A Framework for Compiling Object Calculi

Luís Lopes, Fernando Silva, Vasco T. Vasconcelos

Technical Report Series: DCC-97-12

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

A Framework for Compiling Object Calculi *

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

Vasco T. Vasconcelos — DI-FCUL, Universidade de Lisboa, vv@di.fc.ul.pt

November 20, 1997

Abstract

We give a specification for an abstract machine for object calculi. The instruction set is intended to work as a target assembly language to which high level concurrent languages may be compiled. As a case example the compilation of the TyCO object calculus is treated. Some preliminary results on a prototype implementation are given.

Keywords: Object, Concurrency, Abstract-Machine, Language, Implementation.

1 Introduction

Efforts to provide foundations for concurrent object-oriented languages have relied on two fundamental approaches: the concept of *actors* [2] with the associated computational model, and encodings of objects into Milner, Parrow and Walker's π-calculus [15] or an equivalent asynchronous formulation due to Honda and Tokoro [6].

Despite extensive theoretical research on the subject, abstract machine specifications and implementations of actual models are still scarce. Some actor languages have been implemented with some success such as the ABCL family [27]. Moreover, most existing concurrent object-oriented languages incorporate aspects of the actors model in their specifications. A few programming languages based on process calculi have been purposed. Some, such as Pict [22] and Oz [12] have gained some prominence. Pict [22] is a programming language based on the asynchronous π-calculus. The run-time system is based on a formal abstract machine specification and is implemented in C. However it is not self-contained (since it borrows many features from the C language). Oz [13] is based on the γ-calculus and combines the functional, object-oriented and constraint logic programming paradigms in a single system. The Oz abstract machine is fully self contained, WAM [5]-like, and its implementation is inspired in the AKL [9] machine.

In recent years researchers have devoted a lot of effort in providing semantics for objects within the asynchronous π-calculus [6, 14, 19, 25], variations [3, 23] and extensions [4] (for constraint programming). Typed Concurrent Objects - TyCO [24], is essentially the (asynchronous) π-calculus plus branching structures. Branching structures appear as a pair of operators: selection (or labeled messages), which replace the output processes of the π-calculus, and branching (or objects) which replace the input processes (or prefix) of the π-calculus. TyCO has the same computational power

*This work was partially supported by projects Dolphin (contract PRAXIS/2/2.1/TIT/1577/95) and Escola (contract PRAXIS/2/2.1/MAT/46/94).

as the asynchronous π-calculus [24] and is reminiscent of the Abadi and Cardelli's ς-calculus [1] in that objects are sums of labeled methods (the name *self* being interpreted as the channel where the object is held) and messages can be seen as asynchronous method invocations. The ς-calculus however presents a method update operation which is not present in TyCO.

We developed the theoretical framework for an abstract machine capable of executing programs written in a TyCO idiom [11]. The machine is sound and has important run-time properties such as the impossibility of deadlock in well-typed programs [10]. In this paper we present a specification for a self-contained abstract machine for object calculi. In its present form the machine has a single-threaded architecture, with a very small instruction set and a simple memory layout. We show how a very simple language based on TyCO can be compiled to the instruction set of the abstract machine. Our goal is to provide an universal framework for compiling concurrent, process and communication-based programming languages, and providing them with a run-time system. Several such languages can be easily encoded in TyCO, which may be used as an intermediate language. We developed an implementation for the abstract machine (a run-time system), and a compiler for TyCO with the abstract machine instruction set as the target language. We compare some preliminary performance results with Pict and Haskell.

The remainder of the paper is organized as follows: section 2 describes the architecture and semantics of the abstract machine; section 3 introduces some important optimizations in the design of the machine; section 4 introduces TyCO and describes its compilation into the abstract machine instruction set; section 5 presents some preliminary performance results; and finally, sections 6 and 7 describe, respectively, some related research and future research directions in our own work.

2 The Abstract Machine

This section provides the specification for the abstract machine. We assume that the implementation language offers the following basic capabilities: a) memory can be accessed with offsets from a base position (indexed access); b) static arrays of addresses can be created at compile-time, and; c) static space can be reserved in memory and used at run-time. To the best of our knowledge all current programming languages and assemblers provide these facilities directly or indirectly.

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

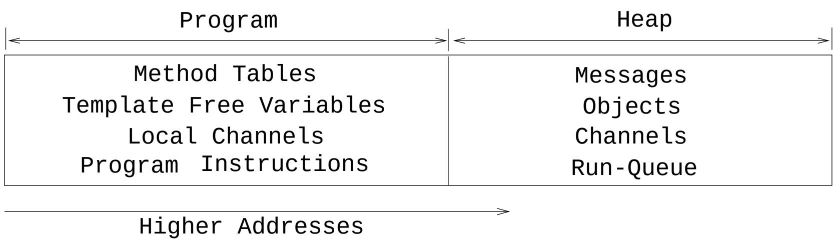

Figure 1: Memory Layout

The abstract machine operates on two basic internal datatypes: `Word` and `Frame`. We use `Word` whenever we wish to refer to a single machine word whereas `Frame` is used to point 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 `table` in a frame held in a register `CF` is represented by `CF.table`. The machine is parameterized by the size chosen for `Word` making it easy to switch from (to) 32 or 64-bit implementations.

2.1 Static Data

Threads. Threads are sequences of instructions. A program is composed of a collection of threads. Execution starts with the special thread `T0`. Other runnable threads are generated through reduction 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 the parameters are kept in two frames pointed to by the machine registers `FB` and `AB`, respectively. This keeps the thread interface uniform throughout the program and also profits from the representation of messages and objects since the free variables for a thread are taken from the object (redex component) frame whereas the parameters are taken from the message (redex other component) frame. Registers `FB` and `AB` are updated each time a new thread is taken from the run-queue for execution.

Local Channels. A running thread typically creates new channels. These channels are bound to variables local to the thread or to some subset of instructions within it. *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 maximum number of channels created by each thread is known at compile time we use a statically allocated frame to hold (pointers to) channels, instead of local variables. This frame is referred to as `NC` (New Channels). Register `LC` (Local Channels) points to the first available word in `NC`. Entries are pointers to heap allocated channels. This space behaves in a *pseudo-stack* policy, i.e., we allocate new channels at the top of the buffer. Register `LC` is initialized to `NC` before running a new thread thus clearing the temporary bindings (but not the channels themselves).

Method Tables. A method table is a frame with at least one word (objects without methods do not have method tables). Each word in the frame contains the address of the first instruction of a particular method. Words in method tables can be viewed as being labeled with the natural numbers `0`, `1`, `2`, ... , starting from the first word. Given a message label `n`, the address for the corresponding method is obtained by inspecting the word labeled with `n`.

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

2.2 Dynamic Data

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 or garbage collected. Garbage collection, when implemented, should not be visible from the abstract machine, rather it should be concealed in the heap implementation.

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:

`Empty` :`Queue` $\mapsto$ `Bool`	`NewQueue` :`Void` $\mapsto$ `Queue`
`Enqueue` :`Queue` $\times$ `Frame` $\mapsto$ `Void`	`Dequeue` :`Queue` $\mapsto$ `Frame`

`Empty` checks whether a given queue is empty. `NewQueue` initializes an empty queue. `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.

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 as `code`, `freevars`, and `arguments`, respectively (see fig. 2). When a reduction occurs, a new frame is placed in the run-queue. Then the machine uses the integer `label` in the message frame and the method-table pointer in the object frame (`table`) to find out which thread to run. The address for this thread is placed in the word `code` of the new frame. The word `freevars` (`arguments`) 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.

Object Frames. Object frames have at least two words. The first is named `continuation` and is used in queue manipulations; the second is named `table` since it is a pointer to a method table (or `NULL` if the object has no methods). 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 `table`) as `freevars` (see fig. 2).

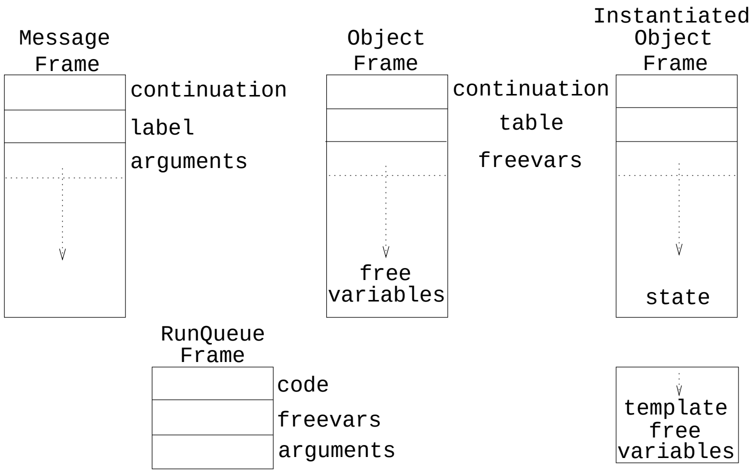

Figure 2: Message, object and run-queue frames

Message Frames. A message frame is a block with at least two words. The first one is named `continuation` and is used in queue manipulations; the second is called `label` and is a natural number index to some method table. Additional words may exist and in this case they are the message arguments. We name the first such word (right after `label`) as `arguments` (see fig. 2).

Channels. Channels implement (process calculi) names. A channel is a queue of object or message frames extended with the following functions:

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

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, or `Objects` if the queue contains at least one object frame[1]. 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 [10].

Using 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. An efficient management (including garbage collection) of channels will likely require powerful compilation techniques combined with type inference information [18, 22].

2.3 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 under execution. The register is initialized with the address of thread `T0`, 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 `FB` (Free variable Bindings) and `AB` (Argument Bindings) are used to hold the free variable and parameter bindings, respectively. Register `LC` (Local Channel) is a pointer to the first available word in the statically allocated frame for local channels (`NC`).

Register	Description
`PC`	Program Counter
`CC`	Current Channel
`CF`	Current Frame
`OF`	Other (redex component) Frame
`NT`	New runnable Thread
`FB`	Free variable Bindings
`AB`	Argument Bindings
`LC`	Local Channels

Table 1: The register set

2.4 Instruction Set

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

[1] We assume the queue updates its state internally. A queue cannot identify a frame as a message or object frame by itself. We should have something like `EnqueueObj()` and `EnqueueMsg()` to do that. We decided to skip these details.

`newf(n)` allocates a frame with `n` words and places a pointer to it in register `CF`.

```
CF := New(n);
PC := PC+2;
```

`new` creates a new channel in the heap and places a pointer to it in the static array `NC`.

```
LC := NewQueue();
LC := LC+1;
PC := PC+1;
```

`loadf(w,v)` loads the contents of the word `v` into the word labeled `w` in the frame currently pointed to by `CF`.

```
CF.w := v;
PC := PC+3;
```

`loada(X,w,v)` loads the contents of the word `v` into the word labeled `w` of the template free variable frame `X`.

```
X.w := v;
PC := PC+4;
```

`loadc(c)` loads the `CC` register with a pointer to the channel frame `c`.

```
CC := c;
PC := PC+2;
```

`try_red_obj` checks the status of the channel queue pointed by `CC`. If the value 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`. If the value is either `Empty` or `Objects` the current frame is enqueued at `CC`.

`try_red_msg` checks the status of the channel queue pointed by `CC`. If the value 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`.

```
try_red_obj:                          try_red_msg:
case  State(CC)  of                   case  State(CC)  of
  Empty:                                Empty:
    Enqueue(CC,CF);                       Enqueue(CC,CF);
  Objects:                              Messages:
    Enqueue(CC,CF);                       Enqueue(CC,CF);
  Messages:                             Objects:
    OF := Dequeue(CC);                     OF := Dequeue(CC);
    NT := New(3);                          NT := New(3);
    NT.code := (CF.table)[OF.label];       NT.code := (OF.table)[CF.label];
    NT.freevars  := CF;                    NT.freevars  := OF;
    NT.arguments := OF;                    NT.arguments := CF;
    Enqueue(RunQueue,NT);                  Enqueue(RunQueue,NT);
PC := PC+1;                            PC := PC+1;
```

Notice the symmetry of the code. Only registers `CF` and `OF` exchange roles as they alternatively point to message frames or to object frames.

`freeof` frees the object (free variables) frame from the current thread.

```
Free(FB);
PC := PC+1;
```

`freemf` frees the message (arguments) frame from the current thread.

```
Free(AB);
PC := PC+1;
```

`next` 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`, `FB` and `AB` are updated with the values in the words `code`, `freevars` and `arguments`, respectively.

```
if  Empty(RunQueue)
   goto  halt;
else
   NT := Dequeue(RunQueue);
   PC := NT.code;
   FB := NT.freevars;
   AB := NT.arguments;
   LC := NC;
PC := PC+1;
```

2.5 The Emulator Loop

The engine of the abstract machine is a standard [5, 9] emulator loop which starts with the first machine instruction. The opcode is determined and the code for the instruction executed. The arguments of the instruction, if any, follow the opcode in the same order as they appear in the program. After an instruction is executed the program counter is incremented by the size of the instruction and the engine loops to `emulate` to execute the next instruction.

The program terminates by jumping off the `emulate` loop (to `halt`) whenever the run-queue empties.

```
  RunQueue := NewQueue();
  PC := T0;
emulate:
  case *PC of
    newf:  CF := New(n);
           PC := PC+2;
           goto emulate;
    loadf: ...
    ...
halt:  // jump here when machine halts
```

3 Optimizations

This section describes a couple of simple optimizations.

3.1 Compact Messages and Objects

Given the frequency with which channels holding a single message or a single object occur in programs the abstract machine is fine tuned for these particular cases. The idea is to completely avoid the overhead of enqueuing and dequeuing frames, and to access their contents directly. With this objective in mind we must change the definition of channels. In our new definition a channel can be a queue or a frame of three words. Channels are represented as queues if they hold more than one frame, i.e., if their state is either `Messages` or `Objects`. They are represented as three word frames if they are either `Empty` or hold only one object (message) frame in which case we say their state is `OneObject` (`OneMessage`). In this case, we name the three words in the frame as `status`, `selector` and `bindings`, respectively. We define the following extra functions for state and conversions between representations:

$$
\begin{aligned}
\texttt{State} &: \texttt{Frame} \mapsto \{\texttt{Empty}, \texttt{OneMessage}, \texttt{OneObject}, \texttt{Messages}, \texttt{Objects}\} \\
\texttt{Compact} &: \texttt{Queue} \times \texttt{Frame} \mapsto \texttt{Void} \\
\texttt{DeCompact} &: \texttt{Frame} \mapsto \texttt{Void}
\end{aligned}
$$

In the case of `OneMessage`, the method index is placed in the word `selector` and a pointer to the first value of the communication (if any) in the word `bindings`. With `OneObject` the word `selector` keeps the pointer to the method table whereas the word `bindings` points to the first free-variable binding (see fig. 3). This representation is especially valuable for fast synchronous communication and built-in operations.

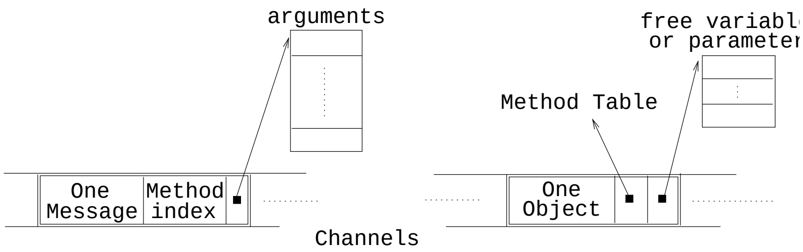

Figure 3: Compacted Views of Messages and Objects

The code for the `try_red_obj` and `try_red_msg` instructions is re-written as follows:

```
try_red_obj:
case  State(CC)  of
  Empty:
    Compact(CC,CF);
  OneObject:
    DeCompact(CC);
    Enqueue(CC,CF);
  OneMessage:
    NT := New(3);
    NT.code := (CF.table)[CC.selector];
    NT.freevars := CF;
    NT.arguments := CC.bindings;
    Enqueue(RunQueue,NT);
  Objects:
    Enqueue(CC,CF);
  Messages:
    OF := Dequeue(CC);
    NT := New(3);
    NT.code := (CF.table)[OF.label];
    NT.freevars := CF;
    NT.arguments := OF;
    Enqueue(RunQueue,NT);
PC := PC+1;
```

```
try_red_msg:
case  State(CC)  of
  Empty:
    Compact(CC,CF);
  OneMessage:
    DeCompact(CC);
    Enqueue(CC,CF);
  OneObject:
    NT := New(3);
    NT.code := (CC.selector)[CF.label];
    NT.freevars := CC.bindings;
    NT.arguments := CF;
    Enqueue(RunQueue,NT);
  Messages:
    Enqueue(CC,CF);
  Objects:
    OF := Dequeue(CC);
    NT := New(3);
    NT.code := (CF.table)[OF.label];
    NT.freevars := CF;
    NT.arguments := OF;
    Enqueue(RunQueue,NT);
PC := PC+1
```

3.2 Replication

Some process calculi use replication instead of recursion (the π-calculus version used in Pict [20] is an example). Replicated objects survive reduction, as opposed to plain (ephemeral) objects who die during reduction. Such objects have exclusive use of the channels where they are held. In other words replicated objects are created in a channel and remain there, alone, for the duration of the computation (such an invariant can be checked statically [22]). The abstract machine compacts replicated objects in a way similar to previous section. However, upon reduction, the object is not removed from the channel.

As multiple messages may be sent to one such object, at one time there may be multiple runnable threads in the run-queue that use the same object frame to hold their free variable bindings. Note that these bindings are taken at the moment the object is first created, and hence loss of bindings is not a question. The abstract machine must distinguish replicated from ephemeral objects. This can be viewed as a property of the channel being inspected and hence we change the function `State` to become:

$$\texttt{State} : \texttt{Frame} \mapsto \{\texttt{Empty}, \texttt{OneMessage}, \texttt{OneObject}, \texttt{ReplObject}, \texttt{Messages}, \texttt{Objects}\}$$

The code specification given above for the `try_red_obj` instruction must be changed. We introduce a new instruction `try_red_replobj` to handle the case in which the replicated object is first placed in the channel. Note that the channel in question is empty when the object is created and this is the only case to be considered. In the case of `try_red_msg` we must change the code to introduce the possibility of reduction with a replicated object. This case is similar to the above in that we take the `table` and `freevars` pointers directly from the channel. However, now we do not change the state of the channel. The code is as follows:

```
try_red_replobj:
  ReplObject(CC,CF);
```

```
try_red_msg:
  case State(CC) of
    ...............
    ReplObject:
      NT := New(3);
      NT.code := (CC.selector)[CF.label];
      NT.freevars := CC.bindings;
      NT.arguments := CF;
      Enqueue(RunQueue,NT);
    ...............
PC := PC+1;
```

The most important change, however, is one that concerns the compilation of the methods in the replicated object. Each method thread ends with the instructions: `freemf; next;` which frees the message (argument) frame but not the object (free variable) frame as in ordinary threads. The object remains compacted in the channel.

4 Compiling the TyCO Object Calculus

This section describes a simple compiler for the TyCO process calculus [24].

4.1 TYped Concurrent Objects

Programs are collections of threads. *Threads* are sequences of processes. *Processes* are built from *constants*, *value variables*, *labels*, and *template variables*. Let x, y, z range over the set of value variables, and u, v range over the set of values (constants and variables). Let $\vec{x}$ denote the sequence $x_1 \cdots x_k$ $(k \geq 0)$ of pairwise distinct variables. Template variables are denoted by $X, Y, \ldots,$ and labels by $l, m, \ldots$. The set of processes is given by the following grammar:

$$
\begin{array}{rcl}
P & ::= & x \texttt{ ! } l[\vec{v}] \mid x \texttt{ ? } M \mid X[\vec{v}] \mid \texttt{new } x \texttt{ in } \vec{P} \mid \texttt{def } D \texttt{ in } \vec{P} \\
\vec{P} & ::= & P_1 \mid \cdots \mid P_k \\
M & ::= & \{l_1(\vec{x_1}) : \vec{P_1}, \ldots, l_k(\vec{x_k}) : \vec{P_k}\} \\
D & ::= & X_1(\vec{x_1}) = y_1 \texttt{ ? } M_1 \texttt{ and} \cdots \texttt{and } X_k(\vec{x_k}) = y_k \texttt{ ? } M_k
\end{array}
$$

Parenthesis constitute the binding of the calculus: variables $\vec{x}$ are bound in the part $\vec{P}$ (the part y ? M) of a method $l(\vec{x}) : \vec{P}$ (a definition $X(\vec{x}) = y$? M, respectively). The free variables of a process P (denoted $\texttt{fv}(P)$), and the substitution of variables by constants (denoted $P[\vec{v}/\vec{x}]$) are defined accordingly.

Processes are considered equal up to structural congruence [16]. We omit the exact definition [10], it suffices to say that it includes α-conversion, permutation in threads, the pushing of `new`s outwards, and the expanding of definitions. Reduction is defined by the following axiom:

$$
a \texttt{ ! } l_i[\vec{v}] \mid a \texttt{ ? } \{l_1(\vec{x_1}) : \vec{P_1}, \ldots, l_k(\vec{x_k}) : \vec{P_k}\} \mid \vec{P} \rightarrow \vec{P_i}[{}^{\vec{v}}/_{\vec{x_i}}] \mid \vec{P} \qquad (1 \leq i \leq k)
$$

and by allowing reduction under `new` and `def`. See the appendix ?? for an example.

4.2 Intermediate Macros

A small set of intermediate macros makes the code more compact and object calculi flavored.

`sched` frees the binding frames from the last thread and fetches another for execution.

```
#define sched         { freemf; freeof; next; }
```

`bindk(X,y1,...,yk)` loads the contents of the variables `y1,...,yk` (in this order) into the template free variable array `X`.

```
#define bindk(X,y1,...,yk)  { loada(X,0,y1);...;loada(X,k-1,yk); }
```

`msgk(c,l,u1,...,uk)` (`objk(c,M,y1,...,yk)`) creates a new message (object) frame and fills it with the label number `l` and the values `u1,...,uk` (method array pointer `M` and the contents of the free variables `y1,...,yk`).

```
#define  msgk(c,l,u1,...,uk)
{
  newf(k+2);
  loadf(continuation,NULL);
  loadf(label,l);
  loadf(arguments,u1);
  ...........
  loadf(arguments+k-1,uk);
  loadc(c);
  try_red_msg;
}
```

```
#define  objk(c,M,y1,...,yk)
{
  newf(k+2);
  loadf(continuation,NULL);
  loadf(table,M);
  loadf(freevars,y1);
  ...........
  loadf(freevars+k-1,yk);
  loadc(c);
  try_red_obj;
}
```

4.3 The Compiler

We rely on a symbol table mapping identifiers in the program with strings used to generate code, and providing the operations `Insert` and `Lookup`:

$$\texttt{Insert} : \texttt{Id} \times \texttt{String} \mapsto \texttt{Void} \qquad\qquad \texttt{Lookup} : \texttt{Id} \mapsto \texttt{String}$$

We assume a function `Gen` that takes an arbitrary number of strings and integers, and outputs them. Translating labels in programs to natural number offsets to the method tables requires information gathered by a type-inference system. Here, we assume that this translation is performed by the function `IndexOf`.

The compiler keeps two counters used to assign distinct strings to local channels (`CI`) and to method tables (`TI`). `CI` is initialized with zero at the beginning of a thread and reset at the end of the thread. `TI` is initialized to zero when the compilation starts.

There are five code-producing functions.

$$\begin{aligned}
\texttt{Insts} &: \texttt{Thread} \mapsto \texttt{Void} & \texttt{Inst} &: \texttt{Proc} \mapsto \texttt{ListOf(String} \times \texttt{Method)}\\
\texttt{Meths} &: \texttt{ListOf(String} \times \texttt{Method)} \mapsto \texttt{Void} & \texttt{Meth} &: \texttt{String} \times \texttt{MethodBody} \mapsto \texttt{Void}\\
\texttt{Bind} &: \texttt{Binding} \mapsto \texttt{Void} & &
\end{aligned}$$

`Insts` compiles a thread, using `Inst` to compile individual processes. Methods are compiled separately, and as such, the compilation of object processes returns the object's methods together with a string by which the methods are known in the code for the object (this string is none other than the label of the method table in the final code). `Meths` generates the method table, and compiles collections of methods using `Meth` to compile individual methods. Finally `Bind` compiles each of the bindings in a process definition.

To compile a program, reset the table index and compile the thread.

$$\texttt{Compile}(\vec{P}) \quad = \quad \texttt{TI} := 0;\ \texttt{Insts}(\vec{P})$$

Threads are compiled by resetting the local channel index, compiling its processes and gathering the as yet uncompiled methods. Then the methods are compiled.

$$\begin{aligned}
\texttt{Insts}(P_1 \mid \cdots \mid P_k) = \quad &\textbf{let} \quad \texttt{CI} := 0;\ L_1 = \texttt{Inst}(P_1); \ldots; L_k = \texttt{Inst}(P_k);\\
&\textbf{in} \quad \texttt{CI} := 0; \texttt{Meths}(\texttt{Concatenate}(L_1, \ldots, L_k));
\end{aligned}$$

To compile a (local) channel creation `new` x `in` $\vec{P}$, we insert the binding for x in the symbol table, generate a new channel, and proceed with the code for $\vec{P}$.

$$\begin{aligned}
\texttt{Inst}(\texttt{new}\ x\ \texttt{in}\ \vec{P}) \quad = \quad &\texttt{Insert}(x, \texttt{"LC["} + \texttt{CI} + \texttt{"]"});\ \texttt{CI} := \texttt{CI} + 1;\ \texttt{Gen}(\texttt{"new"});\\
&\texttt{Insts}(\vec{P}); [\,]
\end{aligned}$$

Messages are compiled into `msg` macros; the arguments must be fetched from the symbol table:

$$\begin{aligned}
\texttt{Inst}(x\ \texttt{!}\ l[u_1 \cdots u_k]) = \quad &\texttt{Gen}(\quad \texttt{"msgk("}, \texttt{Lookup(x)}, \texttt{","}, \texttt{IndexOf(l)}, \texttt{","}\\
&\qquad\quad \texttt{Lookup(u}_1\texttt{)}, \texttt{","}, \ldots, \texttt{","}, \texttt{Lookup(u}_k\texttt{)}, \texttt{")"} \quad);\\
&[\,]
\end{aligned}$$

Objects are compiled into `obj` macros; the method table label is decided here; the function returns this label together with the as yet uncompiled methods:

$$\begin{aligned}
\texttt{Inst}(x\ \texttt{?}\ M) = \quad &\textbf{let} \quad \texttt{table} = \texttt{"T"} + \texttt{TI};\\
&\qquad\quad y_1, \ldots, y_k = \texttt{fv}(M)\\
&\textbf{in} \quad\ \texttt{Gen}(\quad \texttt{"objk("}, \texttt{Lookup(x)}, \texttt{","}, \texttt{table}, \texttt{","}\\
&\qquad\qquad\quad \texttt{Lookup(y}_1\texttt{)}, \texttt{","}, \ldots, \texttt{","}, \texttt{Lookup(y}_k\texttt{)}, \texttt{")"} \quad);\\
&\qquad\quad \texttt{TI} := \texttt{TI} + 1;\ [(\texttt{table}, M)]
\end{aligned}$$

Objects instances of templates ($X[\vec{u}]$) are compiled similarly, the difference being that the arguments $\vec{u}$ now refer to the template parameters rather than to the object's free variables. Given that $X(\vec{x}) = y\ ?\ M$, the strings for M (the object table) and for y (the **self** channel) have been inserted in the symbol table associated with the identifiers X and $Xself$, respectively (see **Bind** bellow).

$$\begin{array}{rll} \texttt{Inst}(X[u_1 \cdots u_k]) = & \texttt{Gen}(& \texttt{"objk("}, \texttt{Lookup(Xself)}, \texttt{","}, \texttt{Lookup(X)}, \texttt{","} \\ & & \texttt{Lookup(u}_1\texttt{)}, \texttt{","}, \ldots, \texttt{","}, \texttt{Lookup(u}_k\texttt{)}, \texttt{")"} \quad); \\ & [\,] & \end{array}$$

Finally, to compile a definition we compile the bindings, gathering the as yet to be compiled methods. After compiling the thread we return the methods.

$$\begin{array}{rll} \texttt{Inst(def } D_1 \texttt{ and } \cdots \texttt{ and } D_k \texttt{ in } \vec{P}) = & \textbf{let} & B = \texttt{Concatenate(Bind(}D_1\texttt{)}, \ldots, \texttt{Bind(}D_k\texttt{))}; \\ & \textbf{in} & \texttt{Insts}(\vec{P});\ B \end{array}$$

To compile a binding we reserve a frame for the free variables in the binding, and then store their values in this frame using a **bind** macro. The actual generation of the code for the methods is postponed until the end of the compilation of the current thread.

$$\begin{array}{rll} \texttt{Bind}(X = (x_1 \cdots x_k) y\ ?\ M) = & \textbf{let} & \texttt{table} = \texttt{"T"} + \texttt{TI}; \\ & & y_1, \ldots, y_k = \texttt{fv}((x_1 \cdots x_k) y\ ?\ M) \\ & \textbf{in} & \texttt{Insert}(X, \texttt{table}); \texttt{Insert}(Xself, \texttt{Lookup}(y)); \\ & & \texttt{Insert}(x_1, \texttt{"FB[0]"}); \ldots; \texttt{Insert}(x_k, \texttt{"FB[}k-1\texttt{]"}); \\ & & \texttt{Gen}(X, \texttt{" : "}, n); \\ & & \texttt{Gen}(\texttt{"bind"}, k, \texttt{"("}, X, \texttt{","}, \texttt{Lookup}(y_1), \texttt{","}, \ldots, \texttt{","}, \texttt{Lookup}(y_n), \texttt{")"}); \\ & & \texttt{TI} := \texttt{TI} + 1;\ [(\texttt{table}, M)] \end{array}$$

Compiling a collection of methods involves generating the method table and the actual code for the methods (the operator + appends two strings):

$$\begin{array}{rcl} \texttt{Meths}(t, \{l_1(\vec{x_1}) : \vec{P_1}\ ; \cdots ;\ l_k(\vec{x_k}) : \vec{P_k}\}) & = & \texttt{Gen}(t, \texttt{" : "}, t + \texttt{IndexOf}(l_1), \texttt{","}, \ldots, \texttt{","}, t + \texttt{IndexOf}(l_k)); \\ & & \texttt{Meth}(t + \texttt{IndexOf}(l_1), (\vec{x_1}) : \vec{P_1}); \\ & & \cdots \\ & & \texttt{Meth}(t + \texttt{IndexOf}(l_k), (\vec{x_k}) : \vec{P_k}); \end{array}$$

Finally, to compile a method we first establish the bindings for the method arguments and then generate the code for the method body:

$$\begin{array}{rcl} \texttt{Meth}(t, (x_1 \cdots x_k) : \vec{P}) & = & \texttt{Insert}(x_1, \texttt{"AB[0]"}); \ldots; \texttt{Insert}(x_k, \texttt{"AB[}k-1\texttt{]"}); \\ & & \texttt{Gen}(t, \texttt{" : "}); \\ & & \texttt{Insts}(\vec{P}); \end{array}$$

5 Preliminary Results

In this section we present some preliminary results obtained with an implementation of the abstract machine. The current implementation uses C to provide a flow of execution (the program counter), has booleans and integers as built-in datatypes, and implements the optimization described in section 3 for compacting single objects and messages into channels. The implementation of the abstract machine is quite compact, the run-time system being implemented in just 650 C code lines.

	Haskell ghc -O2	Pict -O2	TyCO -O2	TyCO$^+$ -O2	TyCO^{++} -O2	$\frac{\text{TyCO}^{++}}{\text{Pict}}$
Sieve 1000 (size)	0.090s (390k/10k)	0.110s (14k)	0.493s (13k)	0.291s (12k)	0.255s (13k)	(2.3)
Queens 8	0.189s (304k/15k)	0.194s (15k)	2.378s (15k)	0.753s (13k)	0.557s (14k)	(2.8)
Fib 20	0.079s (305k/6k)	0.077s (11k)	0.532s (11k)	0.204s (10k)	0.170s (10k)	(2.1)

Table 2: Preliminary results

We used `Fib`, the `Sieve` of Eratosthenes and `Queens` all implemented in TyCO, Pict (version 4.0) and Haskell (the Glorious Glasgow Haskell compiler for `Linux 2.08`) to gauge the speed of the implementation, and the size of the binaries produced. Table 2 summarizes the results we got:

The column labeled TyCO presents the results for an implementation of built-in booleans and integers evaluated using explicit messages. TyCO$^+$ refers to an implementation where built-in operations are performed inline as argument evaluations. In TyCO^{++} we added the optimization for replicated objects to TyCO$^+$. The table shows that with TyCO^{++} we are 2 to 3 times slower than Pict (and Haskell, for that mather). The `-O2` flag refers to the optimization level with which the C code generated by the TyCO compiler was compiled. The time for `Queens` is worse than average because the benchmark uses list constructors extensively and these are built-in both in Pict and Haskell, but not in TyCO. Also, in general, the current implementation does not benefit from type information which allows a number of optimizations to be performed, and other optimizations such as tail recursion are not yet implemented. Note that Pict and TyCO produce considerably smaller binaries than those produced by the Glasgow Haskell compiler. This is because the latter requires a large number of modules to be statically linked. We are currently performance-tuning the machine so that its speed may reach that of Haskell or Pict which are currently the state of the art in process-based and functional languages, respectively. We will ultimately need a powerful type system to assist us in this task.

6 Related Work

Although there is extensive theoretical research on the subject, abstract machine specifications and implementations of actual models are still scarce. A few programming languages based on process calculi have been purposed. Some, such as Pict [22] and Oz [12], have gained some prominence.

Pict [22, 20] is a pure concurrent programming language based on the asynchronous π-calculus, with the basic abstractions being processes and names (channels). Processes communicate by sending values along shared channels. Recursion is modeled by process replication.

Oz [12, 13] combines three major programming paradigms: functional, object-oriented, and constraint logic programming, into an high-level programming language. Oz is based in the γ-calculus, where the basic abstractions are names, variables (logical), procedural abstraction and cells. Constraints are built over logical variables by first-order logic equations, using a set of predefined predicates. Cells are primitive entities that maintain state and provide atomic read-write operations.

Our abstract machine is based on TyCO [24], which is essentially the asynchronous π-calculus plus branching structures. Branching structures appear as a pair of operators: selection (or labeled messages), which replace the output processes of the π-calculus and; branching (or objects) which replace the input processes (or prefix) of the π-calculus. Potentially unbounded behavior is modeled through explicit instantiation of recursive templates.

Pict implements only persistent objects where methods are input processes assigned to distinct channels. Mutual exclusive execution of methods in an object is achieved by sending the object a message that carries its state. Only then can a method be invoked. This makes method invocation a two-way protocol [26] in Pict, as opposed to one-way in TyCO. Oz procedures have encapsulated state so that objects can be defined directly as sets of methods (procedures) acting over their state (represented as a set of logical variables). This representation is quite close to objects in our own abstract machine if we view the state held in logical variables as template parameters.

In Pict recursion is supported through replication. This has the disadvantage that, once created, replicated objects will exist for the duration of the program even if they are only required for a small part of the computation. These may have to be garbage collected. TyCO supports recursion through explicit instantiation of template objects. Each recursive call within the body of the method must create a new replica of the original object, ensuring that no unnecessary copies are ever created.

The Pict abstract machine tries to maximize the number of processes executed within a thread of control. This is achieved by allowing a process to be executed at once instead of building a closure and placing it in the run-queue. The only exception occurs when a replicated process closure is taken from a queue upon reception of a communication. On the other hand the Oz abstract machine executes programs made up of threads - stacks of expressions. State, in the form of cells, and equations are kept in a *blackboard* which is updated by concurrent threads. New threads are created whenever an expression that occurs in the current thread cannot be immediately reduced. In this case a new thread is created for the expression. Our engine executes threads - sequences of processes. All newly created threads are placed at the end of the run-queue. Processes within threads are executed without interference from the scheduler, hence threads cannot be blocked. Threads are identified with object's methods. This fact allows the definition of a very clean model for object-oriented languages where objects are collections of methods atomically executed with some *state* carried by template parameters. Moreover, in a concurrent setting, non-determinism originates in interactions of threads on shared names.

7 Future Work

Future work will explore several research directions. A complete performance evaluation of the current implementation will be conducted to identify feasible optimizations. This work will benefit substantially from the development of profiling, debugging and visualization tools to gauge the run-time system.

The implementation of the type-inference algorithm described in [10] will allow the type-checking of programs and the consequent detection of run-time errors at an early stage of the compilation. The incorporation of linear types [18] would allow substantial optimizations in channel management which is commonly very space inefficient. Moreover, speed could also be greatly improved, for example, by using registers to hold frame pointers instead of allocating new memory channels.

Finally, a most interesting research project involves the development of high level idioms [7] on top of this object *assembly* to form full fledged object-based/oriented languages. Data-type declarations similar to those existing in current functional languages [8, 21, 17] will allow the construction of complex data-structures. Functional expressions and pattern matching will complete the implementation of a functional subset of TyCO. Finally, true polymorphic classes and inheritance will allow a more powerful, object-oriented, style of programming.

References

[1] M. Abadi and L. Cardelli. *A Theory Of Objects*. Springer-Verlag, 1996.

[2] Gul A. Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems*. The MIT Press, 1986.

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

[4] G. Smolka. A Fundation for Concurrent Constraint Programming. In *Constraints in Computational Logics*, volume 845 of *Lecture Notes in Computer Science*. Springer-Verlag, September 1994. Invited Talk.

[5] Hassan Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction*. Logic Programming Series. MIT Press, 1991.

[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] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type disciplines for structured communication-based programming. Submitted.

[8] P. Hudak and S. Jones et al. Report on the Programming Language Haskell: a non-strict, purely functional language. *ACM Sigplan Notices*, 27(5), 1992.

[9] Sverker Janson. *AKL - A Multiparadigm Programming Language*. PhD thesis, SICS Swedish Institute of Computer Science, Uppsala University, 1994.

[10] L. Lopes and V. Vasconcelos. An Abstract Machine for an Object Calculus. Submitted.

[11] L. Lopes and V. Vasconcelos. TyCOAM - The Definition. Technical Report DCC-97-1, DCC-FC & LIACC, Universidade do Porto, May 1997.

[12] G. Smolka M. Henz and J. Würtz. Object-Oriented Concurrent Constraint Programming in Oz. pages 29–48, 1993.

[13] R. Scheidhauer M. Mehl and C. Schulte. An Abstract Machine for Oz. Technical report, German Research Center for Artificial Intelligence (DFKI), June 1995.

[14] M. Papathomas. A Unifying Framework for Process Calculus Semantics of Concurrent Object-Oriented Languages. In M. Tokoro and O. Niestrasz and P. Wegner, editor, *Proceedings of the ECOOP'91 Workshop on Object-Based Concurrent Computing*, number 612 in Lecture Notes in Computer Science, pages 53–79. Springer-Verlag, 1992.

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

[16] Robin Milner. Functions as Processes. *Journal of Mathematical Structures in Computer Science*, 2(2):119–141, 1992.

[17] Robin Milner and Mads Tofte. *The Definition of Standard ML*. 1991.

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

[19] Oscar Niestrasz. Towards an Object Calculus. In M. Tokoro and O. Niestrasz and P. Wegner, editor, *Proceedings of the ECOOP'91 Workshop on Object-Based Concurrent Computing*, number 612 in Lecture Notes in Computer Science, pages 1–20. Springer-Verlag, 1992.

[20] Benjamin C. Pierce and David N. Turner. Pict: A Programming Language Based on the Pi-Calculus. Technical Report CSCI 476, Computer Science Department, Indiana University, 1997. To appear in *Proceedings of Language and Interaction: Essays in Honour of Robin Milner*, Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, MIT Press.

[21] D. Turner. Miranda: a Non-Strict Functional Language with Polymorphic Types. In *IFIP International Conference on Functional Programming and Computer Architecture*, volume 201 of *LNCS, Springer-Verlag*, 1985.

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

[23] V. Vasconcelos. Typed Concurrent Objects. In *8th European Conference on Object-Oriented Programming, LNCS*, volume 821, pages 100–117. Springer-Verlag, July 1994.

[24] 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.

[25] David Walker. π-Calculus Semantics of Object-Oriented Programming Languages. In *Theoretical Aspects of Computer Software*, volume 526, pages 532–547, 1992.

[26] David Walker. Objects in the π-calculus. *Journal of Information and Computation*, 116(2):253–271, 1995.

[27] Akinori Yonezawa. *ABCL, an Object Oriented Concurrent System.* MIT Press, 1990.