A SEQUENTIAL ABSTRACT MACHINE FOR FLAT CONCURRENT PROLOG

AVSHALOM HOURI AND EHUD SHAPIRO

▷ This paper describes a uniprocessor implementation of Flat Concurrent Prolog, based on an abstract machine and a compiler for it. The machine instruction set includes the functionality necessary to implement efficiently the parallel semantics of the language. In addition, the design includes a novel approach to the integration of a module system into a language. Both the compiler and the emulator for the abstract machine have been implemented and form the basis of Logix, a practical programming environment for Flat Concurrent Prolog. Its performance suggests that a process-oriented language need not be less efficient then a procedure-oriented language, even on a uniprocessor. In particular, it shows that a process queue and process spawning can be implemented as effectively as an activation stack and procedure calling, and thus debunks the "expensive-process-spawn myth". ◁

1. INTRODUCTION

Compilation techniques for implementing logic-programming languages have been investigated extensively. The most important techniques were developed by Warren [18, 20] for the compilation of Prolog. Each clause in the source program is compiled into a set of abstract machine instructions. These instructions compile away much of the run-time overhead of unification and provide the basic control flow required by the language. The methods have significantly improved performance in both time and space; as a result, Prolog is widely accepted as giving performance comparable to Lisp. To test the compilation process and to give a level of portability an emulator for the abstract machine can be built. This emulator is generally written in a lower-level implementation language.

Address correspondence to Ehud Shapiro, Department of Applied Mathematics and Computer Science, The Weizmann Institute, Rehovot 76100, Israel.
Received 15 July 1986; accepted 23 November 1987.

However, Prolog does not explicitly provide the notion of concurrency. To add this capability to logic programming, a family of languages have evolved which originated in the Relational Language of Clark and Gregory [1, 2, 10, 17]. These languages have a parallel semantics and use dataflow synchronization with stream-oriented communication.

Flat Concurrent Prolog (FCP) [6, 11] is a simple concurrent logic-programming language. Its simplicity lies in the nonhiearchical structure of its process and data environment. The language, like the concurrent logic languages mentioned above, incorporates nondeterministic goal selection, committed-choice nondeterminism in clause selection, dataflow synchronization, and stream-based communication.

The guards of FCP clauses are only primitive actions, in contrast to most of the other languages. This provides two major benefits: only a single environment needs to exist during execution, and there is no need to distribute the commit operation. These simplifications allow nondeterminism to be simulated by a simple iterative deterministic algorithm. This algorithm was originally designed and developed in an interpreter-based implementation of FCP [6]; however, this implementation was too slow for practical use. The algorithm admits a simple transformation from the source code to abstract machine code and provides an opportunity for efficient compilation.

The abstract machine described in this paper utilizes a number of novel techniques to compile the parallel semantics and provides a novel approach to the implementation of modules. Its instruction set implements the parallel and nondeterministic features of the language.

The implementation has been compared with one of the fastest commercially available Prolog compiler. Although slower in absolute terms, it should be recognized that the emulator is written in C and a variety of straightforward optimizations are not yet utilized. In contrast, the Prolog emulator is written largely in machine code and uses a heavily optimized compiler.

The implementation forms the lower end of the Logix system [14], an FCP program development environment. Logix is in regular use at the Weizmann Institute and several other places. It has been used to develop several applications, including the Logix system itself, whose total size exceeds 30,000 lines of FCP source code at the time of writing.

2. LANGUAGE DESCRIPTION

This section provides a brief description of the language syntax and informal semantics.

2.1. *Definitions*

An FCP *clause* is a guarded Horn clause of the form

$$H \leftarrow G_1, G_2, \ldots, G_n | B_1, B_2, \ldots, B_m, \qquad m, n \geq 0,$$

where H is the *head* of the clause, each G_i is a system-defined test called a *guard*, and each B_i is a general *body goal.*

An FCP *program* is a finite set of clauses.

An FCP *term* is either a variable or a structure. A *variable* is a single assignment variable, which may have two types of occurrences: a *writable* occurrence by which the variable can be assigned, and a *read-only* occurrence that cannot be used for assigning the variable.

A *structure* is either a *constant* or a *compound* structure, which is a composition of terms.

2.2. Semantics

An informal operational semantics for FCP is described in [11]. A general overview follows.

Each state of the computation consists of a multiset of processes which form the resolvent and a program.

A possible state transition involves rewriting *some* process from the resolvent using *some* clause of the program. Rewriting a process using a clause consists of the following operations:

(1) The process is unified with the head of the clause.

(2) The guards are evaluated with respect to terms in the goal.

(3) After the unification and guard evaluation succeed, the process is replaced in the resolvent by the processes in the body of the clause, and variables in the resolvent are assigned in accordance with the unifying substitution.

The language employs read-only unification as the basic operation. Read-only unification is an extension of standard unification to read-only variables. The read-only unification of two terms containing read-only variables can either succeed, fail, or suspend. A suspended unification may succeed or fail later, as new bindings are produced by concurrently executing unifications. For a full definition of read-only unification see [11, 14].

A *success* state is a state in which the resolvent is empty.

A *deadlock* state is a state in which no reduction transition is possible.

A *failure* state is a terminal state reached when a failing process is encountered. The computation is just; hence when a process is continuously enabled, it will eventually be reduced.

No order is imposed on the processes to be reduced; thus a computation proceeds by nondeterministic process selection. If there are independently reducible goals, this nondeterminism enables the reduction of processes in parallel; the resolvent may be regarded as *And-parallel*. In addition, all attempts to perform rewriting using a clause may be carried out in any arbitrary order; thus clause selection is based on *Or-nondeterministic* choice.

3. ABSTRACT MACHINE CONCEPTS

The abstract machine description is derived below from the semantic properties of the language, and its basic properties are outlined.

3.1. Realizing Nondeterministic Process Selection

The set of active processes is realized by a queue of processes. Nondeterministic process selection can be realized by a number of process reduction attempts. Each reduction attempt may have one of three possible outcomes:

Success. The process is replaced in the process queue by the clause body.

Suspension. The process should be retried later.

Failure. The process can never be reduced. The abstract machine enters a terminal failure state.

The abstract machine cannot guess which processes can reduce in a given state; thus the first process in the queue is dequeued and a reduction attempt is made. If the process cannot be reduced at this point, it can be enqueued again and tried later. This scheme, called busy waiting, achieves the correct behavior but is inefficient.

To avoid processes from being repeatedly dequeued and enqueued to the process queue, a suspension mechanism is used. When a reduction attempt suspends, the process is associated with the variables whose instantiation can cause it to succeed later. Upon instantiation of any of these variables the process is enqueued to the process queue.

The abstract machine does not handle process failure. A method for containing and handling process failure within FCP is described by Hirsch et al. [3].

3.2. Realizing Nondeterministic Clause Selection

Nondeterministic clause selection can be achieved by any arbitrary clause-try sequence. For simplicity the clauses are tried in textual order. Upon success of any clause try the process commits and the clause body is enqueued to the process queue. When a clause try is unsuccessful, no changes to the global environment should be made. This is achieved by recording global changes in a stack called the *trail*; upon suspension or failure of a clause try, the trail is used to undo global changes.

3.3. Tail Recursion Optimization

Upon commitment of a reduction attempt to one of the clauses, the process record of the committed process becomes available for reuse; moreover, the execution may proceed with any one of the processes being spawned. Instead of enqueuing the process to the process queue, execution may proceed with some chosen process. This scheme reduces process switching and process record management. To ensure the justice of the scheduling policy the number of times that this optimization can be employed must be bounded. A *time slice* is used to bound the number of times this optimization is employed. When a process is dequeued from the process queue, its time slice is initialized to a compiler constant. The time slice is decremented each time the execution iterates, and when the time slice is exhausted (i.e., becomes zero), the tail recursion optimization is not carried out; instead, the process is enqueued into the process queue.

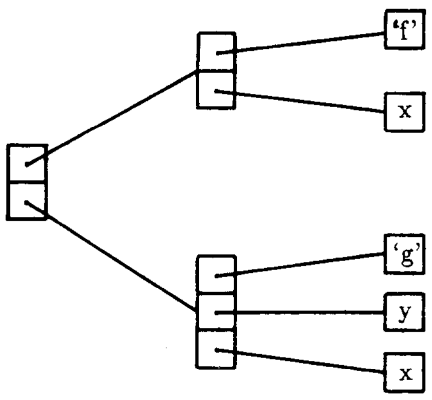

FIGURE 1. A tree of unification operations.

The above optimization leads to a bounded depth-first control strategy. The method is particularly useful for tight iterative loops, since process switching does not affect their efficiency, and process reduction can be implemented by a simple *goto*.

3.4. Clause Encoding

Each clause in the source program is individually encoded by the compiler into a sequence of abstract machine instructions. The code corresponding to a single clause contains two parts: the *clause try* and *body spawning*; the latter is executed upon commitment.

Clause Try. Each clause try consists of instructions which performs unification of the process with the head of the clause and calls for each guard test, similar to [20]. The instructions are extended to implement read-only unification. If a read-only unification suspends, the address of the first variable it suspends upon is stored in the suspension table, and the clause try fails. Upon failure of all the clause tries, the process is suspended if the suspension table is not empty, and fails if it is.

Unification is a recursive algorithm that executes operations corresponding to different data states. Part of the execution of the algorithm is known at compile-time due to the clause structure. This knowledge is used to advantage by encoding unification instructions at compile time instead of interpreting data structures at run time. This leads to a view of the execution as a tree whose vertices are known; unknown parts of the execution tree occur at the leaves. Figure 1 shows the tree of operations which must be carried out in order to unify the clause head[1] $p(\{f, X\}, \{g, Y, X\})$ with a goal.

A depth-first walk on the tree of execution can be flattened to produce a sequence of abstract machine instructions. Each node of the tree corresponds to one or more

[1]Note that *tuples* rather then *functors* are used to represent compound terms; thus the term $f(X_1, X_2, \ldots, X_n)$ is equivalent to $\{f, X_1, X_2, \ldots, X_n\}$.

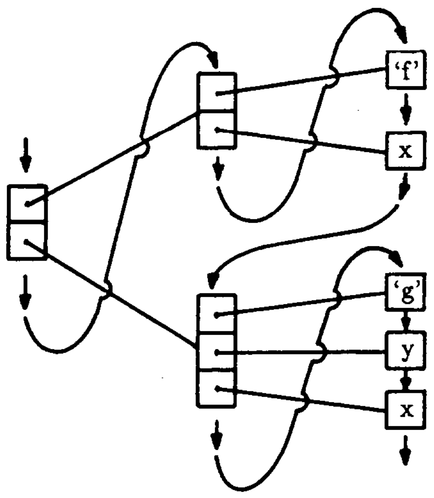

FIGURE 2. Depth-first traversal of the unification tree.

instructions that execute the needed operation of the unification algorithm. Thus the known part of the unification algorithm is executed by using static structures rather than using dynamic stack. The needed operations are implicitly encoded in the abstract machine instructions and need not be analyzed at run time. Figure 2 illustrates the depth-first walk on the execution tree.

Flattening the tree of execution must preserve the mode in which the unification operations are executed. Each vertex in the tree (instruction in the sequence) can be reached either when structures needed previously were available in the global environment (*read mode*) or when a variable was encountered (*write model*). In the former case the instruction is executed in read mode, which causes it to analyze the global data. In the latter case a variable assignment has been carried out previously; thus the instruction is carried out in write mode, causing structures corresponding to the clause structure to be allocated. The clause structure is implicitly encoded in the instructions.

Body Spawning. The abstract instructions for spawning of the clause body may perform some combination of the following operations:

Allocate a process record.

Create process arguments.

Enqueue a process into the process queue.

Iterate on one of the body processes if the time slice is not exhausted.

Halt the execution of the current process and return control to the scheduler.

3.5. Module system

To facilitate a distributed implementation of the language, global data objects such as a global symbol table or a global code area have been avoided. Instead, each program is represented as a single unit containing all the data needed for its execution, called a *module*. Since there is no central symbol table, string comparison may need to be carried out during unification. Due to the use of a hash code in a string representation, the resulting overhead is negligible.

Different modules should be able to communicate and execute remote procedure calls. The machine supports a module system [9] which provides this functionality. The module system is viewed as a set of communicating code segments. The approach taken is to handle the different aspects of the module system at the language level where possible. A design decision was made to utilize the inherent stream communication mechanism of FCP, and use incomplete messages [10] between modules to implement remote procedure calls. When a module is activated, an associated process, called the *module manager*, is spawned and is given two streams: an input stream for incoming requests and an output stream for requests to other modules. A convention was adopted that the procedure for the module manager is the first procedure of the module; its code is produced automatically by the compiler.

The module manager can easily handle input procedure calls by recognizing messages on the input stream and calling the appropriate local procedures. For each procedure with name p and arity n in the module which does not perform, directly or indirectly, remote procedure calls, the module manager includes one clause:

module_manager([$p(X_1, X_2, \ldots, X_n)$|in], Out) $\leftarrow$
  $p(X_1, X_2, \ldots, X_n)$,
  module_manager(In?, Out).

Remote procedure calls are handled by merging them onto the module manager's output stream. The above method enables the programming environment to organize intermodule communication simply by merging and distributing the appropriate streams to various modules. This methodology is incorporated in the Logix system.

The above scheme differentiates between the *code* of a module and its *activation*. The module code is a special data structure representing the compiled program. An activation is an abstract notion corresponding to the set of processes whose code resides in the module. The module code is a result of the compilation process and does not represent an active object. The system-defined call *activate* accepts as its arguments a module and two streams for input and output; it spawns a process which inherits these arguments and whose procedure is the first procedure of the module.

To show the elegance of this solution consider an operating system. Modules may be compiled and debugged; due to the chosen representation, modules can be replaced without a need for complex linking operations. The processes of a module activation continue running with the code of the old module, but the streams can be redirected to the new module when replaced. All these operations can be written in the language and do not require additional support in the abstract machine. When the system closes the input stream of a module activation and all its processes terminate, the module's code is garbage-collected automatically.

The viability of this approach is shown also in a parallel implementation of the language, where code management is a much more complex problem. The ability to define in the language processes that maintain and activate code in each processor and communicate code between processors is an essential basis for the code management mechanisms described in [15].

4. ABSTRACT MACHINE IMPLEMENTATION

4.1. Execution Algorithms and Structures

The main loop of the execution proceeds in the following steps:

(1) The first process in the process queue is dequeued, and a reduction attempt is made by trying the clauses in the associated procedure one after another.

(2) If a clause try succeeds, the clause body is enqueued into the process queue and any processes that were suspended on variables which have been changed are woken up. If a clause try fails, the variables that the process needs to be suspended upon are recorded in the suspension table and the next clause is tried.

(3) If all clause tries fail, the process is suspended on the variables recorded in the suspension table. If the table is empty when trying to suspend a process, the abstract machine enters a terminal failure state.

The compiler generates abstract instructions which perform the correct control for the clause tries as explained in Appendix 6.

Processes. A process record is a compound structure containing the following:

The address of a procedure.

The arguments of the process.

A reference to the next process record in the process queue.

The procedure address corresponds to a saved program counter, and the process arguments correspond to saved registers of conventional operating-system processes. In a direct implementation of the abstract machine this correspondence can be used to advantage, especially if the process model of the underlying processor and that of the abstract machine are similar.

Read-Only Unification. Recall from the semantics of the language that at unification time writing on a writable variable and reading its corresponding read-only variable cannot be done at the same unification. Hence a read-only variable has to be distinct from its correspondence writable variable, and the fact that a writable variable has been instantiated should not be observable from its read-only counterpart prior to the completion of the unification.

A pair of writable and read-only variables are represented as two distinct data objects, whose type is *writable* and *read-only*, respectively. The writable variable contains the address of its counterpart read-only variable if it has been allocated; otherwise it is null. The read-only variable points, indirectly, to the processes suspended on it, as explained below. The diagram in Figure 3 illustrates the representation of a pair of writable and read-only variables. Note that the writable variable cannot be accessed from its read-only counterpart, a fact which might be useful when we are concerned with security.

During unification writable variables are instantiated and this fact is trailed. Only if the unification completes does the clause commit and are the read-only variables bound to the value that their corresponding writable variables were bound to. This is done by scanning the trail.

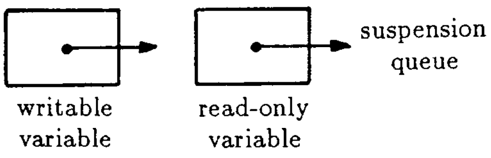

FIGURE 3. A pair of writable and read-only variables.

Suspension Queues. Processes suspend upon a set of variables waiting for one of them to be instantiated. More than one process may suspend on a single variable. When any variable is instantiated, all the processes suspended on it must be woken up, i.e., enqueued to the process queue. In order to perform these operations, suspension notes are used to record which processes have suspended on a given variable. These suspension notes are held as a suspension queue, the most recently suspended process first. A pointer to the beginning of the suspension queue is associated with the read-only variable. When a note is added, it is placed at the start of the suspension queue and the pointer is updated.

A process may attempt to reduce more then once, and each reduction attempt may suspend on some variables. When a process is woken up due to the instantiation of a variable, a mechanism is required to ensure that the process is not rewoken when any of the other variables are instantiated. This is achieved by a level of indirection from a suspension note to a process. All the suspension notes for a single reduction attempt point to a pointer, called a *hanger*, which points to the actual process. When a process is woken up via a suspension note, the hanger is nulled. Subsequent attempts to wake up the process via this hanger perform no operation.

Figure 4 illustrates how a process is suspended on two variables.

In a single reduction attempt different clause tries may cause suspension on the same variable. If a note is produced for each entry in the suspension table, more

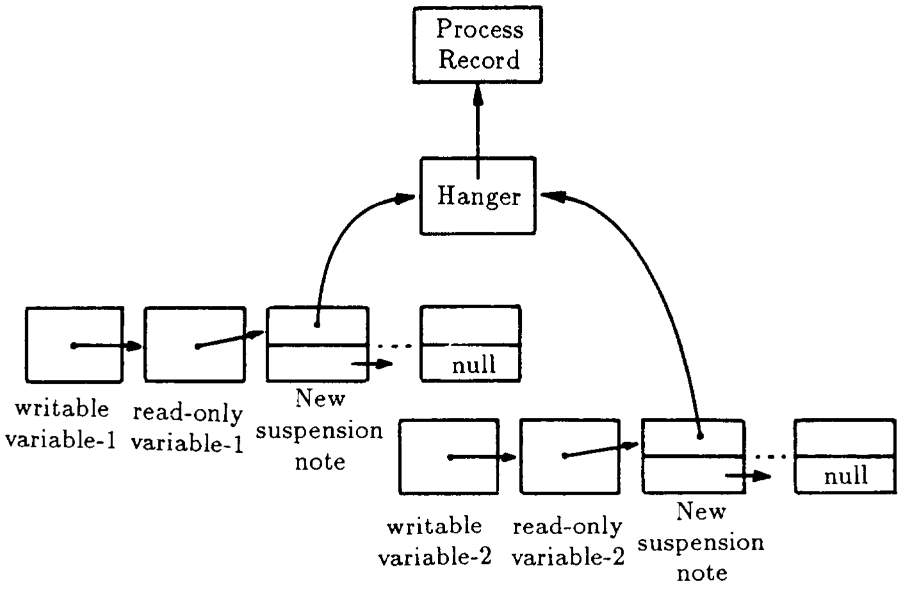

FIGURE 4. Suspending a process on two variables.

than one note to the same process may be associated with the same variable. This scheme is correct, but producing multiple notes is inefficient both in time and in space. To prevent the same reduction attempt from producing multiple notes on the same variable, the first element of a suspension queue is inspected before adding a new note to the list. If this note references the same hanger as the suspension note about to be added, no operation is carried out. Thus no duplicate notes for a single reduction attempt can occur in a single suspension queue.

When a clause try succeeds, suspended processes need to be woken up. All writable variables which have been instantiated during the clause try are recorded in the trail, and thus their corresponding read-only variables and their suspension queues are accessible. By scanning the trail at commit time the values of read-only variables are updated to those of their corresponding writable ones, and the necessary process activations are carried out. The order of processes in a suspension queue is the reverse of the order in which they were suspended. At the time of activation, the suspension queue is enqueued in reverse order to the active queue. Hence the most recently suspended processes are enqueued last.

Free Lists. When a process is reduced, tail-recursion optimization is carried out and the process record is reused for one of the body goals. If the body of the clause is empty, the process halts and the record cannot be reused. A free list of process records is maintained, and the process record is added to this free list. When a process record needs to be allocated for a newly created process, the free list is checked, and if it is nonempty, a process record from the free list is used. If the free list is empty, a new process record is allocated from the heap. For efficiency all process records are of identical size, and there is one process free list. Some processes may require more arguments then available in the process record; they are transformed by the compiler to conform to the bound. Since structures are used to hold the extra arguments, tail-recursion optimization on such processes may be less effective.

The same technique is used for allocation of suspension records. This scheme reduces the number of process records and suspension records allocated from the heap and thus reduces the frequency of garbage collection.

4.2. Heap and registers

All system run-time structures are constructed from tagged data objects; thus there is no need to allocate run-time structures from separate areas. A single data area, the *heap*, is used for terms, programs, and processes. The heap grows when objects are allocated, and shrinks only by garbage collection.

In addition to the heap, two data structures are required to support run-time algorithms: the *trail* and the *suspension table*. The trail is a stack which is empty at the beginning of a clause try. When a heap word is changed, an entry containing the address of the word and its previous value is pushed onto the trail. If the clause try fails, the trailed changes are undone, resetting the global environment to its previous state. The suspension table is empty at the beginning of a process try, and when a clause try needs the value of a variable to proceed, the writable or the read-only variable is added to the suspension table.

The current state of the machine is defined by the contents of its heap, trail, suspension table, and registers. The registers can be divided into three groups according to the time in which their value is significant:

The general registers, which are significant throughout all the machine's operation.

The process-try registers, which are significant only during a reduction attempt.

The clause-try registers, which are significant only during a clause try.

General Registers

HB	Heap Backtrack, contains the value of the top of the heap prior to a process try.
HP	Heap Pointer, points at the current top of the heap.
QF	Queue Front, points at the first process in the process queue.
QB	Queue Back, points at the last process in the process queue.
PFL	Process Free List, the address of the first process in the process free list, or null.
SFL	Suspension Free List, the address of the first suspension note in the suspension free list, or null.

Process-Try Registers

CP	Current Process, points at the process currently being tried.
TS	Time Slice, the remaining number of times that the current process may use tail-recursion optimization.
PC	Program Counter, the address of the next instruction to be executed.
STP	Suspension Table Pointer, points at the top of the suspension table.

Clause-Try Registers

FL	Failure Label, the address of the instruction to branch to in case of failure of the clause try.
TRP	Trail Pointer, points at the top of the trail.
A	Argument pointer, points at an argument of a process or of a guard. This register is used by the *get* and *put* instructions.
SP	Structure Pointer, points at an argument of a compound structure. This register is used by the *unify* instructions.
Mode	Mode register, can specify either *read* or *write* mode. This register is used by the *unify* instructions.
X_i	*X* registers, a set of temporary registers used for holding intermediate values during a clause try.

4.3. Garbage Collection

Since FCP programs tend to generate a large amount of garbage relative to the amount of useful data, stop-and-copy garbage collection [6] is used in the machine. Even though stop-and-copy requires two heaps for its operation, it is more suitable for the machine, since its complexity is linear with respect to the useful data rather than the total memory.

Since the machine can access data only through processes residing in the process queue, the only data which need to be copied are those accessible from the current process and the process queue. Both free lists are discarded when a garbage collection is started. Any suspension note whose hanger is null is not copied.

Processes which are suspended only on inaccessible variables are not copied during garbage collection. By maintaining a count of the number of existing processes and comparing it with the number of processes that are copied during garbage collection it is possible to detect if processes are lost. As mentioned before, unreferenced code is also garbage-collected; no special code-management mechanism is necessary to support interactive program development.

5. ENCODING

The encoding scheme follows largely that of Warren [19]. Since there is no backtracking, no environments should be handled and the encoding of a clause becomes much simpler. Indexing instructions were not implemented either.

5.1. Encoding Scheme

This subsection provides an overview of the encoding scheme; the set of abstract instructions is described in the following subsection.

Procedure Encoding. The code for a procedure is a sequence of abstract instructions. When a clause try fails, execution should proceed in the next clause try. This is achieved by the instruction *try_me_else Failure_Label*; when the clause try following the instruction fails, execution proceeds from *Failure_Label.*

Upon failure of all the clause tries, the process is to be suspended on the variables in the suspension table. After the last clause try of a procedure, a *suspend* instruction is planted to suspend the process. The failure label of the last clause refers to this instruction:

```
P:   (label of the procedure)
     try_me_else C2
     (code for first clause)
C2:
     try_me_else C3
     (code for second clause)
```

```
            ⋮
    Cn:
            try_me_else S:
            (code for last clause)
    S:
            suspend
```

Clause Encoding. The code for a single clause try is constructed from abstract instructions for head unification and guard execution, followed by instructions for commit and spawning of the body processes.

A clause is encoded as followed:

```
(code for head unification and guard execution)
commit
(code for body spawn)
```

Head Unification. Recall the unification algorithm presented in Section 3. Flattening of the execution tree could be carried out using a stack of pairs. Each pair corresponds to an argument in the clause and an argument in the goal which is to be unified. Apart from the values of bound variables, flattened structure is known at compile time; as a result, a stack is not required, but rather a set of registers can be used to hold values and pointers to the arguments. Since the data type of the clause argument is also known at compile time, the clause part of the pair can be omitted and can be represented by different abstract instructions.

Some optimizations can be carried out before the final encoding. When a constant appears in the clause, the goal argument is pushed onto the stack and later popped either to match or assign a goal variable. These push and pop operations can be carried out together and encoded into a single instruction. When a variable appears in the clause which does not require general unification, the operation succeeds trivially and thus can be omitted.

The following primitive instruction types form the basic operations on the above stack and encode the structure of the data required. The dynamic stack is replaced by the set of X_i registers, and the goal argument is pointed to by an argument register which is advanced upon a successful completion of the instruction.

compound Arity corresponds to a pop of the stack and match or assignment of a compound structure.

constant Value corresponds to a match or assignment of a constant.

var X_i corresponds to push onto the stack of an argument.

val X_i corresponds to general unification for subsequent occurrences of a variable in a clause.

ro_var X_i corresponds to push onto the stack of an argument when the clause argument is a read-only variable.

ro_val X_i corresponds to general unification for subsequent occurrences of a variable in a clause when annotated read-only.

The abstract machine uses more than one instruction for each of the above primitive instruction types. The different instructions correspond to the primitive data types in the language (see Appendices 1 and 3). In addition different instructions are used to distinguish the first and subsequent levels in the execution tree.

There are two types of head instructions: *get* and *unify*. Unify instructions correspond to the actions that would normally be executed by the unification algorithm and may thus execute in either read or write mode. The get instructions occur only at the first level of the head structure and thus always operate in read mode.

Most clause heads have some compound structure as an argument. Pushing and popping the argument at this level would be an overhead. In order to avoid this overhead a different argument register is used by the get instructions. Thus the encoding of a compound structure appearing at the first level of the head is a *get_compound* instruction immediately followed by the unify instructions for its arguments.

The encoding of the arguments of the clause head $p(f(X, X?), X)$ is:

```
get_compound 3
unify_constant 'f'      % f
unify_var X_n           % X
unify_ro_val X_n        % X?
get_val X_n             % X
```

A nested compound structure is encoded by a *push* X_i instruction which loads the mode on which the current sequence of unify instructions operate and the continuation of the current compound structure into the X_i register. A corresponding *pop* X_i, following the next unify sequence, will get the mode and continuation of the current compound structure, enabling the current sequence of unify instructions to continue. Thus the encoding of the arguments of the clause head

$p(f(1, f(X?), X), 1)$

is:

```
get_compound 4
unify_constant 'f'      % f
unify_constant 1        % 1
push X_m                % (save ",X)")
unify_compound 2
unify_constant 'f'      % f
unify_ro_var X_n        % X?
pop X_m                 % (load ",X)")
unify_val X_n           % X
get_constant 1          % 1
```

Note that when the nested compound structure is the last argument of the current compound structure, there is no need to save the continuation; thus the pop and push instructions are not needed.

Argument Creation. An argument creation in the guard or in the body of the clause cannot be executed in read mode. Thus another type of argument instructions, *put*, is added, which corresponds to the operation of the unify instructions in write mode. The justification for having these instructions explicitly is similar to that of get instructions. The put and get instructions use the same argument register for their operation. The put instructions also correspond to the arguments in the first level of the clause. The instructions for creating the arguments for the goal $q(a, g(1, g(X)))$ are:

```
put_constant 'a'        % a
put_compound 3
unity_constant 'g'      % g
unify_constant 1        % 1
unify_compound 2
unify_constant 'g'      % g
unify_var X_n           % X
```

Guard. The encoding of a guard call is composed of two parts: creating the arguments for the guard, and an instruction which calls the guard.

In order to enable optimizations of the arguments allocation for a guard, the X registers are used for transferring arguments to the guard. If the arguments the guard needs have already been loaded into X registers, no arguments allocation is needed.

Two additional instructions are used for guard encoding: *set* X_i sets the argument register to the X_i register. The following put and unify instructions will create arguments starting at X_i. The actual call to the guard is carried out by the *call Index*, *Args* instruction, where *Index* is the guard's internal number and *Args* is a list of X registers which refer to the arguments for the guard. The encoding of the clause $p(X, Y) \leftarrow guard(Y, X) | q(X, Y)$. is

```
try_me_else Label
get_var X_0             % X
get_var X_1             % Y
call Index X_1 X_0
commit
(code for body)
```

The encoding of the clause $p(X, Y) \leftarrow guard(1, X, Y) |$ *true*. is:

```
try_me_else Label
get_var X_0             % X
get_var X_1             % Y
set X_2                 % start creating arguments for the guard.
put_constant 1
call Index X_2 X_0 X_1
commit
halt
```

Body. An empty body is encoded into a single instruction called *halt*; this instruction frees the current process record for future use and returns control to the scheduler. A nonempty body is encoded to do the following operations:

Allocate process records for body goals.

Create the arguments for each process.

Enqueue the body processes into the process queue.

Iterate on one of the body processes if the time slice is not exhausted.

The allocation and enqueuing of new processes is carried out by the *spawn* instruction. The creation of arguments is carried out by the put and unify instructions. Since the current process record is used for tail recursion optimization, there is no need to allocate a process record for the iterated goal. By convention this is the first goal in the body. The commit instruction is followed by the instructions which create arguments for the first goal in the body. The last instruction in the encoding of a nonempty body is an *execute* or *iterate* instruction which tries to iterate on the first body goal.

The body of a general clause is encoded as follows:

```
(code for args of first goal)      % put arguments of the first
                                   % goals in the current process
spawn Proc_2                       % Proc_2 is the procedure of
                                   % the second goal.
(code for args of second goal)     % put arguments
                                   % of the second goal in the process.
   .
   .
   .
spawn Proc_n                       % Proc_n is the procedure for
                                   % the last goal.
(code for args of last goal)
execute Proc_1                     % Proc_1 is the procedure for
                                   % the first goal.
```

5.2. *Abstract Instructions*

The abstract-instruction set can be divided into four types: *clause* (or *indexing*), *argument*, *guard*, and *process*. A description of each type of instruction is given below.

Clause (Indexing) Instructions. Clause instructions are concerned with the flow of control through a procedure. Currently there is only one clause instruction, *try_me_else Label*. The code section following this instruction corresponds to a clause try. Upon failure of the clause try, execution proceeds at the point labeled by *Label*. More clause instructions can be added as in [20] to perform clause selection according to the arguments of a process.

Argument Instructions. The argument instructions are used for two purposes: execution of the known part of the unification algorithm and allocation of data structures. For the verification of the global data state *get* and *unify* instructions are used. Allocation of arguments of processes and guards is performed using *put* and *unify* instructions. The data structure to be verified or allocated is implicitly encoded into the different instructions. The *get* and *put* instructions use the A register, which refers to the arguments of the process record, while the *unify* instructions use the SP register, which refers to general data structures. Each of the argument instructions increments the register it uses upon successful completion. Some of the instructions explicitly use the set of X registers to write or read values.

There are two main differences in the format of the argument instructions between Warren's instructions and ours. The first is the use of an implicit argument register A for the *put* and the *get* instructions instead of using a set of explicit argument registers (A_i) in Warren's instructions. The other difference is the use of the *push* and *pop* instructions for the encoding of nested substructures.

Since most of the instructions are similar to those of Warren, their explicit description is omitted. Only the completely new instructions are shown:

get_ro_var X_i This instruction encodes the first occurrence of a variable in the clause when annotated read-only. If the argument given by A contains a writable variable, that variable is bound to a read-only variable while X_i is set to its writable counterpart. Otherwise, the argument given by A is stored in X_i.

get_ro_val X_i This instruction encodes subsequent occurrences of a variable when annotated read-only. General unification is applied with the read-only counterpart of this variable and the current argument.

put_ro_var X_i This instruction encodes the first occurrence of a variable in the clause when annotated read-only. A new writable variable is allocated; a reference to it is stored in X_i while a reference to its read-only counterpart is stored in the argument referenced by A.

put_ro_value X_i This instruction encodes subsequent occurrences of a variable in the clause when annotated read-only. If the argument given by X_i is a writable variable, its read-only counterpart is stored in the argument referenced by A. Otherwise a reference to the argument given by X_i is stored in the argument given by A.

push X_i This instruction starts the encoding of a nested substructure. The state in which the next *unify* instruction would have operated is saved in the X_i register. This state consists of the value of the SP register incremented by one and the value of the *Mode* register, which is either read of write.

pop X_i This instruction ends the encoding of a nested substructure, and it corresponds to a previous *push* instruction. The state of the higher sequence of unify instructions is loaded from the X_i register.

set X_i This instruction starts the creation of argıments that will be referred by a consecutive X registers starting at X_i. The A :gister is initialized to X_i. Recall that the *put* instructions will create arguments at the place pointed to by the A register.

Guard Instruction The guard instruction provides the mechanism for calling a guard. The instruction is:

call Index, *Args* Call the guard procedure whose internal number is *Index* and whose arguments are the X registers specified by *Args*.

Process Instructions The process instructions correspond to the possible outcomes of a process try. The instructions are:

commit Commit to the current clause; this instruction appears after the instructions for the head and guards of the clause. The instruction scans the trail while binding read-only variables to the value of their bounded writable counterparts, and activates suspended processes which are reachable from the trail. In addition, tail-recursion optimization is employed, and the current process record is prepared for reuse.

spawn Proc This instruction creates a process record and enqueues it to the process queue. The procedure that is attached to this process is indicated by *Proc*. In addition the *A* register is initialized so that the following *put* and *unify* instructions will create the arguments of the process.

execute Proc This instruction iterates on the procedure indicated by *Proc* if the time slice is not exhausted. Otherwise the current process is enqueued to the process queue and control is given back to the scheduler.

iterate This instruction is similar to the execute instruction, but iterates on the procedure of the current process.

halt This instruction halts the execution of the current process. The process record is freed for future use. Control is given back to the scheduler.

suspend This instruction is executed when all the clause tries fail. The process is suspended on all the variables in the suspension table, and control is given back to the scheduler. If the suspension table is empty, the machine enters a terminal failure state.

6. DATA OBJECTS

In representing data objects we have used the method of *tagged objects* rather than the *tagged pointers* used by the Warren abstract machine. The difference between the two is implied by their names: in tagged pointers the tag which tells the type of the data object is stored in a pointer to the data object, while in tagged objects the tag is stored with the object itself.

Detailed description of the various data objects is given in Appendix 1. Here we describe only the important points.

6.1. Strings

Strings are consecutive bytes of memory. There are several types of strings: character strings, modules, procedures, and frozen terms. A module is the result of compiling an FCP program, and it contains a series of smaller strings which correspond to procedures, character strings, and frozen terms in the program. A procedure may appear only within a module, and it represents the encoding of a single procedure in the program. A frozen term is the result of freezing an FCP term [7].

During a computation many string unifications are carried out; in particular, clause selection is often based on the value of a string argument. Strings created at run time may cause redundant string unifications, since they may be equal to existing strings. To avoid this overhead the following method is used.

When comparing two strings, the machine checks that the two strings are not from the same module. Since strings in a module are unique, two strings from the same module are known to be different. If the strings are not from the same module, the hash and the length of the two strings are compared before comparing the characters. This method saves full string comparisons in most of the cases where the strings are different.

After two strings are compared and found identical, if one of the strings is a "free" string (not within a module), this string becomes a reference to the other string. In this way redundant occurrences of strings are canceled, and time and space are saved.

6.2. Tuples and Lists

The machine uses two main types of compound objects for representing compound terms, namely *tuples* and *lists*. Tuples are used for representing a finite compound object as $a(b)$ or $\{X, Y\}$, while lists represent a list of terms which is not always of a bounded length.

There are two important things to mention about tuples and lists: unification and *car* coding.

Unification. When unifying two compound objects (tuples or list cells), the machine starts the unification by changing the first word of one of the compound objects to a reference to the other. There are two reasons why this is done:

(1) The space of the compound object is saved when garbage collection occurs.

(2) Since the occur check is not performed during unification, circular terms may be created. The method proposed avoids infinite loops when unifying two circular terms.

Car Coding. Lists and streams are used often in FCP; therefore optimization of list representation and manipulations is very important. In this implementation a method called *car* coding is used in order to reduce the space needed for list representation. This method is the tagged-object counterpart of the *cdr*-coding technique found in tagged-pointer implementations of Lisp.

In *car* coding the tag of the list cell is condensed into the tag of the *car*, saving one heap word for each list cell. Furthermore, since the *cdr* is generally a reference to another list cell, this next list cell is put at the place where the *cdr* would have been. This method can be applied inductively. Hence, the list [1, 2, 3] can be represented by four consecutive heap words (the fourth is [ ]) instead of using three separate list cells.

Lists can be created using separate *car*-coded list cells and be condensed at garbage collection. Condensed lists can also be created by recursive procedures which produce only one list cell at every iteration. For example,

$$merge([X|X_s], Ys, [X|Zs]) \leftarrow merge(Ys?, Xs?, Zs).$$

$$merge(Xs, [Y|Ys], [Y|Zs]) \leftarrow merge(Ys?, Xs?, Zs).$$

Since *merge* iterates during its time slice, and allocates only one list cell per reduction, the list cell created by the previous reduction is at the top of the heap; the variable *Zs* will be its *cdr*. Checking whether a variable is at the top of the heap before instantiating it to a new list cell enables the machine to create the list cell at the location of the variable itself instead of instantiating the variable to a reference to the list cell. Hence the sublist created by *merge* during one time slice is condensed.

In addition to condensing lists cells, this scheme allows any one word object, including variables, to be an immediate argument of a tuple. Furthermore any data object can appear as the last argument of a tuple or as the *cdr* of a list cell.

7. PERFORMANCE

Appendix 5 contains a number of performance measurements and compares the results with that of Quintus Prolog [8]. Although slower in absolute terms, FCP has not undergone the extensive optimizations used in the Quintus implementation. In addition, the emulator is written in C rather than assembly language. Quintus Prolog is one of the faster commercially available Prolog compilers; the results demonstrate that the FCP compiler is both practical and efficient despite its simplicity. The benchmarks were run on a VAX-11/750.

The results indicate that Quintus Prolog is at best seven times faster than the FCP emulator. In more complex Prolog programs the speedup is much smaller, unless a cut is used in every Prolog clause.

In absolute terms, the emulator achieves a speed of 2K LIPS (logical inferences, or reductions, per second) on the standard naive reverse benchmark. On real-life programs the average speed is much smaller, about 500 LIPS. Iteration of an empty procedure

$p \leftarrow p.$

achieves a peak performance of 10K LIPS. Iterative process spawning, defined by

$p \leftarrow q, p.$
$q.$

runs at a rate of 5K LIPS, generating about 2500 processes per second.

The entire FCP compiler, including the tokenizer, parser, code generator, and assembler, is written in FCP. It compiles about 100 lines of source code per CPU minute. During the compilation of 100 lines of code, about 10,000 processes are created, and 30,000 process reductions occur.

More detailed measurements of the cost of different operations are yet to be carried out.

8. FUTURE RESEARCH

The main drawback of the instruction set defined is that each clause must be compiled individually. Better performance can be achieved if all clause tries of a procedure are compiled as one unit, into a decision tree. Such a compiler is being developed presently [4].

Another direction pursued is the parallel implementation of the language [16]. The abstract machine described is extended to execute in each processor in a parallel implementation. It must preserve the atomicity of process reduction in spite of the environment being distributed across several memories, and allow several processors to attempt to reduce simultaneously.

9. CONCLUSIONS

The abstract machine described in this paper uses a similar approach to that of Warren [18] in compiling Prolog. The instruction set for FCP uses a number of the Warren instructions to compile unification but enhances them to handle read-only variables. Additional instructions are required, and have been described, to handle process control and execution of guard predicates.

Although it would appear that due to the overhead for suspending and activating processes, FCP is less efficient then Prolog, this is not the case, as that is compensated for by the lack of backtracking in the language. There is no need to keep backtrack information, and it is not necessary to maintain two different global areas (i.e., heap and stack).

The methods used to compile FCP are of general interest and can be applied to other concurrent logic-programming languages. Some of the ideas in this work have already been applied to the compilation of Guarded Horn Clauses [17] by Levy [5].

The methods described have been used to provide the first efficient and practical implementation of a concurrent logic-programming language.

APPENDIX 1. DATA TYPES

FCP data objects are represented as words in memory. The tag of the object is stored in the six least significant bits of its first word. Objects can be atomic or compound.

The following is a BNF definition of FCP data objects:

⟨data-object⟩ = ⟨atomic-object⟩|⟨compound-object⟩.

Atomic Objects

```
⟨atomic-object⟩ =
  ⟨reference⟩|
  ⟨writable⟩|
  ⟨read-only⟩|
  ⟨integer⟩|
  ⟨real⟩|
  ⟨string⟩.
  ⟨nil⟩.
```

Reference

⟨reference⟩ = ⟨address⟩⟨ref⟩.

The reference is used as a general reference to other data objects.

Writable Variable

⟨writable⟩ = ⟨address⟩⟨var⟩.

The address is null if the corresponding read-only variable has not been allocated. Otherwise, it is the address of the read-only variable.

Read-Only Variable

⟨read-only⟩ = ⟨address⟩⟨ro⟩.

The address is null if the read-only variable does not have a suspension queue. Otherwise, it is the address of the last suspension note which was entered to the queue.

Integer

⟨integer⟩ = ⟨integer-value⟩⟨int⟩.

An integer-value is negative if its leftmost bit is set. Sign extension can be used to create the full negative value.

Real

```
⟨real-object⟩ =
  ⟨zero-value⟩⟨real⟩
  ⟨real-value⟩
  ⟨real-value⟩
```

A real value is composed from two words, and its internal representation is machine-dependent.

String

```
⟨string⟩ =
  ⟨string-type⟩⟨string-offset⟩⟨str⟩
  ⟨string-hash⟩⟨string-length⟩
  ⟨characters⟩
  ⟨padding-nulls⟩
⟨string-type⟩ = 0...15
```

⟨string-offset⟩ = 0|positive number

⟨string-hash⟩ = hash(characters)

⟨string-length⟩ = number of characters

⟨characters⟩ = ⟨character⟩|⟨character⟩⟨characters⟩

⟨padding-nulls⟩ = 4 − (string-length mod 4) ⟨zero-bytes⟩

The string type, which consists of four bits, determines the type of the string. The current types are character string, module, procedure, and frozen term. A module is the result of compiling an FCP program, and it contains a series of smaller strings which correspond to procedures, character strings, and frozen terms in the program. A procedure may appear only within a module, and it represents the encoding of a single procedure in the program. A frozen term is the result of freezing an FCP term.

The string offset is used for garbage collection for deciding whether the referred string is part of a module.

Nil

⟨nil⟩ = ⟨null-value⟩⟨nil⟩.

Nil represents the symbol [ ].

Compound Objects

⟨compound-object⟩ =
  ⟨list-object⟩|
  ⟨tuple-object⟩|
  ⟨vector-object⟩.

List

⟨list-object⟩ = ⟨car⟩⟨cdr⟩.

Lists are time and space optimization. They are used in FCP programs for streamlike structures. They could be represented by a binary tuple, but since they are the most heavily used data type, they are optimized extensively.

⟨car⟩ =
  ⟨car reference⟩|
  ⟨car integer⟩|
  ⟨car nil⟩.

⟨car reference⟩ = ⟨address⟩⟨1_ref⟩.

⟨car integer⟩ = ⟨integer-value⟩⟨1_int⟩.

⟨car nil⟩ = ⟨null-value⟩⟨1_nil⟩.

⟨cdr⟩ = ⟨data-object⟩.

Any data object can appear as list-cdr and in particular a car.

Tuple

```
⟨tuple-object⟩ =
  ⟨tuple-length⟩⟨tpl⟩
  ⟨tuple-arguments⟩.
⟨tuple-length⟩ = number of arguments
⟨tuple-arguments⟩ =
  ⟨data-object⟩|
  ⟨tuple-argument⟩⟨tuple-arguments⟩
⟨tuple-argument⟩ =
  ⟨reference⟩|
  ⟨writable⟩|
  ⟨read-only⟩|
  ⟨integer⟩|
  ⟨nil⟩.
```

Note that $f(X)$ and $\{f, X\}$ both have the same representation, which is a tuple of arity 2. Any data object can appear as the last argument of a tuple.

Vector

```
⟨vector-object⟩ =
  ⟨vector-length⟩⟨vctr⟩
  ⟨vector-arguments⟩.
⟨vector-length⟩ = number of arguments
⟨vector-arguments⟩ =
  ⟨vector-argument⟩|
  ⟨vector-argument⟩⟨vector-arguments⟩
⟨vector-argument⟩ =
  ⟨reference⟩|
  ⟨integer⟩|
  ⟨nil⟩.
```

The third type of compound data object, used internally by the machine, is called *vector*. It is used for representing finite compound objects and is similar in structure to tuples. However, a vector cannot contain variables, which are address-dependent objects, as immediate arguments. Vectors are used for representing data structures such as process records, hangers, and mutual-references [12]. These data structures can be destructively charged during the computation, and hence cannot be condensed like tuples.

APPENDIX 2. BOOTSTRAP AND CONTROL FLOW OF THE MACHINE

Conventions

If R is a register, then $*$ is the value of the memory location pointed to by R.
beginning_of_heap, *boot*, *null*, *time_slice*, *trail*, and *suspension_table* are emulator constants. $pc\_of(PR) = PR + 1$, $next\_of(PR) = PR + 2$, $args\_of(PR) = PR + 3$ are addresses of words inside the process record with address PR.

Utility Procedures

The following procedures are used in the machine description:

```
procedure Enqueue(Process)
   if QF = null
         % process queue is empty
   then QF := QB := Process
         % initialize queue to contain the new process
   else *next_of(QB) := Process; QB := Process
         % add new process to the process queue

procedure Dequeue
   if QF = null
         % process queue is empty
   then stop
         % stop the machine
   else
      CP := QF
         % current process is first process
      QF := *next_of(QF)
         % update QF register to point to the next process
   A := args_of(CP)
         % set argument register to point to
         % first argument of current process
   PC := *pc_of(CP)
         % set program counter to procedure of current process
   TS := time_slice
```

Machine Operation

Initially the process queue contains a single process, the booting process, at address 'boot'. The initial values of the registers are as follows:

HB = top of heap

HP = top of heap

QF = boot

QB = boot

```
    PFL = null
    SFL = null
    STP = suspension_table
    TRP = trail.
Initial values of other registers are unimportant.
    Start:
      Dequeue
            % dequeue booting process.
    Next:
      Fetch next instruction
      Case instruction of:
        get(Arg):
          unify(*A, Arg)
            % get *A, the current argument,
            % and unify it with Arg
          A := A + 1
            % Advance argument register
        put(Arg):
          create(*A, Arg)
            % initialize *A, the
            % current argument, to be Arg,
          A := A + 1
            % Advance argument register
        unify(Arg):
          if Mode = 0
            % value of Mode register is "read"
          then unify(*SP, Arg)
            % unify *SP, the current structure
            % argument, with Arg.
          elseif Mode = 1
            % value of Mode register is "write"
          then create (*SP, Arg)
            % initialize *SP, the current
            % structure argument, to be Arg.
          SP := SP + 1
            % Advance structure pointer register
        set(Xn):
          Set A to point to register Xn.
        call(Index, Xi, Xj, ...):
          Call guard guard number Index, with the arguments
          Xi, Xj, ...
        commit:
          Bind read-only counterparts of trailed writable variables.
          Activate processes in suspension queues which are
          reachable from the trail.
    TRP := trail
      % reset trail pointer register
    STP := suspension_table
      % reset suspension table pointer register
    A := args_of(CP)
      % set argument register to first argument
      % of current process
spawn(Proc):
    Allocate a new process record, store its address in NPR.
    *pc_of(NPR) := Proc
      % initialize the program counter
      % of the process to Proc
    Enqueue(NPR)
      % enqueue process NPR
    A := args_of(NPR)
      % set argument register to first
      % argument of NPR
execute(Proc):
 *pc_of(CP) := Proc
      % change procedure of current
      % process to Proc.
    If TS > 0
      % time slice not exhausted
    then TS := TS - 1;
      % decrement time slice
    A := args_of(CP)
      % set argument register to point to
      % first argument of current process
    PC := Proc
      % set program counter to Proc
    else
    Enqueue(CP)
      % enqueue current process

    Dequeue
      % dequeue next process
halt:
    *next_of(CP) := PFL; PFL := CP
      % return current process to process free list
    Dequeue
suspend:
    Suspend current process on the variables in the
    suspension table. If the suspension table is empty
    then fail the computation.
    STP := suspension_table
      % reset suspension table
    Dequeue

goto Next
```

APPENDIX 3. LIST OF INSTRUCTIONS

Procedural Instructions

commit
spawn *Proc*
execute *Proc*
iterate
halt
suspend

Clause Instruction

try_me_else *Label.*

Guard Instruction

call *Index*, X_i, X_j,...

Argument Instructions

set X_i
push X_i
pop X_i
void

Put Instructions

put_var X_i
put_val X_i
put_ro_var X_i
put_ro_val X_i
put_int *Integer-Value*
put_real *Real-Value*
put_str *Address*
put_nil
put_tpl *Arity*
put_car_var X
put_car_val X_i
put_car_ro_var X_i
put_car_ro_val X_i
put_car_int *Integer-Value*
put_car_real *Real-Value*
put_car_str *Address*
put_car_nil
put_car_list X_i
put_car_tpl X_i, *Arity*

The *Xi* register in *put_car_list* and *put_car_tpl* instructions is used for pushing into it the *Mode* of unification and the address of the next argument. The same applies to the *get* and *unify* instructions.

Get Instructions:

get_var X_i
get_val X_i
get_ro_var X_i
get_ro_val X_i
get_int *Integer-Value*
get_real *Real-Value*
get_str *Address*
get_nil
get_tpl *Arity*

get_car_var X_i
get_car_val X_i
get_car_ro_var X_i
get_car_ro_val X_i
get_car_int *Integer-Value*
get_car_real *Real-Value*
get_car_str *Address*
get_car_nil
get_car_list X_i
get_car_tpl X_i, *Arity*

Unify Instructions:

unify_var X_i
unify_val X_i
unify_ro_var X_i
unify_ro_val X_i
unify_int *Integer-Value*
unify_real *Real-Value*
unify_str *Address*
unify_nil
unify_tpl *Arity*

unify_car_var X_i
unify_car_val X_i
unify_car_ro_var X_i
unify_car_ro_val X_i
unify_car_int *Integer-Value*
unify_car_real *Real-Value*
unify_car_str *Address*
unify_car_nil
unify_car_list X_i
unify_car_tpl X_i, *Arity*

APPENDIX 4. ENCODING EXAMPLES

The examples below were produced by the Logix compiler and relate to the data types used in the implementation. The examples were adjusted by hand to remove unneeded information and to make them readable. Note that X registers are allocated only as long as they are needed and an attempt is made to reduce the use of X registers as much as possible. This will be most important when part of the X registers resides in real registers.

Compiling Read-Only Variables in the Head and in the Body

```
merge([X|Xs],Ys,[X|Zs?]) ← merge(Xs?,Ys,Zs).
A0:
    try_me_else A1      % merge(
    get_car_var X0      % X|
    unify_var X2        % Xs],
    get_var X1          % Ys,
    get_car_val X0      % X|
    unify_ro_var X0     % Zs?])
    commit              % ← merge(
    put_ro_val X2       % Xs?,
    put_val X1          % Ys,
    put_val X0          % Zs).
    iterate             %
A1:
suspend.
```

Compiling a Complete Procedure, with Process Spawning

```
qsort([X|Xs],S1,L2) ←
  part(X,Xs?,S,L),qsort(S?,S1,[X|L1]),qsort(L?,L1,L2).
qsort([ ],X,X).
B0:
    try_me_else B1      % qsort(
    get_car_var X1      % X|
    unify_var X4        % Xs],
    get_var X3          % S1,
    get_var X0          % L2)
    commit              % ← part(
    put_val X1          % X,
    put_ro_val X4       % Xs?,
    put_var X4          % S,
        put_var X2          % L),
        spawn B0            % qsort(
        put_ro_val X4       % S?,
        put_val X3          % S1,
        put_car_val X1      % X|
        unify_var X1        % L1]),
        spawn B0            % qsort(
        put_ro_val X2       % L?,
        put_val X1          % L1,
        put_val X0          % L2).
        execute C0          %
    B1:
        try_me_else B2      % qsort(
        get_nil             % [ ]
        get_var X0          % X,
        get_val X0          % X,
        commit              % ).
        halt
    B2:
    suspend.
```

Compilation of Nonempty Guard

part(X, [Y|Ys], [Y|S?], L) ← X > Y|part(X, Ys?, S, L).

part(X, [Y|Ys], S, [Y|L?]) ← X ≤ Y|part(X, Ys?, S, L).

part(X, [ ], [ ], [ ]).

```
    C0:

        try_me_else C1      % part(
        get_var X3          % X,
        get_car_var X4      % Y|
        unify_var X2        % YS],
        get_car_val X4      % Y|
        unify_ro_var X1     % S?],
        get_var X0          % L) ←
        call >,X3,X4        % X > Y
        commit              % |part(
        put_val X3          % X,
    put_ro_val X2          % Ys?,
    put_val X1             % S,
    put_val X0             % L).
    iterate                %

C1:
    try_me_else C2         % part(
    get_var X3             % X,
    get_car_var X4         % Y|
    unify_var X2           % Ys],
    get_var X1             % S,
    get_car_val X4         % Y|
    unify_ro_var X0        % L?]) ←
    call ≤,X3,X4           % X ≤ Y
    commit                 % |part(
    put_val X3             % X,
    put_ro_val X2          % Ys?,
    put_val X1             % S,
    put_val X0             % L).
    iterate                %

C2:
    try_me_else C3         % part(
    get_var X0             % X,
    get_nil                % [ ],
    get_nil                % [ ],
    get_nil                % [ ]).
    commit
    halt

C3:
suspend.
```

Compiling Complex Structures in the Head

$d(U * V, X, (DU * V) + (U * DV)) \leftarrow$
$d(U, X, DU), d(V, X, DV).$

```
D0:
    try_me_else D1         % d(
    get_tuple 3
    unify_string '*'
    unify_var X4           % U*
    unify_var X2           % V,
        get_var X1            % X,
        get_tpl 3
        unify_string '+'
        push X0               % (save "(U*DV)")
        unify_tpl 3
        unify_string '*'
        unify_var X3          % (DU*
        unify_val X2          % V) + (
        pop X0                % (load "(U*DV)")
        unify_tpl 3
        unify_string '*'
        unify_val X4          % U*
        unify_var X0          % DV))
        commit                % ← d(
        put_val X4            % U,
        put_val X1            % X,
        put_val X3            % DU),
        spawn D0              % d(
        put_val X2            % V,
        put_val X1            % X,
        put_val X0            % DV).
        iterate               %.
    D1:
    suspend.
```

Compiling complex structures in the body

test ← do(parse($s(np, vp)$, [birds, fly], [ ])).

```
    E0:
        try_me_else E1              % test
        commit                      % ← do(
        put_tuple 4
        unify_string 'parse'        % parse(
        push X0                     % (save "[birds, fly], [ ]")
        unify_tuple 3
        unify_string 's'            % s(
        unify_string 'np'           % np,
        unify_string 'vp'           % vp),
        pop X0                      % (load "[birds, fly], [ ]")
        push X0                     % (save "[ ]")
        unify_car_string 'birds'    % birds,
    unify_car/string 'fly'      % fly
    unify_nil                   %]
    pop X0                      % (load "[ ]")
    unify_nil                   % [ ]
    execute (offset to do)      %

E1:
suspend.
```

APPENDIX 5: BENCHMARKS

Note. Benchmarks were carried on a slightly older version of the abstract machine.

In Quintus Prolog benchmarks we have tested the program with the following goal: "run(X); run(X)." and the result of both runs is given. The second run is faster, since Prolog has already allocated a stack for the run.

Naive Reverse

```
% FCP (C emulator)
Call: rev(100)
rev(N) ← list(N?, X), rev(X?, _).
list(N, [N|Xs?]) ←
    N > 0, N1 := N − 1 |list(N1, Xs).
list(0, [ ]).
rev([X|Xs], Ys) ←
    rev(Xs?, Zs), append(Zs?, [X], Ys).
rev([ ], [ ]).
append([X|Xs], Ys, [X|Zs]) ←
    append(Xs?, Ys, Zs).
append([ ], Xs, Xs).
```

Without tail recursion optimization (time slice is 1):

```
    Creations:            102
    Suspensions:          101
    Process switches:     5152
    Reductions:           5254
    Speed:                1616.62 LIPS
    Time:                 3250 ms
```

With tail-recursion optimization (time slice is 26):

```
    Creations:            102
    Suspensions:          100
```

```
Process switches:    149
Reductions:          5254
Speed:               1843.51 LIPS
Time:                2850 ms
```

```
% Prolog (Quintus)

run(T) ←
   statistics(runtime, [T1, _]),
   rev(100),
   statistics(runtime, [T2, _]),
   T is T2 − T1.

rev(N) ← list(N, X), rev(X, _).

list(N, [N|Xs]) ←
   N > 0, N1 is N − 1, list (N1, Xs).
list(0, [ ]).

rev[(X|Xs], Ys) ←
   rev(Xs, Zs), append(Zs, [X], Ys).
rev([ ], [ ]).

append([X|Xs], Ys, [X|Zs]) ←
append(Xs, Ys, Zs).
append([ ], Xs, Xs).
```

Time of first run was 700 ms (4.6–4.1 times speedup).
Time of second run was 433 ms (7.5–6.6 times speedup).

Quick Sort

```
%FCP (C emulator)

Call: list(100, X), qsort(X?, _).

qsort([X|List], Sorted − Tail) ←
   partition(X?, List?, Small, Large),
   qsort(Large?, SSorted − Tail),
   qsort(Small?, Sorted − [X|SSorted]).
qsort([ ], X − X).

partition(X, [Y|List], [Y|Small], Large) ←
   X > Y |
   partition(X, List?, Small, Large).
partition(X, [Y|List], Small, [Y|Large]) ←
   X ≤ Y |
   partition(X, List?, Small, Large).
partition(_, [ ], [ ], [ ]).
```

Without tail-recursion optimization (time slice is 1):

```
Creations:      202
Suspensions:    201
```

```
Process switches:    5152
Reductions:          5354
Speed:               634.36 LIPS
Time:                8440 ms
```

With tail recursion optimization (time slice is 26)

```
Creations:           202
Suspensions:         153
Process switches     79
Reductions:          5354
Speed:               707.27 LIPS
Time:                7570 ms
```

```
% Prolog (Quintus)

run(T) ←
   statistics(runtime, [T1, _]),
   qsort(100),
   statistics(runtime, [T2, _]),
   T is T2 - T1.

qsort(N) ← list(N, X), qsort(X, _).

list(N, [N|Xs]) ←
   N > 0, N1 is N - 1, list(N1, Xs).
list(0, [ ]).

qsort([X|List], Sorted-Tail) ←
   partition(X, List, Small, Large),
   qsort(Large, SSorted-Tail),
   qsort(Small, Sorted-[X|SSorted]).
qsort([ ], X - X).

partition(X, [Y|List], [Y|Small], Large) ←
   X > Y,
   partition(X, List, Small, Large).
partition(X, [Y|List], Small, [Y|Large]) ←
   X ≤ Y,
   partition(X, List, Small, Large).
   partition(_, [ ], [ ], [ ]).
```

Time of first run was 4783 ms (1.8–1.6 times speedup).
Time of second run was 1100 ms (7.7–6.9 times speedup).

Hanoi Towers

```
% FCP (C emulator)

boot(_, _) ← hanoi(10).

hanoi(N) ←
   hanoi(N, _).
hanoi(N, X) ←
   hanoi(N, a, c, X).

hanoi(N, From, To, (Before, (From, To), After)) ←
   N > 0, diff(N, 1, N1)|
   free(From, To, Free),
   hanoi(N1,From, Free, Before),
   hanoi(N1, Free, To, After).
hanoi(0, From, To, (From, To)).

free(a, b, c).
free(a, c, b).
free(b, a, c).
free(b, c, a).
free(c, a, b).
free(c, b, a).
```

Without tail recursion optimization (time slice is 1):

```
   Creations:            2047
   Suspensions:          0
   Process switches:     1026
   Reductions:           3073
   Speed:                532.58
   Time:                 5770
```

With tail recursion optimization (time slice is 26):

```
   Creations:            2047
   Suspensions:          0
   Process switches:     0
   Reductions            3073
   Speed:                555.70
   Time:                 5530
```

```
% Prolog (Quintus)

run(T) ←
   statistics(runtime, [T1, _]),
   hanoi(10),
   statistics(runtime, [T2, _]),
   T is T2 - T1.

hanoi(N) ←
   hanoi(N, _).

hanoi(N, X) ←
   hanoi(N, a, c, X).

hanoi(N, From, To, (Before, (From, To), After)) ←
   N > 0, N1 is N - 1,
   free(From, To, Free),
    hanoi(N1, From, Free, Before),
    hanoi(N1, Free, To, After).
hanoi(0, From, To, (From, To)).

free(a, b, c).
free(a, c, b).
free(b, a, c).
free(b, c, a).
free(c, a, b).
free(c, b, a).
```

Time of first run was 2700 ms (2.14–2.0 times speedup).
Time of second run was 1133 ms (5.1–4.9 times speedup).

The design of the abstract machine and the implementation of its emulator were part of a group project, and ideas of many are incorporated in this work. We would like to thank Jim Crammond, Michael Hirsch, Jacob Levy, Colin Mierowsky, William Silverman, and Steve Taylor for their contributions in general, and Shmuel Safra for his contribution to the idea of modules in particular. Shmuel, Steve, and Michael Codish helped with the organization and the writing of this paper.

REFERENCES

1. Clark, K. L. and Gregory, S., A Relational Language for Parallel Programming, in: *Proceedings of the ACM Conference on Functional Languages and Computer Architecture*, 1981, pp. 171–178.
2. Clark, K. L. and Gregory, S., PARLOG: Parallel programming in logic, *ACM Trans. OPLAS* 8(1):1–49 (1986).
3. Hirsch, M., Silverman, W., and Shapiro, E., Layers of Protection and Control in the Logix System, Tech. Report CS86-19, Dept. of Computer Science, Weizmann Inst. of Science, Rehovot, 1986.
4. Kliger, S., Towards a Native-Code Compiler for Flat Concurrent Prolog, M.Sc. Thesis, Dept. of Computer Science, Weizmann Inst. of Science, Rehovot, 1987.
5. Levy, J., A GHC abstract machine and instruction set, in: E. Shapiro (ed.), *Proceedings of the 3rd International Conference on Logic Programming*, Lecture Notes Comput. Sci. 225, Springer-Verlag, 1986, pp. 157–171.
6. Mierowsky, C., Taylor, S., Shapiro, E., Levy J., and Safra, S., The Design and Implementation of Flat Concurrent Prolog, Tech. Report CS85-09, Dept. of Computer Science, Weizmann Inst. of Science, Rehovot, 1985.
7. Nakashima, H., Tomura, S., and Ueda, K., What Is a Variable in Prolog?, in: *Proceedings of the International Conference on Fifth Generation Computer Systems*, Tokyo, 1984, pp. 327–332.
8. *Quintus Prolog Reference Manual*, Quintus Computer Systems Ltd., 1985.
9. Safra, S., Partial Evaluation of Concurrent Prolog and Its Implications, Tech. Report CS86-24, Dept. of Computer Science, Weizmann Inst. of Science, Rehovot, 1986.
10. Shapiro, E., A Subset of Concurrent Prolog and Its Interpreter, ICOT Tech. Report TR-003, Inst. for New Generation Computer Technology, Tokyo, 1983.
11. Shapiro, E., Concurrent Prolog: A Progress Report, *IEEE Computer* 19(8):44–58 (Aug. 1986).

12. Shapiro, E. and Safra, S., Multiway Merge with Constant Delay in Concurrent Prolog, *New Generation Comput.* 4(2):211–216 (1986).

13. Shapiro, E. (ed.), *Concurrent Prolog: Collected Papers*, Volumes 1 and 2, MIT Press, 1987.

14. Silverman, W., Houri, A., Hirsch, M., and Shapiro, E., The Logix System User Manual, Tech. Report CS86-21, Dept. of Computer Science, Weizmann Inst. of Science, Rehovot, 1986.

15. Taylor, S., Av-Ron, E., and Shapiro, E., A Layered Method for Process and Code Mapping, *J. New Generation Comput.*, 5(2) 1987.

16. Taylor, S., Safra, S., and Shapiro, E., A Parallel Implementation of Flat Concurrent Prolog, *J. Parallel Programming* 15(3):245–275 (1987).

17. Ueda, K., Guarded Horn Clauses, in: E. Wada (ed.), *Logic Programming*, *Lecture Notes Comput. Sci. 221*, Springer-Verlag, 1986, pp. 168–179.

18. Warren, D. H. D., Implementing Prolog—Compiling Predicate Logic Programs, Tech. Report DAI 39/40, Dept. of Artificial Intelligence, Univ. of Edinburgh, 1977.

19. Warren, D. H. D., Logic Programming and Compiler Writing, *Software—Practice and Experience* 10:97–125 (1980).

20. Warren, D. H. D., An Abstract Prolog Instruction Set, Tech. Report 309, Artificial Intelligence Center, SRI International, 1983.