International Journal of Parallel Programming, Vol. 18, No. 2, 1989

Facile: A Symmetric Integration of Concurrent and Functional Programming

Alessandro Giacalone,[1] Prateek Mishra,[1] and Sanjiva Prasad[1]

Received November 1988; revised November 1989

Facile is a symmetric integration of concurrent and functional programming. The language supports both function and process abstraction. Functions may be defined and used within processes, and processes can be dynamically created during expression evaluation. In this work we present two different descriptions of the operational semantics of Facile. First, we develop a structural operational semantics for a small core subset of Facile using a labeled transition system. Such a semantics is useful for reasoning about the operational behavior of Facile programs. We then provide an abstract model of implementation for Facile: the *Concurrent and Functional Abstract Machine* (C-FAM). The C-FAM executes concurrent processes evaluating functional expressions. The implementation semantics includes compilation rules from Facile to C-FAM instructions and execution rules for the abstract machine. This level of semantic description is suitable for those interested in implementations.

KEY WORDS: concurrent programming; functional programming; structural operational semantics; abstract machines; compilation; multi-paradigm languages.

1. INTRODUCTION

Concurrent programming, as exemplified by CCS,[1] CSP[2] or occam,[3] and functional programming, as exemplified by the pure functional parts of

This work has been partially supported by NSF CCR-8704309 and NSF CCR-8706973.

[1] Department of Computer Science, The State University of New York at Stony Brook, Stony Brook, New York 11794-4400. e-mail: ag@sbcs.sunysb.edu, mishra@sbcs.sunysb.edu, sanjiva@sbcs.sunysb.edu.

ML[4] or Scheme,[5] have been recognized as expressive and attractive programming techniques. These techniques apply naturally to rather different classes of problems. The strength of functional programming is its support for abstraction, through the definition of abstract data types and functions.[6] On the other hand, most functional languages cannot express indeterminate computation, modeling of shared resources and persistent objects. Moreover, concurrent programming is necessary when dealing with physically distributed systems, or with problems of synchronization.

Consider, for example, the problem of specifying and implementing an "electronic office" on a physically distributed network. The office allows users to prepare and file documents and transmit them to one another. Each user station is naturally modeled as a collection of datatypes: files, documents, in and out "trays." Functional programming may be used to implement these datatypes as well as the different operations on them. Communication between users and the transfer of files and other information is performed over a network with some topology. The operation of this network poses problems such as synchronization and deadlock. Such problems are best addressed by the use of concurrent programming. Cohen *et al.*[7] discuss this example using two different formalisms: VDM[8] and SCCS[9] are used to describe the individual user functionality and the network implementation, respectively. The specification and implementation of such a complex system is better addressed by the use of a language and environment that supports both functional and concurrent programming.

We have developed Facile (Functional And Concurrent Integrated LanguageE), a language framework that is a *symmetric integration* of functional and concurrent programming, that is, it fully supports both programming styles. By integration, we mean that in any context the user has the choice of using functions and abstract data types, or communicating processes, or any combination of both abstractions. By symmetry we mean that a concept may take the form of a function and be treated as such, but may in fact be implemented as a system of communicating processes. Symmetrically, the internals of a process may be implemented using functions.

One specific issue addressed by the Facile project is the design and implementation of heterogeneous systems: systems constructed by combining components specified using different programming paradigms. Many such systems arise in practice, for instance, the electronic office described above. Traditionally, a heterogeneous system is put together in an *ad hoc* manner by writing low-level code to glue together modules that are already implemented. In contrast, Facile supports the integration of functional and concurrent paradigms at the language level. This provides integrated systems with a well-defined semantics, with particular regard for the interplay between the concurrent and functional components.

In the remainder of this introduction, we summarize the contributions of this paper. A discussion of related work appears in Section 10.

1.1. The Language

Facile is a combination of the typed λ-calculus and a core set of constructs for concurrency extracted from occam[3] and CCS.[1] We chose the typed λ-calculus for the functional component of Facile as it forms the basis for modern languages like ML[4] and has a well understood semantics. Thus, Facile is a statically typed, higher-order language. The concurrent part of Facile includes constructs for programming communicating processes. While semantic foundations of concurrent languages are still the subject of research, promising operational/algebraic approaches have been recently developed.[1, 10–14] These techniques appear to be adequate for a semantic description of the constructs included in Facile.

Unlike ML, Facile does not provide facilities for polymorphic definitions; such an addition to the language is a subject for further study. In further contrast to ML, expressions in Facile may involve interaction with other processes and may in general display nondeterministic behavior.

The model of concurrency underlying Facile is one of processes that communicate by synchronized message passing over typed channels. The processes manipulate data in the functional style. Channels, which constitute the interface between processes, are generated dynamically and are first class values. In particular, channel values can be communicated between processes. Sending and receiving values over channels, channel creation, and process creation are expressions.

In Section 3, we demonstrate the expressiveness of Facile through a set of examples. The examples also serve to illustrate the different features of the language.

1.2. A Structural Operational Semantics for Facile

In Section 5, we give a structural operational semantics for Facile in terms of a Plotkin-style transition system.[15] The operational semantics is given using a "core" syntax, presented in Section 2, that includes the essentials of the language. This produces a compact semantic description and allows us to concentrate on the interesting problems that arise from combining functional and concurrent programming. For improved readability, the examples in Section 3 use an extended syntax; in Section 6 we discuss how the extended language used for the examples is mapped into the core language. Of particular interest is the fact that the core syntax does not include a fixpoint operator. Recursion can be expressed as we can program a typed, call-by-value Y combinator within the core language.

The operational semantics defines the evaluation of function expressions and process expressions in terms of a labeled transition system. The labeled transition system defines two relations: an *evaluates* relation on function expressions and a *derives* relation on behavior expressions. Rules are included to map expression evaluation steps into behavior expression derivation steps. The semantics of function expressions is given in terms of a labeled reduction to values. The semantics of processes is given in terms of a notion of observable behavior.(1) Labels occurring in an expression reduction represent the observable effects of expression evaluation. For example, *send* and *receive* operations on channels are expressions whose evaluation generates observable actions. Since expression evaluation may produce observable effects, a fixed order of evaluation is imposed by the semantic definition.

1.3. The Concurrent and Functional Abstract Machine

In Section 7, we describe an implementation semantics for Facile in terms of a *Concurrent and Functional Abstract Machine* (C-FAM). The C-FAM is a generalization of the SECD machine(16) that supports concurrent processes evaluating functional expressions. The C-FAM described in this paper is an abstract machine that describes the implementation models for a class of languages that integrate functional and concurrent programming, e.g. Facile and Amber.(17)

The machine can be called "functional" since it supports function-closures as first-class values, along with other values that a simple functional language uses. It qualifies to be called "concurrent" since it provides support for process abstraction, dynamic process creation, dynamic channel creation, and synchronized inter-process communication on typed channels. The C-FAM supports process definition using process closures which are values; it also support creation of processes by instantiating process definitions, forking and termination of processes, and the non-deterministic section of a continuation.

In Section 8, we describe a function *compile* that maps Facile programs into programs over a small set of C-FAM op-codes. Following Landin and Cardelli(16, 18) a transition system over machine states describes the operations of the abstract machine.

1.4. Relationship Between Structural and Machine Semantics

The contribution of this paper consists of two different descriptions for the operational semantics of Facile, each at a different level and aimed at readers with different backgrounds and interests. The abstract machine style is easily accessible to those interested in implementations, the structural operational semantics provides the basis for reasoning about Facile programs.

A question naturally arises about the formal connection between the two semantic descriptions. However, establishing such a connection is a technical exercise that lies outside the scope of this paper. In Section 9, we discuss informally some of the interesting points in the relationship between the structural and the machine semantics.

2. CORE SYNTAX OF FACILE

We now present the core syntax of Facile in terms of which the semantics of the language are formalized. The core syntax consists of two recursively defined syntactic categories: *expressions* and *behavior expressions*. There are two reasons for using a two-sorted syntax. First, it supports a separation of concerns between expression evaluation and process execution. Expressions describe data manipulation; their evaluation is deterministic upto "equivalence" of values input or output on channels, and choice of generated channels. Behavior expressions describe the external observable behavior of processes. Problems such as nondeterministic choice, synchronization and deadlock are only of interest at the level of behavior expressions.

A two-sorted syntax also supports a compact and accessible description of dynamic semantics. We could work with one-sorted syntax, as in Amber, but at the cost of much more complicated static and dynamic semantics. Another alternative might be a sophisticated static semantics along the lines of that given for FX(19) which factors out "pure expressions" from "expressions with side-effect."

The only binding operator we use is λ binding, as in the functional style. Unlike CCS, we do not have a restriction operator, which is also a binding-scoping operator. Instead, like PFL(20) and Amber,(17) Facile supports channel creation and treats channels as first-class values.

Notation: $\mathscr{I}$ denotes the set of all identifiers; *id* and x are representative identifiers. $\mathscr{C}$ denotes the set of all primitive constants; c is a representative constant. ■

Definition: (*Types*) Υ, the set of types is defined as:

$$t ::= int \mid bool \mid unit \mid t_1 \to t_2 \mid code \mid t\ chan \quad \blacksquare$$

The type *unit* is a one-element type, and $t_1 \to t_2$ is the usual function type. The type *t chan* consists of channels on which values of type t may be transmitted. The type *code* is the type of a behavior expression packaged as an expression. The static semantics (Section 4) contains the typing rules.

Definition: (*Expressions*)

$$\begin{aligned} exp ::= {} & x \mid c \mid \lambda x \cdot exp \mid exp_1\ exp_2 \mid \mathbf{code}(Beh\_Exp) \\ & \mid \mathbf{spawn}(Beh\_Exp) \mid \mathbf{channel}(t) \mid exp_1 ! exp_2 \mid exp? \quad \blacksquare \end{aligned}$$

The set $\mathscr{C}$ includes integers, booleans *true* and *false*, a distinguished value *triv*, and channel-valued constants. In addition, we assume that $\mathscr{C}$ includes functions such as standard arithmetic operations and the **if-then-else** construct. Expressions include variables, λ-abstraction and application, as in functional languages. In addition, we have the following expressions. The *spawn* expression evaluates to *triv*, but has the external effect of creating a process executing concurrently the specified behavior expression. The *channel* expression evaluates to a *new* channel value. The *send* expression $exp_1 ! exp_2$ evaluates to *triv* and transmits the value of exp_2 on the channel given by exp_1's value. The *receive* expression $exp?$ evaluates to the value received on channel *exp*. The *code* constructor packages a behavior expression into a value; such a value may be transmitted over channels or activated.

Definition: (*Behavior Expressions*)

$$\begin{aligned} Beh\_Exp ::= {} & \mathbf{terminate} && (\textit{inaction}) \\ \mid\ & \mathbf{activate}\ exp && (\textit{process invocation}) \\ \mid\ & Beh\_Exp_1 \parallel Beh\_Exp_2 && (\textit{parallel composition}) \\ \mid\ & Beh\_Exp_1\ \%\ Beh\_Exp_2 && (\textit{alternative}) \quad \blacksquare \end{aligned}$$

terminate is the behavior expression signifying termination of a process. Parallel composition of behavior expressions is a behavior expression. The *activate* behavior expression connects the world of behavior expressions to that of expressions; the expression *exp* is evaluated for its effects, and evaluates to a process script, *i.e.*, a *code* value. The current process is then replaced by one with behavior specified in that process script. The "%" constructor defines the nondeterministic selection of either Beh_Exp_1 or Beh_Exp_2.

3. FACILE EXAMPLES

We now present a few examples that demonstrate some of the key features of Facile. The examples focus on the idea of symmetry and illustrate the expressiveness and flexibility of the language. The examples are written using an extended syntax similar in style to ML. For instance, we use *let* and *fun* with their standard meanings, as well as a sequencing combinator ";" for expressions and behavior expressions. Section 6 describes a mapping from these constructs to the "core" syntax for Facile presented in Section 2.

Example 1 (Fig. 1) defines the script of a process that, when provided a nonnegative integer i on input channel a returns the ith Fibonacci number on channel b, and then terminates. The function *fib* is defined in the usual functional programming style as a recursive function. This function is applied to the integer received on the channel a. The result is then output on channel b. The syntax $b!$ *exp* and $a?$ stand for a send and receive operation over channels b and a. Note that $a?$ is an expression whose value is whatever value is received on channel a; it is not a binding operation as in CSP or Amber.

Example 2 (Fig. 2) defines a process script for the same computation but where the Fibonacci function is implemented using a network of processes. *fib* is still a function, but recursive calls are not "stacked." Instead, for each recursive call to *fib*, a new process is created and the integer argument passed to it on an input channel. The channels generated for each recursive call are *new.* The system of processes that compute the Fibonacci function evolves as a binary tree. The expression *channel*(*int*)

```
proc fib_server(a,b) = let fun fib(i) = if ( i = 0 ) or (i = 1) then 1
                                         else fib(i-1) + fib(i-2)
                       in
                          b ! ( fib( a ? ) )
                       end;
                       terminate
```

Fig. 1. Processes use functions.

```
proc fib_server(a,b) =
       let fun fib(i) = if ( i = 0 ) or (i = 1) then 1
                        else
                          let val (in1, out1) = (channel(int), channel(int) );
                              val (in2, out2) = (channel(int), channel(int) )
                           in
                              spawn( out1 ! (fib(in1 ?)) ; terminate );
                              spawn( out2 ! (fib(in2 ?)) ; terminate );
                              ( in1 ! (i-1) ) ;
                              ( in2 ! (i-2) ) ;
                              ( (out1 ?) + (out2 ?) )
                          end
        in
           b ! ( fib( a ? ) )
       end;
       terminate
```

Fig. 2. Functions use processes.

yields a new channel on which integer values can be communicated. *spawn*(*B*) is an expression which returns the value *triv* locally, but has the effect of creating a new process whose behavior is defined by *B*. Example 2 illustrates how processes can be invoked by functions.

Example 3 (Fig. 3) demonstrates how references or memory cells can be implemented in Facile. Other abstract data types and persistent objects can be implemented in a similar manner. The process script *mem_cell* describes the behavior of a memory location. The formal parameters *get* and *put* are channels for accessing and changing the contents of the location. *mem_cell* uses the % operator, which is an "exclusive or" similar to the CCS sum operator. *mem_cell* offers its contents to any reader on channel *get* (first alternative of %), and then reactivates itself. Or else it allows its contents to be updated (the second alternative) through the channel *put*. The first alternative is triggered by the evaluation of *get*! *contents*, the second by the evaluation of expression *put*?, which yields the new contents.

Following the convention in ML, memory locations are manipulated via three functions: *ref*, *deref* and *assign*. ML's **ref** datatype is represented

```
proc mem_cell(get,put,contents) =
              ( get!contents ;
                 activate mem_cell(get,put,contents) )
              %
               activate let val x = put ?
                        in  mem_cell(get,put,x)
                        end

fun ref(x) = let val (get,put) = ( channel(int), channel(int) )
              in    spawn( activate mem_cell(get,put,x) );
                    (get,put)
             end

fun deref(loc) = let val (i,o) = loc  in     i ?   end

fun assign(loc,x) = let val (i,o) = loc  in    o ! x     end
```

Fig. 3. Memory cells.

here by a pair of channels used to read and update the contents of a memory location. Function *ref* creates a memory cell process, by spawning an *activation* of an application of the *mem_cell* abstraction to the appropriate argument values. It then returns the pair of channels with which to access this process. Given the pair of access channels to a memory location, function *deref* returns the contents of that memory location; *assign* changes the contents to the desired value.

Process scripts are first-class values in Facile. In Example 4 (Fig. 4),

```
fun rsh(node_id, cmd)  =  node_id ! cmd

proc listener(mn)  =  activate ( let val x = mn ?
                                 in    code( activate x ||
                                                activate listener(mn) )
                                  end )
```

Fig. 4. Remote shell.

```
proc chainer(i,o, pcode, n) =
    activate  if n = 1 then   code(activate pcode(i,o) )
              else   let val x = channel(int)
                      in   code( activate pcode(i,x) ||
                                      activate chainer(x,o, pcode, n-1)   )
                     end

proc super_chainer(i,o,simproc,m,n) =
   activate let proc m_chain(a,b) = activate chainer(a,b,simproc,m)
            in    chainer(i,o,m_chain,n)
            end
```

Fig. 5. A process combinator for chaining.

this higher-order facility is used to implement a remote shell command. *rsh* is defined as a function which sends a given command *cmd* over a channel *node_id* identifying a given machine. At the receiving end, the process *listener* receives script x over *mn*. The construct *activate* x creates a process running x in parallel with a reinvocation of *listener*. The "| |" operator denotes parallel composition.

Example 5 (Fig. 5) illustrates how higher-order process combinators can be defined in Facile. *chainer* generates a pipeline of n processes with channels i and o as the external interface. The internal connections in the pipeline are generated in the line *let val* $x =$ *channel*(*int*) of *chainer*. Each process in the pipeline exhibits the behavior specified by the parameter *pcode*: a process script with two channels as parameters.

Of course, *pcode* may itself be a pipeline generator, created using *chainer*, as shown in the process script *super_chainer*, which creates a pipeline of pipelines.

4. STATIC SEMANTICS

The static semantics of Facile is quite standard, and is defined in a structural manner[15] in terms of two relations—the *has-type* relation for expressions, and the *well-typed* relation for behavior expressions. The notational convention we use for these relations is described below, and the inference rules defining these relations inductively are presented in Fig. 6.

1. Constants

$$\frac{\Box}{A \vdash c : t_c}$$

where t_c is the type of constant c.

2. Identifiers

$$\frac{\Box}{A \vdash x : A(x)}$$

3. Function Abstraction

$$\frac{A[x \mapsto t_1] \vdash e : t_2}{A \vdash \lambda x.e : t_1 \rightarrow t_2}$$

4. Function Application

$$\frac{A \vdash e_1 : t_1 \rightarrow t_2 \qquad A \vdash e_2 : t_1}{A \vdash e_1\, e_2 : t_2}$$

5. Channel Creation

$$\frac{\Box}{A \vdash \texttt{channel}(t) : t\ chan}$$

6. Send

$$\frac{A \vdash e_1 : t\ chan \qquad A \vdash e_2 : t}{A \vdash e_1\, !\, e_2 : unit}$$

7. Receive

$$\frac{A \vdash e : t\ chan}{A \vdash e\,? : t}$$

8. Process Script

$$\frac{A \Vdash B}{A \vdash code(B) : code}$$

9. Spawn

$$\frac{A \Vdash B}{A \vdash \texttt{spawn}(B) : unit}$$

10. Terminate

$$\frac{\Box}{A \Vdash \texttt{terminate}}$$

11. Activate

$$\frac{A \vdash e : code}{A \Vdash \texttt{activate}\ e}$$

12. Alternative

$$\frac{A \Vdash B_1 \qquad A \Vdash B_2}{A \Vdash B_1 \,\%\, B_2}$$

13. Composition

$$\frac{A \Vdash B_1 \qquad A \Vdash B_2}{A \Vdash B_1 \parallel B_2}$$

Fig. 6. Static semantics for core Facile.

Definition: (*Type Assignment*) A *Type Assignment* is a finite mapping from identifiers $\mathscr{I}$ to types Υ. A is a typical type assignment. $\varnothing$ is the empty type assignment. ■

The relation $A \vdash e{:}t$ is to be read as "under type assignment A, expression e has type t." The relation $A \Vdash B$ is to be read as "under type assignment A, behavior expression B is well-typed."

$f[f']$ is a finite domain function obtained by augmenting function f with f', such that

$$dom(f[f']) = dom(f) \cup dom(f')$$

$$f[f'](x) = \begin{cases} f'(x) & \text{if} \quad x \in dom(f') \\ f(x) & \text{if} \quad x \in dom(f) - dom(f') \end{cases}$$

5. DYNAMIC SEMANTICS

The Dynamic Semantics consists of two labeled transition systems defining the "evaluates" relation for expression evaluation, and the "derives" relation, for process execution. The labels on transitions roughly correspond to the external effects of the computation performed by a process.

5.1. Preliminaries

Definition: (*Values*) *Val*, syntactic values, is a well-typed subset of *exp* that comprises:

$$c,\ \lambda x.\text{exp},\ \mathbf{code}(Beh\_Exp) \quad \blacksquare$$

A *sort* is a set of typed channels. $\mathscr{S}$ denotes the universal set of all possible typed channels. $\mathscr{P}(\mathscr{S})$ denotes the power set of $\mathscr{S}$. $\mathscr{S}_t$ denotes the subset of $\mathscr{S}$ consisting of channels on which values of type t can be communicated. $\mathscr{K}$ is a typical subset of $\mathscr{S}$, and k is a typical channel in $\mathscr{S}$.

Definition: (Communication Labels) *Comm*, the set of communication labels is defined as:

$$Comm = \{k(v), \overline{k(v)} \mid \exists t.(k \in \mathscr{S}_t, v \in Val, \varnothing \vdash v : t)\} \quad \blacksquare$$

Definition: (Labels) $\mathscr{L}$, the set of labels is defined as:

$$\mathscr{L} = Comm \cup \{\tau\} \cup \{\Phi(B) \mid B \in Beh\_Exp\} \quad \blacksquare$$

The labels $\overline{k(v)}$ represent, respectively, a potential input and output action of a value v over a channel k. The value v should be type-consistent with the channel k. $k(v)$ and $\overline{k(v)}$ are called complementary labels.

The τ label, imported from CCS, represents a "hidden" or internal atomic action, such as a communication between two component processes in a system.

The special $\Phi(B)$ label, generated by a *spawn* expression, is used to mark the creation of a process executing code B. This label enables us to provide a concise treatment of process creation in a symmetrically integrated language.

- $Bcon \subseteq \mathscr{P}(\mathscr{S}) \times Beh\_exp$, is the set of behavior configurations of the form $(\mathscr{K}, B)$ where $\mathscr{K}$ is a sort and B is a behavior expression.
- $Econ \subseteq (\mathscr{P}(\mathscr{S}) \times exp)$, is the set of expression configurations of the form $(\mathscr{K}, e)$ where $\mathscr{K}$ is a sort and e is an expression.

1. Function Application

(a) $$\frac{\mathcal{K}, e_1 \overset{\ell}{\hookrightarrow} \mathcal{K}', e_1'}{\mathcal{K}, e_1\ e_2 \overset{\ell}{\hookrightarrow} \mathcal{K}', e_1'\ e_2}$$

(b) $$\frac{\mathcal{K}, e_2 \overset{\ell}{\hookrightarrow} \mathcal{K}', e_2'}{\mathcal{K}, v\ e_2 \overset{\ell}{\hookrightarrow} \mathcal{K}', v\ e_2'}$$

(c) $$\frac{\Box}{\mathcal{K}, (\lambda x.e)\ v \overset{\tau}{\hookrightarrow} \mathcal{K}, e[x \mapsto v]}$$

2. Send

(a) $$\frac{\mathcal{K}, e_1 \overset{\ell}{\hookrightarrow} \mathcal{K}', e_1'}{\mathcal{K}, e_1\ !\ e_2 \overset{\ell}{\hookrightarrow} \mathcal{K}', e_1'\ !\ e_2}$$

(b) $$\frac{\mathcal{K}, e_2 \overset{\ell}{\hookrightarrow} \mathcal{K}', e_2'}{\mathcal{K}, k\ !\ e_2 \overset{\ell}{\hookrightarrow} \mathcal{K}', k\ !\ e_2'}$$

(c) $$\frac{\Box}{\mathcal{K}, k\ !\ v \overset{\overline{k(v)}}{\hookrightarrow} \mathcal{K}, triv}$$

provided $k \in \mathcal{K}$.

3. Receive

(a) $$\frac{\mathcal{K}, e \overset{\ell}{\hookrightarrow} \mathcal{K}', e'}{\mathcal{K}, e\ ? \overset{\ell}{\hookrightarrow} \mathcal{K}', e'\ ?}$$

(b) $$\frac{\Box}{\mathcal{K}, k\ ? \overset{k(v)}{\hookrightarrow} \mathcal{K}, v}$$

provided $k \in \mathcal{K}$, value v transmittable on k.

4. Channel Creation

$$\frac{\Box}{\mathcal{K}, \mathtt{channel}(t) \overset{\tau}{\hookrightarrow} \mathcal{K} \cup \{k\}, k}$$

where $k \notin \mathcal{K}$ and $k \in \mathcal{S}_t$.

5. Process Creation

$$\frac{\Box}{\mathcal{K}, \mathtt{spawn}(B) \overset{\Phi(B)}{\hookrightarrow} \mathcal{K}, triv}$$

6. Activate

(a) $$\frac{\mathcal{K}, e \overset{\ell}{\hookrightarrow} \mathcal{K}', e'}{\mathcal{K}, \mathtt{activate}\ e \overset{\ell}{\twoheadrightarrow} \mathcal{K}', \mathtt{activate}\ e'}$$

(b) $$\frac{\Box}{\mathcal{K}, \mathtt{activate}\ \mathtt{code}(\ B\ ) \overset{\tau}{\twoheadrightarrow} \mathcal{K}, B}$$

7. Concurrent Composition

(a) $$\frac{\mathcal{K}, B_1 \overset{\ell}{\twoheadrightarrow} \mathcal{K}', B_1'}{\mathcal{K}, B_1 \parallel B_2 \overset{\ell}{\twoheadrightarrow} \mathcal{K}', B_1' \parallel B_2}$$

(b) $$\frac{\mathcal{K}, B_2 \overset{\ell}{\twoheadrightarrow} \mathcal{K}', B_2'}{\mathcal{K}, B_1 \parallel B_2 \overset{\ell}{\twoheadrightarrow} \mathcal{K}', B_1 \parallel B_2'}$$

8. Complementary Communication

$$\frac{\begin{array}{c}\mathcal{K}, B_1 \overset{\ell}{\twoheadrightarrow} \mathcal{K}', B_1' \\ \mathcal{K}, B_2 \overset{\ell'}{\twoheadrightarrow} \mathcal{K}', B_2'\end{array}}{\mathcal{K}, B_1 \parallel B_2 \overset{\tau}{\twoheadrightarrow} \mathcal{K}', B_1' \parallel B_2'}$$

where ℓ and ℓ' are complementary labels.

9. Alternative

(a) $$\frac{\mathcal{K}, B_1 \overset{\ell}{\twoheadrightarrow} \mathcal{K}', B_1'}{\mathcal{K}, B_1\ \%\ B_2 \overset{\ell}{\twoheadrightarrow} \mathcal{K}', B_1'}$$

(b) $$\frac{\mathcal{K}, B_2 \overset{\ell}{\twoheadrightarrow} \mathcal{K}', B_2'}{\mathcal{K}, B_1\ \%\ B_2 \overset{\ell}{\twoheadrightarrow} \mathcal{K}', B_2'}$$

10. Discharge of Φ-labels

$$\frac{\mathcal{K}, B_1 \overset{\Phi(B)}{\twoheadrightarrow} \mathcal{K}', B_1'}{\mathcal{K}, B_1 \overset{\tau}{\twoheadrightarrow} \mathcal{K}', B_1' \parallel B}$$

11. Sort Augmentation

$$\frac{\mathcal{K}, B \overset{\ell}{\twoheadrightarrow} \mathcal{K}', B'}{\mathcal{K} \cup \mathcal{D}, B \overset{\ell}{\twoheadrightarrow} \mathcal{K}' \cup \mathcal{D}, B'}$$

provided $(\mathcal{K}' - \mathcal{K}) \cap \mathcal{D} = \emptyset$.

Fig. 7. Dynamic Semantics.

- $\hookrightarrow \subseteq Econ \times \mathscr{L} \times Econ$ is the "evaluates" relation. $\mathscr{K}, e \overset{l}{\hookrightarrow} \mathscr{K}', e'$ is to be read as "evaluation of expression e, given the sort set $\mathscr{K}$, results in expression e' and sort set $\mathscr{K}'$, on the occurrence of event l."
- $\twoheadrightarrow \subseteq Bcon \times \mathscr{L} \times Bcon$ is the "derives" relation. $\mathscr{K}, B \overset{l}{\twoheadrightarrow} \mathscr{K}', B'$ is to be read as "Given the sort set $\mathscr{K}$, behavior expression B derives behavior expression B' with sort set $\mathscr{K}'$, on the occurrence of event l."

Labels l and l' range over the set of labels $\mathscr{L}$. The letter e, possibly subscripted, stands for any expression, v for values, x for identifiers, k for channel values, and B for behavior expressions.

The "evaluates" and "derives" relations are defined inductively in terms of inference rules. In Fig. 7, we present the rules for "core" Facile, omitting here the evaluation rules induced by the constants in $\mathscr{C}$, and those for derived constructs.

5.2. Evaluation and Derivation Rules

5.2.1. Evaluation Rules for Function Expressions

Rules (1a) and (1b) state that in application $e_1 e_2$, first the operator e_1 is reduced to a λ-form. Then the operator e_2 is reduced to a value v. Rule (1c) defines the binding between formal and actual parameters. The l labeling the transitions in rules (1a) and (1b) represents the labels generated during one step in the reduction of e_1 and e_2. The τ label in rule (1c) says that parameter binding has no observable effect. The initial and final configurations are related by substitution.

Rules (2a-c) describe the evaluation of a send expression $e_1 ! e_2$. First e_1 is reduced to a channel value k using (2a), then e_2 is reduced to a value v by (2b). Finally, by rule (2c), the expression $k!v$ evaluates to value *triv* on the occurrence of the communication event labeled by $\overline{k(v)}$. The proviso ensures the existence of channel k.

Rule (3a) says that when evaluating a receive expression $e?$, the expression e is first reduced to a channel value k. Rule (3b) says that $k?$ may evaluate to any permitted value v with observable effect $k(v)$. The proviso ensures that k is a valid existing channel and that value v can be received over k.

Rule (4), an axiom, describes channel creation. A new channel k, not already in the sort $\mathscr{K}$, is generated and returned as the value of the channel expression. The τ label says that this evaluation does not have any external effect.

Rule (5) says that a *spawn* expression evaluates to value *triv*. In doing so it generates a label that describes the process to be created. In Rule (10), Φ labels are "discharged" resulting in the creation of the new process.

5.2.2. Rules Relating Function Expressions and Behavior Expressions

Rule (6a) states: if label l is produced in one evaluation step for expression e, then l labels the derivation step for behavior expression *activate e*. Rule (6b) says that the activation of a *code* expression yields the corresponding behavior expression, without any observable effect.

5.2.3. Derivation Rules for Behavior Expressions

Rules (7a-b) define the observable behavior of an asynchronous composition of processes in terms of the transitions of component processes. Rule (7a) says that if a process B_1 can make a transition labeled l, then when it is composed with another process B_2, it can still make the same transition. Rule (7b) is the symmetric case.

Rule (8) defines a synchronized communication discipline. For communication to occur, one process should attempt to send a value over a channel and another should attempt to receive over the same channel. This is expressed in the antecedent of rule (8). The τ label in Rule (8) indicates that a communication between two component processes of a system is not observable outside the system. Note that the sorts must match in both clauses of the antecedent.

Rules (9a-b) define the behavior of a process that can either behave as B_1 or as B_2. Rule (9a) says that if a process B_1 can make a transition labeled l to become B'_1, then process $B_1 \% B_2$, can also make the transition labeled l to B'_1. Rule (9b) is the symmetric case.

By rule (5) the dynamic creation of a process B generates a label $\Phi(B)$. Rules (10) and (11) are not syntax-directed. Rule (10) states that whenever a behavior expression B_1 derives a behavior expression B'_1 through a label $\Phi(B)$, we interpret this as B_1 silently deriving $B'_1 \| B$, i.e., a composition of its own derivative and the new process B. Observe that it is adequate to use rule (10) immediately following an application of Rule (6a).

Rule (11) is a derived rule, and can be proven as a theorem from the other inference rules. It says that given a transition possible from a configuration, the sort $\mathscr{K}$ can be augmented with a sort $\mathscr{D}$, but in a disciplined way, to obtain the corresponding transitions. The side condition ensures that the augmenting sort $\mathscr{D}$ does not include channel values that may be generated by the executing process, i.e. $(\mathscr{K}' - \mathscr{K})$.

Rule (11) ensures that distinct processes when composed together cannot independently generate identical channel values. While Rule (4) states that a new channel created by a process cannot be in the process's original sort set, it does not prohibit two different processes from simultaneously generating the same channel value. Rule (11) ensures that we cannot compose together two such processes. It also plays an important role in expanding the sort of a process when a channel value is received by Rule (3), and in obtaining the desired channel sets for communication between processes by Rule (8).

6. MAPPING EXTENDED LANGUAGE INTO CORE LANGUAGE

In the examples, we have used an extended language that includes tupling, sequencing, let-expressions and recursion. These constructs can be expressed within the core language by the mapping shown in Fig. 8.

Observe that recursive functions are expressed using a Y combinator. One of the interesting aspects of the Facile language is that we can program a Y combinator without using recursion or recursive types (Fig. 9). Our Y combinator is well-typed and is a call-by-value combinator. The typed lambda-calculus is known to be *strongly normalizing*; every computation sequence beginning with a typed term T terminates. It is

Construct	Equivalent Form	Comment
(e_1, e_2)	$(\lambda x_1.\lambda x_2.\lambda f.f\ x_1\ x_2)\ e_1\ e_2$	*pairing*
$\texttt{proj}_{1,2}$	$\lambda z.z\ (\lambda x.\lambda y.x)$	*first*
$\texttt{proj}_{2,2}$	$\lambda z.z\ (\lambda x.\lambda y.y)$	*second*
let val $x = e_1$ **in** e_2 **end**	$(\lambda x.e_2)\ e_1$	*let*
fix e	$Y\ e$	*fixpoint*
$e_1\ ;\ e_2$	$(\lambda z.e_2)\ e_1$	*Sequencing*
	where $z \notin FV(e_2)$	
$e\ ;\ B$	**activate** ($\lambda z.code(B)$) e	*Sequenced*
	where $z \notin FV(B)$	*behavior*
fun $f(x) = e$	**val** f = **fix** $\lambda f.\lambda x.e$	*fun definition*
proc $p(x) = B$	**fun** $p(x) = code(B)$	*proc definition*

Fig. 8. Mapping the extended syntax to the core syntax.

```
val Y = lambda g. let a = channel(t)
                  in
                  spawn( a! ( lambda v.  (let x = a?
                                          in
                                            spawn( a!x ; terminate );
                                            g x v
                                          end)
                            ); terminate
                       );
                  let x = a?
                  in
                    spawn( a!x ; terminate );
                    g x
                  end
                  end
```

Fig. 9. A combinator for recursion.

interesting to observe that extending the typed calculus with facilities to spawn processes, as in Facile, leads to a breakdown of this property.

A call-by-value Y combinator is defined by Felleisen.(21)

$$Y_{cbv} = \lambda f.\lambda x.(\lambda g.f(\lambda x.g\,g\,x))(\lambda g.f(\lambda x.g\,g\,x))x$$

However, this combinator is for an untyped lambda calculus and makes explicit use of self-application. It is not possible to type this program without introducing recursive types. In Facile, the higher-order facility combined with the ability to create asynchronously executing processes enable us to define Y, patterned after Thomsen's recursion context.(22) The presence of lambda abstraction in Facile enables us to express Y as a combinator rather than a context. Moreover, Thomsen's CHOCS is a call-by-name calculus and lacks any notion of type.

When applied to a lambda abstraction, $Q \equiv \lambda z.P$, the behavior expression

$$B \equiv (YQ); terminate$$

evolves in the following manner. Q is bound to g on line 1, and the spawn construct creates process B' (lines 3-9) that executes in parallel to B. Evaluation of B continues on line 10. Processes B and B' now "recycle" copies of Q (via the spawn construct) and these are used for recursive invocations.

7. THE CONCURRENT AND FUNCTIONAL ABSTRACT MACHINE

In this section, we describe an abstract machine for a family of languages like Amber, PML and Facile that integrate functional and concurrent programming. The version of C-FAM presented here contains only those operations necessary for expressing core Facile. A version of the C-FAM that includes op-codes for the extended language appears in an earlier paper.(23)

The description of the C-FAM is generally along the lines of the SECD machine.(16) A C-FAM can be viewed as including several concurrently executing SECD-like machines along with the mechanism by which these interact with one another. The description here assumes the existence of an underlying scheduler which selects processes for execution. It is an issue at the level of implementation to ensure that the scheduler is fair.

The C-FAM description differs from the conventional SECD machine description(24) in that the components are not specified in terms of data structures but as abstract structures. In particular, the environment is a function from identifiers to values and not, say, a list of lists. Any realization of the machine will need to express the components as data structures; identifiers appearing in expressions can be replaced by information for accessing these data structures. Implementations of the machine may also differ in many respects: environments may be represented and manipulated differently, and the treatment of recursive definitions may differ. "Op-codes" may be added for recursion, and for extensions to the base language (*e.g.* exceptions, datatypes, pattern-matching). Since recursion is not in core Facile, we do not include any op-codes for recursive definitions. In practice, however, there will be op-codes for this purpose as discussed in the above mentioned paper.(23)

In many respects, the machine is similar to the Functional Abstract Machine (FAM).(18) The machine is built atop an unlimited "heap" of *typed* cells of values which include integers, Booleans, the unit value, closures, etc. Typed channels and *ProcClosures* are among the datatypes supported by the machine—hence it can support dynamic channel creation, channel passing, and dynamic process creation and invocation. The specification of the C-FAM does not define the implementation of the heap, channel creation and garbage collection.

7.1. Configurations and Datatypes

Definition: (Datatypes supported by the C-FAM) $\mathscr{V}$ is the domain of values, or datatypes supported by the C-FAM.

$$\mathscr{V} = int + bool + unit + Closures + ProcClosures + \mathscr{S}$$

We write $\mathscr{V}_t$, where $t \in \Upsilon$, to denote the set of values of type t. ■

int is the set of integers, *bool* consists of the Boolean values **true** and **false**, and *unit* comprises the distinguished value **triv**.

$\mathscr{S}$, *Closures*, *ProcClosures* are the value domains corresponding to the syntactic type expressions t *chan*, $t_1 \rightarrow t_2$, *code*, respectively (where t, t_1, $t_2 \in \Upsilon$).

As before, $\mathscr{S}$ denotes the universal set of all possible typed channels. $\mathscr{S}_t$ denotes the subset of $\mathscr{S}$ consisting of channels on which values of type t can be communicated. $\mathscr{K}$ is a typical subset of $\mathscr{S}$ and k is a typical channel in $\mathscr{S}$.

Closures are the values corresponding to λ-expressions, *i.e.* functions. A closure is a triple consisting of the list of formals, the body of the function, and the environment which gives the values of the free variables appearing in the body.

ProcClosures are similar to closures; they are the process-definition counterparts to closures. A *ProcClosure* is a pair consisting of the body of the process script being defined, and the definition-time environment.

Definition: (Environments) $\mathscr{E}$, the set of all "environments", includes all the finite domain functions from subsets of $\mathscr{I}$ to $\mathscr{V}$. $e \in \mathscr{E}$ is a representative environment. ■

Notation: If L is a sequence, then $\boxed{x}\, L$ is also a sequence, with x as its first element. "[ ]" denotes an empty sequence. ■

The operation of the C-FAM is defined by a transition system whose configurations have two components: a set of active processes, and a set of channels being used by these processes. For both configurations and

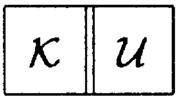


Fig. 10. Components of a C-FAM.

transitions rules, we use a two-dimensional syntax, which we introduce as we proceed. A depiction of a machine configuration is shown in Fig. 10.

$\mathscr{K} \subseteq \mathscr{S}$, is a set of typed channels, called the sort.

$\mathscr{U}$ is a set of labeled quadruples, each representing the $\langle s, e, c, d \rangle$ configuration of a process.

> s, the Argument Stack, is a stack of Values.
>
> $e \in \mathscr{E}$, the Environment, is a finite-domain function from $\mathscr{I}$ to $\mathscr{V}$.
>
> c, the Control-List, is a linear sequence of C-FAM instructions.
>
> d, the Dump, is a stack of (s, e, c) tuples.

Figure 11 shows how a particular quadruple is highlighted in a configuration. We omit the label of the quadruple, the purpose of which is to distinguish different instances of $\langle s, e, c, d \rangle$ tuples. In a machine configuration, we assume that all labels are unique.

7.2. Machine Transitions

The transitions of the C-FAM are defined by the relation "$\Rightarrow$." The transitions depend on the machine state and the current instruction in a quadruple selected by the scheduler. We introduce the op-codes of the machine as we present the transition rules.

Figure 12 shows how a rule is structured. Again, the label of the quadruple in focus is omitted. Instead, we follow the convention that the

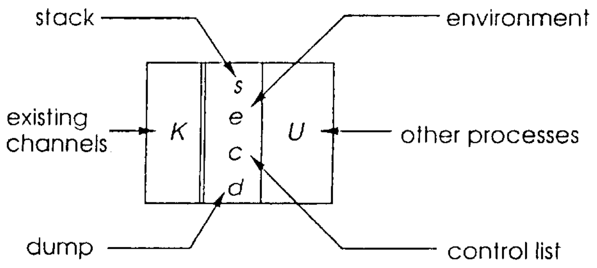


Fig. 11. A quadruple in a C-FAM configuration.

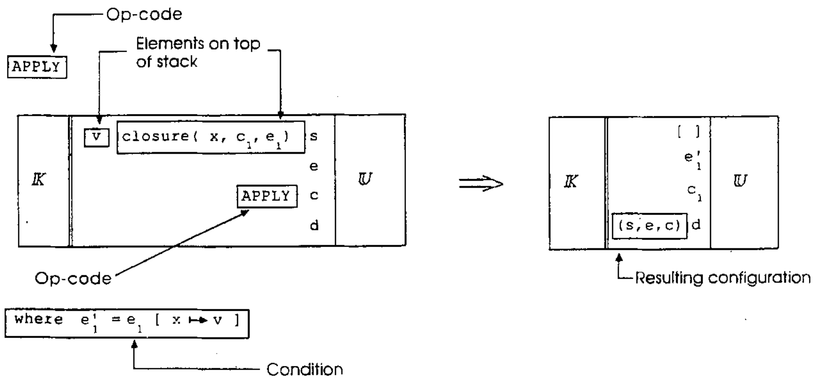


Fig. 12. A transition rule.

quadruples depicted as occupying the same position in the initial and resulting configurations of a transition have the same label.

7.2.1. Process Scripts: The rule in Fig. 13 describes the creation of a ProcClosure from process script c_1, which is an argument to op-code *PROCDEF*. The current environment e is packaged with c_1 to form a ProcClosure that is placed at the top of the stack. We will observe the similarity with the Function Abstraction rule for op-code *ABST* defined later, where a closure is returned.

7.2.2. Process Activation: The Process Activation Rule (Fig. 14) describes the replacement of the current process by one executing the specified process script. The ProcClosure representing the process script is on the top of the stack. In the resulting state, the process has empty stack and dump. Its control list is obtained from the ProcClosure, and the environment is the one packaged in the ProcClosure.

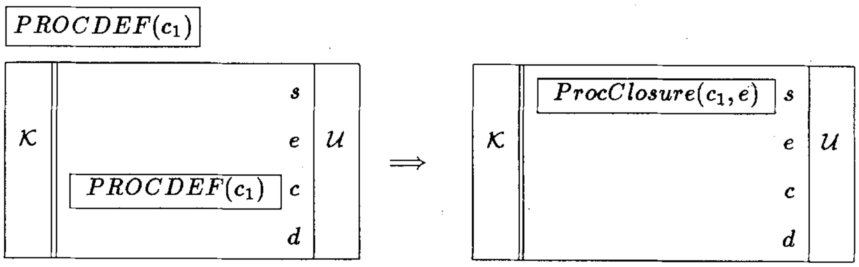


Fig. 13. Process Scripts.

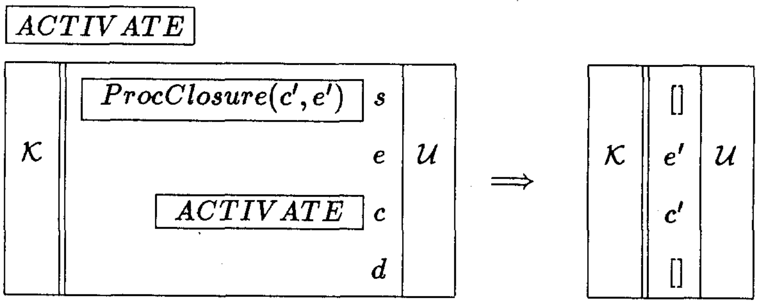


Fig. 14. Process Activation.

7.2.3. Creation of Parallel Processes: The rule in Fig. 15 describes the creation of new processes, which we assume will be labeled with new and unique labels. The stack and dump of the new process are initially empty, the environment is obtained from the process executing the *FORK*, and the control list is obtained from the parameter of the *FORK* op-code.

7.2.4. Non-Deterministic Alternatives: The Alternatives rule (Fig. 16) says that either alternative may be selected as the continuation of the process. For the *i*th alternative to be selected, the precondition of the rule must be satisfied. The precondition says that the machine with a process evaluating the code c_i, can make a sequence of transitions to a new state. We use the relation $\Rightarrow^{\oplus}$ to denote a sequence of $\Rightarrow$ transitions which includes at least one transition caused by an op-code in c_i other than *IDENT*, *PROCDEF*, *ABST*, *CONSTANT* and *TERMINATE*. Then, according to the rule, the machine with a process executing an *ALT* op-code, can make a transition to the same resulting state. Note that the new

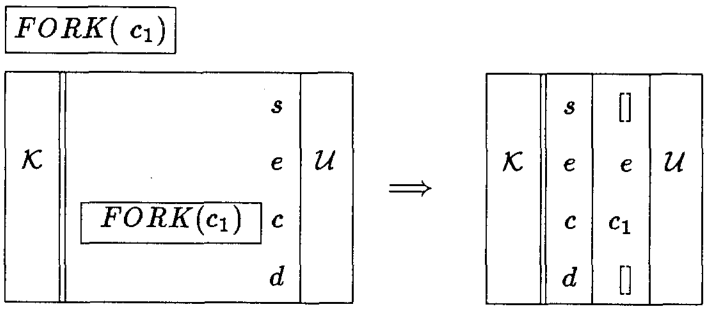
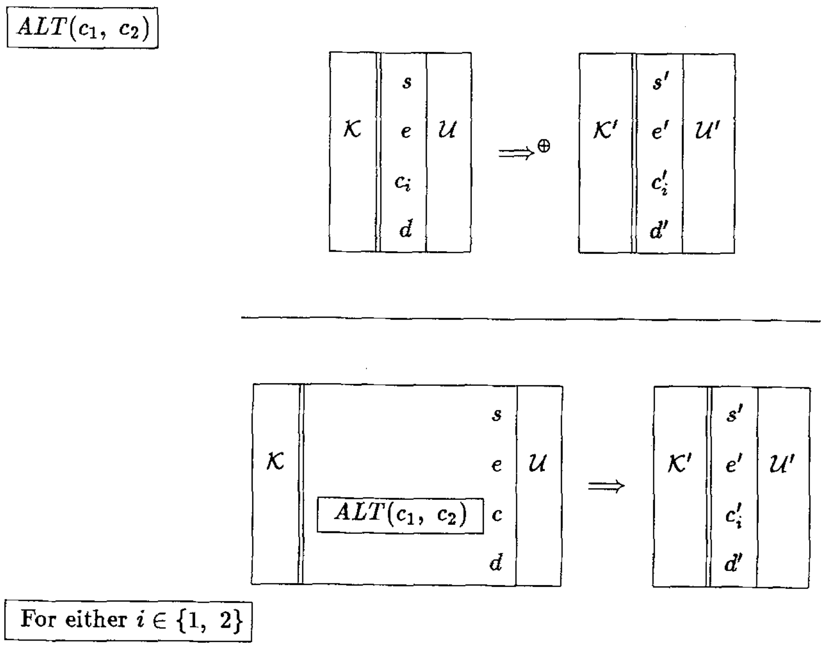


Fig. 15. Creation of Parallel Processes.

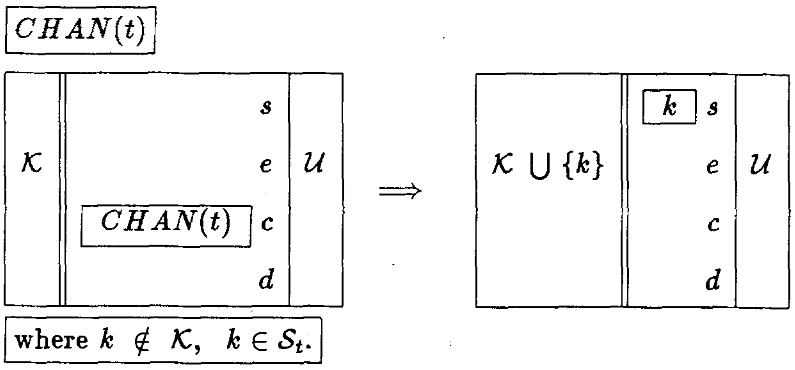


$ALT(c_1, c_2)$
$\mathcal{K}$ | s e c_i d | $\mathcal{U}$
$\Longrightarrow^{\oplus}$
$\mathcal{K}'$ | s' e' c_i' d' | $\mathcal{U}'$
$\mathcal{K}$ | s e $\boxed{ALT(c_1, c_2)}$ c d | $\mathcal{U}$
$\Longrightarrow$
$\mathcal{K}'$ | s' e' c_i' d' | $\mathcal{U}'$
For either $i \in \{1, 2\}$

Fig. 16. Non-Deterministic Alternatives.

C-FAM state may include effects on other processes ($\mathscr{U}'$) as well as the sort ($\mathscr{K}'$).

Any implementation of the ALT rule will require extra machinery in the scheduler in order to create quadruples for each one of c_1 and c_2 and then to keep track of their relationship. As soon as any one of the related quadruples makes a transition outside the list given above (*IDENT*,



$CHAN(t)$
$\mathcal{K}$ | s e $\boxed{CHAN(t)}$ c d | $\mathcal{U}$
$\Longrightarrow$
$\mathcal{K} \cup \{k\}$ | $\boxed{k}$ s e c d | $\mathcal{U}$
where $k \notin \mathcal{K}$, $k \in S_t$.

Fig. 17. Channel Creation.

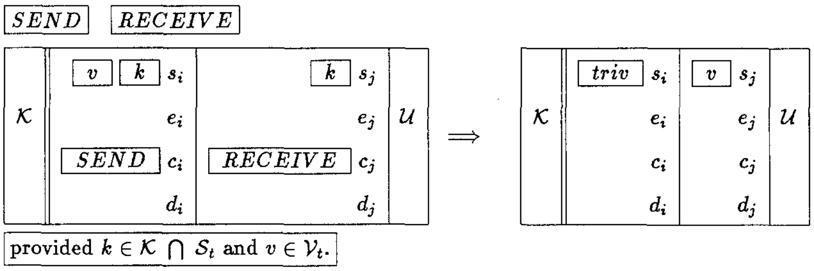


Fig. 18. Communication.

PROCDEF, *ABST*, *CONSTANT* and *TERMINATE*), the other quadruples are removed from the configuration. Further discussion on implementing *ALT* may be found in Section 9.

7.2.5. Channel Creation: The Channel Creation rule (Fig. 17) says that any channel may be returned, on which one may transmit values of the specified type, provided it is not already in the sort $\mathscr{K}$. The sort is augmented with the addition of the new channel.

7.2.6. Communication: The rule for Communication (Fig. 18) says that any two processes may communicate if one of them is attempting to execute op-code *RECEIVE* with channel value k at the top of its stack, and the other is attempting to execute op-code *SEND* with a value v at the top of the stack, with the channel k immediately below. The proviso ensures that the channel k is a valid channel in the sort $\mathscr{K}$, and that value v is transmittable on it. In the resulting state, the receiving process has the value v on its stack, and the sender process has the value **triv**. A process blocks if there is no process it can communicate with.

7.2.7. Process Termination: The rule in Fig. 19 says that when a process

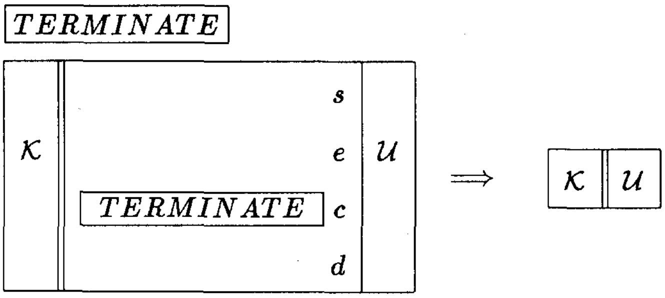


Fig. 19. Process Termination.

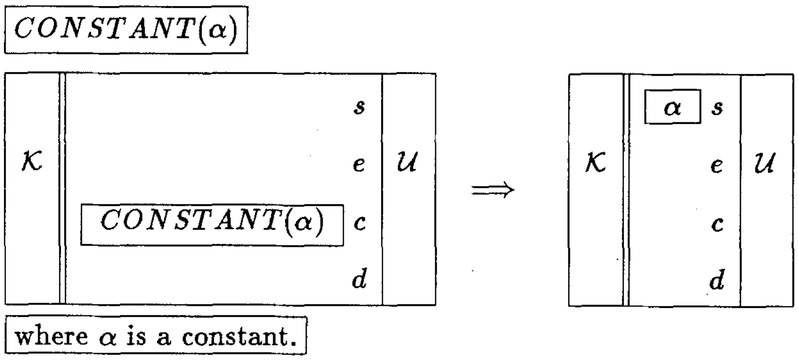


Fig. 20. Constants.

executes a *TERMINATE* op-code, that process quadruple is deleted from the machine configuration.

The remaining rules are essentially similar to those of the SECD machine,(16,24) and deal with the functional aspects such as building closures, application of closures, loading constants and looking up the environment.

7.2.8. Constant: The constant α is loaded onto the stack when the *CONSTANT* op-code is executed (Fig. 20).

7.2.9. Identifiers: When the op-code *IDENT*(x) is executed by a process (Fig. 21), the value v bound to identifier x, obtained from looking up environment e, is loaded onto the stack.

7.2.10. Function Abstraction: As shown in Fig. 22, the *ABST*(x, c') op-

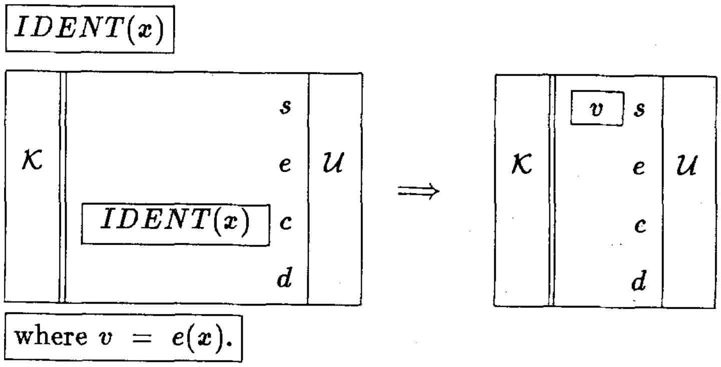


Fig. 21. Identifiers.

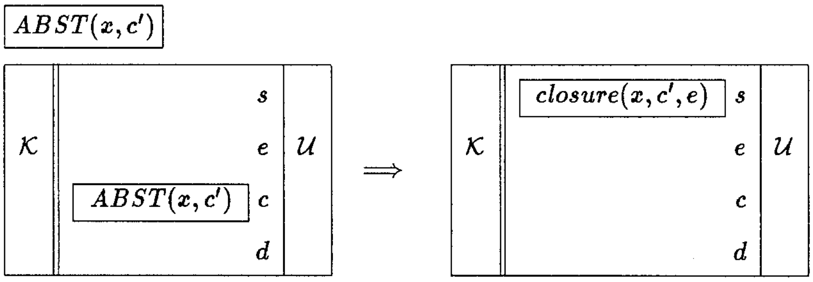


Fig. 22. Function Abstraction.

code results in the formation of a closure, where the closure-formation-time environment e is packaged in with the specified code c', and the formal parameter x. The closure is then placed on the stack.

7.2.11. Function-Call: Function call is achieved by the *APPLY* op-code. In the initial configuration, as shown in Fig. 23, there should be the actual argument value v on top of the stack, and a closure below it. In the resulting configuration, the old stack, environment and continuation are saved on the dump. These represent the point of return from function call. The new stack is empty, the code-list is obtained from the closure, and the new environment is the one packaged into the closure, with the argument v bound to formal parameter x.

7.2.12. Return: Return from a function call is effected by the op-code *RETURN* (Fig. 24). The initial configuration is assumed to have v, the result of the function call on the stack, and a triple (s_0, e_0, c_0) on the dump, placed there by a previous *APPLY* operation. Execution of the *RETURN* op-code pops the triple from the dump, and uses it to restore the

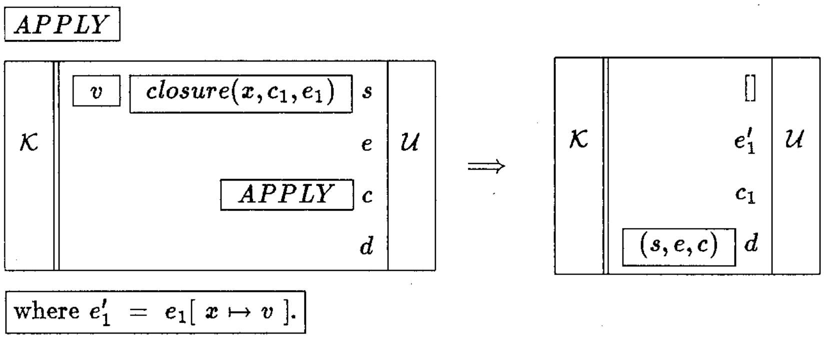


Fig. 23. Function-Call.

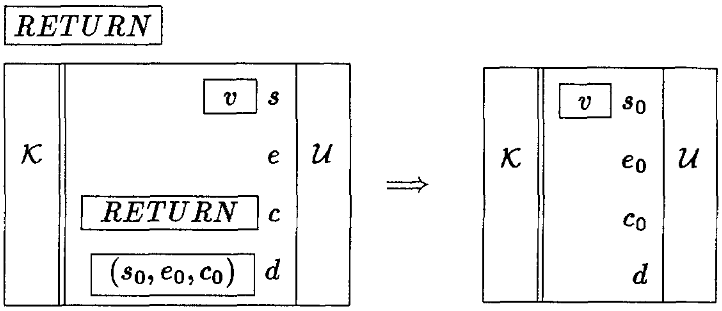


Fig. 24. Return.

state of this process to have environment e_0, code-list c_0 and copies v onto the old stack s_0. Every *APPLY* must be paired with a *RETURN*, which by our method of compilation will be the last op-code in a closure.

8. COMPILING FACILE PROGRAMS

In this section, we describe how a Facile program can be compiled into a sequence of C-FAM instructions. Here we highlight only some interesting features of the compilation, *i.e.* compiling the core language.

Definition: The function *compile* maps Facile constructs to a sequence of C-FAM instructions. It is defined inductively by the following equations. ■

Notation: Sequences are enclosed in brackets, with "|" as the infix cons operator. "@" is the infix append operator. Some examples of sequences are: $[e \mid f]$, $[a]$ @$[c \mid d]$, $[\ ]$. ■

$$compile(B_1 \parallel B_2) = [FORK(compile(B_1)) \mid FORK(compile(B_2)) \mid TERMINATE]$$
$$compile(\text{spawn}(B)) = [FORK(compile(B)) \mid CONSTANT(triv)]$$

In executing the parallel construct, the current process is replaced by a set of concurrent processes. This is compiled into a *FORK* instruction, with the arguments to the *FORK* op-code being the compiled code to be executed by each process. The original process is then terminated by op-code *TERMINATE*.

The *spawn* expression creates a concurrently executing process with the specified code and returns the value *triv*. This too is compiled into a

FORK with. the argument being the compiled argument of the *spawn*. The constant **triv** is then returned by the execution of op-code *CONSTANT*(*triv*).

$$compile(B_1 \,\%\, B_2) = [ALT(compile(B_1), compile(B_2))]$$

The % construct is compiled into an *ALT* op-code with the alternatives compiled to form the arguments to the *ALT* op-code.

$$compile(\mathbf{activate}\, e) = compile(e) \;@\; [ACTIVATE]$$
$$compile(f\, e) = compile(f) \;@\; compile(e) \;@\; [APPLY]$$

In the case of process activation, the expression *e* is compiled, and its evaluation should result in a ProcClosure on which op-code *ACTIVATE* acts. In function application, the operator is compiled, and then the operand. Execution of the C-FAM will result in the argument above a closure, following which the *APPLY* instruction is performed. This will save the return state on the dump. Following the return of a value, the calling time environment, continuation and stack are restored.

$$compile(\lambda x.e) = [ABST(x, (compile(e) \;@\; [RETURN]))]$$
$$compile(code(B)) = [PROCDEF(compile(B))]$$

The λ-abstraction is compiled into an *ABST* instruction, with the formal parameter *x* and the compiled body as arguments. Since function application yields a value, the *op-code RETURN* is appended onto the compiled code of the body. While a closure is created using an *ABST* instruction, a ProcClosure is created with a *PROCDEF* instruction. A *code*

$compile(c)$	$=$	$[CONSTANT(c)]$
$compile(x)$	$=$	$[IDENT(x)]$
$compile(\texttt{channel(}\ t\ \texttt{)})$	$=$	$[CHAN(t)]$
$compile(e_1!e_2)$	$=$	$compile(e_1)@compile(e_2)@[SEND]$
$compile(e?)$	$=$	$compile(e)@[RECEIVE]$
$compile(\texttt{terminate})$	$=$	$[TERMINATE]$

Fig. 25. The remaining compilation rules.

expression is compiled into a *PROCDEF*, with the argument to this instruction being the compiled version of behavior expression *B*.

The other constructs are compiled as shown in the table in Fig. 25.

8.1. An Example

We now show how the Facile expression for the function *ref*, defined in Example 3 of Section 3, is compiled using the compilation rules. In this translation, for the sake of clarity, we have used certain modifications and optimizations of the op-codes and the compilation procedure: (i) The *ABST* op-code has been generalized to take tuples of formal parameters instead of a single identifier. (ii) We use an op-code *TUPLE*(n) to form n-tuples, rather than the encoding in Section 6. (iii) For clarity, the op-code POP, which throws away the topmost element on the stack, is used for expression and behavior sequencing. The resulting C-FAM code is given in Fig. 26.

The function *ref* is compiled into the C-FAM instructions in lines 1-12 of Fig. 26. Its formal parameter is x, and its body compiles into lines 2-12. The op-code *RETURN* is appended at the end of the body of the function, as in line 12. The *let* expression is sugared syntax for a *lambda* form; the let-bound names *get* and *put* appear in line 2, and the pair of new channels they are bound to are compiled into line 10. The binding is accomplished by the *APPLY* in line 11. The body of the *let* expression compiles into

```
1.   [ ABST( (x),
2.          [ ABST( (get,put),
3.                 [ FORK( IDENT(mem_cell) |
4.                        IDENT(get) | IDENT(put) | IDENT(x)  TUPLE(3) |
5.                        APPLY |
6.                        ACTIVATE ) | CONSTANT(triv) |
7.                   POP |
8.                   IDENT(get) | IDENT(put) | TUPLE(2) |
9.                   RETURN ] )
10.          CHAN(int) | CHAN(int) | TUPLE(2) |
11.          APPLY |
12.          RETURN ] ) ]
```

Fig. 26. An example of compilation into C-FAM code.

lines 3-8. The *spawn* expression, compiles into a FORK followed by a *CONSTANT*(*triv*), as shown in lines 3-6. The argument of the *FORK* is the compiled form of an activation (line 6) of a code object. The latter is the result of an application (line 5) of a process-script named *mem_cell* (line 3) to the triple (get, put, x) which compiles into line 4. The *POP* op-code in line 7 is for expression sequencing. The result of the *ref* function is the pair of channels (get, put) created by the instructions in line 8.

9. RELATING THE STRUCTURAL SEMANTICS AND THE C-FAM

In this section we discuss the relationship between the structural semantics and the implementation semantics based on the C-FAM. A formal relationship between structural semantics and an SECD-style abstract machine semantics has been studied by Plotkin[25] for the lambda calculus. There the structural semantics is expressed as a transition system without labels and the execution of the abstract machine is deterministic. In contrast, our structural semantics is based on a labeled transition system and the execution sequence of the C-FAM is not deterministic. Therefore, our discusion concentrates on those constructs that produce nontrivial labels, essentially the concurrent part of the language.

Fundamental to providing a comparison between the labeled transition system and the C-FAM is a comprehension of the role of labels. Roughly, a C-FAM transition corresponds to a τ-labeled transition in the structural semantics. However, the C-FAM transitions are of a finer granularity: a single transition in the structural semantics may correspond to several machine transitions. There are C-FAM transitions which do not correspond to any transitions in the structural rules, *e.g.* environment lookup (*IDENT*) and constant loading (*CONSTANT*).

For some of the constructs of the language, the rules in Fig. 7 have a straightforward correspondence with C-FAM rules. For example, process activation, as defined by Rule (6b), corresponds to the C-FAM rule for *ACTIVATE*, while for channel creation, Rule (4) corresponds to the C-FAM rule for *CHAN*.

Process composition does not correspond to an explicit op-code in the C-FAM. The concurrent execution of processes is expressed by a set of quadruples in a state of the C-FAM. As a consequence, there is no explicit C-FAM analog for Rules (7a-b). In the language, process creation is obtained by activating a composition of processes, or by spawning processes. The C-FAM implements this by the *FORK* op-code. A composition of processes is implemented as a *FORK* for each process. As for spawning, the C-FAM rule for *FORK* captures the combined effect of Rules (5) and (10), *viz* generation and discharge of a $\Phi(B)$ label.

Communication in the labeled transition system is defined by the combined effect of Rules (2c), (3b) and (8). Rules (2c) and (3b) express the *potential* communication of values on channels, as described by the labels. There is no direct C-FAM analog for these rules, since the machine implements not potential but only actual communications. Rule (8) describes actual communication between two processes. It corresponds to the C-FAM communication rule for *SEND* and *RECEIVE*, which describes the transitions made by two processes when they actually communicate a value v on a channel k.

Rules (9a-b) describe a semantics for the % construct where an alternative can be selected provided that it can make a transition. The finer granularity of machine transitions is the reason for the added complexity of the C-FAM ALT rule. The use of the $\Rightarrow^{\oplus}$ relation is necessary to ensure that the selection of an alternative is triggered only by an event that is meaningful in the structural semantics.

One issue in the treatment of the % construct at the C-FAM level arises from the occurrence of phrases of the form $(P \| Q) \% R$, where one summand in an alternative is a parallel composition. If we compile such a phrase we get a mismatch with the structural semantics. Compiling $P \| Q$ we have:

$$[FORK(compile(P)) \mid FORK(compile(Q)) \mid TERMINATE]$$

The problem here is that the *FORK* opcode triggers the selection of the $(P \| Q)$ alternative. In contrast, the structural semantics specifies that only actions of P or Q can trigger the selection. One solution is to transform the source program using algebraic rules ("Expansion Theorems")(1) to replace all occurrences of parallel composition within an alternative by occurrences of sums (%) of single processes. An expansion theorem for Facile is discussed in recent work.(26)

10. RELATED WORK

There are several general purpose languages (*e.g.*, Ada,(27) CHILL,(28) MODULA2(29)) that support some expression evaluation together with facilities for concurrent programming. However, these languages as well as simple languages for concurrent programming such as occam(3) are imperative; computation is by effect on an invisible state rather than on structured data values.(6) They do not support full functional-style programming (*e.g.*, functions are not first class values) and the facilities for concurrency are often restricted (*e.g.*, Tasking in Ada(30)).

CSP,[2,12] ACP[31] and CCS[1] are formalisms for concurrency which explore a structured set of primitives for concurrent programming with synchronized communications between asynchronously executing sequential process. These formalisms are aimed at developing the foundations of a theory of concurrency and hence have not focused on providing support for data manipulation in the applicative style. From these formalisms, concurrent programming languages that support synchronous process execution such as Esterel, MEIJE[32, 33] and SCCS[9] have been developed.

Recently, there have been several extensions to CCS that enhance the set of values that may be communicated. Many of these approaches are based on Milner's suggestion of using indexes families of channels.[9,34] Further refinements have been proposed by Engberg and Nielsen[35] and Astesiano and Zucca[36] for CCS with label passing. Recently, Thomsen introduced a higher-order CCS,[22] where processes are communicable values. A systematic treatment of behaviors as a first-class applicative-style abstract datatype is presented in SMoLCS,[37] an algebraic framework within which concurrent calculi may be defined.

During the past decade there has been interest in parallel interpreters for functional programs. One approach has dealt with extracting the "implicit parallelism" found in pure functional programs. Research in Data Flow, such as the approaches described in a recent survey,[38] has concentrated on exploiting this implicit parallelism. Other researchers have advocated the control of parallelism found in purely functional programs by the use of pragmas.[39,40]

Several approaches that combine functional programming with concurrency are based on Kahn's model[41,42] extended with a pseudo-functional *merge* operator.[43,44] Communication between processes in such languages takes place asynchronously on unbounded buffers or streams. A process is expressed as a function from an input stream to an output stream. Often, unbounded streams are modeled by supporting some form of "lazy" evaluation.[45,46] Other approaches include Flow Graph languages like FGL and Marigold.[47,48]

Turner[49] has described a concurrent functional language, based on Stoye's approach,[50] which takes as its core a lazy functional language. However, this approach differs from the others in that it uses no referentially opaque or indeterminate operators like ML's assignments or a *merge*. Interprocess communication, which is synchronous and through message passing, is managed by an abstract machine.

In a different direction, there has also been interest in the parallelization of programs written in Lisp with side-effects. In Qlisp and Multilisp,[51,52] explicit control constructs for parallelism like *qlambda*, *pcall* and *futures* have been added to Lisp. Connection Machine Lisp[53] organizes parallelism around data structures, rather than control constructs. This idea is also central in Symmetric Lisp[54,55] and in Linda systems.[56] In contrast, systems like CURARE[57] do not use any explicit constructs but instead restructure Lisp programs for concurrent execution.

Kennaway and Sleep use LNET, a language inspired by CCS, to express the parallelism of functional programs.[58] All applicative expressions are treated as processes, and a translation of combinators to LNET processes is presented. Recently, Boundol[59] has presented a concurrent calculus based on CCS to express the lambda calculus.

10.1. Integrated Languages

While the approaches above may be characterized as embedding one or more features of one style of programming in the other style, there are other approaches which integrate a complete concurrent language with a complete functional language. Meira[60] describes a functional language nested within a "meta-level" process language. This approach is also followed in LOTOS,[61] where process computations are expressed in CCS, with the internals of processes written in ACT ONE. The approach supports reasoning at the process execution level as well as functional style reasoning.

Our work differs from these approaches by combining the applicative style of data manipulation with the concurrent style of CCS in a symmetric fashion. A process computes with data in the applicative style, and applicative style expressions may be implemented using processes. The advantage is that we can connect functional level reasoning with process level reasoning.

The language Amber[17] is the closest in spirit to our work. In a sense, the idea of symmetric integration of processes and functions is implicitly present in Amber. Facile renders this idea explicit and can be viewed as a basis for the systematic study of function-process integration.

Reppy motivates the need for generalizing the treatment of interprocess communication in Amber. The language PML[62] introduces the notion of "event values" to express function abstractions involving interprocess communication. Facile, by contrast, does not treat communication events as values, since event values may be obtained by using lambda abstraction. We are not aware however of any formal semantics for Amber or PML.

In the language PFL,[20] process behaviors are expressed in the applicative style; as with SMoLCS-driven concurrent calculi, behaviors are a datatype, which are manipulated using λ abstraction and application. PFL may be considered an applicative version of occam. The continuation-passing style is used to express sequentiality of communication. The operational semantics of PFL is described in a manner similar to CCS as labeled reduction. An important point of difference between Facile and PFL is that in PFL the idea of functions using processes (*e.g.* example 2 in Section 3) has not been explored.

Recently, Nielson(63) has described the language TPL that combines CCS and the typed lambda-calculus. In TPL, evaluation is lazy; the treatment of communication of expressions is closely allied to that in CHOCS.(22) The language, like CCS, includes only static port names. In contrast, Facile has the notion of a channel value which is dynamically created and may be exchanged between processes. Another similarity between TPL and CHOCS is that there is a form of dynamic binding of channel names. In Facile, channel values are bound to names by the static λ-binding. TPL(63) introduces an interesting notion of type for processes, along with a type inclusion relation.

10.2. Implementations

Implementations of functional languages have often been described in terms of the SECD machine.(16,24) This description has served as the basis for abstract machines that are more optimized and implementation oriented such as Cardelli's FAM.(18) Abstract machines for extensions to pure functional languages, *e.g.* the *secd-m* machine(64) and the *Chaos* machine,(65) also derive from the SECD description. In the Categorical Abstract Machine,(66) the approach taken is slightly different from that in Landin's SECD machine.

Abstract machines have also been defined to specify and support the implementation of concurrent languages. For example, the A-Code machine(67) has been used to define the semantics of Ada and CHILL. A *Concurrent Abstract Machine* (CAM)(68) has been used to support an interactive simulation environment based on CCS,(69) and a similar abstract machine is reported by Cardelli.(70)

The C-FAM description in this paper focuses attention on issues of concurrency, omitted from the description of the Chaos machine.(65) Our description is at a more abstract level, without a description of implementations of datatypes, concrete data structures for the environment, etc. The C-FAM differs from the abstract machine style operational semantics used for POOL(71) in that neither the labeled $\langle s, e, c, d \rangle$ quadruples nor their labels are themselves a datatype supported by the C-FAM. This is because while code objects are expressions in Facile, active processes are not. Moreover, there is no concept of sub-process such as that found in object-oriented languages. In Ref. 71, it is suggested that dynamic process creation is simple to express in terms of an abstract machine, but difficult in terms of a structural semantics. Our use of a two-level semantics and the special $\Phi(B)$ label provides a solution to this problem.

11. CONCLUDING REMARKS

We have described the key features of Facile, a powerful language that integrates both functional and process-oriented constructs. We have also introduced its operational semantics at two levels, structural and machine-oriented. Based on the structural semantics, we have recently defined an algebraic semantics for the core language.(26) The algebraic semantics extends the notion of *observable behavior* and *bisimulation* used in CCS.

One technical issue that should be clarified concerns the semantics of the % construct in Facile. We have chosen a simple and general semantics based on the CCS sum operator. There have been several other forms of the alternative construct proposed in the literature, for instance guarded choice in CSP(2) or internal choice.(72) While it is clear that the topic merits study in its own right, in this work we have concentrated on the relationship between concurrent and functional computation. Clearly, the semantics of the Facile % construct may need refinement depending on such issues such as implementational feasibility or semantic clarity.

11.1. The Facile Environment

We see Facile as a practical language for specifying and implementing complex systems. We have begun work on an implementation and plan to develop a programming environment built around the language. This environment will itself be implemented in Facile and will have a graphical user interface. An interactiveng source-level debugging system will allow one to execute system specifications at any stage of refinement.

Currently, a prototype of the interpreter that supports the debugging system has been implemented in ML (2,600 lines of Standard "New Jersey" ML code) on a Sun 3 workstation. The interpreter is based on a variation of the C-FAM called *Augmented C-FAM* (AC-FAM).(73) The configurations of the AC-FAM are essentially those of the C-FAM, except that its code consists of operations on sets of Facile abstract syntax trees, rather than "assembler-level" op-codes. The level at which executions are modeled by the AC-FAM is intermediate between term reduction and C-FAM transitions. Roughly speaking, the AC-FAM implements the reduction semantics but takes from the C-FAM the concept of *environment* as a component of the run-time state. Our viewpoint is that the AC-FAM constitutes a form of semantic specification suitable for interactive debugging.

$$\frac{\begin{array}{c}\boxed{\mathcal{K}}\boxed{\mathcal{U}_1} \overset{*}{\Longrightarrow} \boxed{\mathcal{K}_1}\boxed{\mathcal{U}_1'} \\ \boxed{\mathcal{K}}\boxed{\mathcal{U}_2} \overset{*}{\Longrightarrow} \boxed{\mathcal{K}_2}\boxed{\mathcal{U}_2'}\end{array}}{\boxed{\mathcal{K}}\boxed{\mathcal{U}_1 \cup \mathcal{U}_2} \overset{*}{\Longrightarrow} \boxed{\mathcal{K}_1 \cup \mathcal{K}_2}\boxed{\mathcal{U}_1' \cup \mathcal{U}_2'}}$$

$$\boxed{\text{provided } \mathcal{U}_1 \cap \mathcal{U}_2 = \emptyset \text{ and } (\mathcal{K}_1 \cap \mathcal{K}_2) - \mathcal{K} = \emptyset}$$

Fig. 27. Concurrent Composition of Independent Processes.

11.2. A Distributed C-FAM

Another issue of some interest is whether the interpreter can itself be a distributed system. Note that the C-FAM really specified that processes can either evolve asynchronously through internal evaluation, or communicate with each other in a hand-shaking fashion. In other words, the machine does not specify real parallel/evaluation. Parallel and distributed execution of the C-FAM is expressed by the property depicted by the rule in Fig. 27.

This property says that two concurrently executing independent machines may be combined. The resulting machine can make any series of transitions that the original machines could make, provided (*i*) the processes in the original machines are distinct, and (*ii*) that the sets of channels generated by the original machines while making their transitions are disjoint.

A distributed implementation should ensure that channels generated at different processors are distinct. There also should be facilities for interprocessor sharing of information, since the sort $\mathcal{K}$ is a shared component. A distributed implementation requires robust protocols that ensure sharing of information, as well as the correct application of rules which affect more than one process, such as the rules for *SEND-RECEIVE* and *ALT*.

REFERENCES

1. R. Milner, *A Calculus of Communicating Systems*, Volume 92 of *Lecture Notes in Computer Science*, Springer-Verlag (1980).
2. C. Hoare, Communicating Sequential Processes, *Communications of the ACM*, **21**(8):666–677 (August 1978).
3. *occam Programming Manual*, (1984). Prentice-Hall Series in Computer Science, C.A.R. Hoare (Series Editor).
4. R. Milner, *A proposal for Standard ML*, Internal Report CSR-157-83, University of Edinburgh (1984).

5. The Revised^3 Report on the Algorithmic Language Scheme, AI Memo 848a, MIT (1986): Also in *SIGPLAN Notices*, **21**(12):37–79 (December 1986).
6. J. Backus, Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs, *Communications of the ACM*, **21**(8):613–641 (August 1978).
7. B. Cohen, W. Harwood, and M. Jackson, *The Specification of Complex Systems*, Addison-Wesley (1986).
8. D. Bjørner and C. B. Jones, *The Vienna Development Method: The Meta-Language*, Volume 61 of *Lecture Notes in Computer Science*, Springer-Verlag (1978).
9. R. Milner, Calculi for Synchrony and Asynchrony, *Theoretical Computer Science*, **25**:269–310 (1983).
10. S. Brookes, C. Hoare, and A. Roscoe, A Theory of Communicating Sequential Processes, *J. of the ACM*, **31**(3):560–599 (July 1984).
11. M. Hennessy, *Algebraic Theory of Processes*, MIT Press (1988).
12. C. Hoare, *Communicating Sequential Processes*, Series in Computer Science, Prentice-Hall (1985).
13. G. Plotkin, An Operational Semantics for CSP, In D. Bjørner (ed.), *Formal Description of Programming Language Concepts II*, pp. 199–223, North-Holland, Amsterdam (1983).
14. P. Degano, R. de Nicola, and U. Montanari, A Distributed Operational Semantics for CCS based on Condition/Event Systems, *Acta Informatica* **26**:59–91 (1988).
15. G. Plotkin, *A Structural Approach to Operational Semantics*, Technical Report DAIMI FN-19, Aarhus University (September 1981).
16. P. Landin, The Mechanical Evaluation of Expressions, *Computer Journal*, **6**(4):308–320 (1964).
17. L. Cardelli, Amber. In Cousineau, Curien, and Robinet (eds.) *LNCS 242: Combinators and Functional Programming Languages*, pp. 21–47, Springer-Verlag (1986).
18. L. Cardelli, The Functional Abstract Machine, Technical Report TR-107, Bell Labs (1983).
19. J. M. Lucassen and D. K. Gifford, Polymorphic Effect Systems, In *Conf. Record of the Fifteenth Annual ACM Symp. on Principle of Programming Languages*, pp. 47–57 (January 1988).
20. S. Holmström, *PFL: A Functional Language for Parallel Programming, and Its Implementation*, Programming Methodoly Group 7, University of Göteborg and Chalmers University of Technology (September 1983).
21. M. Felleisen, The Theory and Practice of First-Class Prompts, In *Conf. Record of the Fifteenth Annual ACM Symp. on Principle of Programming Languages*, pp. 180–190 (January 1988).
22. B. Thomsen, A Calculus of Higher Order Communicating Systems, In *Conf. Record of the Sixteenth Annual ACM Symp. on Principle of Programming Languages*, pp. 143–154 (January 1989).
23. A. Giacalone, P. Mishra, and S. Prasad, FACILE: A Symmetric Integration of Concurrent and Functional Programming, In J. Diaz and F. Orejas (eds.), *LNCS 352: TAPSOFT '89*, pp. 184–209, Springer-Verlag, Berlin (March 1989).
24. P. Henderson, *Functional Programming: Application and Implementation*, Prentice Hall International, London (1980).
25. G. Plotkin, Call by name, call by value, and the λ calculus, *Theoretical Computer Science*, **1**:125–159 (1975).
26. A. Giacalone, P. Mishra, and S. Prasad, *Operational and Algebraic Semantics of Facile*, (June 1989) (Manuscript in preparation).

27. Ada Reference Manual, In Ellis Horowitz (ed.), "*Programming Languages: A Grand Tour*" (1983).
28. *CHILL Language Definition: CCITT Recommendation Z.200*, Volume 5, Number 1, (January 1985).
29. N. Wirth, *Programming in MODULA-2. Texts and Monographs in Computer Science*, Springer-Verlag, second, corrected edition (1982).
30. A. Burns, A. M. Lister, and A. J. Wellings, *A Review of Ada Tasking*, Volume 262 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin (1987).
31. J. A. Bergstra and J. W. Klop, Process Algebra for Synchronous Communication, *Information and Control*, **60**:109–137 (1984).
32. G. Berry and L. Cosserat, The ESTEREL Synchronous Programming Language and its Mathematical Semantics, In *LNCS 197: Proc. of the Seminar on Concurrency*, pp. 389–448, Springer-Verlag, Berlin (1985).
33. G. Boudol and D. Austry, Algebre de Processus et Synchronization, *Theoretical Computer Science*, **30**:91–131 (1984).
34. R. Milner, Lectures on a Calculus for Communicating Systems, In *LNCS 197: Proc. of the Seminar on Concurrency*, pp. 197–220, Springer-Verlag, Berlin (1985).
35. U. Engberg and M. Nielsen, *A Calculus of Communicating Systems with Label Passing*, Technical Report DAIMI PB-208, Aarhus University Computer Science Department (1986).
36. E. Astesiano and E. Zucca, Parametric Channels in CCS and Their Applications, In *Proc. of the 2nd Conf. on Foundations of Software Tech. and Theoret. Computer Sci.* (December 1982).
37. E. Astesiano and G. Reggio, SMoLCS-Driven Concurrent Calculi, In *LNCS 249: TAPSOFT '87*, pp. 169–201, Springer-Verlag, Berlin (1987).
38. Arvind and D. Culler, *Data Flow Architectures*, In Traub, Grosz, Lampson, and Nilsson (eds.), *Annual Review of Computer Science*, **1**:225–253, Annual Reviews Inc., Palo Alto, California, USA (1986).
39. P. Hudak and S. Anderson, Pomset Interpretations of Parallel Functional Programs, In G. Kahn, (ed.), *LNCS 274: Functional Programming Languages and Computer Architecture*, pp. 234–256, Springer-Verlag, Berlin (1987).
40. P. Hudak and L. Smith, Para-Functional Programming: A Paradigm for Programming Multiprocessor Systems, In *Conf. Record of the Thirteenth Annual ACM Symp. on Principle of Programming Languages*, pp. 243–254 (January 1986).
41. G. Kahn, The Semantics of a Simple Language for Parallel Programming, In *Proc. of the IFIP Conference*, pp. 471–475 (1974).
42. G. Kahn and D. MacQueen, *Coroutines and Networks of Parallel Processes*, IRIA Report 202 (November 1976).
43. R. Keller, Denotational Semantics for Parallel Programs with Indeterminate Operators, In E. Neuhold (ed.), *Formal Descriptions of Programming Concepts*, pp. 337–366, North-Holland (1978).
44. P. Henderson, Purely Functional Operating Systems, In Darlington, Henderson, and Turner (eds.), *Functional Programming and Its applications*, pp. 177–192, Cambridge University Press (1982).
45. J. Morris and P. Henderson, A Lazy Evaluator, In *Proc. of the Third ACM Conf. on Principles of Programming Languages* (January 1976).
46. D. Friedman and D. Wise, CONS should not evaluate its arguments, in *Third Intl. Colloquium on Automata, Languages and Programming*, Edinburgh University Press (1976).
47. R. Keller, Some Theoretical Aspects of Applicative Multiprocessing, In *LNCS 88: Mathematical Foundations of Computer Science*, pp. 58–74, Springer-Verlag, Berlin (1980).
48. R. B. Kieburtz, Marigold: A Functional Flow-Graph Language, Technical Report

OGC TR CS/E 83-005, Oregon Graduate Center, 19600 N.W. Walker Road, Beaverton, Oregon 97006 (1983).

49. D. Turner, Functional Programming and Communicating Processes, In de Bakker, Nijman, and Treleaven (eds.), *LNCS 259: PARLE Parallel Architectures and Languages Europe*, pp. 54–74, Springer-Verlag, Berlin (June 1987).
50. W. Stoye, *The Implementation of Functional Languages using Custom Hardware*, PhD thesis, Cambridge University Computer Laboratory (1985).
51. R. P. Gabriel and J. McCarthy, Queue-based Multi-processing Lisp, In *Conference Record of the 1984 ACM Symp. on LISP and Functional Programming*, pp. 25–44 (1984).
52. R. H. Halstead, Multilisp: A Language for Concurrent Symbolic Computation, *ACM Transactions on Programming Languages and Systems*, 7(4):501–538 (October 1985).
53. G. L. Steele and W. D. Hillis, Connection Machine LISP: Fine Grained Parallel Symbolic Processing, In *Conf. Record of the 1986 ACM Symp. on LISP and Functional Programming*, pp. 279–297 (August 1986).
54. D. Gelernter and T. London, The Parallel World of Symmetric Lisp, Technical Report, Yale University, New Haven (February 1985).
55. D. Gelernter, S. Jagannathan, and T. London, Environments as First Class Objects, In *Proc. of the Fourteenth Annual ACM Conf. on Principles of Programming Languages*, pp. 98–110 (January 1987).
56. N. Carriero, D. Gelernter, and J. Leichter, Distributed Data Structures in Linda, In *Conf. Record of the Thirteenth Annual ACM Symp. on Principles of Programming Languages*, pp. 236–242 (January 1986).
57. J. R. Larus and P. N. Hilfinger, Restructuring Lisp Programs for Concurrent Execution, In *Proc. of the ACM/SIGPLAN PPEALS* 1988, pp. 100–110 (July 1988).
58. J. R. Kennaway and M. R. Sleep, Expression as Processes, In *Conference Record of the 1982 ACM Symposium on LISP and Functional Programming*, pp. 21–28 (August 1982).
59. G. Boudol, Towards a Lambda-Calculus for Concurrent and Communicating Systems, In J. Diaz and F. Orejas (eds.), *LNCS 351: TAPSOFT '89*, pp. 149–161, Springer-Verlag, Berlin (March 1989).
60. S. Meira, Processes and Functions, In J. Diaz and F. Orejas (eds.), *LNCS 352: TAPSOFT '89*, pp. 286–297, Springer-Verlag, Berlin (March 1989).
61. T. Bolognesi and E. Brinksma, Intro. to the ISO Spec. Language LOTOS, In *Computer Networks and ISDN Systems* 14, pp. 25–59, North-Holland, Amsterdam (1987).
62. J. Reppy, Synchronous Operations as First-class Values, In *Proc. of the SIGPLAN Conf. on Programming Language Design and Implementation*, pp. 250–259 (June 1988).
63. F. Nielson, The Typed λ-Calculus with First-Class Processes, (June 198) (To appear in PARLE '89).
64. S. Abramsky and R. Sykes, Secd-m: a Virtual Machine for Applicative Programming, In J. Jouannaud (ed.), *LNCS 201: Functional Programming Languages and Computer Architecture*, pp. 81–98, Springer-Verlag, Berlin (September 1985).
65. L. Cardelli, The Amber Machine. In Cousineau, Curien, and Robinet, (eds.), *LNCS 242: Combinators and Functional Programming Languages*, pp. 48–70, Springer-Verlag (1986).
66. G. Cousineau, P. L. Curien, and M. Mauny, The Categorical Abstract Machine, In *Proc. of the IFIP Conf. on Functional Programming Languages and Computer Architecture*, pp. 50–64, (September 1985).
67. D. Bjørner and O. Oest, (eds.), *LNCS 98: Towards a Formal Description of Ada*, Springer-Verlag, Berlin (1980).
68. A. Giacalone, A Concurrent Abstract Machine and an Interactive Environment for Simulating Concurrent Systems, Technical Report TR 87/13, Department of Computer Science, SUNY at Stony Brook (December 1987).
69. A. Giacalone and S. A. Smolka, Integrated Environments for Formally Well-Founded

Design and Simulation of Concurrent Systems: A Non-Procedural Approach, *IEEE Transactions on Software Engineering*, **14**(6):787–802 (June 1988).

70. L. Cardelli, An Implementation Model of Rendezvous Communication, In *LNCS 197: Proc. of the Seminar on Concurrency*, pp. 449–457, Springer-Verlag, Berlin (1985).
71. P. America, J. de Bakker, J. N. Kok, and J. Rutten, Operational Semantics of a Parallel Object Oriented Language, In *Conference Record of the Thirteenth Annual ACM Symp. on Principles of Programming Languages*, pp. 194–208, (January 1986).
72. R. de Nicola and M. Hennessy, *CCS without τ's*, In *LNCS 249: TAPSOFT '87*, pp. 138–152, Springer-Verlag, Berlin (1987).
73. T. Chen, *Facile: A User's and Implementor's Guide*, (1989). (To appear as a Technical Report of the Department of Computer Science, SUNY at Stony Brook, New York 11794.)

Printed in Belgium