Compiling Object Calculi

Luís Lopes   Fernando Silva   Vasco T. Vasconcelos

Technical Report Series: DCC-98-3

Departamento de Ciência de Computadores – Faculdade de Ciências

&

Laboratório de Inteligência Artificial e Ciência de Computadores

Universidade do Porto

Rua do Campo Alegre, 823 4150 Porto, Portugal

Tel: +351+2+6078830 – Fax: +351+2+6003654

http://www.ncc.up.pt/fcup/DCC/Pubs/treports.html

Compiling Process Calculi

Luís Lopes, Fernando Silva
DCC-FC & LIACC
Universidade do Porto
{lblopes,fds}@ncc.up.pt

Vasco T. Vasconcelos
DI-FC
Universidade de Lisboa
vv@di.fc.ul.pt

Abstract

Despite extensive theoretical work on process-calculi, abstract machine specifications and implementations of actual computational models are still scarce. Moreover, a general framework for compiling languages based on these calculi is still elusive.

We propose a core assembly language onto which process-calculi based languages may be compiled and present a specification for a self-contained abstract machine that runs this assembly code. We also include compilation examples of features of well known calculi and discuss possible optimizations. Finally, we report preliminary performance results obtained with an implementation of the abstract machine coupled with an experimental language based on the TyCO-calculus.

Keywords: Process-Calculi, Concurrency, Abstract-Machine, Compilation, Implementation.

1 Introduction

In recent years researchers have devoted a great effort in providing semantics for pure concurrent programming languages within the realm of process-calculi. Milner, Parrow and Walker's π-calculus [9] or an equivalent asynchronous formulation due to Honda and Tokoro [6] has been the starting point for most of these attempts. Other calculi have been proposed that are closely related to the the asynchronous π-calculus. Boudol's Blue-calculus [4] introduces a functional style of programming in the π-calculus by distinguishing between synchronization and communication. Vasconcelos' TyCO-calculus [14] introduces branching structures as an efficient way to model objects and uses explicit recursion to model unbounded behavior. Fournet and Gonthier's Join-calculus [2] is based in the reflexive CHAM [1] and is related to the π-calculus by the fact that restriction, reception, and replication are combined into a single concept. Smolka's γ-calculus [5] is obtained from the π-calculus by distinguishing variables from names, and by making variables logical, thus supporting the treatment of state, constraints, and reflexive computations.

Despite the extensive theoretical work, abstract machine specifications and implementations of actual computational models are still scarce. Moreover, a general framework for compiling languages based on process-calculi is still elusive. This contrasts vividly with other programming paradigms where generic compilation schemes and abstract machine specifications are widely used. The G-machine [7] and the WAM [16] are good examples for functional and logic programming, respectively. Programming languages based on process-calculi have been proposed, such as Pict [13], Oz [12] and Join [3], but their implementations are manifold and often, as in the case of Pict, not based on a self contained design.

We aim to set the basis for a general compilation framework for process-calculi that would provide a universal core system onto which process and communication-based programming languages may be mapped. We identify a basic assembly language involved in the compilation of process-calculi, one that will ultimately be executed by an abstract machine. To demonstrate the flexibility and potential efficiency of this assembly we show how various features of existing calculi can be compiled into the instruction set of the abstract machine and describe feasible optimizations.

We also provide a specification for a self-contained abstract machine that runs programs written in this assembly code. The machine has a very small instruction set and a simple memory model, being essentially a von Neumann architecture. We currently have a full implementation of the abstract machine and a compiler for the experimental language TyCO [15].

The remainder of the paper is organized as follows: section 2 describes the architecture and semantics of the target abstract machine, and its assembly language; section 3 exemplifies the compilation of features from a few calculi in this scheme and presents some optimizations; section 4 presents some performance results we got with the experimental TyCO language, and finally; section 5 overviews the paper's main results and describes future work.

2 The Abstract Machine

This section provides a specification of a self-contained abstract machine that supports the execution of process calculi based programming languages. The machine runs byte-code files compiled either directly from a high-level programming language or assembled from an intermediate machine assembly language.

We want the abstract machine to provide direct support for the following features from process calculi.

Creation of New Names. The creation of new names (here called channels) is directly supported by an instruction. Scope restriction is implemented through variable renaming.

Branching Structures. Objects[1] and messages[2] provide a more general model for the implementation of input and output processes, when compared to the simple prefix of the π-calculus. Method invocation in such objects is a one-way protocol as opposed to a two-way protocol with simple prefix processes. Branching structures also avoid the production of computational garbage in the form of unused names and input processes.

[1]sums of labeled methods
[2]method invocations

Template Processes. Templates are recursive processes abstracted on a set of variables. Although they may be modeled in a template-less framework, their use is so pervasive and useful we chose to directly support it.

Recursion and Replication. Unbounded behavior is modeled using recursive template definitions and replication. The former avoids the proliferation of unused replicated input processes, and provides a more natural formulation for persistence. The latter is useful to provide for efficient compilation of stateless persistent objects (or functions).

Next we discuss the program layout for the machine assembly and its relation with the layout of the source programs.

2.1 Program Layout

Programs in process-calculi are typically composed of a main thread[3]. Often, these processes have nested structures such as: objects in TyCO; definitions in TyCO, Blue and Join; and input-processes in π. When a method invocation or an instantiation occurs these processes generate new threads for execution. So, a program is composed of a main thread plus a set of virtual threads whose execution depends on the flow of computation.

The structure of assembly programs in our framework closely follows the one of original programs in the sense that threads constitute the main organization block for the code. The code for nested threads in a assembly program is flattened. A typical code block for a thread $P_1 \mid \cdots \mid P_k$ is shown below:

```
label:  load environment from frames
        free environment frames
        code for P1
        ...
        code for Pk
        return
```

Each thread is identified by a unique label `label` and is composed of four contiguous sub-blocks. The first block contains instructions that load the arguments and the free variables for the thread from the heap into machine registers. The latter are quite close to actual hardware registers and can be mapped efficiently. Once this is done the heap frames that held the environment can be reused. Some optimizations require that a part of the bindings remain active so that we must have instructions that allow the manipulation of parameter and free variable blocks independently.

Next follows a sequence of instructions with the code for each process in the thread. Since the machine is sequential the instructions are issued for $P_1 \mid \cdots \mid P_k$ in the same order as the processes appear in the source code. Notice that this code is made up of simple, non-nested instructions. Finally, we return from the thread code. On returning, the machine checks if there are any more threads available for execution. If there are other threads it picks one and runs it, otherwise the machine halts.

[3] a set of concurrent processes at the same nesting level

Compiling objects requires the use of tables of code labels to implement method tables. The former appear in between thread blocks in the assembly code. They may not appear at all in compilations of calculi which do not require the full power of branching structures, e.g., the π-calculus. The layout of such tables is as follows:

$\texttt{label:}\ \ \texttt{label}_1\texttt{,...,label}_k$

2.2 The Computational Model

Threads in a program are executed without interruption starting with the `main` thread. Within each thread, processes are executed sequentially. The main drivers of the computation are method invocation (reduction) and instantiation. These produce new runnable threads which are added to a run-queue. Processes also perform other operations such as the introduction of local variables bound to new channels, the binding of variables with values and the definition of new template processes.

Reduction implies that compatible messages and objects must be syntactically adjacent. In a real system, run-time re-ordering of processes is not feasible. We follow Turner's[13] solution for representing channels as queues where messages (respectively objects) wait for compatible objects (respectively messages) to reduce. Reduction is eager in the sense that it occurs as soon as a redex is found.

Processes manipulate channels through variables dynamically introduced with the scope restriction construct. Following the representation of channels a new empty queue is associated with each newly introduced variable. Variables have well defined contexts and unique channel bindings. They are ephemeral, as opposed to the channels they are bound to which may last for the entire program.

Static binding is imposed so that referential transparency is preserved. This means that the bindings for the free variables of objects and definitions must be saved at the time the object (definition) is found and not when reduction (instantiation) occurs.

When a reduction occurs, the method invoked by a message becomes a new thread of computation with an environment formed by the bindings for both parameters (from the message) and free variables (held with the object). A template instantiation produces a new thread immediately. This is the body of the template plus the bindings for the parameters and those for the free variables.

New threads are kept in a queue and selected by a scheduler for execution. The computation stops whenever the scheduler is invoked and there are no more runnable threads.

2.3 Memory Organization

The abstract machine has a very simple memory layout, being composed of two memory areas (see fig. 1). The first area keeps the program with the instructions for threads, space for template free variables, and method tables. The layout of the individual instructions is discussed below. The second area is an heap and is used for dynamically growing structures and includes the run-queue and the communication queues.

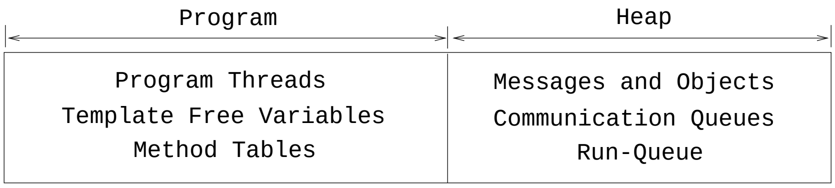

Figure 1: Memory Layout

2.4 Memory Word Format

The abstract machine operates on two basic internal data-types: `Word` and `Frame`. We use `Word` whenever we wish to refer to a single machine word whereas `Frame` points to blocks of contiguous machine words. To make the code more readable we sometimes label particular words or subsets of (contiguous) words of a frame. To represent access to the contents of a labeled word in a frame we use the common field selection operator "`.`". For example, the word labeled `t` (for table) in a frame held in a register `CF` is represented by `CF.t`. The machine is parameterized by the size chosen for `Word` making it easy to switch from (to) 32 or 64-bit implementations. Words have no tags.

2.5 Static Data: the Program

Threads. Threads are sequences of contiguous instructions ending in a `ret` instruction. A program is composed of a collection of threads. Execution starts with the `main` thread. Other runnable threads are generated through reduction and instantiation, and are placed in the run-queue to be executed. Once the current thread finishes, the next thread in the run-queue is taken and executed. Threads are executed within an environment that keeps bindings for the variables. The bindings for the free variables and for the parameters are kept in two dynamic memory frames pointed to by the machine registers `FV` and `PM`, respectively. This keeps the thread interface uniform throughout the program and simplifies the representation of messages and objects. The free variable bindings for a thread are taken from the object frame whereas, the parameter bindings are taken from the message frame. Instantiation parameters are kept in a new frame whereas free variables are taken from previously stored locations in the static area. Registers `FV` and `PM` are updated each time a new thread is taken from the run-queue for execution.

Local Variables. A running thread typically creates new local variables. These variables stand for originally empty communication queues (channels) that can be used for future computations. The scope of such variables is the thread they are created in. The channels however remain in existence. *Scope extrusion* allows channels to escape the context of a thread and become visible to other threads (e.g., by sending a local channel in a message targeted outside the scope of the channel). Since the exact number of channels created by each thread is known at compile time we use generic registers to rename new variables. These registers are reused for each new thread thus eliminating previous bindings, but not the underlying channels themselves.

Method Tables. A method table is a frame with at least one word (objects with a single method do not have a table). Each word in the frame contains the offset of the first instruction of a particular method. Words in method tables are labeled with naturals starting from 0. Given a message label n, the address for the corresponding method is obtained by inspecting the n^{th} word in the method table.

Template Free Variables. Templates are threads parameterized on a sequence of pair-wise distinct variables. The free variables of a template are bound to their values where the template is defined in the program. The bindings for the template free variables are stored in statically allocated space in the program. The size of these frames is known at compile time. This allows us to treat parameter bindings in an object instantiation of a template the same way as free variables are treated in common objects(see fig. 2).

2.6 Dynamic Data: The Heap

Frame Heap. An heap is an address space where the abstract machine data structures evolve dynamically. The abstract machine requires that heap implementations provide two operations satisfying the following signatures:

$$\mathtt{New} : \mathtt{Nat} \mapsto \mathtt{Frame} \qquad\qquad \mathtt{Free} : \mathtt{Frame} \mapsto \mathtt{Void}$$

`New` receives a natural number specifying the size of the block required and returns a (pointer to a) frame of that size. `Free` takes a (pointer to a) frame and releases it so that it may be re-used.

2.7 Frames.

Object Frames. Object frames have at least one word. It is named `t` (for table) since it is a pointer to a method table. Additional words may exist and in this case they are the bindings for the object's free variables. We name the first such word (right after `t`) as `fv` (see fig. 2).

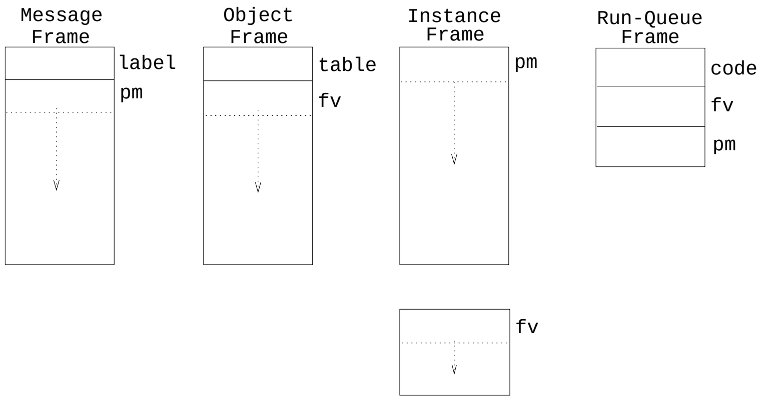

Figure 2: Message, Object, Instance and Runnable Thread frames

Message Frames. A message frame is a block with at least one word. This word is named `l` (for label) and is a natural number index to some method table. Additional words may exist in which case they are the message arguments. We name the first such word (right after `l`) as `pm` (see fig. 2).

Instance Frames. Instance frames are blocks with at least one word (instances without parameters do not require frames). These words contain the instance arguments and are generically referred to as `pm` (see fig. 2).

Queues. Queues are used to implement both the run-queue and the channel (communication) queues. The abstract machine requires that implementations of queues provide four operations with the following signatures:

$$\begin{array}{rlrl} \texttt{NewQueue}: & \texttt{Void} \mapsto \texttt{Queue} & \texttt{Enqueue}: & \texttt{Queue} \times \texttt{Frame} \mapsto \texttt{Void} \\ \texttt{Empty}: & \texttt{Queue} \mapsto \texttt{Bool} & \texttt{Dequeue}: & \texttt{Queue} \mapsto \texttt{Frame} \end{array}$$

`NewQueue` initializes an empty queue. `Empty` checks whether a given queue is empty. `Enqueue` takes a queue and a frame and places the frame in the (back of the) queue. `Dequeue` takes a queue and returns a frame from (the front of) the queue.

Communication Queues. Communication queues or channels implement process calculi names. A channel is a queue of object or message frames extended with the following operation:

$$\texttt{State}: \texttt{Queue} \mapsto \{\texttt{Empty}, \texttt{Messages}, \texttt{Objects}\}$$

where `Empty`, `Messages`, and `Objects` constitute an enumeration. `State` returns `Empty` if the queue is empty, `Messages` if it contains at least one message frame, and `Objects` if the queue contains at least one object frame. An invariant of the abstract machine states that, at any given moment of the computation, no channel can contain a mixture of objects and messages [8, 13].

The use of queues to hold message and object frames in channels introduces some fairness into the abstract machine in the sense that all messages or objects waiting on a channel may eventually reduce to a runnable thread.

Process calculi are well known for their pervasive use of channels even for relatively simple programs. Efficient management (including garbage collection) of channels requires powerful compilation techniques combined with type inference information [10, 13].

The Run Queue. The items in the run-queue are frames of size three, and represent runnable threads. For convenience we name the words in these frames respectively as `cd`, `fv`, and `pm` (see fig. 2). When a reduction occurs, a new frame is placed in the run-queue. Then the machine uses the integer `l` in the message frame and the method table offset in the object frame (`t`) to find out which thread to run. The offset for this thread is placed in the word `cd` of the new frame. The word `fv` (`pm`) keeps the pointer to the object (message) frame where the free variable (parameter) bindings are kept. The name `RunQueue` is used to refer to the machine's run-queue.

2.8 Registers

The abstract machine uses a small set of registers to control the program flow, to hold frame pointers and local channels, and to manage the run-queue (see table 1). Register `PC` (Program Counter) points to the instruction to be executed. The register is initialized with the address of thread `main`, the first to be invoked for every program (the main thread). Register `CC` (Current Channel) points to the channel which is currently being used to try a reduction. Register `CF` (Current Frame) holds (object or message) frames temporarily until they are enqueued or used in a reduction. If a reduction takes place, the frame for the other redex's component is kept in the register `OF` (Other Frame). Frames for runnable threads are temporarily kept at register `NT` (New Thread). Frames are placed in `NT` during reduction, when a new thread is being built and also when a new thread is fetched from the run-queue to be executed. Registers `FV` (Free Variable bindings) and `PM` (ParaMeter bindings) are used to hold the free variable and parameter bindings, respectively. Generic registers of the form `R`n, where n is a natural, are used to hold parameter and free variable values as well as pointers to local channels.

Register	Description
`PC`	Program Counter
`CC`	Current Channel
`CF`	Current Frame
`OF`	Other Redex Component Frame
`FV`	Free Variable Bindings
`PM`	Parameter Bindings
`NT`	New Runnable Thread
`R`n	Generic Registers

Table 1: The register set

2.9 Instruction Set

Instructions are of fixed size and identified by the emulator by inspecting the opcode, which is the word at the offset of the program counter `PC`. The type of instruction determines the number of arguments that follow in contiguous words.

`newf n` allocates a frame with `n` words and places a pointer to it in register `CF`.	`CF := New(n);` `PC := PC+2;`
`newc r` creates a new channel in the heap and places a pointer to it in register `r`.	`r := NewQueue();` `PC := PC+1;`
`loadpm n,r` loads the contents of the word at offset `n` in the frame `PM` in the register `r`.	`r := PM.n;` `PC := PC+3;`
`loadfv n,r` loads the contents of the word at offset `n` in the frame `FV` in the register `r`.	`r := CF.n;` `PC := PC+3;`
`loadtfv n,r,X` loads the contents of the word at offset `n` in template free variable frame for `X` into the register `r`.	`r := X.n;` `PC := PC+4;`
`store n,u` stores the value `u` in the word at offset `n` in the frame currently pointed to by `CF`.	`CF.n := u;` `PC := PC+3;`
`store n,u,X` stores the value `u` in the word at offset `n` in the template free variable frame for `X`.	`X.n := u;` `PC := PC+4;`
`freefv` frees the free variable bindings frame of the current environment.	`Free(FV);` `PC := PC+1;`
`freepm` frees the parameter bindings frame of the current environment.	`Free(PM);` `PC := PC+1;`

`trobj r` loads the channel pointer in register `r` into `CC` . Then it checks the state of the channel. If the state is `Messages` then it dequeues a message frame (from `CC`) and creates a new runnable thread in the run-queue as the result of the reduction between the object at `CF` and the message at `OF`. The registers `CF` and `OF` are complementary in the sense that each holds a redex component for reduction. If the channel states is either `Empty` or `Objects` the current frame is enqueued at `CC`.

```
CC := r;
CASE State(CC) OF
 Empty:
  Enqueue(CC,CF);
 Objects:
  Enqueue(CC,CF);
 Messages:
  OF:=Dequeue(CC);
  NT:=New(3);
  NT.cd:=(CF.t)[OF.l];
  NT.fv:=CF;
  NT.pm:=OF;
  Enqueue(RunQueue,NT);
PC:=PC+2;
```

`trmsg r` loads the channel pointer in register `r` into `CC` . Then it checks the state of the channel. If the state is `Objects` then it dequeues an object frame (from `CC`) and creates a new runnable thread in the run-queue as the result of the reduction between the message at `CF` and the object at `OF`. If the value is either `Empty` or `Messages` the current frame is enqueued at `CC`. Notice the symmetry of the code. Only registers `CF` and `OF` exchange roles as they alternatively point to message frames or to object frames.

```
CC := r;
CASE State(CC) OF
 Empty:
  Enqueue(CC,CF);
 Messages:
  Enqueue(CC,CF);
 Objects:
  OF:=Dequeue(CC);
  NT:=New(3);
  NT.cd:=(OF.t)[CF.l];
  NT.fv:=OF;
  NT.pm:=CF;
  Enqueue(RunQueue,NT);
PC:=PC+2;
```

`ret` checks whether the run-queue is not empty in which case it dequeues a runnable thread. If the run-queue is empty it halts the machine. Before running the thread the registers `PC`, `FV` and `PM` are updated with the values in the words `cd`, `fv` and `pm`, respectively.

```
IF Empty(RunQueue)
 goto halt;
ELSE
 NT:=Dequeue(RunQueue);
 PC:=NT.cd;
 FV:=NT.fv;
 PM:=NT.pm;
```

2.10 The Emulator Loop

The engine of the abstract machine is a standard [11, 16] emulator loop. Instructions are executed sequentially following a program counter register. Each instruction is identified by its opcode. The arguments, if they exist, follow in the next adjacent words. After an instruction is executed the program counter is incremented by the size of the instruction and the engine jumps to the address of the next instruction. The program terminates by jumping off the emulator loop, when the current thread ends and the run-queue is empty.

3 Compiling Process Calculi

In this section we demonstrate the flexibility and potential efficiency of the assembly language of the abstract machine, by giving several compilation examples of features from a number of existing process calculi. The calculi studied here are the asynchronous π-calculus [9, 6], Boudol's Blue-calculus [4] and Vasconcelos' TyCO [14].

3.1 Creating a New Channel

Perhaps the most pervasive operation in a process calculus, channel creation: $(\nu x)P$, is implemented by the instruction **newc** which has as a register argument, say **r**, where the pointer to the new channel is kept.

```
newc   r    % create new channel and make r point to it
Code   P
```

3.2 Guarded Input Process

A π-calculus guarded input process $u?(\vec{x}).P$, is compiled to the following code, assuming u in register $\mathtt{r}_0$:

```
     newf       2        % new frame of size 2
     store      0,m0     % table: pointer to the code of P
     store      1,r0     % fv[0]: free variable r0
     trobjs     r0       % simple prefix optimization

m0:  code for P
```

The instruction `trobjs`, and its dual `trmsgs`, is an optimization for the case of simple prefix processes. Here the target object of a message contains only one method. For this reason the code for the object contains a direct reference to the method's code and the label entries for messages to such an object are omitted. The code for `trobj` and `trmsg` is changed just by adjusting the way the label for the thread's code is obtained: `NT.cd := CF.t` for `trobjs` and `NT.cd := OF.t` for `trmsgs`.

3.3 Guarded Output Processes

Guarded output processes such as $x![\vec{u}].Q$ may be used for synchronous communication. The compilation is slightly different. First we observe that Q does not have input parameters, just free variables if any. Second, on reduction with a corresponding guarded input process, Q must be executed with those bindings untouched, as follows:

$$x![\vec{u}].Q \mid x?(\vec{y}).P \longrightarrow \{\vec{u}/\vec{y}\}P \mid Q$$

So we must store Q's free variable bindings in the frame for the message. Assume the arguments $\vec{u}$ in $\mathtt{r}_1, \ldots, \mathtt{r}_k$, x in register $\mathtt{r}_0$ and the free variables of Q in $\mathtt{r}_{k+1}, \ldots, \mathtt{r}_{k+l}$:

```
newf       2+k+l        % new frame of size 2+k+l
store      0,k          % offset to P's free variable bindings
store      1,r1         % PM[0]
...
```

```
     store       k,r_k           % PM[k-1]
     store       k+1,m_0         % reference to code for Q
     store       k+2,r_{k+1}     % FV[0]
     ...
     store       k+l+2,r_{k+l+1} % FV[l-1]
     trmsgc      r_0   % try selection with continuation

m_0: Code for Q
```

When reduction occurs, the message frame is used to keep the bindings for two threads that will be created with the code for P and the code for Q. The code for **trmsgsc** is similar to that of **trmsgs** (see above) but has to handle the particular layout of the message frame. We have for the dual instructions the following code for reduction:

```
Objects:                          Messages:
    k = *CF;                          k = *OF;
    NT.cd := *OF;                     NT.cd := *CF;
    NT.pm := CF+1;                    NT.pm := OF+1;
    NT.fv := OF+1;                    NT.fv := CF+1;
    Enqueue(RunQueue,NT);             Enqueue(RunQueue,NT);
    NT.cd := *(CF+k+1);               NT.cd := *(OF+k+1);
    NT.pm := 0;                       NT.pm := 0;
    NT.fv := CF+k+2;                  NT.fv := OF+k+2;
    Enqueue(RunQueue,NT);             Enqueue(RunQueue,NT);
```

3.4 Replicated Input Processes

We want the assembly for a process of the form $!x(y_1 \ldots y_k).P$. Replicated processes allow considerable optimization since: a) they have exclusive use of a channel; b) they always occur before any message to the channel. From this it follows that the code for an object of this kind just places a persistent closure for the object at a given channel and any incoming message will automatically generate a new thread of computation. We use the instruction **trobjsr** to create a replicated simple prefix process at x, and the dual **trmsgsr** to reduce with such processes. We assume x a channel in register $\mathbf{r}_0$ and a set of k free variables placed in registers $\mathbf{r}_1, \ldots ,\mathbf{r}_k$.

```
     newf        k+1       % make frame for single object
     store       0,m_0     % pointer to code of P
     store       1,r_0     % FV[0]
     store       2,r_1     % FV[1]
     ...
     store       k,r_k     % FV[k-1]
     trobjrs     r_0       % try to reduce the replicated input

m_0: code for P            % the code for thread P
     ret                   % return control to scheduler
```

The code for the dual instructions is the following:

```
trobjsr(r)                 trmsgsr(r)
   CC := r;                   CC := r;
   Enqueue(CF,CC);            DeQueue(CC,OF);
   PC := PC+2;                NT.cd := *OF;
                              NT.pm := CF;
                              NT.fv := OF+1;
                              EnQueue(RunQueue,NT);
```

3.5 Branching Structures in TyCO

TyCO's branching structures:

$$u?\{p(\vec{x}) = P, q(\vec{y}) = Q, r(\vec{z}) = R\}$$

are compiled very straightforwardly. We start by computing the object's free variables and store their values along with the method table reference in a frame. Then we insert an instruction that tries to reduce the object thus formed. Each method of the object is then (recursively) compiled. Here we assume u at $\mathtt{r}_0$ and the free variables in $\mathtt{r}_1, \ldots, \mathtt{r}_k$:

```
      newf      k+1       % new frame for object
      store     0,o_u     % method table reference
      store     1,r_1     % fv[0]
      ...
      store     k,r_k     % fv[k-1]
      trobj     r_0       % try to reduce the object

o_u:  p, q, r

p:    code for P
      ret
q:    code for Q
      ret
r:    code for R
      ret
```

3.6 Definitions in Blue

Definition molecules in the Blue calculus provide the means to evaluate a resource on a name and to use it in the context of a continuation process. Given a resource fetching molecule such as $\langle x = N \rangle$, we can generate code by observing that such an operation corresponds essentially to the definition of a replicated input process. Molecules such as $\langle x \Leftarrow N \rangle$ are defined similarly but are not replicated since access to the resource is granted only once. So assume N to have k free variables previously stored in registers $\mathtt{r}_1, \ldots, \mathtt{r}_k$, and x stored in $\mathtt{r}_0$. The code for $\langle x = N \rangle$ is as follows:

```
      newf      k+1          % new frame for a object
      store     0,m_0        % with a method at m_0
      store     1,r_1        % FV[0]
      store     2,r_2        % FV[1]
      ...
      store     k,r_k        % FV[k-1]
      trobjsr   r_0          % this is a replicated input

m_0:  code for N             % code for N
      ret
```

Here we used the replicated, simple prefix process optimization to model persistent resources in Blue.

3.7 "Small" β-reduction in Blue

Blue's "small" β-reduction: $(\lambda x_1 \ldots x_n)Pu_1 \ldots u_n \to P[\vec{u}/\vec{x}]$, can be compiled efficiently the following way. We have three cases to consider:

$$(\lambda x_1 \ldots x_n)Pu_1 \ldots u_k$$

where either $k > n$ (more arguments than parameters), $k = n$ (full application) and $k < n$ (partial application). The case $k > n$ reduces to $k = n$ using associativity:

$$(\lambda x_1 \dots x_n)Pu_1 \dots u_k \equiv ((\lambda x_1 \dots x_n)Pu_1 \dots u_n)u_{n+1} \dots u_k$$

We compile abstractions as special replicated input processes on fixed channels and having parameters $x_1 \dots x_n$. The reduction code for such processes however is specialized to deal with partial application. When an abstraction is found, a replicated process is placed in a channel with a special layout. As usual, the first word keeps state and garbage collection information. The following words keep (respectively) a pointer to a frame where the code for the body of the abstraction (P) and the bindings for its free variables are kept and, finally, a pointer to a frame with the parameter bindings collected so far (null otherwise). So, an abstraction $(\lambda x_1 \dots x_n)P$ is created at r_0 by creating a frame with the free variables for P assumed to be in registers $\mathtt{r}_1, \dots, \mathtt{r}_k$ and executing the instruction `func`.

```
     newf   l+1            % new instance frame
     store  0,n            % number of arguments
     store  1,m0   % the code for P
     store  2,r1           % first free variable
     ...
     store  l+1,rl         % last free variable
     func   r0         % create an abstraction

m0:  code for P
     ret
```

When an application with k arguments in registers $\mathtt{r}_1, \dots, \mathtt{r}_k$ occurs we have to test k to check if either a new thread or a new partial function will be created. This is done with the instruction `apply`. We would have the code:

```
     newf   k              % new instance frame
     store  0,r1           % first argument
     ...
     store  k-1,rk       % last argument
     apply  k,r0         % create a function with n-k arguments at r0 or a thread
```

The code for `func` and `apply` can be written as follows:

```
func(r)                 apply(k,r)
   CC := r;                 OF := CC.last;
   CC.last = CF;            n  := *OF;
   PC := PC+1;              if( k = n )
                               NT.cd := *(OF+1);
                               NT.pm := CF;
                               NT.fv := OF+2;
                               EnQueue(RunQueue,NT);
                            else
                               CC := r;
                               CC.last := OF;
                               *(CC.last) := n-k;
                               CC.first := CF;
```

Notice that all the channels created here are linear.

4 Implementation and Performance

We have a full implementation of the abstract machine specification. The abstract machine is quite compact, the system being implemented in just 2k C code lines. The architecture of the system is depicted in figure 3.

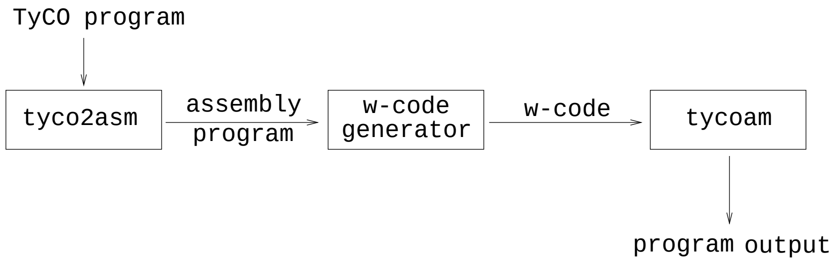

Figure 3: The System Architecture

4.1 The w-Code

The abstract machine executes what we call *word-code* (w-code hereafter), since the basic instruction block has the size of a machine word. The w-code files are obtained from assembly text files using a w-code generator (an assembler). While this is not the best approach in terms of space (e.g., opcodes do not need a full machine word) it avoids in a first approach the overheads of tagging words and the complexities of memory alignment when manipulating distinct types of values. The assembler program is basically a one-pass parser armed with a symbol table to catch and resolve labels as program offsets. The syntax of an instruction's arguments in an assembly program defines the addressing mode being used. There are only three modes allowed by the abstract machine: constant, register and indexed access. The assembler unfolds each assembly instruction into one or more variants according to the addressing mode being used.

The layout of individual instructions is quite simple. A first word for the opcode, followed by one word per argument. The opcode of an instruction determines exactly how many arguments it is followed by. Instructions in a thread are all contiguous. The space in between thread code can be occupied by method tables, slots for template free variables or constant strings. These are also kept in contiguous words and do not interfere with the program's execution as far as the labels in it are correctly resolved by the assembler. The first word of a program is a special word where a magic number is inserted, to support the distribution of w-code files across distinct hardware architectures.

4.2 Preliminary Performance

To prove the validity of our approach we developed an experimental programming language based on the TyCO [14] process calculus and compiled it into the machine assembly. The current implementation only benefits from very basic machine optimizations and two built-in data-types: booleans and integers. Optimizations such as tail recursion, synchronous communication and others requiring type information are not yet implemented. The implementation of a type-inference system for TyCO is underway. Once this module is added to the system a larger spectrum of optimizations becomes feasible and better performances may be achieved.

We used a set of benchmarks to gauge the time and memory performance of the implementation as well as the size of the executable files produced relative to the π-calculus based language - Pict. We did not use Join in these tests since it is not a performance oriented language and so it would not be a fair term for comparison.

The benchmark programs are: `esieve` n, the Sieve of Eratosthenes, finds all prime numbers up to the natural n; `queens` n computes the number of solutions of the nqueens problem for a natural n, and; `mirror` n`k` takes a tree with $2n$ nodes, leafs and buds, and mirrors it, recursively. They were implemented both in TyCO and Pict (version 4.1) and only use booleans and integers as built-in data-types.

We were mainly interested in the time and memory performance of branching structures when handling complex data structures such as the trees in `mirror` and which are pervasive in any high-level programming language. We used objects encoded with simple prefix processes in Pict as the basis for comparison.

Another important issue is the size of the executable files generated by both compilers, since we are interested in supporting architecture independent distributed computing.

	esieve 10000	queens 8	mirror 5k
pict	2.28s 46(15)kb/280(245)gc	0.25s 50(16)kb/39(33)gc	0.56s 713(220)kb/32(27)gc
tycoam	4.36s 0.9(0.3)kb/29gc	0.73s 4(0.7)kb/5gc	0.52s 813(108)kb/3gc
TyCO/Pict	1.91	2.92	0.93

Table 2: Performance Results

Table 2 shows, for each benchmark, the execution time[4] in seconds and, in a second row, the size of the w-code and binary files (uncompressed and compressed in `kbytes`) and the number of garbage collections performed. For Pict we also give, in between parenthesis, the number of fast garbage collections. These just involve a shift of a memory block in the heap. The last row in the table shows the TyCO-to-Pict ratio. All the values observed were obtained with a default heap space of 256k words, in a 166MHz i586 machine with 32Mb main memory and running Linux. The overall results show that although TyCO is still in an early stage of development its performance is within range of that of Pict.

TyCO's results for the `mirror` benchmark are encouraging since this benchmark manipulates a large and complex data-structure that is best mapped as a branching structure. This is proved by the consistently smaller number of garbage collections performed by TyCO as opposed to Pict. Furthermore, currently TyCO implements very few optimizations and does not benefit from type information. The latter is particularly important in this benchmark where linear channels are pervasive, and thus TyCO's performance could be highly increased.

Benchmarks such as `esieve` and `queens` just use simple prefix input processes, and achieve better performances in Pict. However, most of the channels produced by the benchmarks are linear and there are also many opportunities for tail recursion optimization, both of which are not yet implemented in TyCO. This is even clearer with `queens` as it uses mainly *functional* patterns of communication.

Another important issue to note here is the size of binary files as opposed to w-code files

[4]the smallest observed value in a sequence of 10 runs.

(currently not optimized for space, e.g., an instruction opcode takes a full machine word). With the exception of `mirror` which has a large input data-structure, the other benchmarks are a full order of magnitude smaller than Pict's binaries. When we analise the size of the compressed files then it becomes clear that the TyCO assembler generates quite small, compressable w-code files that may be efficiently distributed with the advantage of being arquitecture independent. This is a major issue to us since we want to support efficient distributed computing in the future.

5 Conclusions and Future Work

We presented a general framework for the compilation of process-calculi based programming languages. This framework couples with a specification of a self-contained abstract machine and a w-code assembler for run-time support. The compactness and flexibility of the machine assembly was demonstrated with the compilation of features from well known process calculi. Finally, the validity of this approach was tested with an experimental language based on the TyCO process calculus. Preliminary results are promising and there is scope for manifold optimizations.

Future work will focus on performance evaluation and fine tuning of the abstract machine implementation and TyCO compiler. This work will comprise: the optimization of the basic abstract machine data-structures such as the heap or the communication queues; the introduction of type (or type related) annotations in the programs to allow assembly optimizations (e.g., new specialized instructions); a greater sophistication of the TyCO compiler in recognizing patterns that may be optimized such as synchronous communication or tail recursion. A type-system [15] for TyCO currently being developed will latter be incorporated into the compiler and the program annotations dropped. After these optimizations have been incorporated in the implementation, the next step will be the development of a small high level concurrent programming language based on TyCO with constructs for abstract datatypes, a functional core and common communication patterns.

Acknowledgements The authors are very grateful to Vítor Santos Costa for his comments on this paper. Discussions with Gérard Boudol, Luís Caires, António Ravara and Silvano Dal-Zilio have been very beneficial, for which we are grateful. The authors are partially supported by projects Dolphin (contract PRAXIS/2/2.1/TIT/1577/95) and Escola (contract PRAXIS/2/2.1/MAT/46/94).

References

[1] Gérard Berry and Gérard Boudol. The Chemical Abstract Machine. *Theoretical Computer Science*, (96):217–248, 1992.

[2] Cédric Fournet and Georges Gonthier. The Reflexive Chemical Abstract Machine and the Join-Calculus. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, pages 372–385, St. Petersburg Beach, Florida, January 21-24 1996. ACM.

[3] Cédric Fournet and Luc Maranget. *The Join-Calculus Language (release 1.02)*. Institute National de Recherche en Informatique et Automatique, June 1997.

[4] G. Boudol. The π-calculus in Direct Style. In *Proceedings of POPL'97*. Springer-Verlag, January 1997.

[5] G. Smolka. A Fundation for Higher-Order Concurrent Constraint Programming. In *1st International Conference on Constraints in Computational Logics*, volume 845 of *Lecture Notes in Computer Science*, pages 50–72. Springer-Verlag, September 1994.

[6] K. Honda and M. Tokoro. An Object Calculus for Asynchronous Communication. In *5th European Conference on Object-Oriented Programming*, volume 512 of *LNCS, Springer-Verlag*, pages 141–162, 1991.

[7] Simon P. Jones. *The Implementation Of Functional Programming Languages*. Prentice-Hall, London, Great Britain, 1987.

[8] L. Lopes and V. Vasconcelos. An Abstract Machine for an Object-Calculus. Technical report, DCC-FC & LIACC, Universidade do Porto, May 1997.

[9] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes (parts I and II). *Information and Computation*, (100):1–77, 1992.

[10] B. Pierce N. Kobayashi and D. Turner. Linearity and the π-calculus. In *ACM Symposium on Principles of Programming Languages*, 1996.

[11] Simon L. Peyton Jones. Implementing Lazy Functional Languages on Stock Hardware: the Spineless Tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, July 92.

[12] G. Smolka. The Definition of Kernel Oz. Technical report, Programming Systems Lab, German Research Center for Artificial Intelligence (DFKI), November 1994.

[13] David Turner. *The Polymorphic Pi-calculus: Theory and Implementation*. PhD thesis, University of Edinburgh, 1995.

[14] V. Vasconcelos and M. Tokoro. A Typing System for a Calculus of Objects. In *1st International Symposium on Object Technologies for Advanced Software, LNCS*, volume 742, pages 460–474. Springer-Verlag, November 1993.

[15] Vasco T. Vasconcelos and Rui Bastos. Core-TyCO - The Language Definition. Technical Report TR-98-3, Departamento de Informática / FCUL, March 1998.

[16] David H. D. Warren. An Abstract Prolog Instruction Set. Technical Report 309, SRI International, Menlo Park, CA, October 1983.