

$$\begin{aligned} &\theta', \mathcal{K}, \mathcal{P}_v', \mathcal{P}_r' + \langle \pi \, ; \, E'[\texttt{channel}\ x\ \texttt{in}\ e'] \rangle \implies \\ &\theta', \mathcal{K}, \mathcal{P}_t' + \langle \pi \, ; \, E[e'[x \mapsto \kappa]] \rangle \end{aligned}$$

By 5.2.28 we get

$$TE \vdash e[x \mapsto \kappa] \hookrightarrow e'[x \mapsto \kappa], \tau, \varphi \tag{5.2.29}$$

and by Lemma 5.2.5

$$TE \vdash E[e[x \mapsto \kappa]] \hookrightarrow E[e'[x \mapsto \kappa]] : \tau, \varphi$$

therefore the resulting configurations are consistent.

$\boxed{\text{Case } c_s = \theta, \mathcal{K}, \mathcal{P}_v, \mathcal{P}_r + \langle \pi \, ; \, E[\texttt{spawn}\ \lambda x.e] \rangle}$ and the transition

$$\begin{aligned} &\theta, \mathcal{K}, \mathcal{P}_v, \mathcal{P}_r + \langle \pi \, ; \, E[\texttt{spawn}\ \lambda x.e] \rangle \implies \\ &\theta, \mathcal{K}, \mathcal{P}_v, \mathcal{P}_r + \langle \pi \, ; \, E[()] \rangle + \langle \pi' \, ; \, \lambda x.e\ ()] \rangle \end{aligned}$$

If by translation rule 5.1.6

$$TE \vdash E[\texttt{spawn}\ \lambda x.e] \hookrightarrow E'[\texttt{spawn}\ \lambda x.e'] : \tau, \varphi \tag{5.2.30}$$

then by consistency the corresponding target configuration is

$$\theta', \mathcal{K}, \mathcal{P}'_v, \mathcal{P}'_r + \langle \pi\,;\, E'[\texttt{spawn}\ \lambda x.e'] \rangle$$

with the target transition

$$\begin{aligned} &\theta', \mathcal{K}, \mathcal{P}'_v, \mathcal{P}'_r + \langle \pi\,;\, E'[\texttt{spawn}\ \lambda x.e'] \rangle && \Longrightarrow \\ &\theta', \mathcal{K}, \mathcal{P}'_v, \mathcal{P}'_r + \langle \pi\,;\, E'[()] \rangle + \langle \pi''\,;\, \lambda x.e'\ ()] \rangle && \end{aligned}$$

By the translation rule for `unit` and 5.2.30 we get

$$TE \vdash () \hookrightarrow () : \texttt{unit}, \emptyset$$

$$TE \vdash \lambda x.e\ () \hookrightarrow \lambda x.e'\ (), \tau, \varphi$$

and hence by Lemma 5.2.4

$$TE \vdash E[()] \hookrightarrow E'[()] : \tau, \varphi$$

therefore the resulting configurations are consistent. □

CHAPTER 6

Implementation

Currently we have a prototype implementation that has been used to experiment with the ideas presented in this thesis. The implementation consists of a parser which utilizes sml-lex [**AMT94**] and sml-yacc [**TA94**], a static analysis and translation module which utilizes the sml basis library, and a run-time system module which extends CML with support for FCS. The architecture diagram (Figure 6.0.1) illustrates a source-to-source transformation and a run-time system consisting of CML extended for FCS loaded into SML/NJ. Our modules consist of approximately 2000 lines of ML code.

To implement the semantics of the target language, the run-time system (RTS) must know which store locations could affect the value of a synchronization condition. Our implementation represents this dependency by a link from store regions to internal condEval structures. Figure 6.0.2 illustrates the topology of the RTS data structures. The synchronization structure `condEvt1` is dependent upon store regions `R1` and `R2`. Currently `Thread1` and `Thread2` are blocked and waiting for the condition in `condEvt1` to change from `false` to `true`. The `condEvt2` structure depends on store regions `R2` and `R3`. Similarly, the condition in `condEvt2` is `false` and `Thread3` and `Thread4` will remain blocked until the synchronization condition becomes `true`.

If an update to region `R1` occurs then the condition for `condEvt1` is re-evaluated. A result of `true` would unblock `Thread1` and `Thread2`. An update to `R2` would

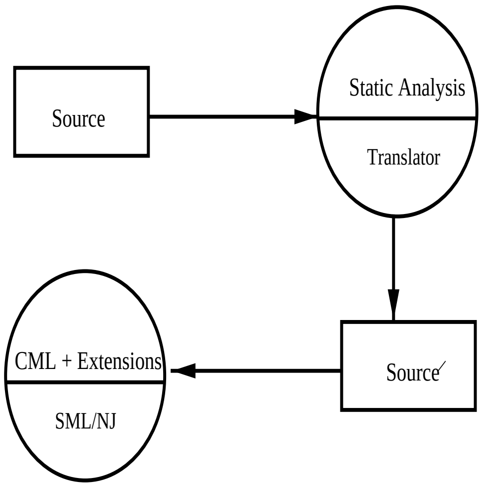

FIGURE 6.0.1. System architecture

force the re-evaluation of the conditions in both `condEvt1` and `condEvt2` which could result in all four threads becoming unblocked. Once a condition is determined to be `true` the RTS may delete its corresponding internal condEval representation since there are no longer any threads waiting on the condition. The code that creates internal `condEvt` structures is encapsulated inside `the event` value and is triggered at synchronization time. Thus whenever a threads attempts to `sync` on a conditional event the internals condEval structure is created (if necessary. Another approach would have been to leave the internal condEval structures intact until the garbage collector determines the structures may be removed. We chose the former approach since it does not require extending the garbage collector.

The target example in Figure 6.0.3 is the translated version of the `makeBarrier` example from Subsection 2.1.2. The annotations consist of the extra binding for the region variable ρ which is then passed in the argument list to `condEvt`′ since ρ is being read by the synchronization condition. Also the call to `set` in the source was translated to call `set`′ which has a region parameter.

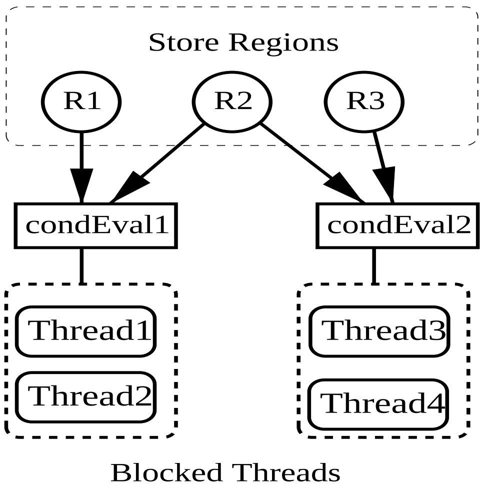

FIGURE 6.0.2. Topology of run-time system data structures

```
fun makeBarrier n =
  let
    val counter = new 0
    val incCh = channel()
    fun inc () = (
    recv incCh;
    set'(counter,get counter + 1,ρ);
      inc())
    val barrierEvt = condEvt'(fn () => get counter = n,[ρ])
  in
    spawn inc;
    wrap(sendEvt(incCh,()), fn()=>sync barrierEvt)
  end
```

FIGURE 6.0.3. Example target program

	1 car	2 cars	3 cars	4 cars	5 cars
number of re-evaluations	6	12	80	224	425
time (seconds)	5.5	12.3	31.4	46.3	69.2

TABLE 1. Measurements for unannotated programs

6.1. Benchmarks

To quantitatively measure the benefit of performing the translation we executed several versions of the translated traffic light program (Figure 2.1.6) against identical

	1 car	2 cars	3 cars	4 cars	5 cars
number of re-evaluations	2	8	15	32	50
time (seconds)	5.8	12.5	18.5	28.2	58.3

TABLE 2. Measurements for annotated (translated) programs

CML programs that were not annotated by translation. The run-time system for the unannotated programs simply polls *all* pending synchronization conditions when any sync ref variable is updated. Two measurements were made for each test case:

- the number of re-evaluations of synchronization conditions
- the time for 1 designated car to pass through all the traffic lights

A computational delay was inserted for each re-evaluation in order to magnify the time penalty for performing a re-evaluation. The lights were arranged in an alternating red/green pattern with one car initially positioned at each red light. In addition, the alternating pattern was maintained by changing all lights simultaneously. Each test case has twice as many lights as cars. The goal is to gradually increase the number of pending synchronization conditions for each successive test case. This is achieved by having a car waiting at each red light. Note that the synchronization conditions are disjoint since each condition references the status of only one light. Tables 1 and 2 contain the resulting data.

For the test case consisting of 1 car and 2 lights the annotated and unannotated programs performed closely in terms of both re-evaluations and time. However, as more lights and cars were introduced the benefit of introducing the region annotations becomes manifest. The test case consisting of 5 cars and 10 lights shows the translated program performing 50 re-evaluations as opposed to 425 for the unannotated counterpart. The timings for this test case are also more distinct in favor of the translated program.

We conclude that the translation produces a performance benefit for concurrent programs whose execution profile exhibits multiple pending synchronization conditions. At least some of the synchronization conditions must be disjoint to show improved performance relative to the unannotated program. Otherwise, a region update results in re-evaluation of all pending synchronization conditions which is exactly what the run-time system for the unannotated program does.

CHAPTER 7

Conclusion

We now undertake a comparison of FCS with the synchronization mechanisms of the categories of concurrent programming languages examined in Part I.

7.1. Comparison with concurrent object-oriented languages

FCS is a more flexible form of synchronization as compared conditional synchronization in Orca and Ada95 for several reasons. First, the synchronization mechanism is freed from the syntactic "baggage" of its second-class form. Second, our analysis and translation allow non-local variables to be used in the synchronization condition where this is disallowed in Ada95. Orca does not have nested scopes.

In addition, in first-class form conditional synchronization could enhance COOL's since it would allow an object to export partial information about its internal state without violating encapsulation. This would be accomplished by having a method return a conditional event. The synchronization in this case would involve private instance variables of a server object. Hence a client object could establish that a server object has reached a certain state.

7.2. Comparison with concurrent constraint languages

On the surface, CCP's `ask` seems similar to conditional synchronization since both involve synchronizing on boolean expressions. However there are fundamental differences in the operational semantics. In the case of conditional synchronization on a boolean expression `b`, there are two possible outcomes:

1. `b` evaluates to `true` and the thread attempting synchronization proceeds.
2. `b` evaluates to `false` and the thread blocks indefinitely until the value of `b` toggles to `true`.

Unlike `ask` in CCP, `b` must evaluate to either `true` or `false`. Conversely, an agent in CCP may `ask` about a constraint containing variables yet to be fixed in the store. Furthermore, if an agent asks about an expression that is inconsistent with the store then the agent is aborted. There is no notion of rejection in FCS. Moreover, `ask` synchronization is a second-class mechanism.

The similarity between FCS and `ask` is that in both cases a thread of control is suspended based on the status of a boolean expression. Thus both `ask` and FCS allow a thread to establish an invariant condition before crossing a program point. In spite of substantial differences, the similarity suggests that using conditional expressions to control synchronization is useful across different frameworks for concurrent programming.

7.3. Comparison with concurrent functional languages

The introduction of FCS is new to functional programming since none of the concurrent functional languages have a form of conditional synchronization nor a broadcast mechanism. Thus FCS complements the existing synchronization mechanisms of these languages.

```
module
ABRO:
input A, B, R;
output O;
loop
  [ await A || await B ];
  emit O
each R
end module
```

FIGURE 7.4.1. A simple controller in Esterel

7.4. Other related work

The notion of a language mechanism which broadcasts an event in a concurrent system also exists in the *synchronous* language Esterel [**BG92**] [**Ber98**]. In this Section we describe the semantic framework of Esterel and then undertake a comparison of the broadcast mechanisms found in FCS and Esterel.

Esterel is one of the so-called synchronous languages. Other synchronous languages are Signal [**GBBG85**] and Lustre [**CPHJ87**]. The synchronous model has also been incorporated into concurrent constraint programming [**SJG95**]. Synchronous languages appeared in the 1980's and have applications in the control systems domain (embedded systems). Synchronous languages are concurrent, yet deterministic. Prior to synchronous languages, one had to choose between concurrency and determinism since the classical concurrent languages (i.e. Ada) are non-deterministic. The motivation for incorporating determinism into concurrent systems is to create systems which are easier to specify, debug, and analyze. Determinism also allows synchronous languages to be efficiently compiled into deterministic finite-state automata. A synchronous program will always produce the same output for a given set of inputs [**MP86**].

Synchronous languages are designed for programming *reactive* systems where programs react (in a timely fashion) to stimuli from the external environment. This is in contrast to *interactive* systems such as operating systems where submitted requests are processed at some point in the future. Thus reactive systems are driven by external input and the pace of the interaction is determined by the environment (not by the system).

The essence of synchronous languages is expressed by the *synchrony hypothesis* which states that:

- all reactions are instantaneous and take zero time with respect to the external environment
- sub processes react instantly to each other
- interprocess communication is done instantaneously by broadcasting events

To illustrate the properties of synchronous languages and in particular Esterel, we next examine present the English specification and Esterel implementation of a simple controller from [**Ber98**]. The specification is given as:

- *Emit the output 0 as soon as both the inputs A and B have been received.*
- *Reset the behavior whenever the input R is received.*

The Esterel realization of the controller is given in Figure 7.4.1. During each reaction a signal has a unique status of being either *present* or *absent*. The status of an input signal is determined by the environment while other signals are generated when a process executes an `emit` statement. Unlike variables (which may be repeatedly updated), a signal's status is fixed during each reaction. The `await` statement waits for the specified signal and then terminates. Parallel composition is denoted by the vertical bars `||`, thus:

```
await A || await B
```

waits for the presence of both `A` and `B` and then terminates. The `loop each` statement is used for preemption [**Ber93**]. In the controller the body of the `loop` is restarted from the beginning each time the `R` signal is received.

We now compare the signal broadcasting mechanism of Esterel to the broadcast mechanism of FCS. There are salient distinctions between the two. First, sync ref variables may updated at random times while signals are established at for each reaction.

Hence our static analysis and translation does not apply since the signal changes happen at well defined times and `await` statements may not consist of boolean expressions.

The broadcast mechanism of Esterel is inherently different from conditional synchronization. Although signals may have values, the statement `await S` suspends until the signal `S` is present. Boolean expressions are not allowed in an `await` statement. The difference between signals and variables is illustrated by the fact that there are two different ways to test for the presence of a signal and the value of a boolean expression. The statement:

```
present S else stmt end
```

tests for the presence of `S` in the current reaction (executing `stmt` if `S` is not present). This is in contrast to the `if` statement used for testing the value of boolean expression.

7.5. Summary

FCS is a general purpose synchronization mechanism. As in the example of the discrete event simulator, FCS is appropriate for applications where it is required to suspend groups of threads in queues where a queue is flushed by establishing a condition. Thus the event of a condition becoming `true` is broadcast to all of the threads in the corresponding queue while the queuing machinery is hidden in the RTS internals. Moreover, FCS is the first higher-order form of conditional synchronization. Thus allowing greater expressiveness than the previous forms of conditional synchronization. The advantages of first-class synchronous values were previously demonstrated by the advent of CML [**Rep92**]:

- abstraction - the fact that a function performs a synchronous operation is reflected in its type specification.
- composability - by applying the CML `choose` or `wrap` combinators, new event values may be created by combining existing event values.

Our conditional `event` incorporates broadcast synchronization which complements the existing CML synchronization facilities associated with channel communication, M-variables, and I-variables.

With the introduction of concurrency into mainstream languages like Java [**GJS96**] concurrency has manifested itself as being useful for a broad class of application development. Yet the basis for synchronization in Java, the monitor [**Hoa74**], is not well-integrated with the type system and consequently does not aid the programmer sufficiently in building concurrent-access abstractions. By incorporating first-class mechanisms like FCS the programming language can facilitate programming with concurrency.

We introduced FCS and achieved the main benefits cited in the Introduction that can be achieved when introducing a concept into a programming language:

- improved performance since the compiler can perform optimizing transformations - our optimizing transformation for FCS results in re-evaluating sync conditions only when regions being read by a condition are updated.
- guarantees of correctness properties - since the semantics of programs can be analyzed and verified - in particular our transient theorem guarantees that a thread waiting for a condition to become `true` will not miss any changes where the condition becomes briefly `true` and then `false` again.
- abstraction - our examples of barrier synchronization and the discrete-event simulator (Subsection 2.1.2) illustrate that FCS facilitates the development of applications where broadcast synchronization is required.

A topic for future research is a distributed framework for FCS, which presents some interesting issues. In this context, a synchronization condition becoming `true` may require notification messages to be sent to remote hosts. Hence additional bookeeping would need to be maintained to track the network locations of threads synchronizing on conditional events.

Bibliography

[AMT94] A. W. Appel, J. S. Mattson, and D. R. Tarditi. A lexical analyzer generator for Standard ML.version 1.6.0. October 1994.

[AP89] R. Nikhil Arvind and K. Pingali. I-structures - data structues for parallel computing. *TOPLAS*, 11:598–632, 1989.

[AP95] P. Achten and R. Plasmeijer. Concurrent interactive processes in a pure functional language. In *Proc. of Computing Science in the Netherlands (CSN '95)*, pages 10–21, 1995.

[AWWV96] J. L. Armstrong, M. C. Williams, C. Wikstrom, and S. R. Virding. *Concurrent Programming in Erlang.* Prentice-Hall, 2nd edition, 1996.

[Bac86] M. J. Bach. *The Design of the Unix Operating System.* Prentice-Hall, 1986.

[Bar84] H. P. Barendregt. *The Lambda Calculus*, volume 103 of *Studies In Logic And The Foundations of Mathematics.* North-Holland, 1984.

[Bar89] J. G. P. Barnes. *Programming in Ada.* Addison-Wesley, 3rd edition, 1989.

[Bar96] J. Barnes. *Programming in Ada 95.* Addison-Wesley, 1996.

[BB90] G. Berry and G. Boudol. The chemical abstract machine. In *Principles of Programming Languages*, pages 81–94. ACM, 1990.

[BC93] F. Benhammou and A. Colmerauer. *Constraint Logic Programming - Selected Research.* Logic Programming. MIT Press, 1993.

[BDG$^+$91] A. Beguelin, J. J. Dongarra, A. Geist, R. Mancheck, and V. Sunderam. A users' guide to PVM (parallel virtual machine). Technical Report ORNL/TM-11826, Oak Ridge National Laboratory, 1991.

[Ber93] G. Berry. Preemption and concurrency. In *FSTTCS 93, Lecture Notes in Computer Science*, volume 761. Springer-Verlag, 1993.

[Ber98] G. Berry. *The Foundations of Esterel.* MIT Press, 1998.

[BF96] P. D. Blasio and K. Fisher. A calculus for concurent objects. In *Concurrency Theory - CONCUR '96*, volume 1119 of *LNCS*, pages 655–670, 1996.

[BG92] G. Berry and G. Gonthier. The synchronous programming language Esterel: Design, semantics, and implementation. *Sience of Computer Programming*, 19(2):83–152, 1992.

[BKT88] H. Bal, M. F. Kaashock, and A. Tanenbaum. Orca: A language for parallel programming of distributed systems. In *Usenix/SERC Workshop on Experiences with Building Distributed and Multiprocessor Systems.* Vrije University, Netherlands, 1988.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattic model for static analysis of programs by construction of approximation of fixpoints. In *ACM Symposium on Principles of Programming Languages.* ACM, 1977.

[CPHJ87] P. Caspi, D. Pilaud, N. Halbwachs, and J.Plaice. Lustre: a declarative language for real-time programming. In *Proceedings of Principles of Programming Languages (POPL'87).* ACM, 1987.

[DM82] L. Damas and R. Milner. Principal type schemes for functional programs. In *ACM Symposium on Principles of Programming Languages*, 1982.

[EGLT76] K.P. Eswaran, J. N. Gray, R.A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *CACM*, 19(11):624–633, Nov 1976.

[FF86] M. Felleisen and D. Friedman. *Formal Description of Programming Concepts III*, chapter Control operators, the SECD-machine, and the λ calculus. North-Holland, 1986.

[GA89] A. Gottlieb and G. S. Almasi. *Highly Parallel Computing*, chapter 2. Benjamin/Cummings Publishing, 1989.

[GBBG85] P. Le Guernic, A. Benveniste, P. Bournal, and T. Gauthier. Signal: A dataflow oriented language for signal processing. Technical Report 246, IRISA, Rennes, France, 1985.

[Gel85] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.

[GJLS87] D. Gifford, P. Jouvelot, J. M. Lucassen, and M. Sheldon. Fx-87 reference manual. Technical Report TR-407, MIT-LCS, 1987.

[GJS96] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* The Java Series. Addison-Wesley, June 1996.

[GMP89] A. Giacalone, P. Mishra, and S. Prasad. Facile: A symmetric integration of concurrent and functional programming. In *International Journal of Parallel Programming*, volume 18. 1989.

[GR93] E. R. Gansner and J. H. Reppy. *A Multi-threaded Higher-order User Interface Toolkit*, volume 1 of *Software Trends*, pages 61–80. John Wiley & Sons, 1993.

[Har86] D. M. Harland. *Concurrency and Programming Languages.* John Wiley and Sons, 1986.

[Hoa74] C. A. R. Hoare. Monitors: an operating system structuring concept. *Communications of the ACM*, 17:549–557, 1974.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. In *Communications of the ACM*, volume 21, pages 666–677. ACM, August 1978.

[HS86] R. J. Hindley and J. P. Seldin. *Introduction to Combinators and λ-Calculus.* Cambridge University Press, 1986.

[HW87] M. Herlihy and J. Wing. Avalon : Language support for reliable distributed system. In *17th Int. Symposium on Fault-Tolerant Computing*, pages 89–94. , Pittsburgh, PA, [7] 1987.

[Int95] Intermetrics, Inc. *Ada 95 Rationale*, Jan. 1995.

[iTCS96] Algol-Like Languages (Progress in Theoretical Computer Science. *P. W. O'Hearn R. D. Tennent.* Birkhauser, 1996.

[JG91] P. Jouvelot and D. Gifford. Algebraic reconstruction of types and effects. In *ACM Symposium on Principles of Programming Languages*, 1991.

[JGF96] S. P. Jones, A. Gordon, and S. Finne. Concurrent Haskell. *ACM Symposium on Principles of Programming Languages*, 1996.

[JL87] J. Jaffar and J. L Lassez. Constraint logic programming. In *ACM Symposium on Principles of Programming Lanaguages*, pages 111–119, January 1987.

[JL96] R. Jones and R. Lins. *Garbage Collection : Algorithms for Automatic Dynamic Memory Management.* John Wiley and Sons, 1996.

[Jor78] H. Jordan. A special purpose architecture for finite element analysis. In *International Conference on Parallel Processing*, pages 263–266, 1978.

[KJ88] D. A. Kranz and R. H. Halstead Jr. Multi-t: A high-performance parallel Lisp. In *Conference on Programming Language Design and Implementation PLDI'88.* ACM, June 1988.

[KPT96] N. Kobayashi, B. C. Pierce, and D. N. Turner. Linearity and the pi-calculus. In *Proceedings of Principles of Programming Languages (POPL'96).* ACM, 1996.

[LG91] J. Lucassen and D. Gifford. Polymorphic effect systems. In *ACM Symposium on Principles of Programming Languages*, 1991.

[LJ96] J. Launchbury and S. P. Jones. I-structures - data structues for parallel computing. *Lisp and Symbolic Computation*, 1996.

[Luc87] J. M. Lucassen. *Types and Effects, towards the integration of functional and imperative programing.* MIT/LCS/TR-408, MIT Laboratory for Computer Science, 1987.

[Mat89] D. C. J. Matthews. Processes for Poly and ML. Technical Report 16, University of Cambridge, 1989.

[Mey88] B. Meyer. *Object-oriented software construction.* Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

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

[MP86] Z. Manna and A. Pneuli. *Concurrency and Programming Languages.* John Wiley and Sons, 1986.

[MW97] S. Marlow and P. Wadler. A practical subtyping system for Erlang. In *2'nd ACM International Conference on Functional Programming*, Amsterdam, June 1997.

[NN94] F. Nielson and H. R. Nielson. Constraints for polymorphic behaviours of Concurrent ML. In *CCL'94*, volume 845 of *LNCS*, pages 73–88. Springer, 1994.

[Pap89] M. Papathomas. *Concurrency Issues in Object-Oriented Programming Languages*, pages 207–245. Centre Universitaire d'Informatique, University of Geneva, July 1989.

[Per87] R. H. Perrott. *Parallel Programming.* Addison-Wesley, Menlo Park, 1987.

[Ram90] N. Ramsey. Concurrent programming in ML. Technical Report CS-TR-262-90, Princeton University, 1990.

[Rep88] J. H. Reppy. Synchronous operations as first-class values. In *SIGPLAN Programming Language Design and Implementation*, pages 250–259. ACM, June 1988.

[Rep92] J. H. Reppy. *Higher-Order Concurrency.* Technical Report TR 92-1285, Cornell University, 1992.

[Rep95] J. H. Reppy. First-class synchronous operations. In *Theory and Practice of Parallel Programming.* Springer-Verlag LNCS, 1995.

[Rep99] J. H. Reppy. *Concurrent Programming in ML.* Cambridge University Press, 1999.

[RHH85] Jr R. H. Halstead. Multilisp: A language for concurrent symbolic computation. In *Transactions of Programming Languages and Systems*, volume 7, October 1985.

[Sar93] V. Saraswat. *Concurrent Constraint Programming.* MIT Press, 1993.

[Sha86] E. Shapiro. Concurrent prolog: A progress report. In *IEEE Computer*, volume 19, August 1986.

[Sha87] E. Shapiro. *Concurrent Prolog.* MIT Press, 1987.

[SJG95] V. Saraswat, R. Jagadeesan, and V. Gupta. Default timed concurrent constraint programming. In *Proceedings of Principles of Programming Languages (POPL'95).* ACM, 1995.

[TA94] D. R. Tarditi and A. W. Appel. ML-Yacc user's manual version 2.3. October 1994.

[TJ92] J. P. Talpin and P. Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(2), 1992.

[TJ97] J. P. Talpin and P. Jouvelot. The type and effect dicipline. In *Information and Computation.* Academic Press, 1997.

[Tof90] M. Tofte. Type inference for polymorphic references. In *Information and Computation*, 1990.

[TT97] M. Tofte and J. P. Talpin. Region-based memory management. In *Information and Computation*, 1997.

[Tur96] F. Turbak. First-class sychronization barriers. *ACM International Conference on Functional Programming*, 1996.

[Wad95] P. Wadler. Monads for functional programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925 of *LNCS*. Springer Verlag, 1995.

[WF92] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. In *Information and Computation*, 1992.

[Wik86] C Wikstrom. Distributed programming in Erlang. In *IEEE Computer*, volume 19, August 1986.

[Wik96] C Wikstrom. Implementing distributed real-time control systems in a functional language. In *IEEE Workshop on Parallel and Distributed Real-Time System*, April 1996.

[Wir83] N. Wirth. *Programming in Modula-2*. Springer-Verlag, 1983.

[Wri92] A. K. Wright. Typing references by effect inference. In *European Symposium on Programming*, 1992.