FACILE: A Symmetric Integration of Concurrent and Functional Programming

Alessandro Giacalone Prateek Mishra Sanjiva Prasad
Department of Computer Science
The State University of New York at Stony Brook
Stony Brook, New York 11794-4400
CSNET: ag@sbcs.sunysb.edu, mishra@sbcs.sunysb.edu, sanjiva@sbcs.sunysb.edu

Abstract

FACILE is a language which symmetrically integrates concurrent and functional programming. It is a typed and statically scoped language. The language supports both function and process abstractions. Recursive functions may be defined and used within processes, and processes can be dynamically created during expression evaluation. Inter-process communication is by hand-shaking on typed channels. Typed channels, functions and process scripts are first-class values.

In this paper, we present the "core" syntax of FACILE. We then describe an abstract machine C-FAM which executes concurrent processes evaluating functional style expressions. The operational definition of FACILE is presented by giving compilation rules from FACILE to C-FAM instructions, and execution rules for the abstract machine. An overview of the FACILE environment is also discussed.

1 Introduction

Concurrent programming, as exemplified by CCS [Mil80], CSP [Hoa85] or occam [INM84], and functional programming, as exemplified by ML [Mil84] or Scheme [Wil85], 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 [Bac78]. Concurrent programming is necessary when dealing with physically distributed systems, or with problems of synchronization and time-dependent behavior.

Most concurrent languages provide only limited support for data abstraction, function definition and general value processing. As a consequence, abstract data types and functions must in general be expressed as collections of processes. This leads to both a loss in clarity and an increase in complexity when reasoning about computations on data values. On the other hand, most functional languages cannot express indeterminate computation, modelling of shared resources, time-dependent behavior and persistent objects. While several proposals to extend the expressive power of functional languages have been made in the past, we believe that none provide the full power of concurrent programming.

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

We have developed FACILE[1], a language framework that is a *symmetric integration* of functional and concurrent programming, that is, it fully supports both programming styles. The model underlying FACILE is one of concurrently executing processes that communicate by synchronous message passing. The processes manipulate data in the functional style. Typed channels, which are data values, constitute the interface through which processes interact.

Our approach is distinct from earlier ones, where one programming style is enriched with constructs that support the other programming style. For example, the approaches described in [Kel78],[AS85] and [Hen82], which are derived from [Kah74,KM76], add the *merge* pseudo-function to a lazy functional language. In contrast, our approach attempts to integrate a full functional language with a full concurrent language.

In this paper we describe the "core" syntax of FACILE, which is a combination of a strongly typed functional language (standard ML) and an expression-oriented concurrent programming language (Occam/CCS). We also describe an operational semantics for FACILE in terms of an abstract implementation. The abstract implementation is based on a definition of a *Concurrent Functional Abstract Machine* (C-FAM), which is a generalization of the SECD machine [Lan64] that supports multi-processing.

Symmetric Integration

FACILE supports both process and function abstractions in a symmetrically integrated fashion. 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. In Section 2 we illustrate these ideas through some examples.

FACILE is more than a language for programming; the ability to choose between expressing a concept in process-oriented or functional terms is especially important at the level of system specification and design. Certain components of a system may be specified in an abstract fashion using functions, while other components may be more naturally described in terms of their temporal behavior as systems of processes. This allows one to take into account requirements that must be met by the structure of system being designed. For example, such requirements may include that the system be physically distributed over a number of processors, that it be implemented on a given architecture or cope with certain synchronization or timing problems.

The symmetric integration of functions and processes in FACILE makes it a powerful language for prototyping and the step-wise refinement of programs. Consider, for example, the problem of specifying a compiler. A compiler may be specified as a process that maps a source program into target code: it accepts the source program over an input channel, applies function *comp* to it and writes out the target code on an output channel. The source and target program may be represented using abstract datatypes common in functional programming: lists, files etc. The function *comp* which accomplishes this mapping could be specified in the functional style but be implemented as a pipeline of processes: lexical scanner, parser, type checker and code generator. In FACILE, the functional specification of *comp* may be replaced by the process implementation without altering the contexts in which *comp* is used. Each component process in the pipeline may itself be implemented as a function or as a combination of functions and processes. Similarly, the various abstract datatypes involved (lists, files, abstract syntax trees) could themselves be represented by processes.

The design of FACILE is part of an ongoing project at Stony Brook concerned with the development of

[1] **F**unctional **A**nd **C**oncurrent **I**ntegrated Languag**E**, pronounced FAH-CHEE-LEH.

interactive environments that support specification/design of complex systems. At present, we are implementing an environment for FACILE which will include a syntax-driven editor and an interactive, graphical source-level debugging system. The user interface of the environment is briefly discussed in the last section.

The Language

As mentioned, FACILE is roughly a combination of ML and an occam-like language. We have taken ML [Mil84] as the functional programming component of FACILE as it is a statically typed, higher-order functional language with excellent facilities for data abstraction, and has a well-understood semantics. The concurrent component of FACILE includes a core set of constructs extracted from occam [INM84] and CCS [Mil80], which provide the necessary support for concurrency. While semantic foundations of concurrent languages are still the subject of research, promising operational/algebraic approaches have been recently developed [BHR84,Hen88,Hoa85,Mil80,Plo82]. These techniques appear to be adequate for a semantic description of the constructs we select.

FACILE exhibits a number of interesting features summarized below.

- It inherits from ML static scoping, static typing and call-by-value semantics.
- In keeping with our goals of symmetry, the syntax is two-sorted: functions and processes. Each syntactic category refers to the other but does not subsume it.
- Inter-process communication is synchronized and takes place over typed channels.
- Channels 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 function expressions.

The Concurrent Functional Abstract Machine

The Concurrent and Functional Abstract Machine is an abstract machine which executes 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.

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 synchronous inter-process communication on typed channels. The C-FAM supports process definition using process closures which are values; it also supports creation of processes by instantiating process definitions with arguments, forking and termination of processes, and the non-deterministic conditional selection of a continuation.

We describe a compiler function that maps FACILE programs into a lower level programs over a small set of C-FAM opcodes. Following Landin and Cardelli [Lan64,Car83] a transition system over machine states describes the operations of the abstract machine.

1.1 Related Work

The language Amber [Car86a] is the closest in spirit to our work. FACILE can be viewed as a generalization of the function-process integration attempted in Amber. The language PML [Rep88], which derives from Amber, describes the use of "event values" to express function abstractions involving inter-process communication. FACILE differs in that it does not have "event values". Recently, Nielson [Nie88] has described a language that combines CCS and the typed lambda-calculus. 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. In [KS82], the parallelism of applicative expressions is expressed by a translation into processes in LNET, a language inspired by CCS. Another approach that relates concurrency with functional-style abstract data type is described in [AR87] : an algebraic framework is presented where behaviors are first-class objects.

There are several general purpose languages (e.g. Ada [ADA83], NIL [SS87], CHILL [CHI85], Modula-2 [Wir82]) that support some expression evaluation together with facilities for concurrent programming. These languages do not support full functional programming (e.g., functions are not first class values) and the facilities for concurrency are often restricted (e.g., Ada).

Implementations of functional languages has traditionally been in terms of the SECD machine [Lan64,Hen80]. This description has served as the basis for abstract machines that are more optimized and implementation oriented such as Cardelli's FAM [Car83]. Abstract machines for extensions to pure functional languages, *e.g.* the *secd-m* machine [AS85] and the *Chaos* machine [Car86b], also derive from the SECD description. A slightly different approach is taken in the Categorical Abstract Machine [CCM85].

Abstract machines have also been defined to specify and support the implementation of concurrent languages. For example, the *A-Code* machine [BO80] has been used to define the semantics of Ada and CHILL, a *Concurrent Abstract Machine* (CAM) [Gia87] has been used to support an interactive simulation environment based on CCS [GS88], and a similar abstract machine is reported in [Car85].

1.2 Organization of the Paper

The remainder of the paper is structured as follows. Sections 2 and 3 contain, respectively, a description of a "core" syntax for FACILE and some examples. Section 4 contains a discussion of the salient features of the C-FAM definition and its execution rules. Section 5 contains a summary of the rules for compiling FACILE into C-FAM codes. The entire definition of C-FAM execution rules is reported in Appendix A. Appendix B contains the entire definition of the compiler rules. Section 6 overviews the FACILE environment and concludes the paper.

2 Syntax

Definition : (IDENTIFIERS) $\mathcal{I}$ is the set of all identifiers. Typical identifiers are represented by x, x_i, *id*. ■

Definition : (TYPES) Υ, the set of type expressions, is defined by the following grammar, where t, t_i are representative types :

$$t \ ::= \ int \mid bool \mid unit \mid (t) \mid t_1 \rightarrow t_2 \mid t \ chan \mid t_1 * ... * t_n \mid t \ proc$$

■

Definition : (EXPRESSIONS) *exp*, the set of function expressions, is defined by the following grammar:

$$
\begin{array}{rcl}
exp & ::= & id \mid constant \mid (exp_1, ..., exp_n) \mid \texttt{project}_{i,n}\ exp \\
& \mid & \texttt{if}\ exp_1\ \texttt{then}\ exp_2\ \texttt{else}\ exp_3 \mid \lambda(id_1, ..., id_n).exp \\
& \mid & exp_1\ exp_2 \mid \Lambda(id_1, ..., id_n).Beh\_Exp \\
& \mid & \texttt{fix}\ (id_1, ..., id_n)\ (exp_1, ..., exp_n) \mid exp_1\ ;\ exp_2 \\
& \mid & \texttt{spawn(}\ Beh\_Exp\ \texttt{)} \mid \texttt{channel(}\ t\ \texttt{)} \mid exp_1\ !\ exp_2 \mid exp\ ? \\
& \mid & exp_1\ +\ exp_2 \mid .....
\end{array}
$$

■

Identifiers are expressions. Constants include integers, booleans *true* and *false*, a special value *triv*, and channel-valued constants. Tuples of expressions are expressions, as also the i^{th} component of a n-tuple. The language contains an `if-then-else` construct for conditional expressions. Function abstractions, *i.e.* λ-forms and function application are also expressions. Recursive functions and process definitions are expressed through the `fix` construct; the tuple of expressions in a `fix` expression must be either λ- or Λ-abstractions.

The Λ-abstraction, also called a Process Script, is the process-level counterpart of the λ-abstraction. The `spawn` expression evaluates to *triv*, but has the effect of creating a process executing the specified behavior expression concurrently with the spawning process. 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*. For the sequential expression exp_1 ; exp_2, expression exp_1 is first evaluated for its effects and then exp_2 is evaluated, with the value of exp_2 returned as the result. The binding of names to values is treated uniformly as λ or Λ bindings.

Definition : (BEHAVIOR EXPRESSIONS) *Beh_Exp*, the set of behavior expressions, is defined by the following grammar:

$$
\begin{array}{rcll}
Beh\_Exp & ::= & \texttt{terminate} & (inaction) \\
& \mid & \texttt{activate}\ exp\ exp & (process\ invocation) \\
& \mid & Beh\_Exp_1 \parallel ... \parallel Beh\_Exp_n & (parallel) \\
& \mid & \texttt{alt}\ exp_1\ \texttt{:-}\ Beh\_Exp_1\ \texttt{\%}\ ... & (alternative) \\
& & \quad \texttt{\%}\ exp_n\ \texttt{:-}\ Beh\_Exp_n\ \texttt{endalt} & \\
& \mid & exp\ ;\ Beh\_Exp & (sequential)
\end{array}
$$

■

`terminate` indicates process termination. An `activate` replaces the current process with one executing an instantiated process script. The first expression must evaluate to a process script, the second to the tuple of arguments for instantiation. Concurrent execution of processes is expressed with the parallel construct. The conditional non-deterministic selection of an alternative is expressed by the `alt` construct. Each alternative is "guarded" by a boolean expression, which can be arbitrarily complex. In the sequential behavior expression, *exp* is evaluated first, for its effects, followed by the execution of *Beh_Exp*.

3 Examples

We now present a few examples of FACILE programs, which illustrate some aspects of symmetry, showing the relation between functions and processes. The use of some constructs is also clarified. We have used `let` and `letrec`, which are "syntactic sugar", to improve readability. The first three examples show different implementations of the *fibonacci* function; the fourth example shows how a ML *ref* cell, a mutable data

structure, can be implemented.

Example 1 defines the script of a process that, when provided a non-negative integer i on input channel "a" returns the i^{th} 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".

```
Λ(a, b).
    letrec
      fib = λ(i).
              if ((i = 0) or (i = 1)) then 1
                 else fib(i − 1) + fib(i − 2)
    in
      b ! (fib(a ?))
    end;
    terminate
```

Example 1

Example 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 λ-abstraction, but recursive calls are not "stacked". Instead, for each recursive call to fib, an asynchronously executing process is created, and the integer argument is passed to it on an input channel. The channels generated for each recursive call are *new*. Example 2 illustrates how processes can be invoked by functions.

```
Λ(a, b).
    letrec
      fib = λ(i).
              if ((i = 0) or (i = 1)) then 1
                 else let
                        (in1, out1) = (channel(int), channel(int));
                        (in2, out2) = (channel(int), channel(int))
                      in
                        spawn( (out1 ! fib(in1?)); terminate );
                        spawn( (out2 ! fib(in2?)); terminate );
                        ( in1!(x − 1) );
                        ( in2!(x − 2) );
                        ( (out1?) + (out2?) )
                      end
    in
      b ! (fib(a ?))
    end;
    terminate
```

Example 2

Example 3 implements the same computation, but with a recursive process script. The *function* fib has been eliminated; the code is quite similar to an implementation in occam.

```
fix(F)( Λ(a,b).
            let x = a?
            in
               if ((x = 0) or (x = 1)) then b!1
               else let
                         (in1, out1) = (channel(int), channel(int));
                         (in2, out2) = (channel(int), channel(int))
                    in
                         spawn(activate F (in1, out1));
                         spawn(activate F (in2, out2));
                         (in1!(x − 1));
                         (in2!(x − 2));
                         b!((out1?) + (out2?))
                    end;
            terminate )
```

Example 3

Example 4 shows how the concept of memory can be implemented in FACILE. Other abstract data types and mutable objects can also be implemented in a similar manner. The example illustrates the use of the process-related constructs **terminate**, **activate** and the non-deterministic **alt**. In particular, the example shows that the "guard" expressions in an **alt** can be very complex.

```
memory ≡ fix(mem)( Λ(get, put, contents).
                        alt ((get!contents); true) :−
                                        activate mem(get, put, contents)
                        % let newcontents = put?
                          in
                                spawn(activate mem(get, put, newcontents));
                                 true
                          end   :−
                                        terminate
                        endalt)

ref ≡ λ(x).
        let
                (read, write) = (channel(int), channel(int))
        in
                ( spawn( activate memory(read, write, x) );
                  (read, write) )
        end

deref ≡ λ(loc).(project_{1,2} loc) ?

assign ≡ λ(loc, newcont). (project_{2,2} loc)!newcont
```

Example 4

Example 4 shows how a memory location can be implemented using processes and channels. The process script

memory describes the behavior of a memory location. The formal parameters *get* and *put* are channels which provide "probes" to access and change the contents of the location. The interface to a memory location is thus represented by a pair of channels used to read and update its contents. Following the convention in ML, memory locations are manipulated via three functions: *ref*, *deref* and *assign*.

Function *ref* activates a *memory* process with the appropriate actual argument values, and returns as the result, the pair of channels with which to access it. 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.

The behavior of the *memory* process script is the following: if an attempt is made to read from channel *get*, it makes the value *contents* available and reactivates itself with the same arguments. If an attempt is made to write on *put*, it reactivates itself with the same access channels but with *contents* replaced by the value *newcontents* received on *put*.

4 The Concurrent and Functional Abstract Machine

The description of the C-FAM is generally along the lines of the SECD machine [Lan64]. A C-FAM can be considered as having several concurrently executing SECD-like machines "embedded" in it, along with the mechanism by which these interact with one another. The machine description differs from the conventional SECD machine description [Hen80] in that the components are not specified in terms of data structures, but as abstract structures; the environment, in particular, is a function from Identifiers to Values, and not, say, a list of lists.

Any realization of the C-FAM will need to express the components as data structures; identifiers appearing in data structures (*e.g.* in a closure) can be replaced by information for accessing these data structures. A realization 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 to handle extensions to the base language (e.g. exceptions, datatypes, pattern-matching).

In many respects, the machine is similar to the Functional Abstract Machine (FAM) [Car83]. The machine is built atop an unlimited "heap" of *typed* cells of values which include integers, booleans, tuples, the unit value, closures, etc. Typed channels and *ProcClosures* are among the "values" 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.

4.1 Notation

Definition : (VALUES) $\mathcal{EV}$ is the domain of expressible values.

$\mathcal{EV} = int + bool + unit + tuples + closures + ProcClosures + \mathcal{S}$ ■

Definition : *int* is the set of integers, *bool* consists of the boolean values **true** and **false**, and *unit* comprises the distinguished value **triv**.
$\mathcal{S}$, *closures*, *ProcClosures*, *tuples* are the value domains corresponding to the syntactic type-expressions t *chan*, $t_1 \rightarrow t_2$, t *proc*, $t_1 \star ... \star t_n$, respectively (where $t, t_1, t_2, t_n \in \Upsilon$).

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

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, *i.e.* they correspond to Λ-expressions. A *ProcClosure* is also a triple consisting of a list of formals, the body of the process being defined, and the definition-time environment. ■

Definition : (ENVIRONMENTS) $\mathcal{E}$, the set of all "environments", includes all the finite domain functions from subsets of $\mathcal{I}$ to $\mathcal{EV}$. $e \in \mathcal{E}$ 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. ■
Notation : $f[f']$ is a finite domain function obtained by augmenting function f with f', such that

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

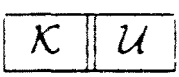
$$f[f'](x) = \begin{cases} f'(x) & \text{if } x \in dom(f') \\ f(x) & \text{if } x \in dom(f) - dom(f') \end{cases}$$

■

Definition : A *sort* is a set of typed channels. ■

4.2 Definition of the Machine

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 transitions rules, we use a two-dimensional syntax.
A machine configuration is depicted below as :



$\mathcal{K} \subseteq \mathcal{S}$, is a sort (set of typed channels).

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

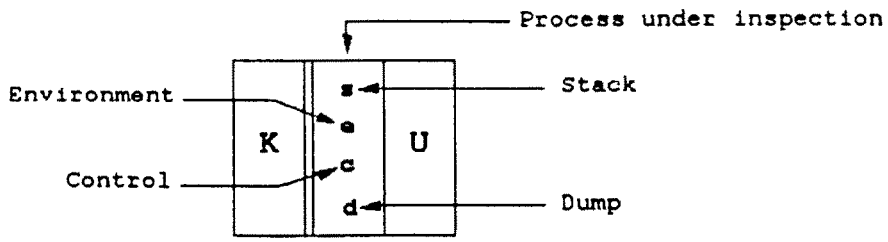
s (Argument Stack) is a stack of Denotable Values

$e \in \mathcal{E}$ is a finite-domain function from Identifiers to Denotable Values (Environment)

c (control-list) is a linear sequence of C-FAM instructions

d (dump) is a stack the elements of which are (i) environments, and (ii) (argument-stack, control-list)–pairs

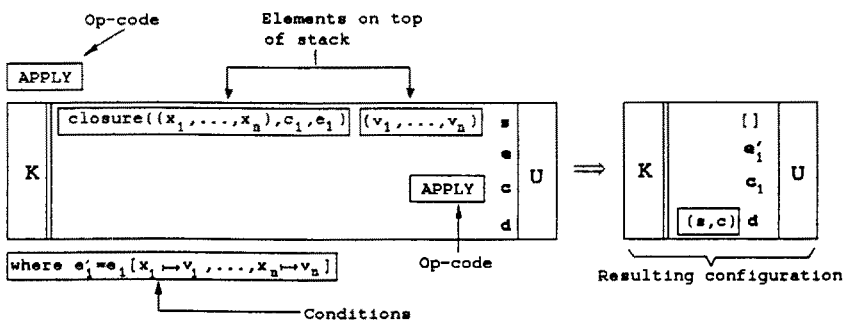
The picture below shows how a quadruple appears in a configuration.



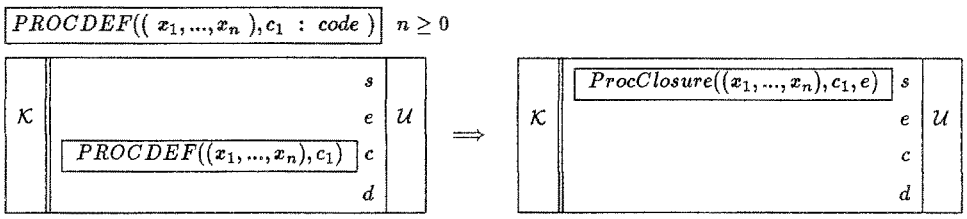
4.3 Transitions

The transitions the machine can make depending on the machine instruction and state are defined by the relation "$\Longrightarrow$", which is defined by the rules in Appendix A.

In this subsection, we discuss some of the rules. The picture below shows how a rule is structured. The rule used in the example is the rule for function application. (Note: An $APPLY$ is preceded by a $SAVE$, according to the compilation rule for function application. So, the current environment e would already have been saved on the dump, according to the execution rules).

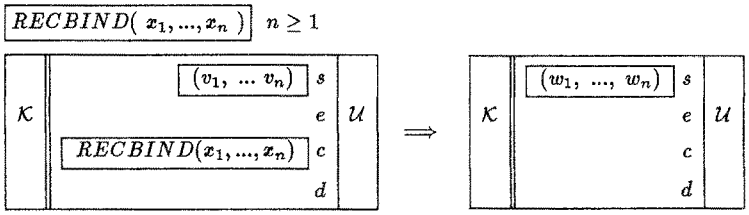


Process Scripts



The rule describes the creation of a ProcClosure from process script c_1, and formal parameters x_1 ... x_n, which are arguments to the op-code *PROCDEF*. The current environment e is packaged with c_1 and x_1 ... x_n to form a ProcClosure that is placed at the top of the stack. Notice the similarity with the Function Abstraction rule for op-code *ABST*, where a closure is returned.

Recursive Environment Augmentation



where

$e' = e[\, x_1 \mapsto w_1, \ \ldots\ , x_n \mapsto w_n]$

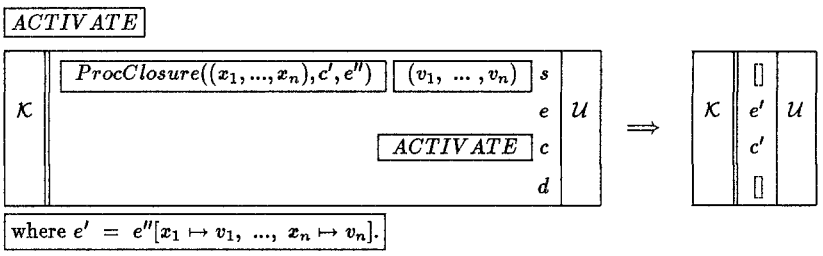
and

w_i is v_i with every instance of a closure $closure(formals_k, c_k, e[f_k])$
appearing in v_i replaced by $closure(formals_k, c_k, e'[f_k])$
and every instance of a proc-closure $ProcClosure(formals_m, c_m, e[f_m])$
appearing in v_i replaced by $ProcClosure(formals_m, c_m, e'[f_m])$

Note the recursively defined environment e'.

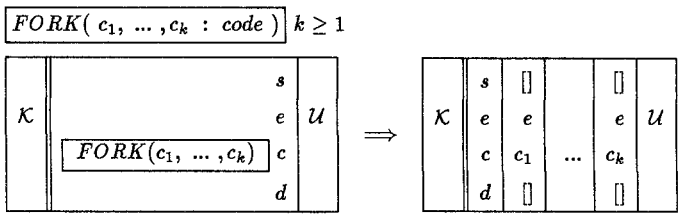
This rule describes the treatment of (mutually) recursive function and process definitions. A tuple of values, each a closure or ProcClosure, is on the stack in the initial state. The op-code *RECBIND* takes as argument the list of identifiers x_1 ... x_n, which are to be recursively bound to these values. The environments packaged in each of these values are updated by building cyclic references into them. The "side-condition" ensures that the cyclic references are built in at the appropriate place.

Process Activation



The Process Activation Rule describes the replacement of the current process by an instantiation of the process script. The ProcClosure is on the top of the stack, with the argument tuple below it. 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, augmented by binding the formal parameters to the actual arguments. Notice the similarity with the Function Application rule.

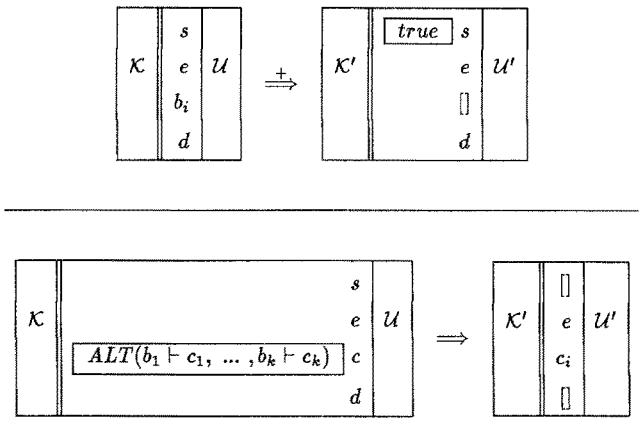
Creation of Parallel Processes



This rule describes the creation of new processes. The stacks and dumps of these new processes are initially empty, the environment obtained from the process executing the $FORK$, and the control lists of these processes are obtained from the parameter of the $FORK$ op-code.

Alternative (Non-Deterministic)

$\boxed{ALT(b_1 : code \vdash c_1 : code \,..\, b_k : code \vdash c_k : code)}\ k \geq 1$

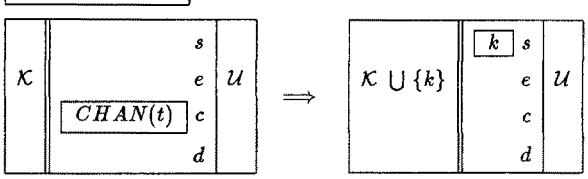


$\boxed{\text{For any } i \in \{1, ..., k\}}$

The Alternative rule says that if the "guard" of any of the alternative continuations for a process can evaluate to **true**, then that alternative may be selected. For the i^{th} alternative to be selected, the precondition of the rule must be satisfied. This says that the machine with a process evaluating the code b_i of the "guard" in a context $\mathcal{U}$ of other processes, $\mathcal{K}$ as the sort, and with stack s, dump d and environment e should make transitions to a state which has the process with value **true** on stack s, the same environment and dump e and d, an empty control list, and with context $\mathcal{U}'$ of other processes, and $\mathcal{K}'$ as the sort. Then, the process executing the ALT op-code selects the i^{th} alternative by making these same transitions. The resulting state has context $\mathcal{U}'$ of other processes, $\mathcal{K}'$ as the sort, but has an empty stack and dump, c_i as the control-list, and e as the environment.

Channel Creation

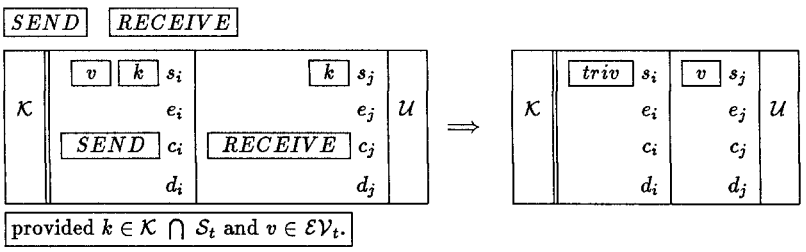
$\boxed{CHAN(t : type)}$



$\boxed{\text{where } k \notin \mathcal{K},\ k \in \mathcal{S}_t.}$

The Channel Creation rule says that that any channel of the specified type may be returned, provided it is not already in the sort $\mathcal{K}$. The sort is augmented with the addition of the new channel.

Communication (Non-Deterministic)



The rule for Communication 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 $\mathcal{K}$, 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.

5 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. The complete definition is given in Appendix B.

Definition : The function *compile* maps FACILE constructs to a sequence of C-FAM instructions. ■

Notation : Sequences are enclosed in (square) brackets, with "|" as the infix cons operator. "@" is the infix append operator. Examples of lists: $[e|f]$, $[a]$ @ $[c|d]$, $[]$. ■

$$
\begin{aligned}
compile(\ Be_1 \parallel ... \parallel Be_n\ ) &= [FORK(compile(Be_1), ..., compile(Be_n))|TERMINATE] \\
compile(\texttt{spawn}(procbody)) &= [FORK(compile(procbody))|CONSTANT(triv)]
\end{aligned}
$$

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 $CONSTANT(triv)$.

$$
\begin{aligned}
compile(\texttt{alt}\ b_1\ \texttt{:-}\ Be_1\ \texttt{\%}..... \quad &= [ALT(\ compile(b_1) \vdash compile(Be_1)\ , ..., \\
\texttt{\%}\ b_n\ \texttt{:-}\ Be_n\ \texttt{endalt}\ ) \quad & \qquad compile(b_n) \vdash compile(Be_n)\ )]
\end{aligned}
$$

The **alt** construct is compiled into an *ALT* op-code with the "guard–alternative" pairs compiled pairwise to form the arguments to the *ALT* op-code.

$$
\begin{aligned}
&compile(\texttt{activate}\ e_1\ e_2\ ) &&= compile(e_2)\ @\ compile(e_1)\ @\ [ACTIVATE] \\
&compile(f\ e) &&= compile(e)\ @\ compile(f)\ @\ [SAVE|APPLY|RESTORE]
\end{aligned}
$$

There is some similarity in the compilation of process activation, *i.e.* the instantiation of a process script, and of function call. The argument is to be evaluated, followed by the evaluation of the operand. In the case of process activation, the ProcClosure is instantiated with the argument tuple by op-code *ACTIVATE*. In function application, the calling environment is first saved, then the closure is applied. Following the return of a value, the calling time environment is restored.

$$
\begin{aligned}
&compile(\lambda(x_1,...,x_n).e) &&= [\ ABST((x_1,...,x_n), compile(e)@[RETURN]\ )\ ] \\
&compile(\ \Lambda(\ x_1,...,x_n\ ).body\ ) &&= [PROCDEF((x_1,...,x_n), compile(body))]
\end{aligned}
$$

The λ- and Λ-Abstractions are compiled in similar ways. Since function application yields a value, the op-code *RETURN* is appended onto the compiled code of the body. A closure is created using an *ABST* instruction, a ProcClosure with a *PROCDEF* instruction.

$$
\begin{aligned}
compile(\texttt{fix}\ (x_1,...,x_n)\ (e_1,...,e_n)) \;=\; & compile(e_1)\ @\ ...\ @compile(e_n)\ @ \\
& [\ TUPLE(n)|RECBIND(x_1,...,x_n)]
\end{aligned}
$$

The recursive **fix** construct is compiled by first compiling the tuple of the body of the definitions, and then filling in the cyclic references with a *RECBIND* instruction.

6 Conclusions and Future Developments

We have described the key features of FACILE, a powerful language that integrates both functional and process-oriented constructs. We have also introduced the C-FAM, an abstract machine to provide an abstract specification of an implementation of our language.

6.1 Distributed C-FAM

Note that the C-FAM really specifies 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 execution/evaluation. Parallel and distributed execution of the C-FAM is expressed by the following property.

Concurrent Composition of Independent Processes

$$\frac{\boxed{\mathcal{K} \parallel \mathcal{U}_1} \overset{*}{\Longrightarrow} \boxed{\mathcal{K}_1 \parallel \mathcal{U}_1'} \qquad \boxed{\mathcal{K} \parallel \mathcal{U}_2} \overset{*}{\Longrightarrow} \boxed{\mathcal{K}_2 \parallel \mathcal{U}_2'}}{\boxed{\mathcal{K} \parallel \mathcal{U}_1 \cup \mathcal{U}_2} \overset{*}{\Longrightarrow} \boxed{\mathcal{K}_1 \cup \mathcal{K}_2 \parallel \mathcal{U}_1' \cup \mathcal{U}_2'}}$$

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

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*.

6.2 The FACILE Environment

As mentioned in the introduction, a project concerned with the implementation of an interactive FACILE environment is in progress. The goal of the FACILE environment is to support and integrate all the activities involved in specifying, designing and implementing a system. The environment will have a graphical user interface. In particular, it will use a two-dimensional syntax for FACILE. Processes are represented by boxes that may enclose sub-systems. Ports can be attached to boxes and joined by communication links. The functional (sequential) part of FACILE will also have a graphical syntax. A syntax-driven editor will allow the "direct" manipulation of system descriptions. An interactive source-level debugging system will allow one to execute system specifications at any stage of refinement. The debugging system will be integrated with an interpreter that is based on a variant of the C-FAM which we describe below.

The semantics of FACILE based on the C-FAM is rather low-level and is suitable mainly as a guide for an implementation. We have also defined a more abstract, "structural" version of FACILE semantics, using Labelled Transition Systems [Plo81]. The semantics derives from the reduction-style semantics of ML and on the notion of *observable behavior* used in CCS. This higher-level semantics is the subject of another paper. It provides a specification of a source-level notion of program execution which is useful, for example, as the basis for developing tools such as source level debuggers.

The interpreter is based on an abstract machine which we will call *AC-FAM* (for Augmented C-FAM). 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; in future work we plan to develop this concept in greater depth.

References

[ADA83] *ADA Reference Manual.* 1983. In Ellis Horowitz, "Programming Languages: A Grand Tour".

[AR87] Egidio Astesiano and Gianna Regio. SMoLCS-Driven Concurrent Calculi. In *LNCS 249 : TAPSOFT '87*, pages 169–201, Springer-Verlag, Berlin, 1987.

[AS85] S. Abramsky and R. Sykes. Secd-m : a Virtual Machine for Applicative Programming. In Jean-Pierre Jouannaud, editor, *LNCS 201: Functional Programming Languages and Computer Architecture*, pages 81–98, Springer-Verlag, Berlin, September 1985.

[Bac78] John 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.

[BHR84] S.D. Brookes, C.A.R. Hoare, and A.W. Roscoe. A Theory of Communicating Sequential Processes. *Journal of the ACM*, 31(3):560–599, July 1984.

[BO80] D. Bjorner and O.N. Oest, editors. *LNCS 98: Towards a Formal Description of ADA. Lecture Notes in Computer Science*, Springer-Verlag, Berlin, 1980.

[Car83] Luca Cardelli. The Functional Abstract Machine. Technical Report Technical Report TR-107, Bell Labs, 1983.

[Car85] Luca Cardelli. An Implementation Model of Rendezvous Communication. In *LNCS 197: Proceedings of the Seminar on Concurrency*, pages 449–457, Springer-Verlag, Berlin, 1985.

[Car86a] Luca Cardelli. Amber. In Cousineau, Curien, and Robinet, editors, *LNCS 242: Combinators and Functional Programming Languages*, pages 21–47, Springer-Verlag, 1986.

[Car86b] Luca Cardelli. The Amber Machine. In Cousineau, Curien, and Robinet, editors, *LNCS 242 : Combinators and Functional Programming Languages*, pages 48–70, Springer-Verlag, 1986.

[CCM85] G. Cousineau, P. L. Curien, and M. Mauny. The Categorical Abstract Machine. In *Proceedings of the IFIP Conference on Functional Programming Languages and Computer Architecture*, IFIP, September 1985.

[CHI85] *CHILL Language Definition: CCITT Recommendation Z. 200.* volume 5 number 1 edition, January 1985.

[Gia87] Alessandro Giacalone. A Concurrent Abstract Machine and an Interactive Environment for Simulating Concurrent Systems. Technical Report TR 87/13, Dept. of Computer Science, SUNY at Stony Brook, December 1987.

[GS88] Alessandro Giacalone and Scott A. Smolka. Integrated Environments for Formally Well-Founded Design and Simulation of Concurrent Systems: A Non-Procedural Approach. *IEEE Transactions on Software Engineering*, June 1988.

[Hen80] Peter Henderson. *Functional Programming: Application and Implementation.* Prentice Hall International, London, 1980.

[Hen82] Peter Henderson. Purely Functional Operating Systems. In Darlington, Henderson, and Turner, editors, *Functional Programming and its applications*, pages 177–192, Cambridge University Press, 1982.

[Hen88] Matthew Hennessy. *Algebraic Theory of Processes.* MIT Press, 1988.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes. Series in Computer Science*, Prentice-Hall, 1985.

[INM84] *occam Programming Manual.* 1984. Prentice-Hall International Series in Computer Science, C.A.R. Hoare (Series Editor).

[Kah74] Gilles Kahn. The Semantics of a Simple Language for Parallel Programming. In *Proceedings of the IFIP Conference*, pages 471–475, IFIP, 1974.

[Kel78] Robert Keller. Denotational Semantics for Parallel Programs with Indeterminate Operators. In E.J. Neuhold, editor, *Formal Descriptions of Programming Concepts*, pages 337–366, North-Holland Publishing Company, 1978.

[KM76] Gilles Kahn and David MacQueen. Coroutines and Networks of Parallel Processes. IRIA Report 202, IRIA, November 1976.

[KS82] J. R. Kennaway and M. R. Sleep. Expressions as Processes. In *Conference Record of the 1982 ACM Symposium on LISP and Functional Programming*, pages 21–28, ACM, August 1982.

[Lan64] P.J. Landin. The Mechanical Evaluation of Expressions. *Computer Journal*, 6(4):308–320, 1964.

[Mil80] Robin Milner. *A Calculus of Communicating Systems.* Volume 92 of *Lecture Notes in Computer Science*, Springer-Verlag, 1980.

[Mil84] Robin Milner. *A proposal for Standard ML.* Internal Report CSR-157-83, University of Edinburgh, 1984.

[Nie88] Flemming Nielson. The Typed λ-Calculus with First-Class Processes. June 1988. Extended Abstract.

[Plo81] G.D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, Aarhus University, September 1981.

[Plo82] G.D. Plotkin. *An Operational Semantics for CSP.* Technical Report CSR-114-82, University of Edinburgh, May 1982.

[Rep88] J.H. Reppy. Synchronous Operations as First-class Values. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 250–259, ACM SIGPLAN, June 1988.

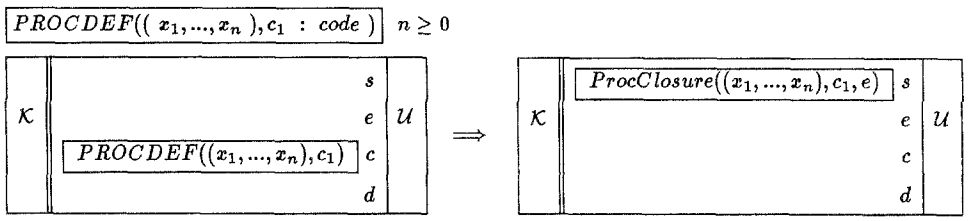
[SS87] Scott A. Smolka and Robert E. Strom. A CCS semantics for NIL. *IBM Journal of Research and Development*, 31(5):556–570, September 1987.

[Wil85] William Clinger et al. The Revised Revised Report on Scheme, or An UnCommon Lisp. AI Memo 848, MIT, Aug 1985.

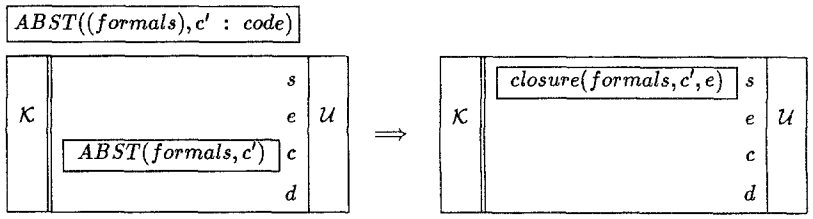
[Wir82] Niklaus Wirth. *Programming in MODULA-2. Texts and Monographs in Computer Science*, Springer-Verlag, second,corrected edition, 1982.

Appendix A : Execution Rules for the C-FAM

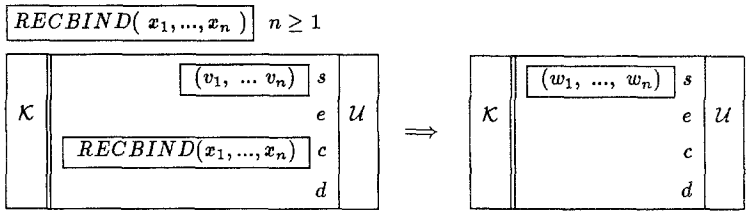
1. **Process Scripts**



2. **Function Abstraction**



3. **Recursive Environment Augmentation**



where

$e' = e[\, x_1 \mapsto w_1, \ \ldots\ , x_n \mapsto w_n]$

and

w_i is v_i with every instance of a closure $closure(formals_k, c_k, e[f_k])$

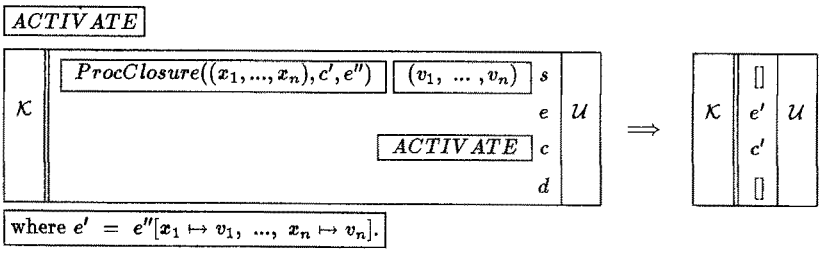
appearing in v_i replaced by $closure(formals_k, c_k, e'[f_k])$

and every instance of a proc-closure $ProcClosure(formals_m, c_m, e[f_m])$

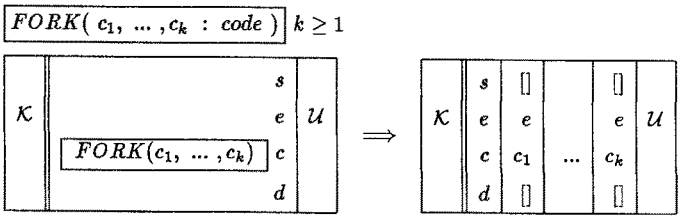
appearing in v_i replaced by $ProcClosure(formals_m, c_m, e'[f_m])$

Note the recursively defined environment e'.

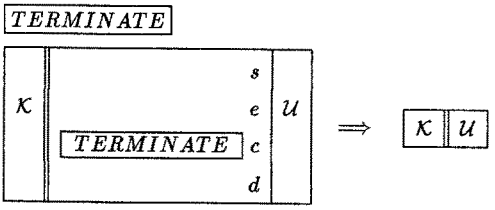
4. **Process Activation**



5. **Creation of Parallel Processes**

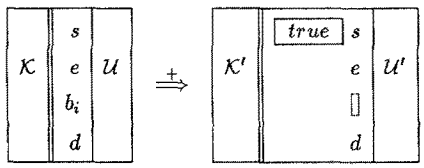


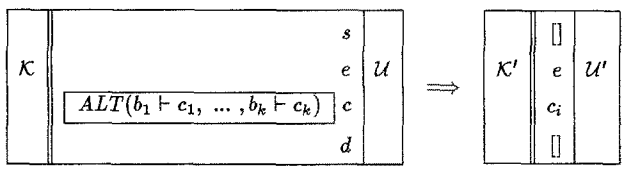
6. **Process Termination**



7. **Alternative (Non-Deterministic)**

$ALT(b_1 : code \vdash c_1 : code \,..\, b_k : code \vdash c_k : code)$ $k \geq 1$

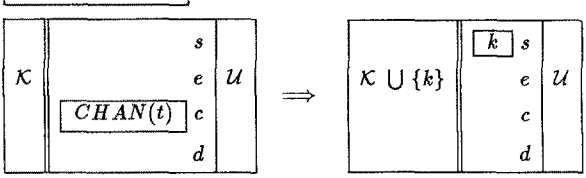




For any $i \in \{1, ..., k\}$

8. **Channel Creation**

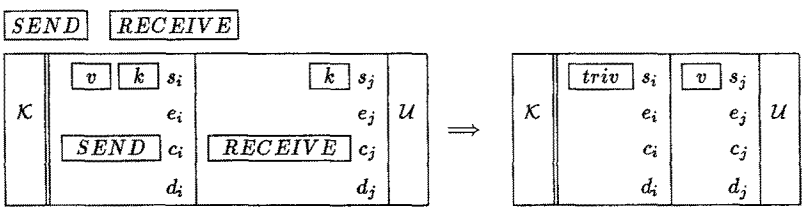
$CHAN(t : type)$



where $k \notin \mathcal{K}$, $k \in chan_t$.

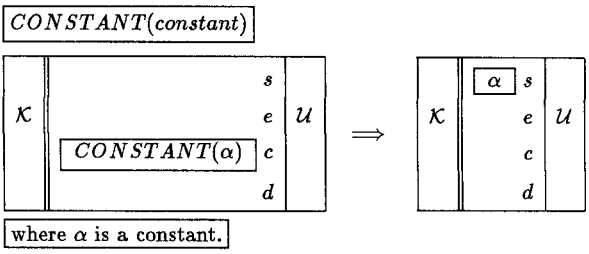
9. **Communication (Non-Deterministic)**

$SEND$ $RECEIVE$

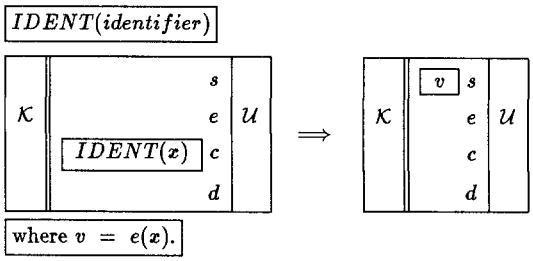


provided $k \in \mathcal{K} \bigcap chan_t$ and $v \in \mathcal{EV}_t$.

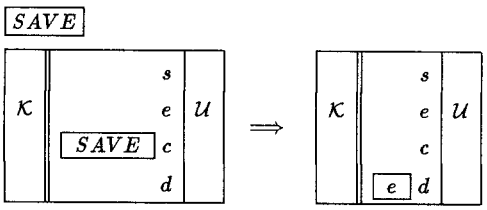
10. **Constants**



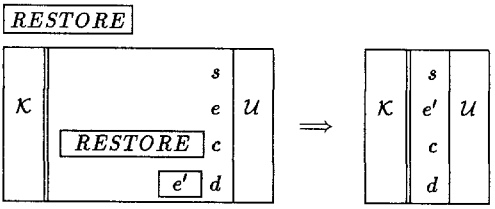
11. **Identifiers**



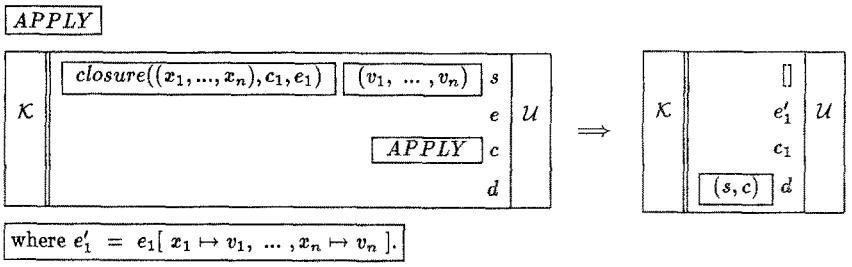
12. **Saving Environments**



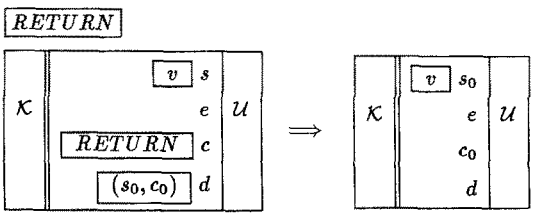
13. **Restoring Environments**



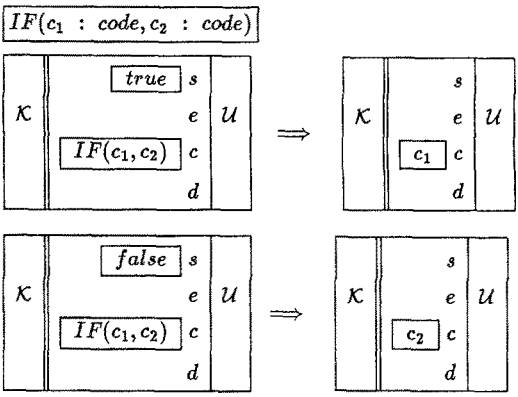
14. **Function-Call**



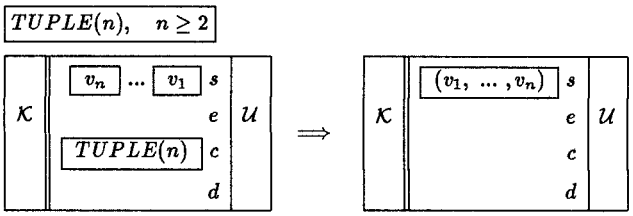
15. **Return**



16. **Conditional**

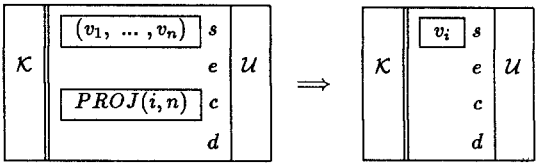


17. **Tupling**

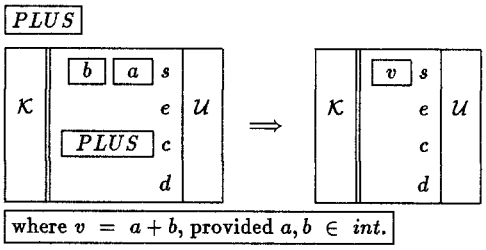


Note: TUPLE(1) can be viewed as identity or a "no-op".
TUPLE(0) can be thought of as CONSTANT(triv)

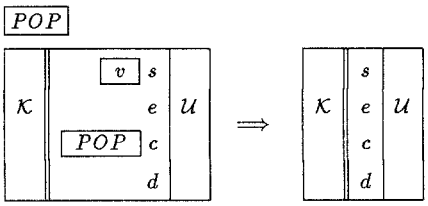
18. **Tuple Projection** $PROJ(i,n),\ 1 \leq i \leq n,\ n \geq 2$



19. **Arithmetic**



20. Stack Manipulation



21. Concurrent Composition of Independent Processes

$$\frac{\boxed{\mathcal{K}}\boxed{\mathcal{U}_1} \Longrightarrow \boxed{\mathcal{K}_1}\boxed{\mathcal{U}_1'} \qquad \boxed{\mathcal{K}}\boxed{\mathcal{U}_2} \Longrightarrow \boxed{\mathcal{K}_2}\boxed{\mathcal{U}_2'}}{\boxed{\mathcal{K}}\boxed{\mathcal{U}_1 \cup \mathcal{U}_2} \Longrightarrow \boxed{\mathcal{K}_1 \cup \mathcal{K}_2}\boxed{\mathcal{U}_1' \cup \mathcal{U}_2'}}$$

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

Appendix B : Compiling FACILE Programs

$$
\begin{array}{lcl}
compile(\ Be_1 \parallel ... \parallel Be_n\ ) & = & [FORK(compile(Be_1),...,compile(Be_n))|TERMINATE] \\
compile(\texttt{alt}\ b_1\ \texttt{:-}\ Be_1\ \%..... & = & [ALT(\ compile(b_1)\ \vdash\ compile(Be_1)\ ,..., \\
\quad \%\ b_n\ \texttt{:-}\ Be_n\ \texttt{endalt}\ ) & & \quad compile(b_n)\ \vdash\ compile(Be_n)\ )] \\
compile(\texttt{terminate}) & = & [TERMINATE] \\
compile(\texttt{activate}\ e_1\ e_2\ ) & = & compile(e_2)\ @\ compile(e_1)\ @\ [ACTIVATE] \\
compile(e\ ;\ Be) & = & compile(e)\ @\ [POP]\ @\ compile(Be)
\end{array}
$$

$$
\begin{array}{lcl}
compile(id) & = & [IDENT(id)] \\
compile(const) & = & [CONSTANT(const)]
\end{array}
$$

$$
\begin{array}{lcl}
compile(\lambda(x_1,...,x_n).e) & = & [\ ABST((x_1,...,x_n),compile(e)@[RETURN]\ )\ ] \\
compile(\Lambda(\ x_1,...,x_n\ ).body\ ) & = & [PROCDEF((x_1,...,x_n),compile(body))]
\end{array}
$$

$$
\begin{array}{lcl}
compile(f\ e) & = & compile(e)\ @\ compile(f)\ @\ [SAVE|APPLY|RESTORE] \\
compile(\texttt{fix}\ (x_1,...,x_n)\ (e_1,...,e_n)) & = & compile(e_1)\ @\ ...\ @compile(e_n)\ @ \\
 & & \quad [\ TUPLE(n)|RECBIND(x_1,...,x_n)]
\end{array}
$$

$$
\begin{array}{lcl}
compile(\texttt{channel}(t)) & = & [CHAN(t)]
\end{array}
$$

$$
\begin{array}{lcl}
compile(\texttt{spawn}(procbody)) & = & [FORK(compile(procbody))|CONSTANT(triv)]
\end{array}
$$

$$
\begin{array}{lcl}
compile(e_1\ !\ e_2) & = & compile(e_1)\ @\ compile(e_2)\ @\ [SEND] \\
compile(e\ ?) & = & compile(e)\ @\ [RECEIVE]
\end{array}
$$

$$
\begin{array}{lcl}
compile(e_1\ ;\ e_2) & = & compile(e_1)\ @\ [POP]\ @\ compile(e_2) \\
compile(\texttt{if}\ b\ \texttt{then}\ e_1\ \texttt{else}\ e_2) & = & compile(b)\ @\ [IF(compile(e_1),compile(e_2))]
\end{array}
$$

$$
\begin{array}{lcl}
compile((e_1,\ ...,\ e_n)) & = & compile(e_1)\ @\ ...\ @\ compile(e_n)\ @\ [TUPLE(n)] \\
compile(\texttt{project}_{i,n}\ e) & = & compile(e)\ @\ [PROJ(i,n)]
\end{array}
$$

$$
\begin{array}{lcl}
compile(e_1 + e_2) & = & compile(e_1)\ @\ compile(e_2)\ @\ [PLUS]
\end{array}
$$