Luís Miguel Barros Lopes

On the Design and Implementation of a Virtual Machine for Process Calculi

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto

Outubro de 1999

Luís Miguel Barros Lopes

On the Design and Implementation of a Virtual Machine for Process Calculi

Tese submetida à Faculdade de Ciências da Universidade do Porto para obtenção do grau de Doutor em Ciência de Computadores

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto

Outubro de 1999

À minha avó Maria, aos meus pais Joaquim e Clara
e à minha irmã Cristina

Agradecimentos

Desejo agradecer a colaboração de várias pessoas que em diferentes momentos contribuiram de forma preciosa para o desenrolar deste trabalho. Em primeiro lugar gostaria de agradecer aos meus orientadores Fernando M. A. Silva e Vasco T. Vasconcelos pelo apoio constante que me foi dado, não raras vezes com algum sacrifício pessoal. Ao Fernando um especial obrigado pelo apoio moral que sempre me manifestou. Ao Vasco um especial obrigado pelo rigor e minúcia, por vezes quase exasperante, com que trabalha especialmente ao nível teórico. Mais uma vez obrigado a ambos. Entre os numerosos colegas que com o seu *feedback* contribuiram para o trabalho apresentado nesta tese gostaria agradecer em especial a Vítor Santos Costa, Rui Bastos, António Ravara, Álvaro Figueira e Luís Caires.

Os meus agradecimentos ao Departamento de Ciência de Computadores por me ter proporcionado todas as condições para a realização deste trabalho, isto apesar das severas limitações de pessoal no período a que se refere o desenvolvimento e escrita desta tese. Finalmente, uma última palavra de agradecimento para a minha família por todo o apoio que me foi dado.

Abstract

There has been an ever growing interest in the study of process-calculi as frameworks for concurrent, distributed, object-oriented languages. These are of particular importance given the adequacy of the object programming model and its widespread use in software development and, ultimately, in Internet applications. The asynchronous π-calculus is a powerful tool to reason about concurrent process interaction. However, it is not an adequate framework for the implementation of object-oriented languages as its encodings of objects are quite inefficient. Efforts in the implementation of languages based on process calculi have so far failed in providing a unifying framework in the form of a virtual machine specification.

This thesis introduces a virtual machine specification and its implementation for a variant of the π-calculus – the TyCO calculus. TyCO features objects and asynchronous messages. Thus, contrary to other calculi such as π, objects are builtin entities and do not require computationally expensive encodings.

This work starts with the definition of a small kernel language based on TyCO and provides its static semantics, in the form of a type-inference system, and its dynamic semantics, in the form of an abstract machine. This formal framework forms the basis for the design of a virtual machine and language compiler. The virtual machine runs TyCO programs translated into byte-code format. It is compact, architecture independent and performs favorably when compared with other related implementations. The machine is fine-tuned for TyCO but may be used to implement other calculi with minor changes or extensions to its semantics.

Finally, an orthogonal extension of the TyCO calculus is defined to provide support for distributed computations and code mobility. In this model, remote communication and code migration are induced by the lexical scope of names in the base calculus. The model can be applied without change to a wide variety of calculi that fulfill a few, very mild, conditions.

Resumo

Existe um interesse crescente no estudo de cálculos de processos como modelos formais para linguagens de programação com concorrência, distribuição e orientação a objectos. Actualmente, estas linguagens assumem uma especial importância pela adequação do seu modelo de programação ao desenvolvimento de grandes projectos de *software*, em especial no domínio das aplicações em rede ou para a *Internet*. O cálculo π assíncrono constitui um instrumento poderoso no estudo de mecanismos de concorrência e interacção entre processos. No entanto, não é um modelo adequado para a implementação de linguagens orientadas a objectos pois as codificações existentes de objectos no cálculo são pouco eficientes. O trabalho realizado na implementação de linguagens baseadas em cálculos de processos tem sido, até à data, infrutífero na obtenção de um modelo de computação unificado sob a forma de uma máquina virtual.

Esta tese introduz uma especificação e implementação de uma máquina virtual para uma forma do cálculo π – o cálculo TyCO. O TyCO tem, para além de nomes, objectos e mensagens assíncronas como abstracções fundamentais. Desta forma, contrariamente a outros cálculos como o π, os objectos não necessitam de codificação.

Inicia-se este trabalho com a definição de uma pequena linguagem de programação baseada no TyCO. A semântica estática da linguagem é definida por um sistema de inferência de tipos ao passo que a semântica dinâmica é introduzida sob a forma de uma máquina abstracta. Este modelo formal proporciona os alicerces para o desenvolvimento da máquina virtual e compilador da linguagem. A máquina virtual executa programas TyCO compilados para um formato *byte-code*, é compacta, independente da arquitectura de *hardware* subjacente e exibe uma performance que se compara favoravelmente com outras implementações. A máquina foi optimizada para o TyCO podendo no entanto ser utilizada na implementação de outros cálculos com pequenas alterações ou extensões à sua semântica.

Finalmente, define-se uma extensão ortogonal ao cálculo TyCO que proporciona o

suporte para computações distribuídas e mobilidade de código. Neste modelo, a comunicação remota e a migração de código são induzidas pelo contexto léxico dos nomes no cálculo base. O modelo pode ser aplicado sem alterações a uma grande variedade de cálculos para os quais se verifiquem algumas pre-condições fracas.

Résumé

On témoigne un intérêt croissant dans l'étude des calculs de processus comme des modèles pour les langages de programmation concurrents, distribués et orientés à objects. Actuellement, ces langages ont une trés grande importance qui résulte de l'adequation de son modèle de programmation au développement de grands projects de logiciel, en particulier des applications pour l'Internet. Le calcul π est un instrument puissant pour l'étude des mécanismes d'interaction entre processus. Pourtant, ce calcul n'est pas un modèle adéquat pour la conception et l'implantation des langages orientés à objects car les codifications d'objects connues sont peux efficaces. Le travail existant d'implantation de langages basés dans des calculs de processus n'a pas réussi, jusqu'a ce moment, à obtenir un modèle de computation unifié sous la forme d'une machine virtuelle.

Cette thése introduit une spécification et une implantation d'une machine virtuelle pour une forme du calcul π – le calcul TyCO. En plus de noms, TyCO a des objects et des messages asynchrones comme des abstractions fundamentales. Les objects dans le calcul TyCO n'ont pas besoin de codification, contrairement à des autres calculs comme le π.

Ce travail s'initie par la définition d'un petit langage de programmation basé dans le calcul TyCO. La sémantique statique du langage est presentée sous la forme d'un système d'inférence de types. La sémantique dynamique est introduite ensuite sous la forme d'une machine abstraite. Ce modèle formel est la base pour le développement de la machine virtuelle et du compilateur du langage. La machine virtuelle exécute des programmes TyCO compilés dans un format de codage em mots. Elle est compacte, indépendant de l'architecture de l'ordinateur subjacent et a une performance comparable à d'autres implantations. La machine a été optimisée pour le calcul TyCO. Cependant elle peut etre utilisée dans l'implantation d'autres calculs avec des petites modifications ou extensions de la sémantique.

Finalement, on étend d'une façon orthogonale le calcul TyCO pour introduire des mécanismes qui supportent des computations distribuées et la mobilité du code. Dans cette extension la communication distante et la migration du code sont induits par le contexte lexical des noms dans le calcul base. Le modèle peut être appliqué sans modifications à une trés grande varieté de calculs qui doivent verifier seulement quelques pré-conditions trés faibles.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In this chapter we briefly overview the concurrent programming paradigm, starting from its origins in the context of operating system development, its use in imperative, concurrent or parallel, programming languages up to the relatively recent development of a *Theory of Concurrent Processes*. We introduce the basic concepts underlying this theory of concurrency since it is of fundamental importance for this thesis. Next we describe the goal of this work and its contribution to this ever growing research field and, finally, we sketch the layout of the thesis.

1.1 The Beginnings of Concurrency Theory

The study of concurrency dates back to the sixties. In those early days the problems studied related to the sharing of resources by processes in an operating system. Resource sharing can result in data inconsistencies, process starvation or even deadlock configurations. The seminal work in this area, mainly by Dijkstra and Hoare [15, 16, 25, 26] provided the algorithms for maintaining data consistency in programs, to avoid starvation through appropriate scheduling and, to diagnose and recover from deadlock configurations.

This work was used extensively for the next generation of operating systems such as MULTICS and its close successor Unix. These systems implemented process scheduling and preemption as well as IPC (Inter-Process Communication) modules that provided services such as shared memory, semaphores and message queues, and enabled both the system and the user level applications to explore concurrency. Programming languages, especially those designed for parallel computing, explored these facilities to provide the users with a concurrent programming paradigm based on either shared memory or distributed memory architectures. The typical approach consisted in implementing concurrent constructs, over an existing sequential programming language or operating system, as an extra feature supported by the run-time system or by an external library. This approach has several disadvantages such as:

- the difficulty in defining the amount of concurrency available and its scheduling among the system's computational resources – hardware or virtual processes;
- concurrency is always explicit in programs, requiring some knowledge of the underlying implementation from the users;
- proof that the system behaves in a certain way is quite difficult. This is mainly due to the absence of an adequate formal framework on which to base the semantics of these languages.

In the late seventies, with the introduction of computer networks, distributed computing and later the Internet, it became obvious that a more general and fundamental theory was required to study the complex and dynamic communication topologies of these systems. The development of pure concurrent languages remained an elusive goal until the basis for an underlying concurrency theory was first set. Researchers were interested in the behavior characterization of concurrent systems, establishing their semantics within a rigorous framework.

1.2 A Theory of Concurrent Processes

A few simple concepts are important for the understanding of this section. The definitions are informal at this point as the objective is to provide the reader with an intuitive understanding of the ideas behind the development of concurrency theory. A rigorous definition of these concepts is presented in chapter 2.

A *Process* is an abstraction that stands for some computation. Computations evolve through interaction on *names*, the end points of communication links between processes. Names may present themselves in two polarity states: input polarity – a, and output polarity – $\bar{a}$. Interacting processes are prefixed with names a that establish links with other processes. For example, the process $a.P$ represents the process P

prefixed with a name a with input polarity. Similarly, the process $\bar{a}.Q$ stands for the process Q prefixed with the same name a with an output polarity. The name a acts as a guard for the processes P and Q, respectively the bodies of $a.P$ and $\bar{a}.Q$. In other words, P and Q can only run after a synchronization occurs at a and this is only possible if the polarities of a are complementary in the interacting processes. A system is composed of any number of processes running concurrently as in $P \mid Q$. Finally some sort of computational mechanism is required to actually make processes interact. $P \to Q$ means that process P evolves into process Q by applying one step of our computational mechanism. The stage is now set for a brief incursion in the history of concurrency theory.

The first step towards a modern concurrency theory was given in the late seventies by Robin Milner. He introduced an algebra named CCS (Calculus of Communicating Systems) [44]. In CCS processes interact by synchronizing on shared names. The interaction does not involve the exchange of data.

$$\bar{a}.Q \mid a.P \to P \mid Q$$

In the example, processes P and Q are freed for execution after a synchronization occurs at name a. The CCS approach of looking at processes as terms in an algebra and their interactions as operations on these terms, much like the λ-calculus, was particularly fruitful and gave way to what are now commonly called *process calculi*.

In 1988, Robin Milner, Joachim Parrow and David Walker [46], proposed a new process calculus, the π-calculus, as an algebraic model for interacting processes. In its original form, a process in the π-calculus could either send (receive) a single name to (from) another process. This was achieved by allowing more complex prefixes. An output prefixed process such as $\bar{a}(b).Q$ besides synchronizing with the input prefixed process $a(x).P$, carries a name that is bound to the variable x local to process P. Naturally, the communication point is a shared name.

$$\bar{a}(b).Q \mid a(x).P \to P\{b/x\} \mid Q$$

This is designated the monadic π-calculus. Since names are the communication links between processes, being able to send them in messages to other processes allows the topology of communication to change dynamically. In this way it is possible to model very complex systems and the calculus becomes very powerful and expressive. Almost immediately after, it was shown that through a simple encoding an equivalent polyadic calculus could be obtained:

$$\bar{a}(\tilde{b}).Q \mid a(\tilde{x}).P \to P\{\tilde{b}/\tilde{x}\} \mid Q$$

These early versions of the π-calculus are called synchronous since messages are *blocking*. In other words, in the output process $\bar{a}(\tilde{b}).Q$, process Q suspends until the message $\bar{a}(\tilde{b})$ is consumed.

Early in the nineties, Honda and Tokoro [27] and Boudol [9] independently showed that the polyadic calculus could be further simplified into an equaly expressive asynchronous formulation, where output processes do not carry continuations and thus have the form $\bar{a}(\tilde{b})$. In this case the one-step reduction is defined by:

$$\bar{a}(\tilde{b}) \mid a(\tilde{x}).P \rightarrow P\{\tilde{b}/\tilde{x}\}$$

This form of the π-calculus quickly became the basic tool for most of the research that followed.

After the core asynchronous calculus was established, progress in the study of concurrent systems was fast and productive. Some of the very fruitful research areas that followed are:

- the study of the semantics of process calculi, and namely of behavior equivalences between processes usually defined in the form of mathematical relations called *bisimulations* [45, 46];
- the study of combinator systems much like those for the λ-calculus, that can generate all of the observed behaviors of interacting processes [29];
- the introduction of type systems that verify whether processes are *well-typed* and that, under subject reduction, guarantee the absence of run-time protocol errors. Types are also used to restrict the contexts in which channels may exhibit certain behaviors thus providing type-safe processes [45, 46, 53, 74].
- the study of the shape and usage of names in programs. This has direct applications in language implementation, namely in optimizing compilers [6, 35, 36].
- the introduction of new process calculi which are either variations or extensions of the π-calculus and aim at providing support for other programming paradigms such as object-oriented or functional or, at increasing the expressiveness of the base calculus [10, 20, 51, 64, 74];
- the design and implementation of pure concurrent programming languages based on this communication model [22, 43, 54];

- the study of calculi supporting distribution and code mobility as a basis for the implementation of web languages and distributed computing [6, 12, 21, 24, 58, 62, 63, 67, 75].

In recent years researchers have devoted a great effort in providing semantics for pure concurrent programming languages within the realm of process-calculi. The asynchronous π-calculus 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 [10] introduces a functional style of programming in the π-calculus by distinguishing between synchronization and communication. Vasconcelos' TyCO-calculus [74] introduces branching structures as an efficient way to model objects and uses explicit recursion to model unbounded behavior. Fournet and Gonthier's Join-calculus [20] is based in the reflexive CHAM [8] and is related to the π-calculus by the fact that restriction, reception and replication are combined into a single concept: a *pattern*. Smolka's γ-calculus [64] 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. Victor and Parrow's Fusion-calculus [51] differs from the π-calculus in that it has a single binding operator and it features a new kind of action, *fusion*s, that are distinct from reductions in π in that names are not bound but rather unified as a side effect of the interaction between processes of opposite polarities.

Pure concurrent languages based in such formal frameworks have many important advantages:

- the programming style is declarative, meaning that only a description of the pattern of interaction between processes in the program is required. The underlying run-time system provides the required control in the form of an algorithm to run the program;

- concurrency is implicit in programs as it is usually one of the composition constructs. The granularity of the individual processes is usually very small giving the user, or better still the compiler, some control over the way these processes may be clustered to increase granularity;

- within the theoretical framework on which these languages are based it is possible to prove that the run-time systems possess important properties such as soundness, completeness and type-safety.

1.3 Goals and Contribution of this Thesis

This thesis uses Vasconcelos' TyCO - Typed Concurrent Objects - process calculus to define and implement a strongly and implicitly typed, polymorphic, concurrent, object-oriented language [71, 74]. The TyCO calculus is a form of the asynchronous π-calculus featuring builtin objects, asynchronous messages and template definitions as fundamental abstractions. The calculus formally describes the concurrent interaction of ephemeral objects through asynchronous communication. Synchronous communication is implemented using explicit reply names. Templates are specifications of processes abstracted on a sequence of variables allowing, for example, classes to be modeled. Unbounded behavior is modeled through explicit instantiation of recursive templates. A type system assigns monomorphic types to variables and polymorphic types to template variables [74]. TyCO is reminiscent of the Abadi and Cardelli's ς-calculus [1] in the sense 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. However, the ς-calculus is not concurrent and forms instead the basis of imperative object-oriented languages. Another important distinction is the existence of a method update operation in ς that is not present in TyCO.

TyCO provides a very clean model for an object-oriented language which combines the benefits of the formal framework of process calculi with the characteristics of actor systems. Actor languages have been implemented with some success as in the ABCL family [78, 79]. Moreover, most existing concurrent object-oriented languages incorporate aspects of the actor model in their specifications. TyCO objects are reminiscent of actors [4, 5, 34] in several ways as in response to a method invocation they may:

- initiate communications with other objects or even to themselves (by sending messages to *self*);
- create new objects either by prototyping or instantiation and propagate *self* channels and;
- change behavior after an incoming communication. This can be achieved by creating a new object at *self* with the same type (interface) but eventually with distinct method implementations. This also allows a form of dynamic behavior inheritance to be modeled.

Our main objectives with this work on TyCO and its implementation are:

- to produce a fine-grained kernel, polymorphic object-based concurrent programming language upon which higher level idioms can be efficiently implemented;
- to provide a self-contained run-time system for the language that has a clean, compact and portable implementation;
- to be efficient, running close to languages such as Pict [54], Oz [43], JoCaml [32] and Haskell [30];
- to provide a type system for the language to insure that well typed programs will not induce the run-time system into generating protocol errors;
- the executable programs (byte-code files) should be compact and architecture independent with a view to future applications in mobile computing.

This thesis describes the design and implementation of the TyCO programming language and its virtual machine. The language is a very low-level kernel language, with a syntax close to the calculus itself and a few derived constructs and builtin datatypes. It is intended as a building block for higher level distributed computing, object-oriented idioms. The language semantics is closely coupled with that of the underlying calculus. A type inference system ensures the type safety of programs, namely, it guarantees that no protocol errors are generated at run-time. Some higher-level idioms may be fully encoded into the kernel language, for example, functional processes and expressions [70]. On the other hand other features such as class hierarchies, mobility and remote interaction constructs must have support from extensions to TyCO such as the one described in chapter 8. The impact of these extensions on the language and especially in the virtual machine is usually quite localized and allows:

1. the virtual machine to remain quite small;
2. the compiler to extract type and usage information for channels from encodings or extra syntactic information;
3. greater flexibility in experimenting with distinct idioms and prototyping.

The virtual machine is implemented as a byte-code emulator with a heap for dynamic data-structures, a run-queue for fair scheduling of byte-code, an array for local variables and, an operand stack for the evaluation of builtin expressions. This is in contrast with other existing implementations such as Pict [54] whose implementation borrows extensively from C, and JoCaml [32] that is based on the Objective Caml implementation. The design is completely self contained in the tradition of other virtual machines proposed for other programming paradigms such as the STG-machine [33] or the WAM [77] for functional and logic programming, respectively. To our knowledge, and despite the extensive theoretical work in process calculi, a general framework for compiling these languages is still elusive, a situation that contrasts vividly with other programming paradigms.

TyCO was developed to serve as a test-bed for experimenting with prototype languages supporting code mobility in distributed environments. The design of the virtual machine was already constrained by requirements of compactness, portability and the use of an architecture independent executable format – the byte-code. Our previous experience in parallel computing makes us believe that compact virtual machine designs are better suited for concurrent object-oriented languages, whether the interest is to explore local and typically very fine grained parallelism or concurrency, or to evolve to mobile computations over fast heterogeneous networks where the latencies must be kept to the lowest possible. A final contribution of this thesis consists of establishing the theoretical framework that allows TyCO computations to occur in a distributed environment, introducing support for remote communication and code mobility. This framework can also easily be applied in the design and implementation of a modular compilation and linking tool for TyCO.

1.4 Thesis Outline

The thesis is structured as follows: chapter 2 describes the TyCO process calculus and its kernel programming language with examples; chapter 3 introduces the dynamic semantics for the TyCO language in the form of a formal abstract machine and proves some important properties such as soundness and the absence of deadlocks for well-typed programs; chapter 4 describes the design and implementation of the TyCO virtual machine, based on the formal specification of the previous chapter; chapter 5 describes the compilation scheme and implementation of the TyCO compiler, the format of the byte-code generated by the TyCO assembler and the implementation of the assembler program itself; chapter 6 describes a set of optimizations to the virtual machine; chapter 7 evaluates the performance of the implementation by comparing it to three other well known programming languages; chapter 8 introduces the notions of distribution and code mobility into TyCO providing the formal framework required for a distributed implementation; chapter 9 reviews some research work that relates directly to the subject of this thesis; finally, chapter 10 elaborates on the main conclusions that can be extracted from the work presented in this thesis, namely if and how the initial goals were achieved. The thesis ends with some pointers to future research and a few concluding remarks.

Chapter 2

The TyCO Calculus and Language

This chapter introduces the TyCO process calculus, its syntax and semantics. The dynamic semantics is given in the form of a reduction relation. The definition of this relation will provide the basic framework for the definition of an abstract machine for TyCO in the next chapter. The static semantics of the language is provided by a type checking system which supports a few builtin types, recursive types, and predicative polymorphism *à la* Damas-Milner over process abstractions.

2.1 The Syntax

In the sequel we describe the syntactic categories required in the definition of the TyCO calculus. *Names* are locations where communication occurs, elements of the set Name ranged over by a, b. Names are introduced in processes implicitly with the creation of each new local variable. Boolean, integer, floating-point or string *constants* are ranged over by c and belong to the set Const. *Variables* stand for names or constants in a given process context. They belong to set Var and are ranged over by x, y. We let $\tilde{x}$ denote the sequence $x_1 \cdots x_k$ $(k \geq 0)$ of pairwise distinct variables. *Values* are either constants or variables, belong to the set Val, and are ranged over by u, v. *Expressions*, in set Expr and ranged over by e, are builtin logical, numeric, relational and string operations. Similarly to variables we let $\tilde{e}$ denote the sequence $e_1 \cdots e_k$ $(k \geq 0)$ of expressions.

Other kinds of identifiers occur in processes. *Template variables*, belonging to the set TVar and ranged over by X, identify processes parameterized in a set of variables.

They are introduced syntactically in template definitions. *Labels*, in set Label and ranged over by l, are used to select methods within objects.

Process abstractions, ranged over by A in set Abs, are processes abstracted on a sequence of variables. *Method collections* form the bodies of objects. They are finite maps from labels into process abstractions, and are ranged over by M. Similarly, *template declarations* are maps from template variables into processes abstractions. They are ranged over by D.

We introduce *Processes*, the topmost category of the calculus, belonging to the set Proc and ranged over by P. Their structure is given by the following grammar:

$$
\begin{array}{rcll}
P & ::= & \mathsf{skip} & \text{Terminated Process} \\
 & \mid & P \mid P & \text{Concurrent Composition} \\
 & \mid & \mathsf{new}\ \tilde{x}\ P & \text{New Local Variables} \\
 & \mid & x?M & \text{Object} \\
 & \mid & x\,!\,l[\tilde{e}] & \text{Asynchronous Message} \\
 & \mid & X[\tilde{e}] & \text{Instantiation} \\
 & \mid & \mathsf{def}\ D\ \mathsf{in}\ P & \text{Template Definition} \\
 & \mid & \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ P & \text{Conditional} \\
 & \mid & (P) & \text{Grouping} \\
 & \mid & \mathsf{io}\,!\,\mathsf{put}[e] & \text{Output} \\
 & \mid & \mathsf{io}\,!\,\mathsf{get}[x] & \text{Input} \\
M & ::= & \{l_1 = A_1\ , \ldots ,\ l_k = A_k\} & \text{Method Collection} \\
D & ::= & X_1 = A_1\ \mathsf{and} \ldots \mathsf{and}\ X_k = A_k & \text{Recursive Definition} \\
A & ::= & (\tilde{x})P & \text{Process Abstraction} \\
e & ::= & e_1\ op\ e_2 \mid op\ e \mid (e) \mid v & \text{Expressions} \\
op & ::= & \mathsf{not} \mid \mathsf{and} \mid \mathsf{or} & \text{Boolean Operators} \\
 & \mid & + \mid - \mid * \mid / \mid \% & \text{Numerical Operators} \\
 & \mid & == \mid != \mid < \mid > \mid <= \mid >= & \text{Relational Operators} \\
 & \mid & \mathsf{cat} \mid \mathsf{len} & \text{String Operators}
\end{array}
$$

Some syntactic conventions are assumed for processes, namely: a) the parameters $\tilde{x}$ in process abstractions are pairwise distinct; b) the labels l_i are pairwise distinct in a method collection $\{l_1 = A_1\ , \ldots ,\ l_k = A_k\}$, and; c) the template variables X_i in a template declaration $X_1 = A_1$ **and** ... **and** $X_k = A_k$ are pairwise distinct.

2.2 Operational Semantics

Scope restriction and abstraction are the *variable binding operators* in the calculus. Variables $\tilde{x}$ are bound in the part P of an abstraction $(\tilde{x})P$ or a scope restriction **new** $\tilde{x}\ P$. The sets $\mathrm{fv}(P)$ and $\mathrm{bv}(P)$ of, respectively, *free variables* and *bound variables* of a process P are defined accordingly. In accordance with the current practice in programming languages, we let the scope of a **new** extend as far to the right as possible. A process P is said to be *closed for variables* if $\mathrm{fv}(P) = \emptyset$. We assume the usual definition of *simultaneous* substitution of values $\tilde{v}$ for the free occurrences of variables $\tilde{x}$ in a process P, denoted by $P\{\tilde{v}/\tilde{x}\}$. The substitution is defined only if the lengths of $\tilde{x}$ and $\tilde{v}$ match.

Declaration is the only *template variable binding operator*. The template variable X is bound in the parts P and Q of a process **def** $X = (\tilde{x})P$ **and** D **in** Q. The sets $\mathrm{ft}(P)$ and $\mathrm{bt}(P)$ of, respectively, *free template variables* and *bound template variables* of a process P are defined accordingly. A process P is said to be *closed for template variables* if $\mathrm{ft}(P) = \emptyset$. *Alpha conversion*, denoted $\equiv_\alpha$, is defined in the usual way for both variables and template variables.

Following Milner [44] we divide the computational rules of the calculus in two parts: the *structural congruence* rules and the *reduction* rules. Structural congruence rules allow the re-writing of processes into semantically equivalent expressions until they may be reduced using reduction rules.

We start by defining the reduction rules for expressions.

$$[\textsc{Const}]\quad c \to c \qquad [\textsc{Name}]\quad a \to a \qquad [\textsc{Sequence}]\quad \frac{e_1 \to e_1' \quad \cdots \quad e_k \to e_k'}{e_1 \cdots e_k \to e_1' \cdots e_k'}$$

$$[\textsc{Unary-C}]\quad op\ c \to \mathsf{op}(c) \qquad [\textsc{Binary-C}]\quad c_1\ op\ c_2 \to \mathsf{op}(c_1, c_2)$$

$$[\textsc{Unary-E}]\quad \frac{e \to e'}{op\ e \to\ op\ e'} \qquad [\textsc{Binary-E}]\quad \frac{e_1 \to e_1' \qquad e_2 \to e_2'}{e_1\ op\ e_2 \to e_1'\ op\ e_2'}$$

$$[\textsc{Expose}]\quad \frac{e \to c}{(e) \to c} \qquad [\textsc{Precedence}]\quad \frac{e \to e'}{(e) \to (e')}$$

We define *multi-step reduction* for expressions, or simply *reduction*, notation $e \to^* e'$, as the transitive closure of $\to$. The rules allow the re-writing of expressions into constants and names. For each operator op in the language we assume the existence of a corresponding builtin operator, op, that provides its implementation. Note that, to be evaluated, an expression cannot have any variables. In other words, the variables in the expression must have been previously substituted for names or constants through process reduction.

CONST reduces constants to themselves and NAME reduces names to themselves. Rule SEQUENCE allows the reduction of a sequence of expressions into another sequence. Rule PRECEDENCE evaluates expressions protected by parenthesis whereas rule EXPOSE gets rid of the parentesis once the expression inside has been computed. Rules UNARY-C and BINARY-C apply a builtin operation, op, to constants. Rule UNARY-E evaluates an expression under an unary operator. Rule BINARY-E does the same for two expressions linked by a binary operator.

Again, following Milner [44], we define $\equiv$ to be the smallest congruence relation over processes induced by the following rules.

$$
\begin{array}{rl}
[\textsc{Alpha}] & P \;\equiv\; Q \qquad \text{if } P \equiv_\alpha Q \\
[\textsc{Group}] & P \mid \textbf{skip} \;\equiv\; P \qquad P \mid Q \;\equiv\; Q \mid P \qquad P \mid (Q \mid R) \equiv (P \mid Q) \mid R \\
[\textsc{New}] & \textbf{new}\ x\ \textbf{skip} \;\equiv\; \textbf{skip} \\
& \textbf{new}\ x\ \textbf{new}\ y\ P \;\equiv\; \textbf{new}\ y\ \textbf{new}\ x\ P \\
& \textbf{new}\ x\ P \mid Q \equiv \textbf{new}\ x\ (P \mid Q) \qquad \text{if } x \notin \mathrm{fv}(Q) \\
[\textsc{Def}] & \textbf{def}\ D\ \textbf{in skip} \;\equiv\; \textbf{skip} \\
& \textbf{def}\ D\ \textbf{in new}\ x\ P \;\equiv\; \textbf{new}\ x\ \textbf{def}\ D\ \textbf{in}\ P \qquad \text{if } x \notin \mathrm{fv}\ D \\
& (\textbf{def}\ D\ \textbf{in}\ P) \mid Q \;\equiv\; \textbf{def}\ D\ \textbf{in}\ (P \mid Q) \qquad \text{if } \mathrm{ft}(D) \cap \mathrm{ft}(Q) = \emptyset \\
& \textbf{def}\ D\ \textbf{in def}\ D'\ \textbf{in}\ P \;\equiv\; \textbf{def}\ D\ \textbf{and}\ D'\ \textbf{in}\ P \qquad \text{if } \mathrm{dom}(D') \cap \mathrm{ft}(D) = \emptyset \\
[\textsc{Perm}] & M \equiv M' \quad \text{if } M' \text{ is a permutation of } M \\
& D \equiv D' \quad \text{if } D' \text{ is a permutation of } D
\end{array}
$$

Rule ALPHA ensures the equivalence of processes under *alpha conversion*. The rules in GROUP respectively garbage collect **skip** processes, and state the commutativity and associativity of the operator "|". Next, in NEW, the first rule garbage collects variables with empty scopes, the second commutes **new** statements and the third allows the scope of a newly introduced variable to be extended further to the right provided that there is no capture of variables. The rules in DEF deal with definitions. The first garbage collects definitions with empty continuations. The second allows **new** statements to be moved in and out of **def** statements provided that there is no variable capture in the bodies of the templates in D. The third rule allows the scope of a template variable to be extended as much to the right as possible, that is, without variable capture. The fourth allows template declarations to coalesce and be pushed to the start of a process. Finally, the rules in PERM, state that permutations of both method collections and template declarations are equivalent.

Reduction rules drive the computation either by consuming *redexes* – pairs object-message interacting on a shared name – and replacing them for their *contractions*, or by creating new instances of process templates. Reduction does not necessarily contract a term. For example, a process of the form **def** $X = X \mid X$ **in** X, expands to the process **def** $X = X \mid X$ **in** $X \mid X$ after one reduction step.

A message $x!l_i[\tilde{e}]$ selects the method l_i in an object $x?\{l_1 = (\tilde{x}_1)P_1, \ldots, l_k = (\tilde{x}_k)P_k\}$. The result is the process P_i where the values $\tilde{v}$ of the expressions in $\tilde{e}$ are bound to the variables in $\tilde{x}_i$. This form of reduction is called *communication*.

$$[\textsc{Comm}] \qquad \frac{\tilde{e} \to^* \tilde{v}}{x\,!\,l_i[\tilde{e}] \mid x?\{l_1 = (\tilde{x_1})P_1\ , \ldots,\ l_k = (\tilde{x_k})P_k\} \to P_i\{\tilde{v}/\tilde{x}_i\}}$$

Another form of reduction occurs when we create a new instance $X[\tilde{e}]$ of a template definition $X = (\tilde{x})P$. The result is the process $P\{\tilde{v}/\tilde{x}\}$ where the values $\tilde{v}$ of the expressions $\tilde{e}$ replace the variables $\tilde{x}$. This form of reduction is called *instantiation*.

$$[\textsc{Inst}] \qquad \frac{\tilde{e} \to^* \tilde{v}}{\textbf{def}\ X = (\tilde{x})P\ \textbf{and}\ D\ \textbf{in}\ X[\tilde{e}] \to \textbf{def}\ X = (\tilde{x})P\ \textbf{and}\ D\ \textbf{in}\ P\{\tilde{v}/\tilde{x}\}}$$

Finally, we have rules that allow reduction over concurrent composition, **new**s and **def**s, as well as the usual structural congruence rule that allows processes to be re-written to form redexes.

$$[\textsc{Conc}] \quad \frac{P \to P'}{P \mid Q \to P' \mid Q} \qquad\qquad [\textsc{New}] \quad \frac{P \to P'}{\textbf{new}\ x\ P \to \textbf{new}\ x\ P'}$$

$$[\textsc{Def}] \quad \frac{P \to P'}{\textbf{def}\ D\ \textbf{in}\ P \to \textbf{def}\ D\ \textbf{in}\ P'} \qquad\qquad [\textsc{Str}] \quad \frac{P' \equiv P \quad P \to Q \quad Q \equiv Q'}{P' \to Q'}$$

We define *multi-step reduction*, or simply *reduction*, notation $P \to^* Q$, as the relation $\equiv \cup \to^+$, where $\to^+$ is the transitive closure of $\to$.

Finally, we single out a name – **io** – to be used for input/output. We assume a persistent object present at this name that services all the IO requests from processes. The signature of this object is of the form:

$$\textbf{io}?\{\textbf{put} = (x)\ldots, \textbf{get} = (x)\ldots\}$$

A message $\textsf{\textbf{io}}!\textsf{\textbf{put}}[$``‘‘hello world’’``$]$ outputs the string ``‘‘hello world’’``, whereas a message $\textsf{\textbf{io}}!\textsf{\textbf{get}}[x]$ gets an input value and sends it in a *val* labeled message to x.

$$\text{PUT} \quad \textsf{\textbf{io}}!\textsf{\textbf{put}}[v] \to \textsf{\textbf{skip}} \qquad\qquad \text{GET} \quad \textsf{\textbf{io}}!\textsf{\textbf{get}}[x] \to x!\textsf{\textit{val}}[v]$$

This IO capability is rather simple and is not suitable for concurrent computing since:

- the **io** name is shared by all processes and there is no way to have exclusive access to the channel while performing IO;
- the **io** name is associated with a single output and input capability. This obviously constitutes a bottleneck for the IO system.

Moreover, the polymorphism of the IO object although supported by the compiler is not a feature of the type system for TyCO, which does not support polymorphism on the parameters of method definitions.

Despite these disadvantages, we chose to implement IO in this simple way since our programs have very limited IO requirements and we were more interested in other aspects of the language. Future developments may introduce a more robust IO system, possibly based on monads [3].

This completes the dynamic semantics of the calculus.

2.3 Type System

This section presents a type checking system for TyCO. The type system features *monomorphic types* including primitive, record and recursive types for channels, comprising a simple extension of the type system for the pure calculus without abstractions [74]. It also supports *polymorphic types*, based on the Damas-Milner type system for the ML programming language [14], and on the predicative polymorphic type system for the π-calculus [69]. Polymorphic definitions are allowed over process abstractions.

2.3.1 Types

Types for variables are built from *type constants*, an infinite set of *type-variables*, and the set of labels introduced in the previous section, by means of a record constructor and a recursive type constructor.

The *type constants* are elements of the set $\{\mathsf{bool}, \mathsf{int}, \mathsf{float}, \mathsf{string}\}$, ranged over by ρ. We fix a denumerable set of *type variables* ranged over by t, and let α, β range over types. A sequence of types $\alpha_1 \cdots \alpha_n$ $(n \geq 0)$ is abbreviated to $\tilde{\alpha}$. Polymorphic types are introduced by universally quantifying ($\forall$) particular type variables in the types for abstractions. The *set of monomorphic types*, ranged over by α, and the *set of polymorphic types*, ranged over by σ, are given by the grammar:

$$\begin{array}{lcl} \rho & ::= & \mathsf{bool} \mid \mathsf{int} \mid \mathsf{float} \mid \mathsf{string} \\ \alpha & ::= & \rho \mid t \mid \{l_1\colon \tilde{\alpha}_1, \ldots, l_n\colon \tilde{\alpha}_n\} \mid \mu t.\alpha \\ \sigma & ::= & \tilde{\alpha} \mid \forall t.\sigma \end{array}$$

where $n \geq 0$, and the labels $l_1, \ldots, l_n$ are pairwise distinct. As a consequence of the definition, universal quantifiers can only occur at the top level of types: polymorphic types are of the form $\forall t_1. \cdots .\forall t_n.\tilde{\alpha}$ for $n \geq 0$.

A type of the form $\{l_1\colon \tilde{\alpha}_1, \ldots, l_n\colon \tilde{\alpha}_n\}$ is intended to describe names representing locations of objects containing n methods labelled with $l_1, \ldots, l_n$, and whose arguments of method l_i belong to the types $\tilde{\alpha}_i$. Types are interpreted as rational (or regular infinite) trees. A type of the form $\mu t.\alpha$ (with $\alpha \neq t$) denotes the rational tree solution of the equation $t = \alpha$ [13, 72]. If α is a type, denote by α^* its associated rational tree. An interpretation of recursive types as infinite trees naturally induces an equivalence relation $\approx$ on types, by putting $\alpha \approx \beta$ if $\alpha^* = \beta^*$.

Type assignments to variables are formulae $x\colon \alpha$, for x a variable and α a type. *Typings*, denoted by Γ, are sets of type assignments where no variable occurs twice. If $\tilde{x} = x_1 \cdots x_n$ is a sequence of pairwise distinct variables, and $\tilde{\alpha} = \alpha_1 \cdots \alpha_n$ is a sequence of types, we write $\{\tilde{x}\colon \tilde{\alpha}\}$ for the typing $\{x_1\colon \alpha_1, \ldots, x_n\colon \alpha_n\}$, and $\Gamma \cdot \tilde{x}\colon \tilde{\alpha}$ for the typing $\Gamma \cup \{\tilde{x}\colon \tilde{\alpha}\}$, provided that variables in $\tilde{x}$ do not appear in Γ. The typing $\Gamma \setminus x$ is obtained from Γ by removing the type assignment associated with x. When we write $\Gamma(x)$ we assume $x : \alpha \in \Gamma$ for some α. The expression $\Gamma(x)$ denotes the type α. Accordingly, $\Gamma(x_1 \cdots x_n)$ represents the sequence of types $\alpha_1 \cdots \alpha_n$ for $x_i : \alpha_i \in \Gamma$ and $i = 1 \ldots n$.

Type assignments to template variables are formulae $X\colon \sigma$, for X an abstraction variable and σ a polymorphic type. *Bases*, denoted by Δ, are sets of type assignments to variables where no variable occurs twice. Similarly to typings, we write $\Delta \cdot X\colon \sigma$ for $\Delta \cup \{X\colon \sigma\}$, provided that X does not appear in Δ, write $\Delta \setminus X$ for the basis Δ with no occurrences of X, and write $\Delta(X)$ for σ if $X : \sigma \in \Delta$.

Typing assignments to processes are statements $\Delta, \Gamma \vdash P$, whereas *type assignment* *to abstractions* (or *methods* or *template variables*) are statements $\Delta, \Gamma \vdash A : \sigma$ (or $\Delta, \Gamma \vdash M : \alpha$ or $\Delta, \Gamma \vdash X : \sigma$, respectively).

A type variable t occurs *free* in a type if t is not in the scope of a μt or a $\forall t$; otherwise t occurs *bound*. A type-variable occurs free in a typing Γ (respectively basis Δ) if it occurs free in some type in Γ (respectively Δ). Finally, we assume the usual notion of substitution of a type α for the free occurrences of a type-variable t in a type σ, and denote it by $\sigma\{\alpha/t\}$.

2.3.2 Typing Expressions

To obtain the typing rules for expressions one must first observe that their types depend only on the Γ portion of the type environment and are monomorphic. So, typing judgments are of the form $\Gamma \vdash e : \alpha$. Also, we define the notation $\tilde{\alpha} \to \tilde{\beta}$ to abbreviate the type $\{\mathit{val} : \tilde{\alpha}\{\mathit{val} : \tilde{\beta}\}\}$. To extract the types for constants and operators we assume the existence of a function typeof that, given a constant or a primitive operator, returns its type. Thus, for example, $\mathsf{typeof}(2) = \mathsf{int}$ and $\mathsf{typeof}(\mathbf{len}) = \mathsf{string} \to \mathsf{int}$. Given these definitions the typing rules for expressions are straightforward:

$$\textsc{Const}\quad \Gamma \vdash c : \mathsf{typeof}(c) \qquad\qquad \textsc{Var}\quad \Gamma \vdash x : \Gamma(x)$$

$$\textsc{SeqE}\quad \frac{\Gamma \vdash e_1 : \alpha_1 \quad \cdots \quad \Gamma \vdash e_k : \alpha_k}{\Gamma \vdash e_1 \cdots e_k : \alpha_1 \cdots \alpha_k} \qquad\qquad \textsc{ParE}\quad \frac{\Gamma \vdash e : \alpha}{\Gamma \vdash (e) : \alpha}$$

$$\textsc{UnE}\quad \frac{\mathsf{typeof}(op) = \alpha \to \beta \quad \Gamma \vdash e : \alpha}{\Gamma \vdash \ op\, e : \beta}$$

$$\textsc{BinE}\quad \frac{\mathsf{typeof}(op) = \alpha_1\alpha_2 \to \beta \quad \Gamma \vdash e_1 : \alpha_1 \quad \Gamma \vdash e_2 : \alpha_2}{\Gamma \vdash e_1\ op\ e_2 : \beta}$$

Note that in the case of the binary operators currently defined in TyCO both parameters have the same type which may (for example, +) or may not (for example, ==) be the type of the result. So we could write in rule BinE the type for the operators as: $\mathsf{typeof}(op) = \alpha\alpha \to \beta$. The rule SeqE is required to assemble the types for sequences of expressions, from the individual types of the elements of the sequence.

2.4 Typing Processes

The typing rules and axioms for processes are as follows:

$$\textsc{Nil}\quad \Delta,\Gamma \vdash \textsf{\textbf{skip}} \qquad \textsc{Res}\quad \frac{\Delta,\Gamma \vdash P}{\Delta,\Gamma\setminus x \vdash \textsf{\textbf{new}}\ x\ P} \qquad \textsc{Par}\quad \frac{\Delta,\Gamma \vdash P \quad \Delta,\Gamma \vdash Q}{\Delta,\Gamma \vdash P \mid Q}$$

$$\textsc{Msg}\quad \frac{\Gamma \vdash \tilde{e} : \tilde{\alpha} \quad \Gamma(x) = \{l_1 \colon \tilde{\alpha}_1, \ldots, l_n \colon \tilde{\alpha}_n\} \quad \tilde{\alpha} \approx \tilde{\alpha}_i}{\Delta,\Gamma \vdash x\,!\,l_i[\tilde{e}]}$$

$$\textsc{Obj}\quad \frac{\Delta,\Gamma \vdash M : \alpha \quad \Gamma(x) \approx \alpha}{\Delta,\Gamma \vdash x?M} \qquad \textsc{Inst}\quad \frac{\Gamma \vdash \tilde{e} : \tilde{\alpha} \quad \Delta,\Gamma \vdash X : \tilde{\beta} \quad \tilde{\alpha} \approx \tilde{\beta}}{\Delta,\Gamma \vdash X[\tilde{e}]}$$

$$\textsc{If}\quad \frac{\Gamma \vdash e : \textsf{bool} \quad \Delta,\Gamma \vdash P \quad \Delta,\Gamma \vdash Q}{\Delta,\Gamma \vdash \textsf{\textbf{if}}\ e\ \textsf{\textbf{then}}\ P\ \textsf{\textbf{else}}\ Q}$$

$$\textsc{Def}\quad \frac{\Delta,\Gamma \vdash A_1 : \sigma_1 \quad \cdots \quad \Delta,\Gamma \vdash A_n : \sigma_n \quad \Delta \setminus \tilde{X} \cdot \tilde{X} : \tilde{\sigma},\Gamma \vdash P}{\Delta \setminus \tilde{X},\Gamma \vdash \textsf{\textbf{def}}\ X_1 = A_1\ \textsf{\textbf{and}} \ldots \textsf{\textbf{and}}\ X_n = A_n\ \textsf{\textbf{in}}\ P}$$

where, in the last rule, $\Delta(X_i) \approx \tilde{\alpha}_i$ and $\sigma_i = \forall\tilde{t}_i.\tilde{\alpha}_i$. The next rules allow the typing of method collections (Meth), abstractions (Abs), and the introduction of new template variables (Var). The final two rules ($\forall$-intro and $\forall$-elim) allow, respectively, to introduce polymorphism by abstraction on type variables in the types of process abstractions and, to eliminate polymorphism in an abstraction by replacing an abstracted type variable with a monomorphic type.

$$\textsc{Meth}\quad \frac{\Delta,\Gamma \vdash A_1 : \tilde{\alpha}_1 \quad \cdots \quad \Delta,\Gamma \vdash A_n : \tilde{\alpha}_n}{\Delta,\Gamma \vdash \{l_1 = A_1, \ldots, l_n = A_n\} : \{l_1 \colon \tilde{\alpha}_1, \ldots, l_n \colon \tilde{\alpha}_n\}}$$

$$\textsc{Var}\quad \Delta \cdot X \colon \sigma,\Gamma \vdash X : \sigma \qquad \textsc{Abs}\quad \frac{\Delta,\Gamma \cdot \tilde{x} : \tilde{\alpha} \vdash P}{\Delta,\Gamma \vdash (\tilde{x})P : \tilde{\alpha}}$$

$$\forall\text{-INTRO}\quad \frac{\Delta,\Gamma \vdash A : \sigma}{\Delta,\Gamma \vdash A : \forall t.\sigma}\ (t \text{ not free in } \Delta,\Gamma) \qquad \forall\text{-ELIM}\quad \frac{\Delta,\Gamma \vdash X : \forall t.\sigma}{\Delta,\Gamma \vdash X : \sigma\{\alpha/t\}}$$

One could be tempted to define the polymorphic system by simply allowing polymorphic types in the Def-rule, but then we would run against the classical problem of polymorphic recursion variables [48].

Example: One-place buffer cell. A cell is an object that holds a value and answers to two kinds of messages: write and read. Together with a write message comes the value to be stored in the cell. The read message carries a name intended to receive the value the cell holds, thus simulating a synchronous method call within an asynchronous calculus.

```
def Cell = (self value)
    self ? {
            write= (newVal) Cell[self newVal],
            read= (replyTo) replyTo!val[value]  | Cell[self value]
    }
```

Denoting by A the cell abstraction, a simple deduction ending with the Abs-rule shows that

$$\mathsf{Cell} : (\{\mathit{write} : t, \mathit{read} : \{\mathit{val} : t\}\}\, t), \emptyset \vdash A : \{\mathit{write} : t, \mathit{read} : \{\mathit{val} : t\}\}\, t.$$

The type for the cell abstraction conveys the idea that cells are objects holding values of type t, and composed of two methods *write* and *read*. Method *write* accepts names of type t, and method *read* accepts names capable of receiving *val*-labelled messages carrying names of type t. This is not the only possible type for the cell abstraction: we may not only replace type variable t by an arbitrary type α, but also extend the type of the *read* method to obtain, say

$$\mathsf{Cell} : \{\mathit{write} : \alpha, \mathit{read} : \{\mathit{val} : \alpha, \mathit{l} : \beta\}\}\, \alpha, \emptyset \vdash A : \{\mathit{write} : \alpha, \mathit{read} : \{\mathit{val} : \alpha, \mathit{l} : \beta\}\}\, \alpha.$$

Comment. In kinded type systems [50, 74] abstraction Cell is assigned a type $(\{\mathit{write} : \alpha, \mathit{read} : t'\}, t)$, and type variable t' is assigned a kind $\{\mathit{val} : \alpha\}$. Actual types for Cell are obtained by replacing t' by records containing at least the component $\mathit{val} : \alpha$ in the kind.

Continuing with the buffer cell example, we may use rule ∀-intro to abstract the type of the value the cell holds, to obtain the polymorphic type:

$$\mathsf{Cell} : (\{\mathit{write} : t, \mathit{read} : \{\mathit{val} : t\}\}\, t), \Gamma \vdash A : \forall t.(\{\mathit{write} : t, \mathit{read} : \{\mathit{val} : t\}\}\, t).$$

Then, using twice the ∀-elim rule we create two cells with particular instances of the polymorphic type.

$$\Gamma \vdash \textbf{def}\ \mathsf{Cell} = A\ \textbf{in}\ \mathsf{Cell[n\ 1]} \mid \mathsf{Cell[b\ true]}$$

where Γ is the typing $\{\mathsf{n} : \{\mathit{write} : \mathsf{int}, \mathit{read} : \{\mathit{val} : \mathsf{int}\}\}, 1 : \mathsf{int}, \mathsf{b} : \{\mathit{write} : \mathsf{bool}, \mathit{read} : \{\mathit{val} : \mathsf{bool}\}\}, \mathsf{true} : \mathsf{bool}\}$.

2.4.1 Some Properties of the Type System

We conclude this section with a brief overview of some important properties of the type systems. These properties establish the *type-safety* property of a TyCO program, defined as the absence of run-time protocol errors. Type safety is a direct result of the *well typedness* of a program and *subject reduction.* This property is of fundamental importance in the specification of the abstract machine for TyCO and in establishing some of its properties (see chapter 3). The type systems just described were first studied by Vasconcelos and proofs, for the monomorphic system, can be found in references [68, 74]. For the polymorphic system (in the context of the π-calculus) the reader may refer to [69].

Subject-reduction ensures that a typing for a process does not change as the process is reduced.

Theorem (Subject-reduction). *If* $\Delta, \Gamma \vdash P$ *and* $P \rightarrow^* Q$*, then* $\Delta, \Gamma \vdash Q$*.*

As a corollary, typable processes do not encounter type errors at runtime. We say a process P contains a possible *runtime error* if

$$P \rightarrow^* \textsf{def } D_1 \textsf{ in} \cdots \textsf{def } D_n \textsf{ in new } \tilde{y}\ x?M \mid x\,!\,l[\tilde{e}] \mid Q$$

and l labels no method in M, or

$$P \rightarrow^* \textsf{def } D_1 \textsf{ in} \cdots \textsf{def } D_n \textsf{ and } X = (\tilde{x})P \textsf{ in new } \tilde{y}\ X[\tilde{e}] \mid Q$$

and the lengths of $\tilde{x}$ and $\tilde{e}$ do not match.

Corollary. *If* $\Gamma \vdash P$*, then* P *contains no runtime error.*

Decidability of typing assignment (given Δ, P, and Γ, is it the case that $\Delta, \Gamma \vdash P$?) and *computability of typing existence* (given Δ and P, is there a Γ such that $\Delta, \Gamma \vdash P$?) are two properties verified by the predicative polymorphic system despite the fact that the system possess no simple notion of subsumption (and hence of principal typings). Such notions can be obtained by introducing, for example, constraints on the types that may replace a given type-variable, in the form of Ohori's *kinds* [50, 74] (see comment in page 40).

2.5 Derived Constructs

We introduced a few derived constructs in TyCO to ease the programming task by making the processes more readable. Some of these constructs are also good candidates for optimization and will be discussed in chapter 6. The builtin conditional construct and its single branch form are standard constructs in concurrent programming. We single out a label — *val* — to be used in objects with a single method. This allows the programmer to abbreviate messages and objects.

$$\begin{array}{lcl}
\textbf{if } e \textbf{ then } P & \equiv & \textbf{if } e \textbf{ then } P \textbf{ else skip} \\
x![\tilde{e}] & \equiv & x!\mathit{val}[\tilde{e}] \\
x?(\tilde{y})P & \equiv & x?\{\mathit{val} = (\tilde{y})P\}
\end{array}$$

The ";" operator is the sequential composition operator. Here it is defined for synchronous communication and synchronous procedure call.

$$\begin{array}{rcl}
x!l[\tilde{e}] \ ; \ P & \equiv & \textbf{new } y \ x!l[\tilde{e}y] \mid y?()P \\
X[\tilde{e}] \ ; \ P & \equiv & \textbf{new } y \ X[\tilde{e}y] \mid y?()P
\end{array}$$

We also define two constructors for two special and particularly pervasive cases of synchronization: synchronous procedure call and pattern matching. The **let** constructor is quite useful in getting back results from synchronous calls; the syntax is taken from Pict [54]. Finally, the **match** constructor is most useful for pattern matching; the syntax is taken from ML [47].

$$\begin{array}{lcl}
\textbf{let } y = x!l[\tilde{e}] \textbf{ in } P & \equiv & \textbf{new } z \ x!l[\tilde{e}z] \mid z?(y)P \\
\textbf{let } y = X[\tilde{e}] \textbf{ in } P & \equiv & \textbf{new } z \ X[\tilde{e}z] \mid z?(y)P \\
\textbf{match } x!l[\tilde{e}] \textbf{ with } M & \equiv & \textbf{new } y \ x!l[\tilde{e}y] \mid y?M \\
\textbf{match } X[\tilde{e}] \textbf{ with } M & \equiv & \textbf{new } y \ X[\tilde{e}y] \mid y?M
\end{array}$$

2.6 Examples

To illustrate the programming style and syntax of the language we sketch a few simple example programs written in TyCO.

The Takeushi function. The so called Takeushi numbers are obtained from a deeply recursive function with three natural parameters.

$$tak(x, y, z) = \begin{cases} z & \text{if } y \geq x, \\ tak(tak(x-1, y, z), tak(y-1, z, x), tak(z-1, x, y)) & \text{otherwise.} \end{cases}$$

In TyCO we can implement this function as a recursive template that takes three parameters x, y and z and whose process body implements the body of the Takeushi function. Since this is a function we need an extra parameter, replyTo, where the result of the call will be sent to.

```
def tak = (x y z replyTo)
    if y ≥ x then
        replyTo![z]
    else
        let v1 = tak[x-1 y z] in
        let v2 = tak[y-1 z x] in
        let v3 = tak[z-1 x y] in
        let v = tak[v1 v2 v3] in
        replyTo![v]
in  let v = tak[22 16 8] in io!put[v]
```

The continuation process just launches the computation by instantiating an initial tak process with the required arguments and waits for the reply in variable v. When this synchronous call receives its reply, its continuation sends the value in v to the standard output.

Adding the nodes in a binary tree. We first define the tree data-structure composed of Leafs with integer attributes and Nodes that recursively branch into a left and right tree.

```
def Leaf=(self n) self ? (replyTo) replyTo!leaf[n]  | Leaf[self n]
and Node=(self l r) self ? (replyTo) replyTo!node[l,r]  | Node[self l r]

in
```

```
def Adder = (t replyTo)
        match t![] with {
                  leaf= (n) replyTo![n],
                  node= (l r) let lval = Adder[l] in
                              let rval = Adder[r] in
                              replyTo![lval+rval]
        }
in
new a b c d e Leaf[a 7] | Leaf[b 2] | Node[c b a] | Leaf[d 3] | Node[e,d,c] |
let sum = Adder[e] in io!put[sum]
```

The Adder process is actually a function that takes a tree t as input, checks its internal structure and either replies an integer immediately or it recursively computes the sums for the left and right sub-trees. The result is sent on the channel replyTo. The final instantiations in the top level process just build an example tree and then invoke the Adder.

The Sieve of Erathostenes. We define a sequence of process templates which generate a stream of natural numbers and check whether they are primes. All primes found are added to a dynamically growing chain of Sieves, one per each prime number found. The sieve chain filters the natural as they are generated in Nats.

```
def Nats = (n m sieve)
   sieve![n] ; if n < m then Nats[n+1 m sieve]

and Sieve = (self prime nextsieve)
        self ? (n done)
             ( if n % prime != 0 then
                 nextsieve![n done]
             else
                 done![] )
             | Sieve[self prime nextsieve]

and Sink = (self)
        self ? (n done)
             io!put[n] |
             new newsieve Sink[newsieve] |
```

```
                    Sieve[self n newsieve]  |
                    done![]

in  io!put[2]  | new sieve Nats[2 1000 sieve]  | Sink[sieve]
```

Nats generates a stream of naturals from n to m. Each natural is sent to the first sieve in the chain at sieve. The generation of a natural is synchronized with the updates in the chain of sieves. Sieve is a template for a sieve in the chain. Each sieve in the chain passes the current natural n to the next sieve, if it does not divide the sieve's prime prime. Otherwise, the natural is forgotten and the sieve signals Nats to proceed. Sink signals the end of the chain. A natural that gets this far is surely a prime and is printed. The sieve chain is then extended by adding a new sieve whose prime is n. The sink is placed after the new sieve and, Nats is signaled to send in the next natural. Finally, at the top level process we start computing the primes by creating a stream of the first 1000 naturals and, an initial sink at sieve.

Chapter 3

The Abstract Machine

In this chapter we define the dynamic semantics of the TyCO programming language in the form of an abstract machine specification. We discuss the motivation for our abstract machine design and its data-structures. Our design follows that of Turner's machine for the asynchronous π-calculus [66] but extends it to support template instantiation and method selection in objects, and refine it by using recursion to model unbounded behavior. Furthermore, our design defines the concurrency unit as a new syntactic category – the *thread* – and matches it with the bodies of template definitions and methods in objects. This matching allows a clean definition of an abstract machine suitable for a concurrent object-oriented language as well as a clear mapping between the abstract machine and its implementation. For example, methods in objects are executed without suspension, and may persist (become available again) after the method executes, which is important for keeping a consistent object state. Another example is the scope of variables which is naturally defined in method and template body units.

3.1 Communication

One of the major strengths of TyCO relative to other calculi, namely π, is that objects are builtin abstractions. From an implementation point of view two points are of interest: the method invocation protocol and the space complexity of objects. In the sequel we compare TyCO to the asynchronous π calculus from the point of view of these properties. Given an object in TyCO:

$$a?\{l_1 = (\tilde{x}_1)P_1, \ldots, l_k = (\tilde{x}_k)P_k\}$$

the invocation of method l_i is achieved with a single asynchronous message: $a\,!\,l_i[\tilde{v}]$. Also, the object requires only one name and a closure for its code. In π, on the other hand the encoding of a similar object would be:

$$\nu b_1 \dots b_k \quad \bar{a}(b_1 \dots b_k) \mid b_1(\tilde{x}_1).P_1 \mid \dots \mid b_k(\tilde{x}_k).P_k$$

To invoke the i-th method of the object we must first get the object interface by consuming the message $\bar{a}(b_1 \dots b_k)$. This removes the message from the environment and avoids interference from other processes. The message actually works as a lock on the object interface. When the interface in the form of the names b_i where the method bodies are held is acquired, an asynchronous message is sent to b_i to invoke the method with some arguments. Thus the code for method invocation can be written as:

$$a(x_1 \dots x_k).\bar{x}_i(\tilde{v})$$

So, in π, besides a two way protocol for method invocation an object with k methods requires $k+1$ names and k processes to be created. Of these k processes only one is actually used in the invocation. The other $k-1$ processes and associated b_j names must be garbage collected.

3.2 Congruence

Computation in the TyCO calculus is driven by reduction closely coupled with congruence. As we have seen in chapter 2 there are two kinds of reduction: communication and instantiation. Assume we have the following parallel composition of processes.

$$a?\{l = (\tilde{x})P\} \mid \cdots \mid a\,!\,l[\tilde{v}] \mid \cdots$$

For a communication to occur the object $a?\{l = (\tilde{x})P\}$ and the message $a\,!\,l[\tilde{v}]$ must be in syntactically adjacent positions in the composition (see the Comm rule in section 2.2) which is not usually the case. However structural congruence allows the re-ordering and the association of processes within a concurrent composition. In this way, in a succession of steps, we can make the object and the message in the previous example syntactically adjoint and reduce using rule Comm. Reduction here is crucial since the expansion of the new process places the processes in the body of method l, previously guarded by the prefix $a?$, available for further reduction.

From an implementation point of view and therefore from the point of view of the virtual machine specification, it is unreasonable to re-order sequences during the computation so that redexes may form and communication may take place. Instead we view each name, where objects may be held and where messages may be sent to, as a queue and we call it a *channel*. When a process such as $a?M$ is found, the result is that the method collection M is placed in the queue associated with a. We proceed similarly for messages $a\,!\,l[\tilde{v}]$. A reduction occurs when one tries to enqueue an object in a queue where a message is present, or the dual case for messages. In this sense reduction is eager.

3.3 Environment

We have said that the method collection M of an object $a?M$ is placed in the queue of a; some care must be taken with this operation. In fact, when we enqueue an object we do not know in general when (or if) it will reduce, much less so in a concurrent computation. If the object must be enqueued, the bindings for its free variables at the time it is found must be preserved, together with its method collection, in a closure. This is required for reduction may later occur in a different environment. The same happens with template definitions that have free variables. The bindings for these variables must be preserved at the time the definition is found and should be shared by all the instances of the template.

3.4 Program Flow

Notice that the processes that may result from reduction are either the bodies of template definitions or the bodies of methods in objects. The scope of the parameter bindings in a template or a method is the body of the template or method. Also, any new variable introduced with a **new** statement has its scope restricted at most by these boundaries. Taking a single process as the basic executable unit of the machine and allowing for concurrency at this level would make the management of process environments rather cumbersome and inefficient. In this case the bindings would have to be managed on a per process basis at the cost of greater heap usage and finer grained work. Moreover, in the TyCO calculus, the new processes resulting from reductions are expanded immediately. From an implementation point of view this corresponds to the suspension of the current sequence of processes, and running the new one that emerged from the reduction. This is not reasonable from an implementation point of view since the resulting machine would not be fair. Instead our machine delays the execution of new threads by placing them in a run-queue. A concurrent implementation can take these threads and run them concurrently.

We take the bodies of template definitions and methods of objects and name them abstract machine *threads*. They constitute the basic execution block of the machine. Processes in a thread are executed sequentially and the thread never suspends. Concurrency and non-determinism exist outside the boundaries of threads. In particular, methods of objects are executed without ever suspending. This definition matches nicely with our aim that objects should have actor-like behavior.

3.5 Preliminary Definitions

We first introduce some preliminary definitions.

- $\tilde{s}$ denotes the sequence $s_1 \cdots s_k$ $(k > 0)$ of elements of some syntactic category, and $\{\tilde{s}\}$ denotes the set $\{s_1, \cdots, s_k\}$. $|s|$ denotes the size of the sequence s.

- Let A,B range over sets. $A \mapsto B$ denotes the set of *finite maps* (partial functions with finite domain) from A to B, ranged over by f,g. Finite maps are often written explicitly in the form $\{a_1 : b_1, \cdots, a_k : b_k\}$ for $k \geq 0$. In particular, the empty map is $\{\}$. When $\tilde{a} = a_1 \cdots a_k$ and $\tilde{b} = b_1 \cdots b_k$, we abbreviate the finite map to $\tilde{a} : \tilde{b}$.

- The finite map f *modified* by g has domain $\mathrm{dom}(f) \cup \mathrm{dom}(g)$ and values
$$(f + g)(a) = \text{if } a \in \mathrm{dom}(g) \text{ then } g(a) \text{ else } f(a).$$

- The map $f \restriction A$, pronounced f *restricted to* A, is the finite map with domain $\mathrm{dom}(f) \cap A$ and values
$$(f \restriction A)(a) = f(a).$$

- Queues are used to implement threads, the run-queue and, the message/object queues, and are defined recursively by:
$$\mathrm{Queue}(\alpha) \quad ::= \quad \alpha :: \mathrm{Queue}(\alpha) \quad | \quad \bullet$$
for elements of some syntactic category α.

3.6 The Target Language

We take the syntax introduced in section 2.1, and scale it down to a kernel calculus without expressions nor builtin datatypes. We borrow the notation from the previous chapter and thus have for the basic categories:

$a \in \text{Name}$	Names
$x, y \in \text{Var}$	Variables
$u, v \in \text{Var} \cup \text{Name}$	Values
$l \in \text{Label}$	Labels
$X \in \text{TVar}$	Template Variables

The main novelty is the introduction of two new syntactic categories: instructions and threads. *Instructions* are closely related to TyCO processes. They are elements of the set Instr and are ranged over by I. *Threads* are elements of the set Thread of queues of instructions and are ranged over by T.

$I \in \text{Instr}$	Instruction
$T \in \text{Thread} = \text{Queue}(\text{Instr})$	Thread
$A \in \text{Abs} = \text{Var}^* \times \text{Thread}$	Thread Abstraction
$M \in \text{Meth} = \text{Label} \mapsto \text{Abs}$	Methods
$D \in \text{DBind} = \text{TVar} \mapsto \text{Abs}$	Definitions

The syntax of the kernel calculus uses threads as the topmost category and is defined by the following grammar:

T	$::=$	$\bullet \;\mid\; I :: T$	Thread
I	$::=$	$\textbf{skip}$	Terminated Process
	$\mid$	$x?M$	Object
	$\mid$	$x\,!\,l[\tilde{v}]$	Asynchronous Message
	$\mid$	$X[\tilde{v}]$	Instantiation
	$\mid$	$\textbf{new}\ x$	New Local Variable
	$\mid$	$\textbf{def}\ D$	Template Definition
M	$::=$	$\{l_1 = A_1\ ,\ \ldots\ ,\ l_k = A_k\}$	Method Collection
D	$::=$	$X_1 = A_1\ \textbf{and} \ldots \textbf{and}\ X_k = A_k$	Template Declaration
A	$::=$	$(\tilde{x})T$	Thread Abstraction

All the relevant definitions, binding operators and syntactic restrictions described in section 2.1 are valid also here. As we can see above, we obtain the set of instructions by separating the processes **new** and **def** from the corresponding continuations. Thus we assume that the variables in the processes are renamed and that the scope of these bindings extends as far to the right as possible.

The encodings $[\![I]\!]$ and $[\![T]\!]$ of instructions and threads, respectively, into TyCO processes are given recursively by :

$$[\![\textsf{skip}]\!] \stackrel{\text{def}}{=} \textsf{skip} \qquad [\![x!l[\tilde{v}]]\!] \stackrel{\text{def}}{=} x!l[\tilde{v}] \qquad [\![x?M]\!] \stackrel{\text{def}}{=} x?[\![M]\!] \qquad [\![X[\tilde{v}]]\!] \stackrel{\text{def}}{=} X[\tilde{v}]$$
$$[\![\textsf{new}\ x :: T]\!] \stackrel{\text{def}}{=} \textsf{new}\ x\ [\![T]\!] \qquad [\![\textsf{def}\ D :: T]\!] \stackrel{\text{def}}{=} \textsf{def}\ [\![D]\!]\ \textsf{in}\ [\![T]\!]$$
$$[\![\bullet]\!] \stackrel{\text{def}}{=} \textsf{skip} \qquad [\![I :: T]\!] \stackrel{\text{def}}{=} I \mid [\![T]\!]$$

with the encoding for method collections, recursive definitions and thread abstractions defined as:

$$[\![M]\!] \stackrel{\text{def}}{=} \{l_1 = [\![A_1]\!], \cdots, l_k = [\![A_k]\!]\}$$
$$[\![X_1 = A_1\ \textsf{and} \cdots \textsf{and}\ X_k = A_k]\!] \stackrel{\text{def}}{=} X_1 = [\![A_1]\!]\ \textsf{and} \cdots \textsf{and}\ X_k = [\![A_k]\!]$$
$$[\![(\tilde{x})T]\!] \stackrel{\text{def}}{=} (\tilde{x})[\![T]\!]$$

3.7 The Machine

In the sequel we present a formal abstract machine specification for TyCO [42]. We start by introducing some preliminary definitions and the syntactic categories used and then proceed with the machine definition proper.

3.7.1 Syntactic Categories

Now we define the syntactic categories required by the abstract machine:

- *Environment* of a thread is represented as a map from variables to values, that is, $\text{VBind} = \text{Var} \mapsto \text{Val}$ and is ranged over by B.
- *Template Bindings* are represented as a map from template variables into pairs of variable bindings and abstractions, that is, $\text{TBind} = \text{TVar} \mapsto \text{VBind} \times \text{Abs}$, and are ranged over by K.

- *Messages* carry a label and a sequence of values and are kept in queues of $\text{QComm} = \text{Queue}(\text{Label} \times \text{Name}^*)$, ranged over by ms.

- *Objects* carry a method table reference and free variable bindings, and are kept in queues $\text{QMeth} = \text{Queue}(\text{VBind} \times \text{Meth})$, ranged over by Ms.

- *Channels* are elements of the set $\text{QBind} = \text{Name} \mapsto (\text{QComm} \cup \text{QMeth})$ of queues of messages or objects, ranged over by Q.

- *Run-Queue* is an element of the set $\text{RunQueue} = \text{Queue}(\text{VBind} \times \text{Thread})$ where threads and their environments are kept waiting for execution. It is named R.

- *Machine State*, denoted S, is a tuple in the set $\text{State} = \text{TBind} \times \text{VBind} \times \text{QBind} \times \text{Thread} \times \text{RunQueue}$.

3.7.2 Notation

The environments B are extended to the domain $\text{Var} \cup \text{Name}$, defined by:

$$B(v) = \begin{cases} v & \text{if } v \in \text{Name} \\ u & \text{if } v : u \in B \end{cases}$$

In the sequel we use the notation $B(\tilde{v})$ to designate the sequence:

$$B(v_1) \cdots B(v_k) \qquad \text{for } k > 0.$$

3.7.3 Reduction

The TyCO program, which is the initial thread fed to the abstract machine, must obey to the following restrictions:

1. the initial thread is closed for template and value variables;

2. all bound value and template variables in the initial thread are pairwise distinct;

3. the encoding in TyCO of the initial thread is typable with respect to the type-system described in section 2.3;

4. the abstract machine has an infinite heap. In particular it is always possible to create a fresh name.

The first three conditions can all be verified statically. The fourth condition allows us to abstract away from memory availability issues at this point. These are treated in detail in the virtual machine design and implementation (see chapter 4).

Given a thread T, the abstract machine starts computing with an empty run-queue, empty variable and template variable environments and no queues in channels. Thus the initial state is:

$$\emptyset, \emptyset, \emptyset, T, \bullet$$

The machine halts when no rule can be applied, that is when a state of the form below is reached.

$$\_, \_, \_, \bullet, \bullet$$

Rules transform states into states. In the sequel, comments accompanying rules indicate conditions that must be met for the reductions to be successful. These are mostly ensured by the assumptions we make about the initial thread.

The Sched rule detects the end of the current thread and starts executing the next one in the run-queue.

$$K, \_, Q, \bullet, (B, T) :: R \longrightarrow K, B, Q, T, R \tag{Sched}$$

The Skip rule just skips any **skip** instruction that appears in the current thread.

$$K, B, Q, \textbf{skip} :: T, R \longrightarrow K, B, Q, T, R \tag{Skip}$$

The New rule introduces a new local variable within a thread. For an instruction **new** x, the rule adds a binding from variable x to a fresh name a, and creates a new empty queue for a in Q. Execution continues with thread T.

$$\frac{a \notin \mathrm{dom}(Q)}{K, B, Q, \textbf{new}\ x :: T, R \longrightarrow K, B + \{x : a\}, Q + \{a : \bullet\}, T, R} \tag{New}$$

Comment. $x \notin \mathrm{dom}(B)$ by the variable renaming assumption (2).

The Def rule introduces new thread abstractions bound to template variables. Given the instruction **def** D, for each definition $X = A$ in D, the rule adds a bind $X : (B \upharpoonright \mathrm{fv}(A), A)$ to K. Execution continues with thread T.

$$K, B, Q, \textbf{def}\ D :: T, R \longrightarrow K + \{X : (B \upharpoonright \mathrm{fv}(A), A) \mid X = A \in D\}, B, Q, T, R \tag{Def}$$

Comment. $\mathrm{dom}(K) \cap \mathrm{dom}(D) = \emptyset$ by the variable renaming assumption (2).

The Inst rule instantiates a new thread from a template definition. Given an instruction $X[\tilde{v}]$ where the template X is defined as $X = (\tilde{x})T'$, this rule creates a new thread in the run-queue with the thread T' and the variable bindings $\{\tilde{x} : B(\tilde{v})\} + B'$. B' are the free variable bindings saved at the time the definition of X was found.

$$\frac{K(X) = (B', (\tilde{x})T')}{K, B, Q, X[\tilde{v}] :: T, R \longrightarrow K, B, Q, T, R :: (B' + \{\tilde{x} : B(\tilde{v})\}, T')} \quad (\textsc{Inst})$$

Comment. $X \in \mathrm{dom}(K)$ since programs are closed for template-variables (1). Similarly, $\{\tilde{v}\} \subseteq \mathrm{dom}(B)$.

The Red rules are responsible for contraction of redexes. The RedMsg (RedObj) rule takes a message (an object) from the current thread, a matching object (message) from the appropriate object (message) queue, and places the body of the invoked method plus the variable bindings in the run-queue for later execution. In both cases execution continues with the current thread.

$$\frac{B(x) = a \qquad Q(a) = (B', M) :: Ms \qquad M(l) = (\tilde{x})T'}{K, B, Q, x\,!\,l[\tilde{v}] :: T, R \longrightarrow K, B, Q + \{a : Ms\}, T, R :: (B' + \{\tilde{x} : B(\tilde{v})\}, T')} \quad (\textsc{RedMsg})$$

$$\frac{B(x) = a \qquad Q(a) = l[\tilde{v}] :: ms \qquad M(l) = (\tilde{x})T'}{K, B, Q, x?M :: T, R \longrightarrow K, B, Q + \{a : ms\}, T, R :: (B \restriction \mathrm{fv}(M) + \{\tilde{x} : \tilde{v}\}, T')} \quad (\textsc{RedObj})$$

Comment. $x \in \mathrm{dom}(B)$ since processes are closed (1). $l \in \mathrm{dom}(M)$ since programs are well typed (3). $\mathrm{length}(\tilde{x})$=$\mathrm{length}(\tilde{v})$ since programs are well typed (3). $a \in \mathrm{dom}(Q)$ since processes are closed and well-typed (1,3), and so the binding must have been placed by the New-rule. $\{\tilde{x}\} \cap \mathrm{dom}(B') = \emptyset$ since $\mathrm{dom}(B') = \mathrm{fv}((\tilde{x})T')$ and $\{\tilde{x}\} \cap \mathrm{fv}(T') = \emptyset$.

The Queue rules are responsible for the queuing of messages and objects. They are used whenever immediate reduction of messages or objects is not possible. The QueueMsg (QueueObj) rule takes a message (an object) and places it in the appropriate message (object) queue. Reduction will occur when a matching object (message) is later scheduled for execution. Execution of the current thread continues.

$$\frac{B(x) = a \qquad Q(a) = ms}{K, B, Q, x\,!\,l[\tilde{v}] :: T, R \longrightarrow K, B, Q + \{a : ms :: l[B(\tilde{v})]\}, T, R}$$

(QueueMsg)

$$\frac{B(x) = a \qquad Q(a) = Ms}{K, B, Q, x?M :: T, R \longrightarrow K, B, Q + \{a : Ms :: (B \restriction \mathrm{fv}(M), M)\}, T, R}$$

(QueueObj)

Comment. $x \in \mathrm{dom}(B)$ since processes are closed (1). $a \in \mathrm{dom}(Q)$ since processes are closed and well-typed (1,3), and so the binding must have been placed by the New-rule.

Definition. *We introduce the notation $\longrightarrow^*$ to denote a finite, possibly zero, number of (single step) abstract machine transitions.*

The abstract machine just presented grows from Turner's abstract machine for Pict [66], but modifies it in the following major ways:

1. objects are builtin and substitute input processes. They are more efficient than in the π-calculus [76] both in reduction and heap usage;
2. we use recursion instead of replication for persistence. This allows a cleaner design of the abstract machine – no need for distinct `?` and `?*` rules – and allows a more rational heap usage;
3. we introduce a new syntactic category – the *Thread*, that represents the basic schedulable and runnable block in the abstract machine. Threads are identified as bodies of template definitions or method implementations;
4. threads cannot be suspended. With this property, our objects are very akin to actors and provide a good model for object oriented concurrent languages [4, 5]. This choice, along with the previous item, also simplifies the treatment of local bindings, introduced with **new** statements, and the management of environments.

3.8 Machine Properties

The abstract machine we have just defined features important run-time properties:

1. the abstract machine is sound, that is, every state transition in the abstract machine can be viewed as a reduction or a congruence between its process counterpart in the base calculus [42];

2. at any time during a computation the queues associated with names are either empty or have only communications or have only method-closures [66];

3. for well-typed programs the abstract machine does not deadlock. This property is intimately linked to the ability of the type system to guarantee that no run-time protocol errors will occur [42];

4. the abstract machine is fair, that is, any given thread always gets executed a finite time slice after it was first placed in the run-queue.

3.8.1 The Abstract Machine is Sound

Taking $B = \{\tilde{x} : \tilde{v}\}$, and the translation $[\![T]\!] = P$ from threads into processes defined in section 3.6, $[\![T]\!]B$ becomes $P\{\tilde{v}/\tilde{x}\}$, where the substitution is defined in section 2.2.

The *Queue* associated with name a is encoded as sequences of messages and objects ending with the **skip** process.

$$\begin{aligned} [\![a : \bullet]\!] &\stackrel{\text{def}}{=} \textsf{\textbf{skip}} \\ [\![a : m :: ms]\!] &\stackrel{\text{def}}{=} a\,!\,m \mid [\![a : ms]\!] \\ [\![a : (B, M) :: Ms]\!] &\stackrel{\text{def}}{=} a?([\![M]\!]B) \mid [\![a : Ms]\!] \end{aligned}$$

where $[\![M]\!]$ is defined in section 3.6. The encoding $[\![Q]\!]$ being the encodings for the queues put in parallel.

$$[\![\{a_1 : q_1, \cdots, a_k : q_k\}]\!] \stackrel{\text{def}}{=} [\![a_1 : q_1]\!] \mid \cdots \mid [\![a_k : q_k]\!]$$

The encoding for *Template Bindings*, $[\![K]\!]$, is:

$$[\![\{X_1 : (B_1, A_1), \cdots, X_k : (B_k, A_k)\}]\!] \stackrel{\text{def}}{=} [\![X_1 = A_1 B_1 \textsf{\textbf{ and}} \cdots \textsf{\textbf{and }} X_k = A_k B_k]\!]$$

and the last encoding is defined in section 3.6.

The encoding $[\![R]\!]$ for the *Run-Queue* is:

$$\begin{aligned} [\![\bullet]\!] &\stackrel{\text{def}}{=} \textsf{\textbf{skip}} \\ [\![(B_1, T_1) :: \cdots :: (B_k, T_k)]\!] &\stackrel{\text{def}}{=} [\![T_1]\!]B_1 \mid \cdots \mid [\![T_k]\!]B_k \end{aligned}$$

Finally the encoding for a machine state, $[\![K, B, Q, T, R]\!]$ is:

$$[\![K, B, Q, T, R]\!] \stackrel{\text{def}}{=} \textsf{\textbf{new}}\ \tilde{a}\ \textsf{\textbf{def}}\ [\![K]\!]\ \textsf{\textbf{in}}\ [\![T]\!]B \mid [\![Q]\!] \mid [\![R]\!]$$

where $\{\tilde{a}\} = \mathrm{dom}(Q)$.

Now we are ready to prove the soundness of the abstract machine relative to the TyCO calculus. The soundness property states that every state transition in the abstract machine can be viewed as a reduction or a congruence between their process encodings in TyCO.

Theorem (Soundness). *Let S and S' be abstract machine states. If $S \longrightarrow S'$, then $[\![S]\!] \equiv [\![S']\!]$ or $[\![S]\!] \to [\![S']\!]$.*

Proof: We enumerate all the rules for the machine and check the validity of the transitions. The proof uses both congruence and reduction rules from section 2.2.

(Sched) we show that:

$$[\![K, B, Q, \bullet, (B', T) :: R]\!] \equiv [\![K, B', Q, T, R]\!]$$

We know that,

$$\begin{aligned}
[\![K, B, Q, \bullet, (B', T) :: R]\!] &\stackrel{\text{def}}{=} \\
\textsf{\textbf{new}}\ \tilde{a}\ \textsf{\textbf{def}}\ [\![K]\!]\ \textsf{\textbf{in}}\ \textsf{\textbf{skip}}B \mid [\![Q]\!] \mid [\![(B', T) :: R]\!] &\equiv \stackrel{\text{def}}{=} \\
\textsf{\textbf{new}}\ \tilde{a}\ \textsf{\textbf{def}}\ [\![K]\!]\ \textsf{\textbf{in}}\ [\![T]\!]B' \mid [\![Q]\!] \mid [\![R]\!] &\stackrel{\text{def}}{=} \\
[\![K, B', Q, T, R]\!] &
\end{aligned}$$

Since, $\textsf{\textbf{skip}}B = \textsf{\textbf{skip}}$ and applying the congruences $\textsf{\textbf{skip}} \mid P \equiv P$ and $P \mid Q \equiv Q \mid P$ in rule Group.

(Skip) we show that:

$$[\![K, B, Q, \textsf{\textbf{skip}} :: T, R]\!] \equiv [\![K, B, Q, T, R]\!]$$

We know that,

$$\begin{aligned}
\llbracket K, B, Q, \textsf{skip} :: T, R \rrbracket &\stackrel{\text{def}}{=} \\
\textsf{new}\ \tilde{a}\ \textsf{def}\ \llbracket K \rrbracket\ \textsf{in}\ (\textsf{skip} \mid \llbracket T \rrbracket)B \mid \llbracket Q \rrbracket \mid \llbracket R \rrbracket &\stackrel{\text{def}}{=} \\
\textsf{new}\ \tilde{a}\ \textsf{def}\ \llbracket K \rrbracket\ \textsf{in}\ \textsf{skip}B \mid \llbracket T \rrbracket B \mid \llbracket Q \rrbracket \mid \llbracket R \rrbracket &\equiv \\
\textsf{new}\ \tilde{a}\ \textsf{def}\ \llbracket K \rrbracket\ \textsf{in}\ \llbracket T \rrbracket B \mid \llbracket Q \rrbracket \mid \llbracket R \rrbracket &\stackrel{\text{def}}{=} \\
\llbracket K, B, Q, T, R \rrbracket &
\end{aligned}$$

again since, $\textsf{skip}B = \textsf{skip}$ and applying the congruence $\textsf{skip} \mid P \equiv P$ from rule Group.

(New) we show that:

$$\llbracket K, B, Q, \textsf{new}\ x :: T, R \rrbracket \equiv \llbracket K, B + \{x : a\}, Q + \{a : \bullet\}, T, R \rrbracket \qquad a\ \ \textit{fresh}$$

We know that:

$$\begin{aligned}
\llbracket K, B, Q, \textsf{new}\ x :: T, R \rrbracket &\stackrel{\text{def}}{=} \\
\textsf{new}\ \tilde{a}\ \textsf{def}\ \llbracket K \rrbracket\ \textsf{in}\ (\textsf{new}\ x\ \llbracket T \rrbracket)B \mid \llbracket Q \rrbracket \mid \llbracket R \rrbracket &\equiv_\alpha
\end{aligned}$$

we choose $a \notin \mathrm{fv}(\llbracket K \rrbracket)$ and replace x for a in $\llbracket T \rrbracket$:

$$\textsf{new}\ \tilde{a}\ \textsf{def}\ \llbracket K \rrbracket\ \textsf{in}\ (\textsf{new}\ a\ \llbracket T \rrbracket\{a/x\})B \mid \llbracket Q \rrbracket \mid \llbracket R \rrbracket \equiv$$

and then we use the congruence $\textsf{def}\ D\ \textsf{in}\ \textsf{new}\ x\ T \equiv \textsf{new}\ x\ \textsf{def}\ D\ \textsf{in}\ T$ in rule Def to get:

$$\begin{aligned}
\textsf{new}\ \tilde{a}a\ \textsf{def}\ \llbracket K \rrbracket\ \textsf{in}\ (\{a/x\}\llbracket T \rrbracket)B \mid \llbracket Q \rrbracket \mid \llbracket R \rrbracket &\stackrel{\text{def}}{=} \\
\textsf{new}\ \tilde{a}a\ \textsf{def}\ \llbracket K \rrbracket\ \textsf{in}\ \llbracket T \rrbracket(B + \{x : a\}) \mid \llbracket Q \rrbracket \mid \llbracket R \rrbracket &\equiv \\
\textsf{new}\ \tilde{a}a\ \textsf{def}\ \llbracket K \rrbracket\ \textsf{in}\ \llbracket T \rrbracket(B + \{x : a\}) \mid \textsf{skip} \mid \llbracket Q \rrbracket \mid \llbracket R \rrbracket &\stackrel{\text{def}}{=}
\end{aligned}$$

and finally, using the congruence $\textsf{skip} \mid P \equiv P$ in rule Group, and the fact that $\llbracket a : \bullet \rrbracket \equiv \textsf{skip}$, we get:

$$\llbracket K, B + \{x : a\}, Q + \{a : \bullet\}, T, R \rrbracket$$

(Def) we show that:

$$\llbracket K, B, Q, \textsf{def}\ D :: T, R \rrbracket \equiv \llbracket K + K', B, Q, T, R \rrbracket$$

where $K' = \{X : (B', A) \mid X = A \in D\}$. We know that:

$$
\begin{aligned}
\llbracket K, B, Q, \textsf{\textbf{def}}\ D :: T, R \rrbracket &\stackrel{\text{def}}{=} \\
\textsf{\textbf{new}}\ \tilde{a}\ \textsf{\textbf{def}}\ \llbracket K \rrbracket\ \textsf{\textbf{in}}\ \llbracket \textsf{\textbf{def}}\ D :: T \rrbracket B \mid \llbracket Q \rrbracket \mid \llbracket R \rrbracket &\stackrel{\text{def}}{=} \\
\textsf{\textbf{new}}\ \tilde{a}\ \textsf{\textbf{def}}\ \llbracket K \rrbracket\ \textsf{\textbf{in}}\ (\textsf{\textbf{def}}\ \llbracket D \rrbracket\ \textsf{\textbf{in}}\ \llbracket T \rrbracket) B \mid \llbracket Q \rrbracket \mid \llbracket R \rrbracket &\stackrel{\text{def}}{=} \\
\textsf{\textbf{new}}\ \tilde{a}\ \textsf{\textbf{def}}\ \llbracket K \rrbracket\ \textsf{\textbf{in}}\ \textsf{\textbf{def}}\ \llbracket D \rrbracket B\ \textsf{\textbf{in}}\ \llbracket T \rrbracket B \mid \llbracket Q \rrbracket \mid \llbracket R \rrbracket &
\end{aligned}
$$

Since $\mathrm{dom}(D) \cap (\mathrm{ft}(Q) \cup \mathrm{ft}(R)) = \emptyset$ (variable renaming assumption) we can extend the scope of the definition $\llbracket D \rrbracket$ to the right. Using the same assumption, we know that $\mathrm{dom}(D) \cap \mathrm{ft}(K) = \emptyset$ and so we use the congruence **def** D **in def** D' **in** $P \equiv$ **def** D **and** D' **in** P in rule DEF to compose the two nested definitions into one.

$$
\begin{aligned}
\textsf{\textbf{new}}\ \tilde{a}\ \textsf{\textbf{def}}\ \llbracket K \rrbracket\ \textsf{\textbf{and}}\ \llbracket D \rrbracket B\ \textsf{\textbf{in}}\ \llbracket T \rrbracket B \mid \llbracket Q \rrbracket \mid \llbracket R \rrbracket &\stackrel{\text{def}}{=} \\
\textsf{\textbf{new}}\ \tilde{a}\ \textsf{\textbf{def}}\ \llbracket K \rrbracket\ \textsf{\textbf{and}}\ \llbracket K' \rrbracket\ \textsf{\textbf{in}}\ \llbracket T \rrbracket B \mid \llbracket Q \rrbracket \mid \llbracket R \rrbracket &\stackrel{\text{def}}{=} \\
\llbracket K + K', B, Q, T, R \rrbracket &
\end{aligned}
$$

(INST) we show that:

$$
\llbracket K, B, Q, X[\tilde{v}] :: T, R \rrbracket \longrightarrow \llbracket K, B, Q, T, R :: (B' + \{\tilde{x} : B(\tilde{v})\}, T') \rrbracket
$$

where $K(X) = (B', (\tilde{x})T')$. We know that:

$$
\begin{aligned}
\llbracket K, B, Q, X[\tilde{v}] :: T, R \rrbracket &\stackrel{\text{def}}{=} \\
\textsf{\textbf{new}}\ \tilde{a}\ \textsf{\textbf{def}}\ \llbracket K \rrbracket\ \textsf{\textbf{in}}\ \llbracket X[\tilde{v}] :: T \rrbracket B \mid \llbracket Q \rrbracket \mid \llbracket R \rrbracket &\stackrel{\text{def}}{=} \\
\textsf{\textbf{new}}\ \tilde{a}\ \textsf{\textbf{def}}\ \llbracket K \rrbracket\ \textsf{\textbf{in}}\ X[B(\tilde{v})] \mid \llbracket T \rrbracket B \mid \llbracket Q \rrbracket \mid \llbracket R \rrbracket &\equiv \\
\textsf{\textbf{new}}\ \tilde{a}\ \textsf{\textbf{def}}\ \llbracket K \rrbracket\ \textsf{\textbf{in}}\ X[B(\tilde{v})] \mid \llbracket T \rrbracket B \mid \llbracket Q \rrbracket \mid \llbracket R \rrbracket &\rightarrow
\end{aligned}
$$

applying rule INST for reduction from section 2.2 we get:

$$
\begin{aligned}
\textsf{\textbf{new}}\ \tilde{a}\ \textsf{\textbf{def}}\ \llbracket K \rrbracket\ \textsf{\textbf{in}}\ ((\tilde{x}) \llbracket T' \rrbracket)[B(\tilde{v})] \mid \llbracket T \rrbracket B \mid \llbracket Q \rrbracket \mid \llbracket R \rrbracket &\equiv \\
\textsf{\textbf{new}}\ \tilde{a}\ \textsf{\textbf{def}}\ \llbracket K \rrbracket\ \textsf{\textbf{in}}\ \llbracket T' \rrbracket (B' - \{\tilde{x}\}) \{B(\tilde{v}) / \tilde{x}\} \mid \llbracket T \rrbracket B \mid \llbracket Q \rrbracket \mid \llbracket R \rrbracket &\stackrel{\text{def}}{=} \\
\textsf{\textbf{new}}\ \tilde{a}\ \textsf{\textbf{def}}\ \llbracket K \rrbracket\ \textsf{\textbf{in}}\ \llbracket T' \rrbracket (B' + \{\tilde{x} : B(\tilde{v})\}) \mid \llbracket T \rrbracket B \mid \llbracket Q \rrbracket \mid \llbracket R \rrbracket &\equiv \stackrel{\text{def}}{=} \\
\llbracket K, B, Q, T, R :: (B' + \{\tilde{x} : B(\tilde{v})\}, T') \rrbracket &
\end{aligned}
$$

Besides INST, we used the definition of substitution and, in the last step, the congruence $P \mid Q \equiv Q \mid P$ in rule GROUP and the encoding of the run-queue.

(RedMsg) we show that:

$$[\![K, B, Q, x\,!\,l[\tilde{v}]::T, R]\!] \longrightarrow [\![K, B, Q + \{a : Ms\}, T, R::(B' + \{\tilde{x} : B(\tilde{v})\}, T')]\!]$$

where $B(x) = a$, $Q(a) = (B', M)::Ms$, and $M(l) = (\tilde{x})T'$. We know that:

$$\begin{aligned}
&[\![K, B, Q, x\,!\,l[\tilde{v}]::T, R]\!] \overset{\text{def}}{=} \\
&\textsf{new}\ \tilde{a}\ \textsf{def}\ [\![K]\!]\ \textsf{in}\ [\![x\,!\,l[\tilde{v}]::T]\!]B \mid [\![Q]\!] \mid [\![R]\!] \overset{\text{def}}{=} \\
&\textsf{new}\ \tilde{a}\ \textsf{def}\ [\![K]\!]\ \textsf{in}\ a\,!\,l[B(\tilde{v})] \mid [\![T]\!]B \mid [\![Q + \{a : ms\}]\!] \mid a?[\![M]\!]B \mid [\![R]\!] \equiv\!\rightarrow
\end{aligned}$$

here we apply rule Comm from section 2.2 and get:

$$\begin{aligned}
&\textsf{new}\ \tilde{a}\ \textsf{def}\ [\![K]\!]\ \textsf{in}\ [\![T]\!]B \mid [\![Q + \{a : Ms\}]\!] \mid (([\![M]\!](l))B')[B(\tilde{v})] \mid [\![R]\!] \overset{\text{def}}{=} \\
&\textsf{new}\ \tilde{a}\ \textsf{def}\ [\![K]\!]\ \textsf{in}\ [\![T]\!]B \mid [\![Q + \{a : Ms\}]\!] \mid ((\tilde{x})[\![T']\!])B'[B(\tilde{v})] \mid [\![R]\!] \overset{\text{def}}{=} \\
&\textsf{new}\ \tilde{a}\ \textsf{def}\ [\![K]\!]\ \textsf{in}\ [\![T]\!]B \mid [\![Q + \{a : Ms\}]\!] \mid ((\tilde{x})([\![T']\!](B' - \{\tilde{x}\})))[B(\tilde{v})] \mid [\![R]\!] \equiv \\
&\textsf{new}\ \tilde{a}\ \textsf{def}\ [\![K]\!]\ \textsf{in}\ [\![T]\!]B \mid [\![Q + \{a : Ms\}]\!] \mid [\![T']\!](B' - \{\tilde{x}\})\{B(\tilde{v})/\tilde{x}\} \mid [\![R]\!] \overset{\text{def}}{=} \equiv \\
&\textsf{new}\ \tilde{a}\ \textsf{def}\ [\![K]\!]\ \textsf{in}\ [\![T]\!]B \mid [\![Q + \{a : Ms\}]\!] \mid [\![T']\!](B' + \{\tilde{x} : B(\tilde{v})\}) \mid [\![R]\!] \overset{\text{def}}{=} \\
&[\![K, B, Q + \{a : Ms\}, T, R::(B' + \{\tilde{x} : B(\tilde{v})\}, T')]\!]
\end{aligned}$$

Besides Comm we used the definition of substitution and the congruence $P \mid Q \equiv Q \mid P$ from rule Group.

(RedObj) we show that:

$$[\![K, B, Q, x?M::T, R]\!] \longrightarrow [\![K, B, Q + \{a : ms\}, T, R::(B + \{\tilde{x} : \tilde{v}\}, T')]\!]$$

where $B(x) = a$, $Q(a) = l[\tilde{v}]::ms$, and $M(l) = (\tilde{x})T'$. We know that:

$$\begin{aligned}
&[\![K, B, Q, x?M::T, R]\!] \overset{\text{def}}{=} \\
&\textsf{new}\ \tilde{a}\ \textsf{def}\ [\![K]\!]\ \textsf{in}\ [\![x?M::T]\!]B \mid [\![Q]\!] \mid [\![R]\!] \overset{\text{def}}{=} \\
&\textsf{new}\ \tilde{a}\ \textsf{def}\ [\![K]\!]\ \textsf{in}\ a?[\![M]\!]B \mid [\![T]\!]B \mid [\![Q + \{a : ms\}]\!] \mid a\,!\,l[\tilde{v}] \mid [\![R]\!] \equiv\!\rightarrow
\end{aligned}$$

again we apply rule Comm from section 2.2 and get:

$$\begin{aligned}
&\textsf{new}\ \tilde{a}\ \textsf{def}\ [\![K]\!]\ \textsf{in}\ [\![T]\!]B \mid [\![Q + \{a : ms\}]\!] \mid [\![M]\!](l)[\tilde{v}] \mid [\![R]\!] \overset{\text{def}}{=} \\
&\textsf{new}\ \tilde{a}\ \textsf{def}\ [\![K]\!]\ \textsf{in}\ [\![T]\!]B \mid [\![Q + \{a : ms\}]\!] \mid ((\tilde{x})[\![T']\!])B[\tilde{v}] \mid [\![R]\!] \overset{\text{def}}{=} \\
&\textsf{new}\ \tilde{a}\ \textsf{def}\ [\![K]\!]\ \textsf{in}\ [\![T]\!]B \mid [\![Q + \{a : ms\}]\!] \mid ((\tilde{x})([\![T']\!](B - \{\tilde{x}\})))[\tilde{v}] \mid [\![R]\!] \equiv \\
&\textsf{new}\ \tilde{a}\ \textsf{def}\ [\![K]\!]\ \textsf{in}\ [\![T]\!]B \mid [\![Q + \{a : ms\}]\!] \mid [\![T']\!](B - \{\tilde{x}\})\{\tilde{v}/\tilde{x}\} \mid [\![R]\!] \overset{\text{def}}{=} \\
&\textsf{new}\ \tilde{a}\ \textsf{def}\ [\![K]\!]\ \textsf{in}\ [\![T]\!]B \mid [\![Q + \{a : ms\}]\!] \mid [\![T']\!](B' + \{\tilde{x} : \tilde{v}\}) \mid [\![R]\!] \overset{\text{def}}{=} \\
&[\![K, B, Q + \{a : ms\}, T, R::(B + \{\tilde{x} : \tilde{v}\}, T')]\!]
\end{aligned}$$

Besides COMM we used the definition of substitution and the congruence $P \mid Q \equiv Q \mid P$ from rule GROUP.

(QUEUEMSG) we show that:

$$[\![K, B, Q, x \,!\, l[\tilde{v}] :: T, R]\!] \longrightarrow [\![K, B, Q + \{a : ms :: (l[B(\tilde{v})])\}, T, R]\!]$$

where $B(x) = a$ and $Q(a) = ms$. We know that:

$$\begin{aligned}
[\![K, B, Q, x \,!\, l[\tilde{v}] :: T, R]\!] &\stackrel{\text{def}}{=} \\
\textsf{\textbf{new}}\ \tilde{a}\ \textsf{\textbf{def}}\ [\![K]\!]\ \textsf{\textbf{in}}\ [\![x \,!\, l[\tilde{v}] :: T]\!] B \mid [\![Q]\!] \mid [\![R]\!] &\stackrel{\text{def}}{=} \\
\textsf{\textbf{new}}\ \tilde{a}\ \textsf{\textbf{def}}\ [\![K]\!]\ \textsf{\textbf{in}}\ a \,!\, l[B(\tilde{v})] \mid [\![T]\!] B \mid [\![Q]\!] \mid [\![R]\!] &\equiv \\
\textsf{\textbf{new}}\ \tilde{a}\ \textsf{\textbf{def}}\ [\![K]\!]\ \textsf{\textbf{in}}\ [\![T]\!] B \mid [\![Q]\!] \mid a \,!\, l[B(\tilde{v})] \mid [\![R]\!] &\stackrel{\text{def}}{=} \\
[\![K, B, Q + \{a : ms :: l[B(\tilde{v})]\}, T, R]\!] &
\end{aligned}$$

using the encodings for the channels and the congruence $P \mid Q \equiv Q \mid P$ from rule GROUP.

(QUEUEOBJ) we show that:

$$[\![K, B, Q, x?M :: T, R]\!] \longrightarrow [\![K, B, Q + \{a : Ms :: (B \restriction \mathrm{fv}(M), M)\}, T, R]\!]$$

where $B(x) = a$ and $Q(a) = Ms$. We know that:

$$\begin{aligned}
[\![K, B, Q, x?M :: T, R]\!] &\stackrel{\text{def}}{=} \\
\textsf{\textbf{new}}\ \tilde{a}\ \textsf{\textbf{def}}\ [\![K]\!]\ \textsf{\textbf{in}}\ [\![x?M :: T]\!] B \mid [\![Q]\!] \mid [\![R]\!] &\stackrel{\text{def}}{=} \\
\textsf{\textbf{new}}\ \tilde{a}\ \textsf{\textbf{def}}\ [\![K]\!]\ \textsf{\textbf{in}}\ a?[\![M]\!] B \mid [\![T]\!] B \mid [\![Q]\!] \mid [\![R]\!] &\equiv \\
\textsf{\textbf{new}}\ \tilde{a}\ \textsf{\textbf{def}}\ [\![K]\!]\ \textsf{\textbf{in}}\ [\![T]\!] B \mid [\![Q]\!] \mid a?[\![M]\!](B \restriction \mathrm{fv}(M)) \mid [\![R]\!] &\stackrel{\text{def}}{=} \\
[\![K, B, Q + \{a : Ms :: (B \restriction \mathrm{fv}(M), M)\}, T, R]\!] &
\end{aligned}$$

using the encodings for the channels and the congruence $P \mid Q \equiv Q \mid P$ from rule GROUP. □

3.8.2 The Abstract Machine is not Complete

The abstract machine is *not* complete with respect to the semantics of TyCO. This essentially results from two restrictions we imposed on the behavior of processes. The first such restriction is easily observed with the following example in TyCO:

$$a\,!\,l[\tilde{v}] \mid a?\{l = (\tilde{x})P\} \mid a?\{l = (\tilde{x})Q\}$$

which may reduce to

$$P\{\tilde{v}/\tilde{x}\} \mid a?\{l = (\tilde{x})Q\} \qquad \text{or} \qquad Q\{\tilde{v}/\tilde{x}\} \mid a?\{l = (\tilde{x})P\} \quad .$$

The same sequence in the abstract machine constitutes a thread and is executed always left to right, deterministically. Thus the only possible outcome of the above example when it is run by the machine is equivalent to the composition:

$$P\{\tilde{v}/\tilde{x}\} \mid a?\{l = (\tilde{x})Q\}$$

In our design we chose to merge processes at the same nesting level into threads, run sequentially and deterministically. Thus we shift the concurrent unit from the process, in the TyCO calculus, to method or template bodies – threads – in the abstract machine. Concurrency exists between threads sharing resources, more specifically, communication channels. Mutual exclusive access to these resources introduces non-determinism in a way we think more natural than explicitly supporting it in reduction rules (for example, in the form of a CONC rule).

The use of a run-queue to hold newly generated threads serializes their execution in a single-threaded implementation of the abstract machine. However in a multi-threaded implementation the run-queue may be transformed in a pool of threads that are scheduled among the computational agents of the machine. This introduces concurrency and additional non-determinism in the computation.

The second important restriction is the fact that our processes are closed for value and template variables. This restricts their behavior in a fundamental way: abstract machine programs are closed entities incapable to interact with external processes. In chapter 8 we address this problem by extending TyCO in such a way that variables from a program may be exported-to/imported-from other programs. Under this extended model the above restriction can be relaxed and may even be eliminated depending on the semantics attached to the export/import operations. For example, if such operations are lazy then it is feasible for a running program to have free variables that may be linked to external interfaces on demand or may in fact never be linked if an implementation of such an interface is not provided.

3.8.3 The Communication Queue Invariant

The abstract machine preserves the invariant that at any time during a computation the queues associated with names have either communications or method-closures (or are empty). Formally, we have the following lemma:

Lemma (Communication Queue Invariant). *Let S_0 be the initial state of the abstract machine. If $S_0 \longrightarrow^* \_, \_, Q, \_, \_$ then $\forall_{a \in \mathrm{dom}(Q)} Q(a) \in$ QCOMM or $Q(a) \in$ QMETH.*

Proof Outline: Induction on the length of the reduction performed by the abstract machine.

(Base) We start with the initial state of the abstract machine:

$$S_0 = \emptyset, \emptyset, \emptyset, T, \bullet$$

where T is the initial thread. There are no queues in this state. The invariant holds trivially.

(Step) Now assume that we are in a generic state: K, B, Q, T, R for which the invariant holds. We must check that after applying any of the reduction rules to such a state the invariant still holds.

The rules SCHED, SKIP, DEF and INST do not affect the queues in Q, so after an application the invariant will still hold. The rule NEW creates a new name and an empty queue with it in Q. Since the empty queue is both in QComm and QMeth the invariant is preserved.

The RED and QUEUE rules affect the queues in a more complex way. We enumerate the possibilities:

- REDMSG is used whenever the queue being inspected is in QMeth . It removes one of the method-closures from the queue. The remainder of the queue (eventually empty) is still in QMeth;
- for REDOBJ the argument is similar but with the queue in QComm.

Thus, the RED rules preserve the invariant. Similarly, for the QUEUE rules we have:

- QUEUEMSG is used whenever the queue being inspected is in QComm. It adds another communication to the end of the queue which obviously remains in QComm;
- for QUEUEOBJ the argument is similar but with the queue in QMeth.

It follows that the QUEUE rules also preserve the invariant. □

3.8.4 Absence of Deadlocks

The abstract machine does not deadlock for typable programs. We first introduce the definition of a deadlock state:

Definition. *S is a* deadlock state *if $S \neq \_, \_, \_, \bullet, \bullet$ and $S \not\longrightarrow$. Let S_0 be the initial state of the machine. We say that the* abstract machine deadlocks *if $S_0 \longrightarrow^* S$ and S is a deadlock state.*

Lemma (Closed Process Invariant). *If $S_0 \longrightarrow^* K, B, Q, T, R$, then $\mathrm{fv}(T) \subseteq \mathrm{dom}(B)$.*

Proof. We proceed by induction on the number of abstract machine transitions.

(Base) $S_0 = \emptyset, \emptyset, \emptyset, T, \bullet$. By assumption 1 for the initial thread of the abstract machine (section 3.7.3) we know that $\mathrm{fv}(T) = \emptyset$. The property holds trivially.

(Step) Now assume $S_0 \longrightarrow^* S = K, B, Q, T', R$ and that $\mathrm{fv}(T') \subseteq \mathrm{dom}(B)$. If $T' = \bullet$ and $R = (B', T'') :: R'$ then we apply rule SCHED and get the state $S' = K, B', Q, T'', R'$. Now, by definition, the environment B' of the thread T'' has domain $\mathrm{dom}(B') = \mathrm{fv}(T'')$, and so the property holds. If $R = \bullet$ the machine halts and this preserves the property. Now assume $T' \neq \bullet$, then we do a case by case analysis on the head of $T' = I :: T''$.

($I = \textsf{skip}$). We apply rule SKIP which does not affect B. Since $\mathrm{fv}(\textsf{skip} :: T'') = \mathrm{fv}(T'')$ we get by hypothesis that $\mathrm{fv}(T'') \subseteq \mathrm{dom}(B)$.

($I = \textsf{new}\ x$). Since we can always get a fresh name a, by the infinite heap abstract machine assumption (section 3.7.3), we get $S' = K, B', Q', T'', R$ by applying rule NEW, with $B' = B + \{x : a\}$ and $Q' = Q + \{a : \bullet\}$. Now, $\mathrm{fv}(\textsf{new}\ x :: T'') \cup \{x\} = \mathrm{fv}(T'')$

and $\mathrm{dom}(B') = \mathrm{dom}(B) \cup \{x\}$ by the variable renaming assumption of the abstract machine. It follows from $\mathrm{fv}(T') \subseteq \mathrm{dom}(B)$ that $\mathrm{fv}(T'') \subseteq \mathrm{dom}(B')$.

($I =$ **def** D). We apply the rule DEF which does affect B. Since $\mathrm{fv}(\textbf{def}\ D :: T'') = \mathrm{fv}(D) \cup \mathrm{fv}(T'')$. It follows that, using the hypothesis, $\mathrm{fv}(T'') \subseteq \mathrm{fv}(T') \subseteq \mathrm{dom}(B)$.

($I = X[\tilde{v}]$). The INST rule is applied. It does not affect the environment B. Moreover, $\mathrm{fv}(T'') \subseteq \mathrm{fv}(X[\tilde{v}] :: T'')$ and therefore, using the hypothesis, $\mathrm{fv}(T'') \subseteq \mathrm{dom}(B)$.

($I = x\,!\,l[\tilde{v}]$). The QUEUEMSG and REDMSG rules do not change the B environment. Moreover, $\mathrm{fv}(T'') \subseteq \mathrm{fv}(x\,!\,l[\tilde{v}] :: T'')$ and hence, using the hypothesis, $\mathrm{fv}(T'') \subseteq \mathrm{dom}(B)$.

($I = x?M$). A similar argument to the case for the message but for rules QUEUEOBJ and REDOBJ allows us to conclude that $\mathrm{fv}(T'') \subseteq \mathrm{dom}(B)$. □

Theorem (Absence of Deadlocks). *Let T be the thread in the initial state of the abstract machine. If the process $[\![T]\!]$ is typable (in the type system described in section 2.3), then the abstract machine does not deadlock.*

Proof. By definition the abstract machine deadlocks if $S_0 \longrightarrow^* S$, and S is a deadlock state. We proceed by induction on the number of abstract machine transitions, to show that S is not a deadlock state.

(Base) $S_0 = \emptyset, \emptyset, \emptyset, T, \bullet$ is not a deadlock state. We observe the shape of T. If $T = \bullet$, then by definition it is not a deadlock state. Now assume $T = I :: T'$. We check all the possibilities for I and show that either it is possible to make a transition or that I cannot occur at the head of T.

($I =$ **skip**). We apply the rule SKIP.

($I =$ **new** x). Since we can always create fresh names in the heap, by one of the abstract machine assumptions, we apply rule NEW.

($I =$ **def** D). We apply rule DEF.

($I = X[\tilde{v}]$). This instruction cannot occur at the head of T since the process $[\![S_0]\!] = X[\tilde{v}] \mid [\![T']\!]$ is not closed for template variables (X is free). This contradicts one of the abstract machine assumptions.

($I = x\,!\,l[\tilde{v}]$). This instruction cannot occur at the head of T since the process $[\![S_0]\!] = x\,!\,l[\tilde{v}] \mid [\![T']\!]$ is not closed for value variables (for example, x is free). Again, this contradicts one of the abstract machine assumptions.

($I = x?M$). This instruction cannot occur at the head of T since the process $[\![S_0]\!] = x?[\![M]\!] \mid [\![T']\!]$ is not closed for value variables (for example, x is free). This contradicts one of the abstract machine assumptions.

Thus, since the initial thread complies with the abstract machine assumptions by hypothesis, only one of the first three cases can occur and therefore a transition is always possible.

(Step) Assume $S_0 \longrightarrow^* S = K, B, Q, T', R$, and S is not a deadlock state (that is, $S = \_, \_, \_, \bullet, \bullet$ or there is a state S' such that $S \longrightarrow S'$).

If $T' = \bullet$, we have two possibilities. If $R = \bullet$, then S is not a deadlock state by definition. On the other hand, if $R = (B', T'') :: R'$ we may apply the rule SCHED and hence S is not a deadlock state.

Now suppose that $T' \neq \bullet$ then, we have to provide a state S' such that $S \longrightarrow S'$. Such a state exists if the pre-conditions for the rules in the abstract machine always hold. We proceed by a case analysis on the head I of the thread $T' = I :: T''$ and use the assumptions for the initial thread from section 3.7.3 and the typing rules from section 2.3:

($I =$ **skip**). We apply the rule SKIP and hence S is not a deadlock state.

($I =$ **new** x). From one of the abstract machine assumptions we may always create fresh names in the heap. Thus we may apply rule NEW and hence S is not a deadlock state.

($I =$ **def** D). We may apply rule DEF and thus S is not a deadlock state.

($I = X[\tilde{v}]$). We must prove that (1) $X \in \mathrm{dom}(K)$, (2) $\tilde{v} \in \mathrm{dom}(B)$ and (3) $|\tilde{x}| = |\tilde{v}|$.

(1) We know that the process $[\![S]\!] =$ **new** $\tilde{a}$ **def** $[\![K]\!]$ **in** $X[B(\tilde{v})] \mid [\![T']\!] B \mid [\![Q]\!] \mid [\![R]\!]$ is closed for template variables from the assumption 1 for the initial thread of the abstract machine (section 3.7.3). This implies that X must be bound in $[\![K]\!]$. Thus $X \in \mathrm{dom}(K)$. Let $\mathrm{dom}(K) = \{X_1, \ldots, X_n\}$ and $X = X_i$ $(1 \le i \le n)$.

(2) That $\tilde{v} \in \mathrm{dom}(B)$ follows immediately from the closed process invariant. In fact, $\mathrm{fv}(X[\tilde{v}] :: T'') = \{\tilde{v}\} \cup \mathrm{fv}(T'') \subseteq \mathrm{dom}(B)$ from the invariant. In particular $\{\tilde{v}\} \subseteq \mathrm{dom}(B)$.

(3) We start with the state S_0. By induction hypothesis, $S_0 \longrightarrow^* S$. It follows, from the soundness theorem, that $[\![S_0]\!] \rightarrow^* [\![S]\!]$. Now, by one of the abstract machine assumptions we know that $[\![S_0]\!]$ is typable. Thus, using the subject reduction theorem

for the type system, we can conclude that $[\![S]\!]$ is also typable.

Since $[\![S]\!]$ is typable, there is a deduction $\Delta, \Gamma \vdash [\![S]\!]$. Such a deduction is a tree of formulae whose leaves are the axioms for the type-system and the root is $\Delta, \Gamma \vdash [\![S]\!]$. Each formula is deduced from those immediately above it by a rule of the type system. Given the structure of $[\![S]\!]$ it follows that, at some point in this tree rule DEF is applied as follows:

$$\dfrac{\cdots \quad \dfrac{\tilde{x}_i : \tilde{\alpha}_i \vdash P_i}{\vdash (\tilde{x}_i)P_i : \tilde{\alpha}_i}\,\textsc{Abs} \quad \cdots \qquad \dfrac{\dfrac{\dfrac{\vdash X_i : \tilde{\alpha}_i}{}\,\textsc{Var}}{\vdash X[B(\tilde{v})]}\,\textsc{App} \qquad \dfrac{}{\vdash [\![T']\!] B \mid \ldots}}{\vdash X[B(\tilde{v})] \mid [\![T']\!] B \mid [\![Q]\!] \mid [\![R]\!]}\,\textsc{Par}}{\Delta \setminus X, \Gamma \vdash \textsf{def}\ [\![K]\!]\ \textsf{in}\ X[B(\tilde{v})] \mid [\![T']\!] B \mid [\![Q]\!] \mid [\![R]\!]}\,\textsc{Def}$$

Finally, the side condition of the rule APP and the rule ABS of the type system guarantee that the application $((\tilde{x})P)[B(\tilde{v})]$ is well defined. From APP follows that $\Gamma(B(\tilde{v})) \approx \tilde{\alpha}_i$ and from ABS that $\Gamma(\tilde{x}_i) = \tilde{\alpha}_i$. Therefore $|\tilde{x}_i| = |B(\tilde{v})|$. Thus, S is not a deadlock state.

$(I = x\,!\,l[\tilde{v}])$. We must prove that one of the two pre-conditions holds:

1. $x \in \mathrm{dom}(B)$, $a \in \mathrm{dom}(Q)$, $Q(a) \in \mathrm{QMeth}$ and $l \in \mathrm{dom}(M)$, where (B', M) is the head of $Q(a)$;
2. $x \in \mathrm{dom}(B)$, $a \in \mathrm{dom}(Q)$, $Q(a) \in \mathrm{QComm}$.

The process $[\![S]\!] = \textsf{new}\ \tilde{a}\ \textsf{def}\ [\![K]\!]\ \textsf{in}\ (x\,!\,l[\tilde{v}])B \mid [\![T']\!] B \mid [\![Q]\!] \mid [\![R]\!]$ is closed for value variables from the closed process invariant. This implies that $x \in \mathrm{dom}(B)$ and therefore $a \in \mathrm{dom}(Q)$.

Now, only one of two possible situations can occur: $Q(a) \in \mathrm{QComm}$ or $Q(a) \in \mathrm{QMeth}$. This results directly from the queue invariant result.

1. if $Q(a) \in \mathrm{QComm}$ we apply the QUEUEMSG since $\{\tilde{v}\} \subseteq \mathrm{dom}(B)$ (from the closed process invariant).
2. otherwise, if $Q(a) \in \mathrm{QMeth}$ then we reduce with rule REDMSG. It remains to prove that $l \in \mathrm{dom}(M)$, with (B', M) the object closure at the head of the queue at a.

   The process $[\![S]\!]$ is written as:

   $$\textsf{new}\ \tilde{a}\ \textsf{def}\ [\![K]\!]\ \textsf{in}\ (x\,!\,l[\tilde{v}])B \mid [\![T'']\!] B \mid [\![Q]\!] \mid [\![R]\!] \stackrel{\text{def}}{=} \equiv$$
   $$\textsf{new}\ \tilde{a}\ \textsf{def}\ [\![K]\!]\ \textsf{in}\ a\,!\,l[B(\tilde{v})] \mid a?[\![M]\!] B' \mid [\![T'']\!] B \mid [\![Q + \{a : Ms\}]\!] \mid [\![R]\!]$$

With an argument similar to the instantiation case we conclude that the derivation tree for $\Delta, \Gamma \vdash [\![S]\!]$ has a subtree of the form:

$$\dfrac{\Delta, \Gamma \vdash a\,!\,l[B(\tilde{v})] \qquad \Delta, \Gamma \vdash a?[\![M]\!]B'}{\Delta, \Gamma \vdash a\,!\,l[B(\tilde{v})] \mid a?[\![M]\!]B'} \ \textsc{Par}$$

The rule Par implies that, in the typings for $a\,!\,l[B(\tilde{v})]$ and $a?[\![M]\!]B'$, a shares the same type. Hence, by an argument similar to the instantiation case now involving rules Msg, Meth and Obj, we conclude that $l \in \mathrm{dom}(M)$ and that the pre-conditions for rule RedMsg hold.

$(I = x?M)$. We must prove that one of the two pre-conditions holds:

1. $x \in \mathrm{dom}(B)$, $a \in \mathrm{dom}(Q)$, $Q(a) \in \mathrm{QComm}$, and $l \in \mathrm{dom}(M)$ for $l[\tilde{v}]$ in $Q(a)$;
2. $x \in \mathrm{dom}(B)$, $a \in \mathrm{dom}(Q)$, and $Q(a) \in \mathrm{QMeth}$.

The first two conditions, $x \in \mathrm{dom}(B)$ and $a \in \mathrm{dom}(Q)$, are common to the case for the message and have already been shown to hold.

Again, only one of two possible situations can occur: $Q(a) \in \mathrm{QComm}$ or $Q(a) \in \mathrm{QMeth}$.

1. if $Q(a) \in \mathrm{QMeth}$ we apply the QueueObj since $\mathrm{fv}(M) \subseteq \mathrm{dom}(B)$ (from the closed process invariant).
2. otherwise, if $Q(a) \in \mathrm{QComm}$ then we have an opportunity to reduce and apply the RedObj rule. It remains to prove that, for a communication $l[\tilde{v}]$ at the head of $Q(a)$, $l \in \mathrm{dom}(M)$.

   The process $[\![S]\!]$ is written as:

   $$\textsf{\textbf{new}}\ \tilde{a}\ \textsf{\textbf{def}}\ [\![K]\!]\ \textsf{\textbf{in}}\ (x?[\![M]\!])B \mid [\![T'']\!]B \mid [\![Q]\!] \mid [\![R]\!] \stackrel{\text{def}}{=} \equiv$$
   $$\textsf{\textbf{new}}\ \tilde{a}\ \textsf{\textbf{def}}\ [\![K]\!]\ \textsf{\textbf{in}}\ a?[\![M]\!]B \mid a\,!\,l[\tilde{v}] \mid [\![T'']\!]B \mid [\![Q + \{a : ms\}]\!] \mid [\![R]\!]$$

   With an argument similar to the case for a message we prove that S is typable and that the type deduced for a in the typings for the message and the object must match.

   Similarly, as the message and object share the same type for a it follows that, from the type system's rules Msg, Meth and Obj, $l \in \mathrm{dom}(M)$ and therefore the pre-conditions for rule RedObj hold.

From the above we conclude that typable programs do not deadlock the abstract machine. □

3.8.5 The Abstract Machine is Fair

Here we prove that the abstract machine is *fair* in the sense that any thread placed in the run-queue gets executed after a finite number of single step transitions in the machine.

Theorem (Fairness). *A thread placed in the run-queue is executed after a finite number of state transitions.*

Proof outline: we compute the number of machine steps required to start executing a thread just placed in the run-queue.

We assume a generic machine state with a (possibly empty) run-queue R, a current thread with j instructions, and a thread we have just placed at the end of the run-queue (B', T'). The machine state is of the form:

$$K, B, Q, I_1 :: \cdots :: I_j, R :: (B', T')$$

Since each instruction triggers a single transition rule in the machine, and we know from the previous theorem that a transition is always possible, after j machine transitions we get to the state:

$$K, B, Q, \bullet, R :: (B', T') :: R' \qquad \text{for some } R'$$

Notice that any new threads resulting from reductions are always added at the end of the run-queue. Now assume that the number of threads in R is k and that the number of instructions in a thread $(B_i, T_i) \in R$ is denoted $|T_i|$. To start computing the thread T' all we require is, for each thread T_i in R:

- to start running the thread with a SCHED instruction;
- to run all the instructions in the thread.

Thus, the total number of machine steps until the thread (B', T') starts running is given by:

$$j + \sum_{i=1}^{k} (1 + |T_i|) \qquad \Box$$

Chapter 4

The Virtual Machine

In this chapter we describe the architecture and implementation of a virtual machine for the TyCO programming language [40, 41]. Our main goal is to design a machine which is both efficient, scalable and that provides all the basic functionality required for concurrent programming. As we have seen in the previous chapters, objects interact by sending messages to shared communication channels. Hence channels, messages and objects are key data units of the machine.

In the design of the virtual machine we take the view that all the data in a program, regardless of where it is stored, is unstructured. The information on how the data should be processed is always embedded in the semantics of the instructions of the virtual machine. For example, an instruction that tries to reduce an object with a message takes a heap address as an argument. The instruction *knows* that this address is that of a channel and accesses its internal fields accordingly. This option not only makes the design cleaner but also makes it easily extensible to support other kinds of data-structures without ever changing the underlying structure. One just has to implement instructions that know how to manipulate such data-structures.

The formal model presented in the previous chapter provides a specification for the TyCO virtual machine. Our implementation follows the formal specification closely in several points, namely:

1. each thread is compiled into a block of contiguous byte-code instructions which we call a *vm_thread* (see section 3.6);

2. objects, messages and channels are implemented as heap frames. In particular, the communication queues are implemented as queues of message or object

frames (rule NEW in section 3.7.3). The methods of objects are compiled into vm_threads;

3. each template definition is compiled into a vm_thread plus a heap frame for the free variables, if they exist (rule DEF in section 3.7.3);

4. the emulator keeps newly generated vm_threads in a run-queue. Each vm_thread is represented by two heap references for the environment frames (arguments and free variables) and a reference to the byte-code to be executed (rules INST and RED in section 3.7.3);

5. the emulator starts with the *main* vm_thread which is the compilation of the program's initial thread (the initial state for the abstract machine in section 3.7.3);

6. a new vm_thread is loaded with its environment when the current vm_thread ends (the SCHED rule in section 3.7.3);

7. the emulator halts when the current vm_thread ends and the run-queue is empty (the halt condition in section 3.7.3).

In the remainder of this chapter we give a specification for a virtual machine for TyCO, describing its architecture and instruction set.

4.1 Memory Architecture

The design and implementation of the virtual machine is inspired in work from the functional language community – the STG-machine [33, 59]; the object-oriented community – the Java Virtual Machine [38], and; from Prolog implementations – the WAM [77]. The basic computational model follows from the formal abstract machine specification of the previous chapter.

The virtual machine uses five logically distinct memory areas to compute (figure 4.1).

4.1.1 The Program Area

The *program* area keeps the byte-code instructions to be executed. The byte-code is composed of instruction blocks and method tables (sequences of pointers to byte-code blocks). The byte-code manipulation datatype is given as:

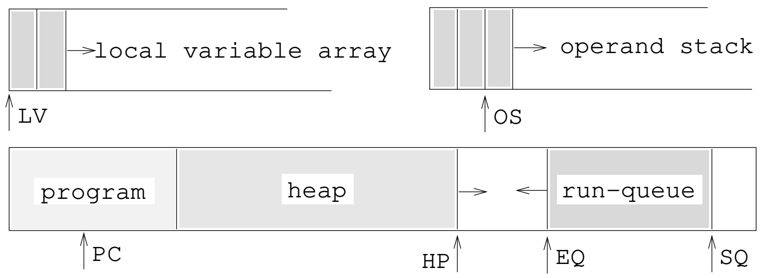

Figure 4.1: Memory Layout of the Virtual Machine

```
typedef  long  *BYTE;
```

The base address of the byte-code is kept in the global variable `Program`.

4.1.2 The Heap

The *heap* is a flat address space where dynamic data-structures such as objects, messages, channels and builtin values are allocated. The basic building block of the heap is a machine *word*. The basic allocation unit in the heap is the *frame* and consists of one or more contiguous heap words with a descriptor for garbage collection.

```
typedef  long   WORD;
typedef  WORD  *FRAME;
```

The virtual machine manipulates three basic kinds of processes at runtime: messages, objects and instantiations. Messages and objects are located in shared communication channels. Externally, the virtual machine sees all these abstractions as simple frames, although their internal structure is known to specialized instructions.

message
`WORD descriptor`
`FRAME next`
`int label`
`WORD` $\mathtt{arg}_1$
...
`WORD` $\mathtt{arg}_k$

object
`WORD descriptor`
`FRAME next`
`BYTE table`
`WORD` $\mathtt{fvar}_1$
...
`WORD` $\mathtt{fvar}_k$

instance
`WORD descriptor`
`BYTE code`
`WORD` $\mathtt{arg}_1$
... ...
`WORD` $\mathtt{arg}_k$

A *message frame* holds the label of the method it is invoking plus a variable number of arguments. An *object frame*, on the other hand, holds a pointer to the byte-code (the location of its method table) plus a variable number of bindings for the free variables in its method collection. An *instance frame* simply holds a reference to the byte-code for the template and a variable number of bindings, the arguments of a template instantiation. Since they may be enqueued in channels, both message and object frames have, besides the descriptor, an extra word that acts as a *next* field. This field allows queues of messages or objects to be formed in the heap as required by the semantics of the abstract machine (see section 3.7.1).

Channels are special frames that hold communication queues and implement process calculi names. Register `CC` points to the current channel being inspected by the virtual machine:

channel
`WORD descriptor`
`FRAME first`
`FRAME last`

Besides the descriptor, a channel has two additional words that point, respectively, to the first and the last frame in a queue. At any given moment, such a queue is either empty or holds only objects or only messages. This is ensured by the queue invariant property of the abstract machine (see section 3.8). The current state of the channel is kept in the descriptor word of the frame. The state indicates the internal configuration and composition of the queue. The basic possibilities can be expressed in the following enumeration:

```
enum {Empty=0, Msg=1, Obj=2} State;
```

A more precise enumeration for the state is defined to take advantage of the very common case when a single message or object occurs in a channel (see Turner [66]). The new definition of `State` is:

```
enum {Empty, OneMsg, OneObj, ManyMsg, ManyObj} State;
```

When only one object or message occurs in a channel, it is kept in a compacted form to optimize for the single message/object case, avoiding queue operations completely. If a second object or message arrives, the channel is quickly reconfigured to form a queue and reductions are processed according to the general case.

generic channel
`WORD  descriptor`
`FRAME first`
`FRAME last`

one object
`WORD  descriptor`
`BYTE  table`
`FRAME fvars`

one message
`WORD  descriptor`
`int   label`
`FRAME args`

When a channel returns to an empty state, it becomes possible to optimize again. So, with this optimization, the channel data structure switches dynamically between three possible representations.

4.1.3 The Run-Queue

The *run-queue* holds vm_threads ready for execution and their environments. It grows downwards in the direction of the heap. Each vm_thread in the run-queue consists of a data structure with the following items: a) a reference to a piece of byte-code in the program area; b) a reference to a heap frame with the parameter bindings, and; c) a reference to a heap frame with the free variable bindings. Thus each run-queue item takes three machine words:

vm_thread
`BYTE  code`
`FRAME args`
`FRAME fvars`

When a reduction (either communication or instantiation) occurs, a new vm_thread is created. The new vm_thread and its environment is placed at the end of the run-queue where it waits to be scheduled for execution. The heap and the run-queue are allocated in the bottommost and topmost areas, respectively, of a single memory block. They grow in opposite directions and garbage collection is triggered when a collision is predicted. Registers `SQ` and `EQ` mark, respectively, the start and the end of the run-queue.

Since we use a run-queue to store all newly created vm_threads and running threads cannot be suspended, the virtual machine is fair (see section 3.8.5) in the sense that every vm_thread created in a computation will eventually be executed after a finite (and typically small) amount of time.

4.1.4 The Local Variable Array

A vm_thread typically creates new local variables. These variables are initially bound to empty communication channels. The bindings for these variables are discarded after the current vm_thread terminates. Despite the short life span of local variables, the channels they are bound to may continue to exist long after the thread ends. Scope extrusion allows channels to escape the context of a thread and become visible to other threads. This is accomplished, for example, when a channel is sent as an argument to a message targeted outside the scope of the channel. Each time a new local variable is created with a **new**, the next available position in the *local variable array* is assigned to this variable and the binding to the channel is kept there until the thread finishes. The machine register `LV` points to the base of the array (figure 4.1). Local variables are renamed at compile time as entries in this array so that, when a vm_thread finishes, its local bindings are discarded simply by being overwritten, once the next vm_thread begins executing.

4.1.5 The Operand Stack

The virtual machine supports the builtin datatypes defined in TyCO directly. This means that there are instructions that implement the usual boolean algebra operations for booleans, the usual integer and floating-point arithmetic and the relational operations for integer, floating-point and strings. For strings there are also instructions for computing their size and for concatenation. The current implementation also partially supports lists as builtin values. For lists, besides the *empty list* value, the machine provides the common functionalities: checking whether a list is empty; constructing a list from an item and another list; getting the head and the tail of a list. In future implementations, strings and lists will be implemented through an external module, rather than directly as instructions of the virtual machine. The types for these builtins are given by:

```
typedef  enum {false=0,true=1}  BOOLEAN;
typedef  long                   INTEGER;
typedef  float                  FLOAT;
typedef  char*                  STRING;
typedef  FRAME                  LIST;
```

The evaluation of builtin expressions is done in a specialized memory area of the virtual machine called the *operand stack*. An operand stack cell is a union of the form:

```
typedef union {
   BOOLEAN   b;
   INTEGER   i;
   FLOAT     f;
   STRING    s;
   LIST      l;
   WORD      w;
} CELL;
```

Simple expressions such as builtin constants or simple variables do not require evaluation and therefore are copied directly to the heap. Using an operand stack to perform builtin operations enables the generation of more compact byte-code since many otherwise explicit arguments (for example, registers for the arguments and the result of an operation) are implicitly located at the top of the stack.

Most of the data-flow at run-time results from messages and object closures being built in the heap. This involves allocating new heap frames and copying the argument or free variable bindings from distinct places in the heap to the local frame. Sometimes, some expressions must be evaluated and so the data moves first from the heap to the operand stack and then back again to the new location at the heap. If a simple expression is involved then the copy is heap to heap.

4.1.6 The Machine Registers

The virtual machine uses a small set of global registers to control the program flow and to manipulate machine and user data-structures (table 4.1). Register `PC` (Program Counter) points to the next instruction to be executed. Register `HP` (Heap Pointer) points to the next available position in the heap. Registers `SQ` (Start Queue) and `EQ` (End Queue) point to the limits of the run-queue. Finally, register `OS` (Operand Stack) points to the last used position in the stack, whereas `LV` (Local Variable array) points to the base of the local variable array.

When a program starts, register `PC` is loaded with the address of the first instruction. Register `CC` (Current Channel) points to the channel which is currently being used to try a reduction. Register `CF` (Current Frame) holds frames temporarily until they are either 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). Registers `FV` (Free Variable bindings) and `PM` (ParaMeter bindings) are used to hold the free variable and parameter bindings, respectively.

Register	Description
`PC`	Program Counter
`HP`	Heap Pointer
`SQ`	Start of runQueue
`EQ`	End of runQueue
`CC`	Current Channel
`CF`	Current Frame
`OF`	Other redex component Frame
`FV`	Free Variable bindings
`PM`	ParaMeter bindings
`OS`	top of Operand Stack
`LV`	base of Local Variable array

Table 4.1: The register set.

4.1.7 The Instruction Set

The virtual machine instruction set was intentionally designed to be minimal in size and to have a very simple layout. Instructions are identified by a unique opcode held in the word pointed to by the program counter `PC`. For most instructions the opcode of an instruction determines the number of arguments that follow in contiguous words. Alternatively, in the case of instructions with a variable number of arguments (for example, `switch`) or in the case of instructions with variable sized arguments (for example, `soutput` – output string) a first argument is inserted in the byte-code that holds the offset for the next instruction. In the sequel we let n,k range over the natural numbers, l over code labels, and w over machine words (representing constants or heap references). We implement virtual machine instructions as parameterless C functions (see chapter 6). Here, only the fundamental code for each instruction is presented; for example, we omit the computation of the address for the next instruction to be executed.

Heap Allocation Instructions. `msgf` n,k allocates a frame for a message with label k and n arguments. Here, as in all the instructions of this group, the register `CF` is initialized pointing to the first word after the descriptor. `objf` n,l allocates a frame for an object with a method table (or just a method) l and n free variables.

```
msgf() {
   HP->desc = FRAME(n + 3);
   HP->next = 0;
   HP->label= k;
   CF       = HP + 1;
   HP      += n + 3;
}
```

```
objf() {
   HP->desc = FRAME(n + 3);
   HP->next = 0;
   HP->table= l;
   CF       = HP + 1;
   HP      += n + 3;
}
```

`instf` n allocates a frame for keeping n free variables of a template definition. `new` k creates a new channel in the heap and keeps a reference to it in the local variable array at LV[k].

```
instf() {
   HP->desc = FRAME(n + 1);
   CF       = HP + 1;
   HP      += n + 1;
}
```

```
new() {
   HP->desc = FRAME(3);
   LV[k]    = HP + 1;
   HP      += 3;
}
```

Data Flow Instructions. These instructions move data, one word at a time, within the heap and between the heap, the operand stack and the local variable array.

```
 put() { CF[k] = w; }

push() { OS++; OS[0] = w; }

 pop() { CF[k] = OS[0]; OS--; }
```

`put` k, w copies the word w directly to the position k in the frame currently being assembled. `push` w moves the data in w to the top of the operand stack. `pop` k moves the word at the top of the operand stack to the position k in the frame currently being assembled.

Control Flow Instructions. `iff` l jumps to the code at label l if the value at the top of the operand stack is (boolean) false, otherwise the execution continues with the next instruction. `switch` $l_1, , \ldots, l_n$ jumps to the code at label l_k where k is the value at the top of the operand stack, and n is the number of code label arguments of the instruction. `jump` l jumps unconditionally to the code at label l.

```
   iff() { if( OS[0] ) PC = l; OS--; }

switch() { PC = PC[OS[0]]; OS--; }

  jump() { PC = l; }
```

Queue Manipulation and State Instructions. For communication queues we need instructions to check and update their state and insert and remove items from the queues. `state` gets the state of the channel pointed to by register `CC` and places the result at the top of the operand stack. `update` k changes the status of the channel pointed to by register `CC` to k. `reset` checks whether the channel pointed to by `CC` holds an empty queue. If this is the case the state is changed to `Empty`. Getting and setting the state are simple operations that simply require bitwise operations on a channel's descriptor word.

```
 state() { OS++; OS[0] = GET_STATE(CC->descriptor); }

update() { SET_STATE(CC->descriptor,k); }

 reset() { if( !CC->first ) SET_STATE(CC->descriptor,Empty); }
```

`enqueue` places the frame pointed to by register `CF` in the queue of the channel pointed to by register `CC`. `dequeue` gets a frame from the queue at the channel pointed to by `CC` and keeps a pointer to the frame in register `OF`.

```
enqueue() {
   if( CC->last ) {
      CC->last->next = CF;
      CC->last = CF;
   } else {
      CC->first = CC->last = CF;
   }
}
```

```
dequeue() {
   OF = CC->first;
   CC->first = CC->first->next;
}
```

Compact Queue Manipulation Instructions. These extra instructions support an optimization of the communication queues for the case in which only one object or one message is held in a channel. In this case we do not build a queue and instead manipulate the object or message directly (see 4.1.2).

```
 cenqueue() { CC->first = CF[2]; CC->last = CF; }

 cdequeue() { OF = CC->last; }

decompact() { CC->first = CC->last; }
```

`cenqueue` places a message or an object frame pointed to by `CF` directly in the channel pointed to by `CC`. Note that `CF[2]` is the word in the frame that contains either a method table pointer (if an object) or a natural label (if a message). The messages and objects are kept in a compacted format. `cdequeue` dequeues a compacted frame from the channel pointed to by `CC` and puts a pointer to it in `OF`. `decompact` reconfigures a compacted object (message) in the channel pointed to by `CC` into a queue with one object (message) frame.

Reduction Instructions. `watch` w copies the pointer to a channel held in the word w to the register `CC`, to prepare a reduction. `redmsg` reduces the object pointed to by `CF` with the message pointed to by `OF`. `redobj` is the dual. It reduces the message pointed to by `CF` with the object at `OF`. These instructions implement communication and generate a new vm_thread that is placed in the run-queue. Finally, `instof` l,n creates a new vm_thread with a byte-code reference l and n arguments allocated in the run-queue. Register `OF` points to the heap frame with the free variables and `CF` is made to point to the first available word for the arguments which are copied to the run-queue by subsequent `put` or `pop` instructions. Note that the space used by the arguments in the run-queue is reclaimed immediately after the corresponding thread is selected for execution. No need for garbage collection.

```
 watch() { CC=w; }

redmsg() { EQ->code=(CF->table)[OF->label]; EQ->args=OF; EQ->fvars=CF; }

redobj() { EQ->code=(OF->table)[CF->label]; EQ->args=CF; EQ->fvars=OF; }

instof() { EQ->code=l; EQ->fvars=OF; CF=EQ; EQ-=n; }
```

Scheduling Instructions. The instruction `ret` is used at the end of each vm_thread to schedule a new vm_thread for execution. It first checks whether the run-queue is empty, in which case the virtual machine halts. If the run-queue is not empty it dequeues a vm_thread by updating the registers: `PC` for the new byte-code to be executed and, `FV` and `PM` that hold the environment bindings for the free variables and parameters, respectively. Notice that before running a vm_thread the machine checks whether a garbage collection is required by inspecting the word immediately before the first instruction in the new byte-code. This word contains the number of words required to run the vm_thread without having to suspend for garbage collection. The value is compared to the gap between the top of the heap pointed to by `HP` and the bottom of the run-queue pointed to by `EQ`.

```
ret() {
   if ( SQ == EQ ) {
      HALT();
   } else {
      PC = SQ->code;
      if( EQ - HP < PC[-1] )
         gc( PC[-1] );
      PM = SQ->args; FV = SQ->fvars; SQ -= 3;
   }
}
```

Instructions on Builtin Datatypes. These instructions operate on the builtin datatypes supported by the virtual machine. Their implementation is straightforward. The operands are taken from the top of the operand stack and the result, when defined, is placed at the top of the operand stack. The first set of instructions operates on boolean values:

```
not() {  OS[0].b = !(OS[0].b); }

and() { OS[-1].b = OS[0].b && OS[-1].b; }

 or() { OS[-1].b = OS[0].b || OS[-1].b; }
```

The basic arithmetic instructions on integer and floating-point values are defined as follows with $\alpha \in \{\mathtt{i}, \mathtt{f}\}$, except `mod` which is defined only for integers:

```
αadd() { OS[-1].α = OS[-1].α + OS[0].α; }

αsub() { OS[-1].α = OS[-1].α - OS[0].α; }

αmul() { OS[-1].α = OS[-1].α * OS[0].α; }

αdiv() { OS[-1].α = OS[-1].α / OS[0].α; }

αneg() {  OS[0].α = - OS[0].α; }

 mod() { OS[-1].i = OS[-1].i % OS[0].i; }
```

Then we have the relational operators for integers, floating-point and strings ($\alpha \in \{\mathtt{i}, \mathtt{f}, \mathtt{s}\}$):

```
 αeq() { OS[-1].b = (OS[-1].α == OS[0].α) ? true: false; }

αneq() { OS[-1].b = (OS[-1].α != OS[0].α) ? true: false; }
```

α`gt() { OS[-1].b = (OS[-1].`α` > OS[0].`α`) ? true: false; }`

α`geq() { OS[-1].b = (OS[-1].`α` >= OS[0].`α`) ? true: false; }`

α`lt() { OS[-1].b = (OS[-1].`α` < OS[0].`α`) ? true: false; }`

α`leq() { OS[-1].b = (OS[-1].`α` <= OS[0].`α`) ? true: false; }`

For string concatenation and length we have:

```
cat() { OS[-1].s = strcat(OS[-1].s,OS[0].s); }

len() {  OS[0].i = strlen(OS[0].s); }
```

Lists are frames with the following internal structure:

list
`WORD  descriptor`
`WORD  item`
`FRAME next`

and are polymorphic on the `item`. A few basic instructions for list manipulation are defined. `empty` checks whether the list pointed to by the word at the top of the operand stack is empty, and returns the boolean answer in the top of the operand stack. Next, `head` and `tail` take a pointer to a list at the top of the operand stack and select, respectively, the `item` or the `next` field from it. The result is placed at the top of the operand stack.

```
empty() { OS[0].b = (OS[0].l == NULL) ? true : false; }

 head() { OS[0].w = (OS[0].l)->item; }

 tail() { OS[0].l = (OS[0].l)->next; }
```

`nil` and `cons` respectively build, an empty list and a list element with a tail pointed to by the word at the top of the operand stack and an item in the position immediately below in the stack.

```
  nil() { OS++; OS[0].l = NULL; }
```

```
cons() {
   HP-=3;
   HP->desc = FRAME(3);
   HP->next =  OS[0].l;
   HP->item = OS[-1].w;
   OS--;
   OS[0].l  = HP;
}
```

Finally, for the input/output instructions we have ($\alpha \in \{\mathtt{b}, \mathtt{i}, \mathtt{f}, \mathtt{s}\}$):

```
αoutput() { fprintf(stdout,format(α),OS[0].α); }

αinput() { scanf(format(α),&OS[0].α); }
```

Besides the above instructions there are also specialized instructions that implement some optimizations. For example, in case a sequence of words is copied into contiguous words in the heap the argument k (that indicates the offset of the destination word in the heap frame) of the instructions `put` and `pop` may be dropped. An optimized version of the instructions just advances to the next word after placing a value. For further discussion on optimizations see chapter 6.

4.2 Garbage Collection

A major issue in the virtual machine is to make the programs run in as short a heap space as possible. Both the virtual machine binary and the memory data-structures are very conservative about memory usage. Efficient garbage collection is essential for such a goal. The virtual machine triggers garbage collection whenever the gap between the top of the heap, pointed to by `HP`, and the end of the run-queue, pointed to by `EQ`, is smaller than the required number of words to execute a thread. Making this test for every instruction that uses the heap before actually executing it would be very costly. Instead, when the byte-code files are assembled the maximum number of heap words required for the execution of the byte-code of each vm_thread is computed and placed in the word immediately preceding the first instruction of the respective piece of byte-code.

At run-time, before starting the execution of a new thread, the virtual machine checks the word `PC[-1]` to see whether there is enough space in the heap to safely run the thread. If there isn't, garbage collection is triggered (see `ret` in section 4.1.7). The number of required words for the thread is passed as an argument to `gc()`. If the garbage collector doesn't free at least `PC[-1]` heap words it stops the machine with an error message. Two kinds of garbage collection can be performed by the system:

- A *Shift Garbage Collection* is performed if the space between the highest memory address and the `SQ` register is enough to run the thread. In this case the garbage collector just shifts the run-queue upwards and returns;
- A *Full Garbage Collection* is performed otherwise.

For the full garbage collection we use a *copying* algorithm. The garbage collector copies active frames from a `FromSpace` (the current heap) to a `ToSpace` (the new heap). Active frames are those that can still be used in future computations within the program. At the boundaries of vm_threads the future run-time behavior of the program depends only on the contents of the run-queue. This is because a given frame is active and may be used in the future only if there is a direct reference to it from the run-queue, or if it is referenced by another active frame.

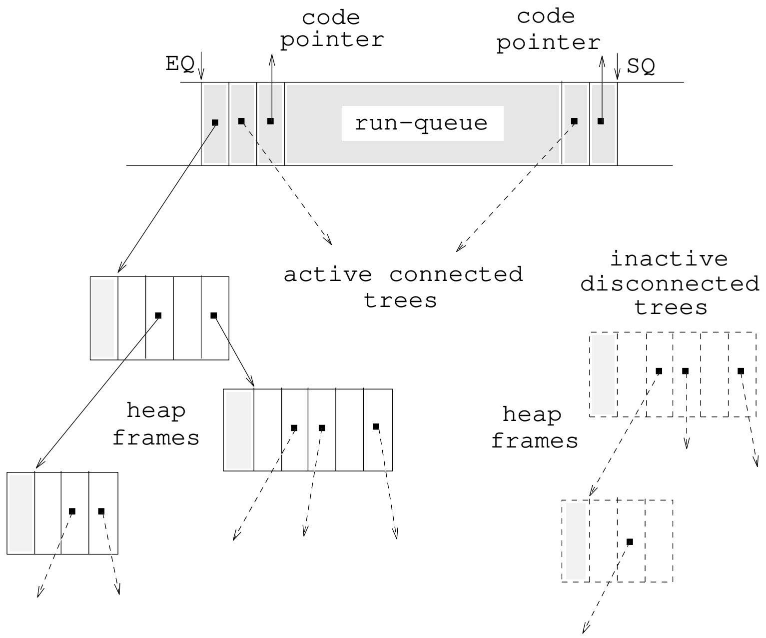

Figure 4.2: Garbage collection

For this reason, each item in the run-queue can be seen as the root of a tree of active heap frames. Any frame which is not in one of these trees will never be used in the future and can be garbage collected (figure 4.2). The garbage collector copies the run-queue from `FromSpace` to the top of `ToSpace` using the function `copy_run_queue`.

```
void  copy_run_queue() {
  register FRAME  ptr1, ptr2;

  for(ptr1 = SQ,ptr2 = ToSpace+HeapSize;ptr1 > EQ;--ptr1,--ptr2)
    *ptr2 = copy_active_tree(*ptr1);

  EQ = ptr2;
  SQ = HeapGcArray + HeapSize;
}
```

For each reference to the heap in the run-queue, the garbage collector copies the entire tree of active frames of which the reference is the root. The reference copied to `ToSpace` is the address of the copied tree in the new address space. If a reference is not a heap reference, for example, a byte-code reference, then it is copied as is to `ToSpace`.

The function `copy_active_tree` takes the root of a tree of active frames and recursively copies it from `FromSpace` to `ToSpace`. Once a frame is copied, it is marked and any further reference to it will just require a simple reference update. This is achieved in one pass through the run-queue. Finally, the pointers to `FromSpace` and `ToSpace` are swapped and the `HP` register updated to the first free position in `ToSpace`.

```
FRAME  copy_active_tree(FRAME ptr) {
  register FRAME  local;

  /* advance before recursion */
  local = To; ToSpace += size;

  /* copy descriptor and mark frame */
  local[0] = ptr[0]; ptr[0] = MARK(local+1);

  for(int i = 1;i < SIZE(ptr);i++) {
     if(INHEAP(ptr[i]))
        if(!MARKED(ptr[i]))
           local[i] = copy_active_tree(ptr[i]);
        else
           local[i] = UNMARK(ptr[i]);
     else
        local[i] = ptr[i];
  }
  return(++local);
}
```

The garbage collector does not use any knowledge about the internal structure of the frames (for example, whether they represent objects or messages).

Chapter 5

The Compiler

This chapter describes the compilation scheme and the implementation of the TyCO compiler. The compiler processes TyCO source code into the intermediate virtual machine assembly language. Next we give a motivation for our decision of using byte-code as the executable format for the virtual machine. The structure of the byte-code and its implementation are described in connection with the implementation of the virtual machine, with an emphasis on its flexibility and scalability. Finally, we discuss the suitability of the virtual machine as a target for compiling other well known process calculi.

5.1 The Assembly Code

The structure of assembly programs in our framework closely follows the one of source programs in the sense that both the nested structure of processes and the scope boundaries, are preserved. *Threads*, sequences of instructions of the form $I_1 :: \cdots :: I_k$, constitute the main organization block for the code. Each nested process sequence in the source code corresponds to a similarly nested sequence of assembly instructions. Nesting levels are delimited by brackets ("{" and "}") in the assembly. The threads corresponding to the bodies of template definitions are labeled in the assembly so that the code can be referred to at any point of the program. Objects and messages, on the other hand are expanded as they appear to the compiler and are unlabeled. The location for the code of a method in an object can only be obtained by making an indexed access to the object's method table.

The top level thread in assembly programs is called `main` and includes the code for the initial thread. For example, the code for the program:

skip

is given by:

```
main = { % no code generated for skip % }
```

Processes without nesting, such as **new** a a![1] are expanded linearly in the generated assembly code:

```
main = {
   new   c1            % create variable a
   msgf  1,0           % create message frame
   put   3,1           % put the value 1 in frame
   trmsg c1            % try to reduce the message at a
}
```

Objects and template definitions in programs introduce new nesting levels for, respectively, the method and template threads. For example, for the following object:

new a a ? { l_1 = (x) **new** b x![b], l_2 = **skip** }

we get the nested assembly code:

```
main = {
   new   c1            % create variable a
   objf  0             % new object frame with no free variables
   trobj c1 = {        % start object body
      { l_1,l_2 }      % method table
      l_1 = {          % start l1 thread
         new c1        % create b
         msgf  1,0     % create message frame for one argument
         put   3,c1    % put b in frame
         trmsg p1      % try to reduce the message at x
      }
      l_2 = {
         % no code for skip
      }
   }
}
```

and for a template definition such as:

def X = (x y) ... **in** X[2 3]

we get the following code:

```
main = {
   def X = {
      % code for body of X
   }
   instof 2,X             % create frame for X and two arguments
   put    1,2             % put 2 in frame
   put    2,3             % put 3 in frame
}
```

Notice how the code for template X can be used anywhere in the program as it is identified with a label. The labels for the methods in the object example are protected in a method table and can only be accessed when a communication occurs. In this case the message label, a natural number, is used as the index into the method table to select the appropriate code label.

In the sequel we compile TyCO programs into a subset of the virtual machine's instruction set. The subset contains all the basic functionality with simple extensions to include built-in datatypes and expressions.

5.2 The Compilation Scheme

In the following discussion we ignore variable scoping issues and assume that the variables in a process have been renamed so that no variable capture can occur. This is one of the premises we assumed as true in the specification for the abstract machine (see section 3.7.3). We first describe the set of data-structures and functions that the compiler will require. First, a symbol table maps identifiers in the program with strings used to generate code and provides the operations Insert and Lookup:

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

We assume a function `Gen` that takes an arbitrary number of strings and integers, and outputs them. Translating program labels into natural number offsets used to access method tables requires information gathered by the type-inference system described in section 2.3. Here, we assume that this information is available, as the programs are assumed to be well typed (see section 3.7.3) and, that the conversion is performed by the function `IndexOf`.

The compiler keeps a counter CI to assign distinct strings to local variables. CI is initialized with zero at the beginning of each thread.

There are seven code-producing functions.

$$\begin{gathered}
\mathsf{Insts} : \mathsf{Thread} \mapsto \mathsf{Void} \\
\mathsf{Inst} : \mathsf{Instr} \mapsto \mathsf{Void} \\
\mathsf{Meths} : \mathsf{ListOf(Method)} \mapsto \mathsf{Void} \\
\mathsf{Meth} : \mathsf{String} \times \mathsf{ListOf(Id)} \times \mathsf{ListOf(Id)} \times \mathsf{Thread} \mapsto \mathsf{Void} \\
\mathsf{Evals} : \mathsf{Expr} \mapsto \mathsf{Void} \\
\mathsf{Eval} : \mathsf{Expr} \mapsto \mathsf{Void} \\
\mathsf{Bind} : \mathsf{Binding} \mapsto \mathsf{Void}
\end{gathered}$$

Insts compiles a thread, using Inst to compile individual instructions. Meths compiles a collection of methods, generating the method table and, using Meth to compile the individual methods. Bind compiles each of the bindings in a process definition. Finally, Eval and Evals compile builtin expressions.

To compile a program, we compile the initial thread.

$$\mathsf{Compile}(T) \quad = \quad \mathsf{Gen}(\text{“}\mathtt{main} = \{\text{”});\ \mathsf{Insts}(T);\ \mathsf{Gen}(\text{“}\}\text{”})$$

Threads are compiled by resetting the local variable index counter and compiling each of the instructions individually.

$$\mathsf{Insts}(I_1 :: \ldots :: I_k) = \mathsf{CI} := 0;\ \mathsf{Inst}(I_1);\ \ldots\ ;\mathsf{Inst}(I_k)$$

To compile a local variable creation **new** x, we insert the binding for x in the symbol table and generate a **new** instruction.

$$\begin{aligned}
\mathsf{Inst}(\textbf{new}\ x) \quad = \quad & \mathsf{Insert}(x, \text{“}c\text{”} + \mathsf{CI});\ \mathsf{CI} := \mathsf{CI} + 1; \\
& \mathsf{Gen}(\text{“}\mathtt{new}\text{”} + \mathsf{Lookup}(x))
\end{aligned}$$

Messages are compiled into a `msgf` instruction that creates a frame in the heap. Next, the instructions for the evaluation of the argument expressions are produced. Finally, the `trmsg` instruction handles reduction. The variables in the argument expressions must be fetched from the symbol table:

$$\begin{aligned}
\mathsf{Inst}(x\,!\,l[e_1 \ldots e_k]) \quad = \quad & \mathsf{Gen}(\text{“}\mathtt{msgf}\text{”}, k+3, \text{“,”}, \mathsf{IndexOf}(l)); \\
& \mathsf{Evals}(e_1);\ \ldots\ ;\mathsf{Evals}(e_k); \\
& \mathsf{Gen}(\text{“}\mathtt{trmsg}\text{”}, \mathsf{Lookup}(x))
\end{aligned}$$

Objects are compiled into an `objf` instruction that creates a frame in the heap. Next, the instructions for the evaluation (copy) of the free variable bindings are produced. Finally, the `trobj` instruction handles reduction. The method table for the object and the bodies of its methods are produced immediately afterwards and recursively. The symbols for the free variables must be fetched from the symbol table:

$$
\begin{aligned}
\mathsf{Inst}(x?M) = \quad & y_1, \ldots, y_k = \mathrm{fv}(M); \\
& \mathsf{Gen}(\text{“}\texttt{objf}\text{”}, k+3, \texttt{table}); \\
& \mathsf{Evals}(\mathsf{Lookup}(y_1));\ \ldots\ ; \mathsf{Evals}(\mathsf{Lookup}(y_k)); \\
& \mathsf{Gen}(\text{“}\texttt{trobj}\text{”}, \mathsf{Lookup}(x), \text{“}= \{\text{”});\ \mathsf{Meths}(M);\ \mathsf{Gen}(\text{“}\}\text{”})
\end{aligned}
$$

Instances $X[\tilde{e}]$ of templates are compiled more straightforwardly. Given that $X = (\tilde{x})T$, we lookup the string that labels X in the code. Also, the argument expressions $\tilde{e}$ must be computed. The symbols for the template's free variable bindings have been inserted in the symbol table when the definition was compiled (see the definition of Bind). In this way we get:

$$
\begin{aligned}
\mathsf{Inst}(X[e_1 \ldots e_k]) \;=\; & \mathsf{Gen}(\text{“}\texttt{instof}\text{”}, k+1, \mathsf{Lookup}(X)); \\
& \mathsf{Evals}(e_1);\ \ldots\ ; \mathsf{Evals}(e_k)
\end{aligned}
$$

Next, to compile a recursive definition **def** D, where $D = D_1$ **and** $\cdots$ **and** D_k we just compile the sequence of bindings in D.

$$
\mathsf{Inst}(\textbf{def}\ D) \;=\; \mathsf{Bind}(D_1);\ \ldots;\ \mathsf{Bind}(D_k)
$$

To compile a binding $X = (\tilde{x})T$ the compiler generates a string (the label) for the code of X and inserts the parameters and free variable bindings in the symbol table. We keep the current bindings for the free variables in a heap frame allocated by instruction `instf`. The labeled template body is compiled immediately assuming $|\tilde{x}| = k$.

$$
\begin{aligned}
\mathsf{Bind}(X = (\tilde{x})T) \;=\; & y_1, \ldots, y_l = \mathrm{fv}(T) \\
& \mathsf{Insert}(X, \texttt{X}); \\
& \mathsf{Insert}(x_1, \text{“}\texttt{p}_1\text{”}); \ldots ; \mathsf{Insert}(x_k, \text{“}\texttt{p}_k\text{”}); \\
& \mathsf{Insert}(y_1, \text{“}\texttt{f}_1\text{”}); \ldots ; \mathsf{Insert}(y_l, \text{“}\texttt{f}_l\text{”}); \\
& \mathsf{Gen}(\text{“}\texttt{instf}\text{”}, l+1); \\
& \mathsf{Evals}(y_1); \ldots ; \mathsf{Evals}(y_l); \\
& \mathsf{Gen}(\mathsf{Lookup}(X), \text{“}= \{\text{”}); \mathsf{Insts}(T); \mathsf{Gen}(\text{“}\}\text{”})
\end{aligned}
$$

Compiling a collection of methods $M = \{l_1 = (\tilde{x}_1)T_1, \ldots, l_k = (\tilde{x}_k)T_k\}$ involves generating the method table and the actual code for the methods. We assume an operator + that appends two strings and that $\tilde{y} = \mathrm{fv}(M)$ are the free variables of the method collection. We then have:

$$\begin{aligned}
\mathsf{Meths}(M) \quad &= \quad \mathsf{Gen}(\text{``\{''}, \text{``}l\text{''} + \mathsf{IndexOf}(l_1), \text{``,''}, \ldots, \text{``,''}, \text{``}l\text{''} + \mathsf{IndexOf}(l_k), \text{``\}''}); \\
&\quad\quad \mathsf{Meth}(\text{``}l\text{''} + \mathsf{IndexOf}(l_1), \tilde{x}_1, \tilde{y}, T_1); \\
&\quad\quad \ldots \\
&\quad\quad \mathsf{Meth}(\text{``}l\text{''} + \mathsf{IndexOf}(l_k), \tilde{x}_k, \tilde{y}, T_k)
\end{aligned}$$

Finally, to compile a method with label l, parameters $\tilde{x}$ ($|\tilde{x}| = k$), free variables $\tilde{y}$ ($|\tilde{y}| = l$) and body T, we first establish the bindings for the method parameters and free variables, and finally generate the code for the method body:

$$\begin{aligned}
\mathsf{Meth}(l, \tilde{x}, \tilde{y}, T) \quad &= \quad \mathsf{Insert}(x_1, \text{``}\mathtt{p}_1\text{''}); \ldots; \mathsf{Insert}(x_k, \text{``}\mathtt{p}_k\text{''}); \\
&\quad\quad \mathsf{Insert}(y_1, \text{``}\mathtt{f}_1\text{''}); \ldots; \mathsf{Insert}(y_l, \text{``}\mathtt{f}_l\text{''}); \\
&\quad\quad \mathsf{Gen}(l, \text{``}= \{\text{''}); \mathsf{Insts}(T); \mathsf{Gen}(\text{``\}''})
\end{aligned}$$

To compile expressions we have two functions: Evals and Eval. The first compiles values into direct heap-to-heap copy instructions and calls Eval for complex expressions.

$$\begin{aligned}
\mathsf{Evals}(c) \quad &= \quad \mathsf{Gen}(\text{``}\mathtt{put}\text{''}, c); \\
\mathsf{Evals}(x) \quad &= \quad \mathsf{Gen}(\text{``}\mathtt{put}\text{''}, \mathsf{Lookup}(x)); \\
\mathsf{Evals}(op\ e_1) \quad &= \quad \mathsf{Eval}(op\ e_1); \mathsf{Gen}(\text{``}\mathtt{pop}\text{''}); \\
\mathsf{Evals}(e_1\ op\ e_2) \quad &= \quad \mathsf{Eval}(e_1\ op\ e_2); \mathsf{Gen}(\text{``}\mathtt{pop}\text{''})
\end{aligned}$$

Eval compiles an expression into code that evaluates the expression on the operand stack and copies the result to the heap.

$$\begin{aligned}
\mathsf{Eval}(c) \quad &= \quad \mathsf{Gen}(\text{``}\mathtt{push}\text{''}, c); \\
\mathsf{Eval}(x) \quad &= \quad \mathsf{Gen}(\text{``}\mathtt{push}\text{''}, \mathsf{Lookup}(x)); \\
\mathsf{Eval}(op\ e_1) \quad &= \quad \mathsf{Eval}(e_1); \mathsf{Gen}(op); \\
\mathsf{Eval}(e_1\ op\ e_2) \quad &= \quad \mathsf{Eval}(e_1); \mathsf{Eval}(e_2); \mathsf{Gen}(op);
\end{aligned}$$

5.3 Compiler Implementation

This section briefly describes the structure and implementation of the TyCO compiler. The compiler organization is based on the datatype hierarchy. Thus, each datatype in the compiler has an associated class with methods for construction, type-inference and code generation, at least. Another possibility would be to organize the compiler in an orthogonal direction, based on the compilation phase rather than the datatype hierarchy [7]. This may prove more productive to work with and is a likely change in the structure of the compiler for the near future. The current implementation is developed in Java.

5.3.1 The Parser

The parser was written using two tools for automatic lexical and syntactic analyzer generation. They are called `jlex` and `jcup` respectively and were introduced for compiler development in Java. The parser builds the syntax tree for a program in memory. All the derived constructs in the source file are translated into basic TyCO at this stage and are represented as such in the syntax tree. Nevertheless, some of this syntactic sugar contains important information on the use of channels. For example, the reply channel used in **let** and **match** constructs is always linear, that is, is used exactly once. This information is gathered by the parser and used in the type-inference stage.

5.3.2 The Type-Inference System

The type-inference system takes the syntax tree produced by the parser and annotates each free identifier in a process with its type inferred from the type system described in section 2.3. Currently, this type information just involves information about the *shape of* channels and template definitions. This information consists in the kind of objects a given channel may carry, with how many methods and the names of these methods; for each method the system also finds out how many parameters it takes and their respective types. In the case of template variables the type system deduces the types of its parameters and supports polymorphic definitions. In some of our experiments, in chapter 7, we use channel usage information to optimize the code generated by the TyCO compiler. For example we are interested in knowing whether a given channel is linear (say of multiplicity 1) meaning that it is used exactly once for reduction, or if it holds a single persistent object. Currently, the type system is being extended to gather this kind of information based mostly in the works of Kobayashi, Pierce and Turner [36] and Amadio [6]. Meanwhile, for the experiments we use explicit type annotations in the programs to gather this information.

5.3.3 The Code Generator

Finally the code generator takes the annotated syntax tree for a program and generates the intermediate assembly code for it. Optimizations are applied based on some syntactic information. For example, an object that has a single method does not require a method table and the extra indirection from consulting the table can be eliminated, thus optimizing the method call. Optimizations may also result from information gathered by the type system. For example, if a given channel is known to be linear then it is possible to avoid allocating it in the heap at all (see section 6.7). Given the pervasiveness of linear channels in most programs the impact in performance of this optimization is quite important (see section 7.2.4).

The intermediate assembly representation produced at this stage is quite useful since it provides a machine-level representation of source programs and maps almost one-to-one with the final byte-code. The intermediate assembly can be used, for example:

- to support visualization and debugging tools for the TyCO language;
- insert an extra level of optimizations based on the analysis of the code generated before it is converted to byte code. Examples of such optimizations may be: register allocation and merging or re-ordering instructions;
- allow other languages to be compiled into TyCO assembly directly and then just use the TyCO byte-code assembler;
- as a possible base representation for proof/type-information carrying code in a system with distribution and code mobility.

Currently a graphical visualization and debugging tool is being developed for the TyCO virtual machine [41] as part of an ongoing project. The user follows the execution of the byte-code through its mapping into the virtual machine assembly.

5.4 Example Compilations

To illustrate the compilation into the virtual machine's instruction set we give the intermediate assembly code generated for some small examples with representative program structures. We present the non-optimized code so that it is more evident the transformation operated on the source code and the readability of the generated code by comparison.

Factorial We compute factorial with the common recursive definition:

```
def Fact = (n replyTo)
    if n == 0 then
      replyTo![1]
    else
      let v = Fact[n-1] in replyTo![n*v]
in  ...
```

The code generated by the TyCO compiler is as follows:

```
main  = {
   def Fact1(2) = {
      push     p0
      push     0
      eqi
      iff      l0      % if n==0
      msgf     1,0     % case true
      put      3,1
      trmsg    p1      % replyTo![1]
      jump     l1
l0:
      newc     c0      % case false
      instof   2,Fact1
      push     p0
      push     1
      subi
      pop      1
      put      2,c0    % Fact[n-1 replyTo]
      objf     2
      put      3,p0
      put      4,p1
      trobj    c0 = { % synchronization
         { val }
         val = {
            msgf     1,0
            push     f0
            push     p0
            muli           % n*v
            pop      3
            trmsg    f1    % replyTo![n*v]
         }
      }
l1:
   }
   ...
}
```

Mutually recursive templates Each of the template definitions in this example calls the other while performing simple arithmetic on the arguments.

def X = (x y)Y[y+1 x-1] **and** Y = (x y)X[y-1 x+1] **in** ...

The code generated by the TyCO compiler is as follows:

```
main  = {
   def X1(2) = {
      instof   2,Y2       % Y[ ...
      push     p1
      push     1
      addi                % y+1
      pop      1
      push     p0
      push     1
      subi                % x-1
      pop      2
   }
   def Y2(2) = {
      instof   2,X1       % X[ ...
      push     p1
      push     1
      subi                % y-1
      pop      1
      push     p0
      push     1
      addi                % x+1
      pop      2
   }
   ...
}
```

The List datatype A List is an object with a single method that when invoked replies either nil – the empty list – or cons[head tail] – for a list with head head and tail tail. The datatype is polymorphic in head.

def Nil = (self) self ? (replyTo) replyTo!*nil*[] | Nil[self] **in**
def Cons = (self head tail) self ? (replyTo) replyTo!*cons*[head tail] | Cons[self head tail]
in ...

The corresponding code generated by the TyCO compiler is as follows:

```
main  = {
   def Nil1(1) = {
      objf     1                  % self ? (replyTo) = ...
      put      3,p0
      trobj    p0 = {
         { val }
         val = {
            msgf     0,1
            trmsg    p0           % replyTo!nil[]
            instof   1,Nil1
            put      1,f0         % Nil[self]
         }
      }
   }
   def Cons2(3) = {
      objf     3                  % self ? (replyTo) = ...
      put      3,p2
      put      4,p1
      put      5,p0
      trobj    p0 = {
         { val }
         val = {
            msgf     2,0
            put      3,f1
            put      4,f0
            trmsg    p0           %   replyTo!cons[head tail]
            instof   3,Cons2
            put      1,f2
            put      2,f1
            put      3,f0         % Cons[self head tail]
         }
      }
   }
   ...
}
```

The Cell datatype A Cell is an object with two methods read and write that, respectively, read the state value of the cell and, update it with another value. The datatype is polymorphic in value.

```
def Cell = (self value) self ? {
          read= (replyTo) replyTo!val[value]  | Cell[self value],
          write= (newValue) Cell[self newValue]
   } in   ...
```

The corresponding code generated by the TyCO compiler is as follows:

```
main  = {
   def Cell1(2) = {
      objf     2
      put      3,p0
      put      4,p1
      trobj    p0 = {
         { read, write }
         read = {
            msgf     1,0
            put      3,f1
            trmsg    p0          % replyTo!val[value]
            instof   2,Cell1
            put      1,f0
            put      2,f1        % Cell[self value]
         }
         write = {
            instof   2,Cell1
            put      1,f0
            put      2,p0        % Cell[self newValue]
         }
      }
   }
   ...
}
```

Procedure call The process below calls a procedure X with an argument 6 and a reply channel replyTo and waits for its result in x before proceeding with some computation guarded by the prefix replyTo ?:

def X = (x replyTo) ... **in new** replyTo X[6 replyTo] | replyTo ? *val*= (x) ...

The code generated by the TyCO compiler is as follows:

```
main  = {
   def X1(2) = {
      % definition of X
   }
   newc     c0       %  new replyTo
   instof   2,X1
   put      1,6
   put      2,c0     %  X[6 replyTo]
   objf     0
   trobj    c0 = {   %  replyTo ? val = (x) ...
      { val }
      val = {
         % computation after synchronization
      }
   }
}
```

5.5 The Byte-Code

When the development of TyCO started we were confronted with two possible options for the implementation of the virtual machine. One corresponds to the approach taken in Pict [66] where the abstract machine is compiled into a C library. Pict programs are compiled to C, then compiled into object files by a C compiler, and finally, linked with the run-time system library to obtain the final executable.

On the other hand, a byte-code implementation such as Join [22] compiles source programs into a hardware independent representation which is then run by an emulator. This design separates the code for the virtual machine from that of the executable files at run-time. While the virtual machine may be hardware dependent only a copy is required to run programs. On the other hand programs are totally hardware independent relying only on an appropriate virtual machine for execution on a given hardware platform.

Byte-code implementations usually introduce run-time overheads resulting mainly from the emulation mechanism and are thus, on average, slower than implementations that generate binary code. However the advantages provided by the byte-code format are far more interesting and justify the shift in design:

- the byte-code for a program is usually much smaller than the corresponding binary code. This is mostly due to the fact that it does not carry with it the code for the virtual machine. In binary format the virtual machine code is replicated in every executable;
- the byte-code is hardware independent which makes it ideal for highly portable systems where only an implementation of the underlying virtual machine for the given architecture is required;
- the virtual machine is self contained. The implementor has full control of the low-level layers of the abstract machine which makes it rather flexible, especially to implement optimizations. In binary implementations one often has to work around language features or limitations to implement or optimize a given feature;
- the virtual machine can be cleanly designed and implemented. This is important for the implementation, since it makes debugging or extending the virtual machine easier. It is also fundamental to allow a precise semantics to be formulated;
- one of the extensions to TyCO we are interested in, and one which is now in

progress, is the inclusion of support for code mobility and distribution across networks. The small size of compiled byte-code, the hardware independence and the flexible structure (allowing, for example, type information to be embedded in the code to be dynamically checked) make it arguably the best choice for distributed implementations.

With the rationale for our option for byte-code in the implementation of TyCO we will describe the layout of the byte-code we used.

5.5.1 The Byte-Code Layout

The virtual machine executes architecture independent byte-code programs. The byte-code is generated from an intermediate assembly representation of TyCO programs and has a very simple layout. The basic unit of the byte-code is the machine word. While this is not optimal in terms of memory usage and file size, as for example opcodes do not need a full machine word, it avoids in a first approach the run-time overheads of keeping bitwise tags for opcodes and the complexities of memory alignment when manipulating certain datatypes such as floating point values.

Instructions are fixed size sequences of byte-code words, with a very simple layout: a first word for the opcode followed by one word per argument, if any. Usually the opcode of an instruction determines exactly how many arguments it is followed by. In the case of the few instructions that have a variable number of arguments (for example, `switch`) or a fixed number of arguments but some with variable sizes (for example, `soutput`), the first word after the opcode carries the offset to the next instruction in the byte-code (figure 5.1).

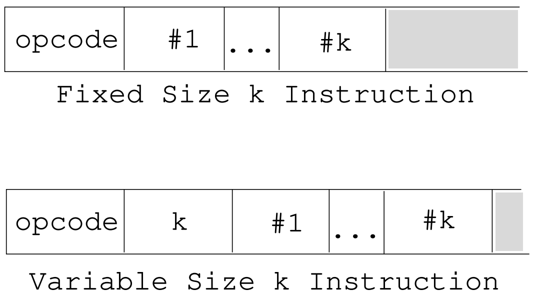

Figure 5.1: Instruction layout.

Byte-code programs are composed of byte-code blocks, corresponding to the compiled code of *vm_threads*, and method tables. Each byte-code block is a sequence of contiguous instructions ending in a `ret`. Every byte-code block starts with a header composed of two machine words. One is a tag used by the byte-code assembler; the other is the maximum number of heap words the vm_thread requires to execute as defined in section 4.2 (figure 5.2). This value is computed for each vm_thread by the assembler program. The method collections of objects in the source program are

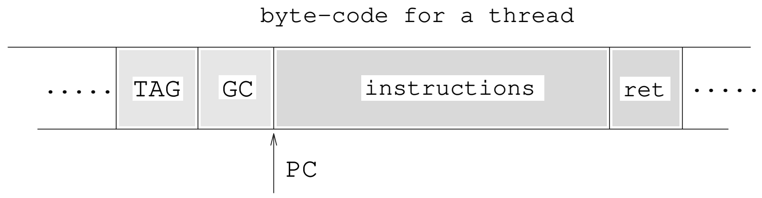

Figure 5.2: A thread.

compiled into tables of references to byte-code blocks. As in the case of a byte-code block, every such table starts with a two word header. One is a tag used by the assembler; the other keeps the size of the table (figure 5.3). When a communication occurs at run-time, the method table for an object is indexed with the label carried in the message and the corresponding method is invoked. The execution of byte-code

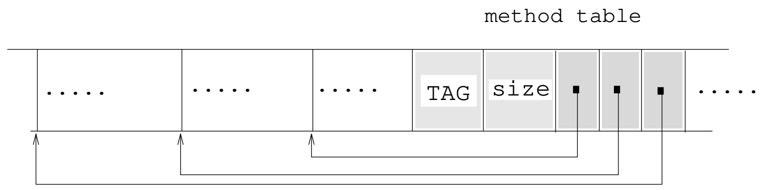

Figure 5.3: A method table

programs starts with the byte-code block at offset 0 in the program. This corresponds to the vm_thread `main`.

The string that identifies the argument of an instruction in the assembly code defines not only the addressing mode being used but also which part of the thread's environment is accessed. There are only two addressing modes allowed by the virtual machine: constant and indexed access. However, the run-time environment of a vm_thread is distributed by three distinct frames: the parameter frame pointed to by `PM`; the free variable frame pointed to by `FV`, and; the local variable array at `LV`. So, the argument string also specifies which of these three components should be accessed for the value. For example, in the following four variants of `put`:

```
put 2      put p1      put f1      put c1
```

the immediate integer value `2` is copied to the heap. The remaining three examples use indexed access to perform the same operation on, respectively, the first parameter of the vm_thread (`PM`[1]), the first free variable of the vm_thread (`FV`[1]) and the first local channel allocated for the vm_thread (`LV`[1]). The assembler unfolds each assembly instruction into the byte-code instruction that matches the addressing mode and environment area in use.

5.5.2 The Byte-Code Assembler

The assembler program is a one-pass parser. It parses the assembly instructions generated by the TyCO compiler one by one and writes the corresponding byte-code instruction to a memory buffer implemented as an *infinite array*. The assembler also performs some operations on the intermediate code. First, it keeps track of all the labels in the code. These are identifiers that mark the position of byte-code blocks or method tables. All occurrences of a label are kept in a list until the defining occurrence of the label is found (figure 5.4). The list is selected from an hash table whose key is the label identifier. Once the label is resolved all its previous occurrences are substituted with the offset and further references are immediately substituted after a quick inspection in the hash table. Programs in TyCO are self-contained in the sense that all labels in a program must have some byte-code attached to it. The assembler

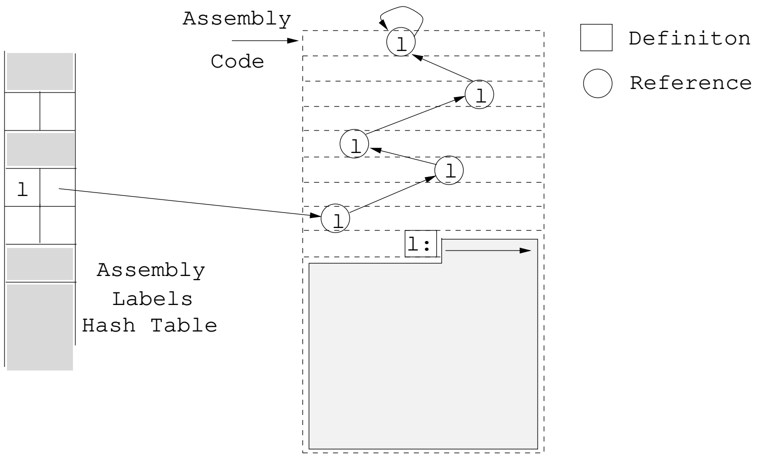

Figure 5.4: Label resolution.

program must also find the boundaries of byte-code blocks where startup/cleanup code must be inserted. The startup code consists basically of instructions that load the new environment and program counter. The cleanup code consists of instructions that check the halting condition for the machine and schedule a new thread for execution.

5.6 Implementing Other Calculi

In the sequel we study the feasibility of implementing a few well known calculi with our virtual machine. For an overview of these calculi and their relation to TyCO the reader may refer to chapter 9. In some cases minor additions are required in the form of new instructions or data-structures. This is because the virtual machine has been designed and optimized with TyCO in mind. For example, certain addressing modes are not used in compiled TyCO code and so there are no instructions to take advantage of the extra functionality.

The design of the virtual machine is based on the principle that communication among processes is channel based. While most process calculi share this communication mechanism, some significantly deviate from the common channel based reduction scheme and require special data-structures to be efficiently implemented and for these we find that our virtual machine is not, in its present state, a suitable implementation framework.

The asynchronous π-calculus [9, 27] is a sub-calculus of TyCO. To translate the asynchronous π-calculus into TyCO we restrict the set of labels to a single element, say *val*. This way, a π-input process $a(\tilde{x}).P$ is equivalent to the TyCO object with a single method $a?\{\mathsf{val} = (\tilde{x})P\}$. Similarly, the π-output process $\bar{a}(\tilde{v})$ is equivalent to the TyCO message $a!\mathsf{val}[\tilde{v}]$. Besides the fact that, at compile time, the type system must now allow only one label to be used (it may even be omitted) the implementation is absolutely for free. Since a π reduction maps into a TyCO method selection the implementation is efficient. In fact, an optimizing compiler can easily avoid the generation of method tables in the corresponding TyCO code since all objects have a single method (see section 6.6). This avoids the extra indirection of consulting the method table and makes reduction as efficiently implemented as it would be in a pure π program.

The π-calculus also exhibits a replication operator "!" that it uses to model unbounded behavior. TyCO does not have builtin support for replication. Instead it uses recursion to model unbounded behavior. An unbounded π-input process $!a(\tilde{x}).P$ can be modeled in TyCO with the template **def** $X = a?\{\mathsf{val} = (\tilde{x})P \mid X\}$. The compilation is efficient since: a) recursion is more conservative in heap usage and; b) the same environment frame for the variables of P can be used (see section 6.8).

The synchronous π-calculus [46] differs from the asynchronous form essentially in that output processes have continuations. Also, replication can be applied to any process. In other words, the output processes are now of the form $\bar{a}(\tilde{v}).Q$, where Q is a computation that is triggered only when the message prefix $\bar{a}(\tilde{v})$ is consumed. The reduction then occurs as follows:

$$a(\tilde{x}).P \mid \bar{a}(\tilde{v}).Q \to P\{\tilde{v}/\tilde{x}\} \mid Q$$

In terms of the virtual machine we have two problems that must be solved to support such a reduction scheme:

1. when an output process $\bar{a}(\tilde{v}).Q$ is found in a program, besides storing the arguments in a frame as we did in the compilation of TyCO, we must now store in another heap frame a pointer to the code for Q and the bindings for its free variables in the current environment (see figure 5.5);

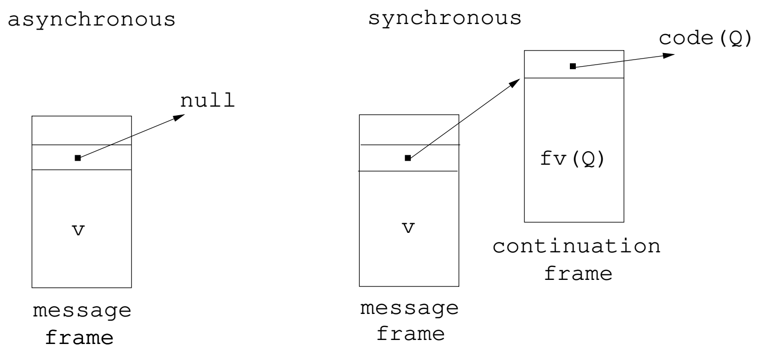

Figure 5.5: Messages in asynchronous and synchronous π.

2. the instruction that reduces output processes must be changed so that besides creating a thread corresponding to $P\{\tilde{v}/\tilde{x}\}$ in the run-queue, if a continuation frame is found with a pointer for the code of some Q, it simultaneously creates the thread for Q with the free variable bindings held in that heap frame. The thread for the continuation is also placed in the run-queue.

Note that the reduction instruction for output processes, is generic and encompasses both the asynchronous and synchronous cases, respectively, when there is not and there is a continuation frame. In what concerns replication, we can efficiently model an unbounded π-process $!P$, with the template **def** $X = P \mid X$ in TyCO.

Sangiorgi's variants [61] is a typed π-calculus aiming at interpreting Abadi and Cardelli's first order object calculus [61]. The main novelties consist in the introduction of a new kind of values called *variants* that are used to model labeled selection and, a **case** construct that selects a computational branch based on a variant value. The calculus is monadic.

In terms of the virtual machine, two main problems arise in the implementation of this calculus:

1. variant values can be composed of several *concatenated* labels followed by a name. These labels must be striped from left to right as the process goes through successive selection events;
2. the **case** construct must be implemented in the framework of the virtual machine.

TyCO is a sub-calculus of variants with only one level of nested variants. In this calculus a TyCO object can be modeled as the following process:

$$a(x).\textbf{case}\ x\ \textbf{of}\ [l_1(x_1) = P_1; \cdots ; l_k(x_k) = P_k]$$

that is, an input process that receives a value bound to x and selects a branch according to the structure of the value received. As we have seen, a value in this calculus is a name prefixed with an arbitrary number of label values. It follows that, for example,

$$\textbf{case}\ l_1 k_2 a\ \textbf{of}\ [l_1(x_1) = P_1; \cdots ; l_k(x_k) = P_k] \to P_1\{k_2 a/x_1\}$$

In TyCO the only values are names. Labels form a distinct syntactic category and are used for method selection. They are not first class.

Given the above correspondence between a TyCO object and an object in this calculus it is easy to see that the expression:

$$\textbf{case}\ l_i v\ \textbf{of}\ [l_1(x_1) = P_1; \cdots ; l_k(x_k) = P_k]$$

is essentially a redex in TyCO:

$$\textbf{new}\ a\ a!l_i[v] \mid a?\{l_1 = (x_1)P_1, \cdots , l_k = (x_k)P_k\}$$

So the compilation of selection is pretty straightforward and so close to TyCO that the resulting code is efficient.

Support for composed values, namely labels, is a more delicate issue. A value such as $l_2 l_1 l_3 a$ can be seen as the body of a message in TyCO of the form $l_2[l_1 l_3\ a]$ where the outer label is ready for a selection and the remaining composed labels are passed explicitly as an extra argument in the message. The final selection binds the name a to some local variable. Thus we see that labels must become first class entities for this to work. There is no restriction whatsoever at the virtual machine level for this. The correctness of such programs would have to be enforced by an appropriate type-system though.

Parrow and Victor's Fusion Calculus [51] is an extension of the π-calculus that features a complete symmetry between input and output processes. Both input and output processes have continuations (the calculus is synchronous) and, there is only one binding operator –ν– for introducing new names. When two processes with opposite polarities interact on a name as in:

$$a(x).P \mid \bar{a}(y).Q \overset{x=y}{\to} P \mid Q$$

the result is the unification of variables x and y in the current environment and the processes P and Q running.

Another important mechanism in the calculus are guards of the form $[x = y]$ that test whether the variables x and y are unified. The process P in $[x = y]P$ gets executed if the variables are unified; it does not run otherwise.

In order to support unification of variables we must change the semantics of the machine operations that copy information. We have two kinds of values: *names* and *links*. Links are reference chains in the heap. At any moment in a computation a variable is either linked directly to a name or linked to one through a reference chain. So, when we keep the bindings for a process in a frame we are now interested in keeping the links between variables rather than the values they point to. For example, a typical instruction for saving a value in a frame would be `put 1,a0` which now reads "place, at offset 1 of the current frame, *a pointer* to (as opposed to *the value at*) `a0`".

To get a value associated with a variable, we need an extra instruction that returns the name at the end of the chain pointed to by the variable, if it exists, or `null` if the chain does not end in a name. The last case is possible, for example, if the variable is free as is explained below.

The ν operator is responsible for the introduction of new names and their binding to variables. In the example above of process interaction, there is no name involved in the interaction. All that remains after the interaction is the environment constraint that $x = y$. However, if we precede the process with a ν we can bind one of the variables, say y, to a name. After the reduction x is bound to that same name:

$$\nu y(a(x).P \mid \bar{a}(y).Q) \stackrel{x=y}{\rightarrow} \nu y(P \mid Q) \equiv (P \mid Q)\{b/x, b/y\}$$

where b is the name associated with the variable y. Finally we need another instruction that checks whether two variables x and y are unified. This is required to implement the guards $[x = y]$. This function just follows each reference chain and checks that they point to same name.

Boudol's Blue calculus [10] is an extension of the π-calculus that directly incorporates the λ-calculus. Thus the calculus has two computational components. The first component is functional and encompasses a name-passing λ-calculus with variables, abstraction and application – small β-reduction

$$(\lambda x P)y \rightarrow P\{y/x\}$$

that drives the computation. The programming style is functional as opposed to *continuation passing* in the pure π-calculus. The other component is communication-based. The main abstractions are names introduced, as usual in the π-calculus, with a scope operator and *resources*. Resources are bound to variables and may either be permanent (notation $\langle x = P\rangle$) or ephemeral, that is available only once (notation $\langle x \Leftarrow P\rangle$). The computation is driven by the *resource fetching* rules

$$(xz_1 \cdots z_k \mid \langle x \Leftarrow P\rangle) \rightarrow Pz_1 \cdots z_k$$
$$(xz_1 \cdots z_k \mid \langle x = P\rangle) \rightarrow Pz_1 \cdots z_k \mid \langle x = P\rangle$$

that describe asynchronous requests $xz_1 \cdots z_k$ for resources $\langle x \Leftarrow P\rangle$ (and $\langle x = P\rangle$) associated to a variable x.

In terms of implementation, the communication component of the Blue calculus could easily and efficiently be implemented in our virtual machine. In fact, a resource $\langle x = P\rangle$ can be seen as an instance of the template:

$$\textbf{def } X = x?\{\mathit{val} = (\tilde{y})P \mid X\}$$

A similar definition can be given for an ephemeral resource but obviously without the recursion at the end of the *val* method. The resource fetching rule then translates straightforwardly into a simple redex reduction in TyCO.

On the other hand, while the functional component of the calculus can be encoded in π and therefore in TyCO, this implementation would be far from optimal. In fact, given the amount of work already done in the implementation of functional programming languages, an implementation of Blue profiting from this work would certainly be more efficient. Moreover the virtual machine as defined is not an adequate framework to support stack based computations. Here perhaps an hybrid channel-based/functional architecture would be more adequate.

The Join calculus [20] is an asynchronous π with multi-way synchronization. It collapses the creation of new names, input and replication into a single construct called a *join pattern*. This means that receptors are defined at the same time as the names they occur in and are therefore unique. A join pattern is composed of a set of clauses, each one expressing a synchronization between several names. For example, the following two clauses define a synchronization between three names a, b and c.

$$\begin{aligned} \textsf{\textbf{let}}\ a(k) \mid b(j) &= p(k, j) \\ \textsf{\textbf{and}}\ a(k) \mid c() &= q(k) \end{aligned}$$

When messages are received at a and b (a and c) the clause $a(k) \mid b(j)$ ($a(k) \mid c()$) is activated and its guarded process p (q) is triggered. So, in Join, activating a clause rather than being a decision based on the status of a given channel, it is a decision that takes into consideration the state of a potentially large set of channels participating in the synchronization. Intuitively the problem of deciding to activate a clause is similar to pattern matching in the sense that we must take the state of each channel in the pattern and find a clause for which all its names have messages pending. Some interesting techniques have been developed to compile such synchronization patterns [19].

An efficient implementation of the Join calculus with the current virtual machine would involve major changes in its architecture. Among other things we would have to change: a) the semantics of the machine instructions that deal with reduction since now it is not sufficient that a redex is formed for a reduction to occur; b) implement the trigger mechanism for clauses, namely finding an adequate clause to activate, and; c) building the environments for the guarded processes from an arbitrary number of frames in the heap, depending on the number of names that participate in a synchronization. For these reasons we do not find the current virtual machine a suitable framework for a Join calculus implementation.

Chapter 6

Machine Optimizations

This section describes a sequence of optimizations to the abstract machine and the byte-code. Some optimizations profit from both the semantics and architecture of the abstract machine whereas others rely mostly on the compiler, namely the type inference module. Type information gathered for channels at compile time, for instance, channel usage patterns such as persistence [6, 60] and linearity [35, 36], is essential to support certain optimizations.

6.1 Threaded Code Optimization

We implement each virtual machine instruction as a parameterless C function that returns the address of the following instruction. The instructions modify the global state of the virtual machine, namely the memory areas and the machine registers.

```
typedef  void *(*INSTRUCTION)(void);

INSTRUCTION  OP() { /* C code for OP */ ; return next_instruction(); }
```

Before running a byte-code file, the virtual machine prepares it for threaded code emulation. Startup code replaces the opcodes of the byte-code instructions for the actual addresses of the virtual machine instructions – the pointers to the C functions. The mapping between opcodes and C function pointers is kept in a global array: `instruction_array[]`.

```
static INSTRUCTION instruction_array[] = { new, msgf, objf, ... }
```

To replace the opcode at offset k in the program's byte-code, pointed to by `Program` (see section 4.1.1), with the corresponding instruction pointer we perform the following operation:

```
Program[k] = instruction_array[Program[k]]
```

The startup code also translates the offsets (labels) in the byte-code into absolute memory addresses. To do so, we just add the offset at k to the base address for the program's byte-code.

```
Program[k] = Program + Program[k];
```

These operations avoid an extra indirection each time the byte-code is addressed and eliminate the need for computing addresses on-the-fly. They are performed by function `link_program_to_emulator` with a single pass on the byte-code.

```
#define  AssignLabel          *ptr = (WORD)instruction_array[*ptr]
#define  AbsoluteAddr(X)      *ptr = (WORD)(Program + (*ptr))

void  link_program_to_emulator( int size ) {
  FRAME    ptr = Program, end = Program + size;

  while( ptr < end )
     switch( *ptr ) {
         case newc:  AssignLabel; ptr+=2; break;
         case msgf:  AssignLabel; ptr+=3; break;
         case objf:  AssignLabel; ptr+=2; break;
         ...
     }
}
```

These transformations in the byte-code format before running a program allow the emulator loop to be defined in a very compact way inspired in the STG machine [33].

```
void emulate() {
    register INSTRUCTION instruction;

    init_machine_data_structures();

    /* run the program */
    instruction = first_instruction();
    while(instruction)
       instruction = (*instruction)();

    free_machine_data_structures();
}
```

The emulation loop starts with the first instruction in the byte-code given by the macro `first_instruction()` defined by:

```
#define first_instruction()      (INSTRUCTION)*(Program + 2)
```

and each instruction returns the address for the next one as given by the macro `next_instruction`. In general, given an instruction with size k, the next instruction is given by:

```
#define   next_instruction()            (INSTRUCTION)*(PC+k)
```

which advances to the immediately following instruction. With control flow and scheduling instructions the jump is made to an absolute address l in the byte-code and so we have the alternative definition:

```
#define   next_instruction()            (INSTRUCTION)(*l)
```

The machine halts when the body of the loop returns a NULL pointer. The NULL pointer is returned only whenever a vm_thread ends and the run-queue is empty.

6.2 C Code Optimizations

The functions that implement the virtual machine instructions have a very simple stack layout. We can take advantage of the fact that:

- the stack for the emulation function does not grow;
- the typical stack frame book-keeping is unnecessary.

The GNU C compiler features an optimization flag – `-fommit-frame-pointer` – that allows the user to switch off the stack frame book-keeping. Moreover, the code for the emulator is compiled with the usual `-O3` flag. A major advantage of this simple emulator architecture is also portability since it is written in ANSI C. Other emulator implementations use non-standard constructs provided by some compilers, such as `gotos` to variable addresses, which provide speedups in execution at the cost of portability and code readability.

As usual in the implementation of emulators the execution of each instruction involves an unconditional jump to a block of C code. This turns out to be the main source of overhead in the execution of byte-code programs. The machine registers are implemented as global variables. The mapping of these into hardware registers has proven to be best left to the C compiler at least for architectures with small register files.

6.3 Compact Messages and Objects

This optimization is due to Turner [66]. Given the frequency with which channels holding a single message or a single object occur at run-time the abstract machine is fine-tuned for these particular cases (see section 4.1.2). The idea is to avoid the overhead of queuing and dequeuing frames when a single object or message is present. Reduction is also speed up. In the case where a single message is held in a channel, we use the first word of the channel to hold the method label and the second word to store a reference to the first argument. In the case of an object we store the method table reference in the first word of the channel and in the second word a reference to the frame with the free variable bindings. If a message arrives while the channel is holding a single message then the channel must be reconfigured to a queue of messages. This is achieved with a single machine instruction. The same happens for queues of objects. In both cases, messages and objects will thereafter reduce according to the general case.

6.4 Optimizing Frame Sizes

The size of the frames is minimized in such a way that fields not required for a given data item are not waisted. For example, messages or objects in a linear channel do not require a `next` field, since they are used at most once. If the receiving object has only one method then the message does not require the `label` field neither. Similar arguments are valid for persistent objects and messages sent to them. Spending just one or two words less per frame can pay off since it minimizes the heap space used which in turn usually diminishes the frequency of garbage collections.

6.5 Merging Instructions

Certain instructions occur in patterns that are very common, sometimes pervasive, in the assembly for programs. This optimization merges a few basic machine instructions into a single one, thus eliminating the overhead of some unconditional jumps (in the emulation). It also improves locality. We present one such optimization where two new instructions, `trobj` and `trmsg`, are obtained from the basic instruction set. `trobj` w and `trmsg` w try to reduce an object or message, respectively, by checking the state of channel w and applying the appropriate action.

```
trobj w:
     watch     w
     state
     switch    l0,l1,l2,l3,l4
l0:  cenqueue  % case Empty
     update    OneObj
     jump      end
l1:  cdequeue  % case OneMsg
     redobj
     update    Empty
     jump      end
l2:  decompact % case OneObj
     enqueue
     update    ManyObj
     jump      end
l3:  dequeue   % case ManyMsg
     redobj
     reset
     jump      end
l4:  enqueue   % case ManyObj
end:
```

```
trmsg w:
     watch     w
     state
     switch    l0,l1,l2,l3,l4
l0:  cenqueue  % case Empty
     update    OneMsg
     jump      end
l1:  decompact % case OneMsg
     enqueue
     update    ManyMsg
     jump      end
l2:  cdequeue  % case OneObj
     redmsg
     update    Empty
     jump      end
l3:  enqueue   % case ManyMsg
     jump      end
l4:  dequeue   % case ManyObj
     redmsg
     reset
end:
```

`trobj` w checks the state of channel w. If the state is either `OneMsg` or `ManyMsg` then it dequeues a message frame and, as a result of the reduction, creates a new runnable vm_thread in the run-queue. If the channel state is either `Empty` or `OneObj` or `ManyObj` the current frame is enqueued. `trmsg` w is the dual macro for messages. It checks the state of channel w. If the state is either `OneObj` or `ManyObj` then it dequeues an object frame and creates a new runnable vm_thread in the run-queue. If the channel state is either `Empty` or `OneMsg` or `ManyMsg` the current frame is enqueued. Notice the symmetry of the code. To apply the optimization, instead of writing these instructions as blocks of independent assembly instructions, we collapse the block into a single new instruction implemented as a C function. This short cuts the code as it avoids all the intermediate unconditional jumps required for the execution of individual machine instructions. The corresponding C code for `trobj` w is written as follows:

```
trobj() {
   CC = w;
   switch( STATE() ) {
      case   Empty: CENQUEUE(); UPDATE(OneObj); break;
      case  OneMsg: CDEQUEUE(); REDOBJ(); UPDATE(Empty); break;
      case  OneObj: DECOMPACT(); ENQUEUE(); UPDATE(ManyObj); break;
      case ManyMsg: DEQUEUE(); REDOBJ(); RESET(); break;
      case ManyObj: ENQUEUE(); break;
           default: vmerror("illegal state in channel");
   }
}
```

where the uppercase operators are macros with the C code for the body of the instructions with the same name as described in section 4.1.7. Other possible optimizations can be considered and depend on a profile analysis of the virtual machine code and the analysis of common patterns in the assembly code.

6.6 Single Method Objects

Objects with a single method do not require a method table. The type system can provide the information that objects at certain channels have only one method. With this information the compiler can produce code for the object without the byte-code for the method table. The reference to the method table in the object's heap frame is replaced with a direct reference to the method. The optimization avoids one extra indirection for method invocations in the objects and to a lesser extent reduces the byte-code size.

Specializations of the reduction instructions `redobj` and `redmsg` are required to support this optimization. `redobjs` reduces a single method object pointed to by `CF` with a message pointed to by `OF`. The dual, `redmsgs`, reduces a message pointed to by `CF` with an object with a single method pointed to by `OF`.

```
redobjs() {
   EQ->code  = CF->table;
   EQ->args  = OF;
   EQ->fvars = CF;
}
```

```
redmsgs() {
   EQ->code  = OF->table;
   EQ->args  = CF;
   EQ->fvars = OF;
}
```

Notice how the byte-code reference for the new thread is directly extracted from the object frame at `CF` in `redobjs` and at `OF` in `redmsgs`, whereas in the common reduction a table lookup is required. Finally, we can define communication macros similar to those in section 6.5, for objects that we know have only one method.

6.7 Fast Reduction

This optimization can be performed when we can assure that a given channel will have exactly one object. The important cases studied here are uniform receptors [60] and linear synchronization channels [36]. The main point of this optimization is that we never allocate a channel in the heap to hold an object or message frame. We just create a frame for the object or message in the heap and use its pointer directly as a binding.

In the non-optimized case, a communication (object-message reduction) immediately places a new thread in the run queue with a pointer to the byte-code of the invoked method and pointers to the frames in the heap that hold the parameter and free variable bindings for the method. In general, trying to reduce a message in a given channel involves switching on the state of the channel and either enqueueing the message or creating a new thread. Since we have no information on the number and arrival order of objects and messages we need a channel queue to keep them while communication is not possible. Figure 6.1 shows the case of reduction of a message. The code that switches on the state of the channel is presented in section 6.5, instruction `trmsg`.

Uniform receptors [60] are channels where a single persistent object (in the sense that it survives reduction) occurs and it may receive an arbitrary number of messages. Messages to these channels can only be issued after the object has been placed in the channel. This must be verified statically by the type inference system.

Reduction occurs exactly k times in linear channels [36] of multiplicity k. Of special interest are linear channels with multiplicity k=1 since they include the important case of synchronization channels. These are used pervasively in functions and pattern matching constructs, among others.

This optimization profits from channel usage information gathered by the type system to simplify the reduction instruction, avoiding the switch altogether and, most importantly, avoiding the allocation of a channel to hold the objects and messages. The later is quite interesting from a performance point of view given the pervasiveness of linear channels and uniform receptors in programs.

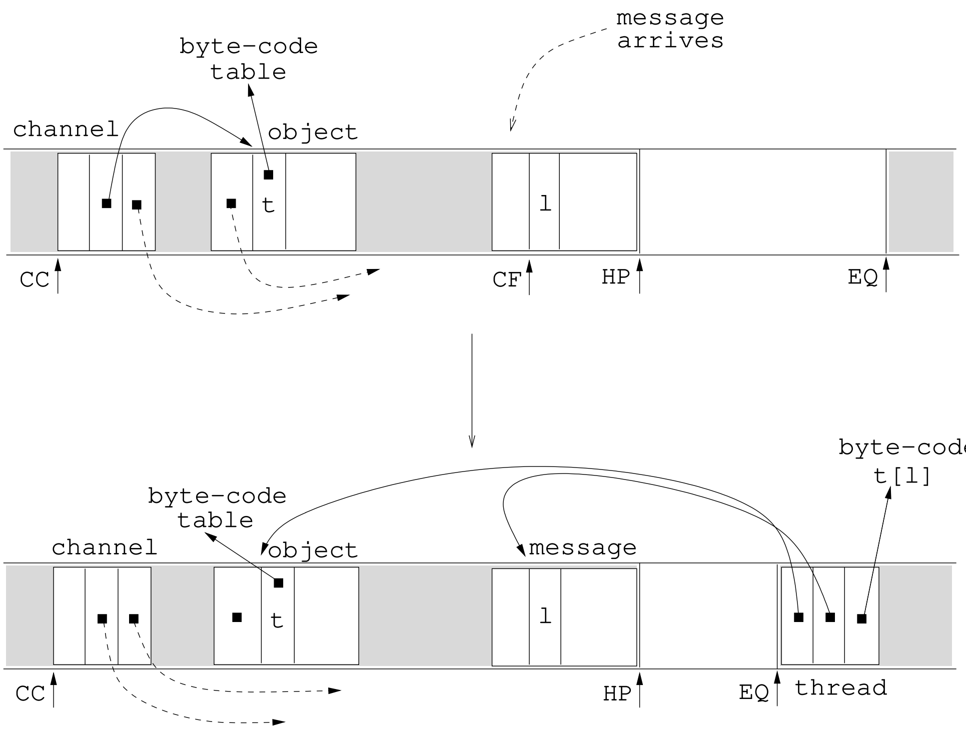

Figure 6.1: Non optimized message reduction.

Here we discuss the reduction on persistent and synchronization channels where the object always appears before any message. This configuration is very common in programs and the compiler can frequently re-order code within a thread to enforce this condition. As a result, no test is required for reduction to occur. We apply the following transformations:

- instead of allocating channels in the heap, we directly use the pointers to message and object frames in the heap to handle reduction. The pointers are passed as free variable or argument bindings in the program and do not require any permanent heap storage;
- we simplify the code for reduction avoiding the switch;
- we inline the arguments of the message in the run-queue.

Two new instructions handle this kind of reduction. The object comes first and we keep a pointer to its frame in the heap. When a message arrives (the only one in the case of a linear channel) it immediately reduces with the object. No check is required, unlike the switch in section 6.5.

Also, since we know that any incomming message to these channels always reduces, we place the arguments of the message in the run-queue rather than in the heap. This allows the space taken by the arguments to be reclaimed immediately after the execution of the corresponding thread. Figure 6.2 illustrates the optimization.

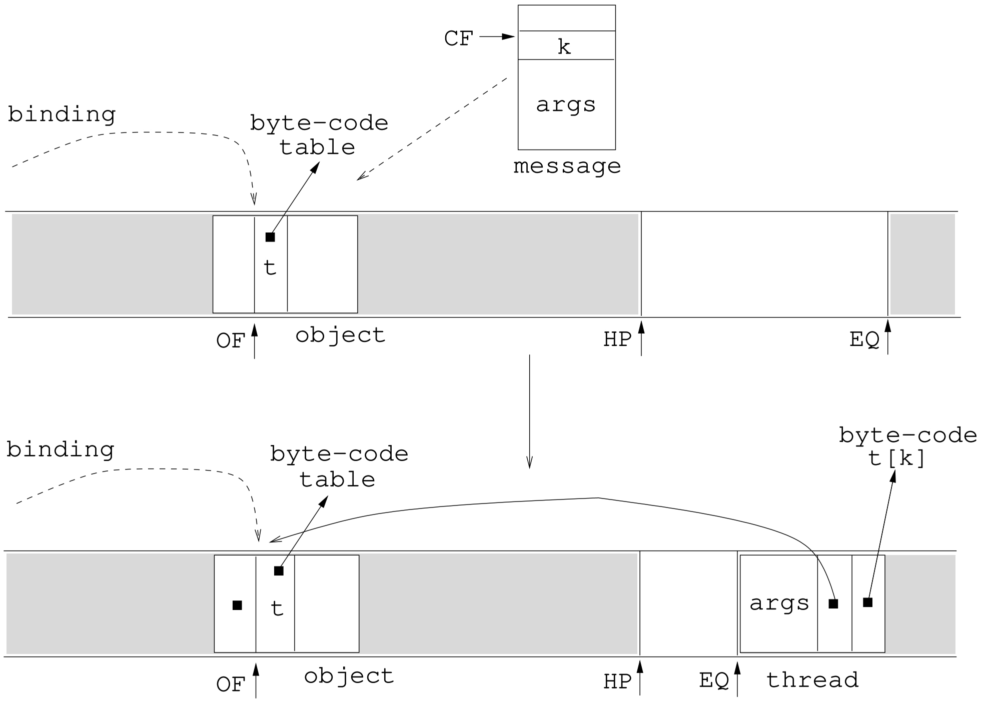

Figure 6.2: Fast reduction.

The specialized instruction `fast_redmsg` w,k,n immediately reduces a message with label k and n arguments with an object pointed to by w.

```
fast_redmsg() {
   EQ->code = w->table[k];
   EQ->fvars= w;
   CF       = EQ;
   EQ      -= n;
}
```

A sequence of `put` or `pop` instructions then copies the arguments to the run-queue. The instruction `fast_redobj` w, on the other hand, simply keeps the heap pointer for an object frame at word w.

```
fast_redobj() { w = CF; }
```

What distinguishes persistent from linear synchronization channels, from an implementation point of view, is that we use recursion to model persistence. A persistent object always has a self referencing pointer in its closure when it is placed in the run-queue. In other words, all the methods of such an object must recur to keep the object alive and so, the *self* channel is preserved as a free variable of the object's method collection. This preserves the frame of the object if garbage collection is triggered. In the case of a linear channel this link cannot exist and so, the object frame only lasts until the thread resulting from the reduction is executed.

Note that, in the general case where it is not known whether an object or a message will come first, it is required that both instructions test the address held in w to check if a redex component is already present. The instructions for uniform receptors and linear channels then differ slightly. This is because an uniform receptor may receive an arbitrary number of messages and so it needs to enqueue them. A linear channel receives exactly one message.

6.8 Recursive Templates

TyCO uses recursion pervasively. A compiler can check statically whether a given template definition is recursive, for example:

def X = (x y) ... | X[y x] **in** X[1,2]
def X = (x) x ? {l_1 = (y) X[y], l_2 = (y z) x![y] | X[z] } **in** **new** a X[a]

In these cases we may optimize the assembly code to reuse the argument or free variable frames for the tail instantiation. Note that this optimization can only be applied if the parameter or free variable frame for the template is allocated in the heap, not the run-queue as is usually the case in the virtual machine, since in the later case the space used by the frame is reclaimed immediately after the thread ends.

For example, for the first case we have the following transformation:

```
main  = {
   def X1(2) = {
      % ...
      tail     X1     % as opposed to     instof   2,X1
      push     p0     %                   put      1,p1
      put      p0,p1  %                   put      2,p0
      pop      p1     %
```

```
   }
   instof   2,X1
   put      1,1
   put      2,2
}
```

The instruction `tail` differs from `instof` in that it does not create a new heap frame for the arguments of the instantiation. Notice that the push and pop instructions are required so that we can swap the values of `p0` (`x`) and `p1` (`y`) safely. Normally we would just need two `put` instructions. For the second example, with an object, we would have the code:

```
main  = {
   def X1(1) = {
      objf     1
      put      3,p0
      trobj    p0 = {
         { l_1, l_2 }
         l_1 = {
            tail     X1      % as opposed to   instof 1,X1
            put      f1,p0   %                 put    1,p0
         }
         l_2 = {
            msgf     1,0
            put      3,p0
            trmsg    f0
            tail     X1      % as opposed to   instof 1,X1
            put      f1,p1   %                 put    1,p1
         }
      }
   }
   newc     c0
   instof   1,X1
   put      1,c0
}
```

The attributes of the new object are now copied to the free variable frame (the `f1` reference in the `put` instructions). This is because, syntactically, the attributes of the object are always free variables in the context of the method.

Also note the more general use of the instructions `put` and `pop`. `put 0,p1` means "place the value at `p1` at offset `0` in the current frame – pointed to by `CF`". `put p0,p1` means "place the value in word `p1` at word `p0` of the parameter frame". Similarly, `pop 1` means "place the value on top of the operand stack in word `1` of the current frame". `pop p1` means "place the value on top of the operand stack in word `p1`, of parameter frame".

6.9 Gathering Partial Results

Consider the following synchronization pattern in TyCO where we collect the results of invoking several methods and only then proceed with the execution of P:

$$\textbf{gather}\ \tilde{\mathsf{y}}_1 = \mathsf{x}_1!\mathit{l}_1[\tilde{\mathsf{e}}_1]\ \ldots\ \tilde{\mathsf{y}}_k = \mathsf{x}_k!\mathit{l}_k[\tilde{\mathsf{e}}_k]\ \textbf{in}\ \mathsf{P} \stackrel{\text{def}}{=}$$
$$\textbf{let}\ \tilde{\mathsf{y}}_1 = \mathsf{x}_1!\mathit{l}_1[\tilde{\mathsf{e}}_1]\ \textbf{in}\ \ldots\ \textbf{let}\ \tilde{\mathsf{y}}_k = \mathsf{x}_k!\mathit{l}_k[\tilde{\mathsf{e}}_k]\ \textbf{in}\ \mathsf{P}$$

Synchronization patterns such these are very common. Process P is executed when every y_i is bound to a value. Typically such a synchronization would involve k intermediate threads, frames and linear channels. Since the destination of the values is P, the compiler with the help of the type-system may collapse the k inputs into a single special input of the form:

$$\mathsf{replyTo}\ ?\ \mathit{val}{=}\ (\tilde{\mathsf{y}}_1\ \ldots\ \tilde{\mathsf{y}}_k)\ \mathsf{P}$$

where replyTo is a linear channel that holds the closure for P with its free variables, and a single frame where all arguments from the k messages will be stored. Messages sent to any of the x_i are replaced with messages to replyTo at compile time. Each of the incoming messages fills a part of the argument frame at replyTo. When the message frame is full, it reduces immediately with the closure for P. The resulting thread is placed in the run-queue. This optimization, saves $k-1$ linear channels and intermediate closures, and $k-1$ additional threads in the run-queue.

6.10 Persistent Data

In TyCO, user defined datatypes are implemented as recursive objects. These objects have a set of attributes that define their state, and have a single method that receives a synchronization channel and sends a message to that destination with a label indicating its internal structure and the attributes as arguments. The state is immutable. For example, the definition of a list is as follows:

```
def Nil = (self) self ? (replyTo) replyTo!nil[] | Nil[self] in
def Cons = (self head tail) self ? (replyTo) replyTo!cons[head tail] | Cons[self head tail]
in ...
```

Interacting with a datatype value involves three frames:

- an object instance that defines the datatype value; this is created once but remains for the rest of the program life;
- the message sent to the value by someone enquiring about its structure and state (for example, a pattern matching construct);
- the reply message sent by the value to the enquirer. This is allocated every time an enquiring message is received. For a given value, this message is always the same since the state does not change.

The reply message from the datatype value is not returned at once since the thread resulting from the reduction of the first two items is placed in the run-queue rather than executed immediately.

The main idea in this optimization is to short cut this process. When we create a datatype value, rather than allocating a channel to hold the object, we bind the variable that represents the datatype with a pointer to its *constant* reply message. This message is a frame in the heap built when the value was created.

In the above example, if we were to create a cons value, instead of building the object self ? (replyTo) replyTo!*cons*[head tail] | Cons[self head tail] with free variables head and tail, we instead keep a pointer to a heap frame with the persistent message: *cons*[head tail].

To apply this optimization, the compiler must first detect that a given message is being sent to a datatype value represented as a heap frame. It then generates code that simply copies this frame to the reply channel replyTo. For each datatype value, this optimization eliminates the need for the object frame for the value and avoids the initial synchronization between the first enquire message and the value.

6.11 Adaptive Channels

Each channel allocated in the heap behaves in a certain way, reflecting its internal state. The instructions that process redexes, upon receiving a new object or message in a channel, first check the state of the channel before performing an appropriate action. The usual way to implement this is by switching on the channel state. This is because the same instruction must work for an arbitrary number of channels with arbitrary channel states. This has a major disadvantage as a test is always performed even if the behavior of the channel is constant, that is, the channel remains in the same state after some point in the execution. For example, an uniform receptor stays in the same state after the arrival of an object. From then on it reduces any incomming message. Linear channels only admit two possible usage patterns: message followed by object or vice-versa. Quasi-linear channels allow multiple linear interactions, one at a time.

The idea of this alternative channel representation is to encode the behavior directly in channels. In this way we are able to dynamically change their behavior, reacting to run-time events (for example, the arrival of an object to an uniform receptor). Since channels react only to two kinds of events – the arrival of a message and the arrival of an object – we need two instruction pointers per channel. One points to the instruction that is executed if the channel receives a message and the other to the corresponding instruction for the reception of an object.

channel
`WORD descriptor`
`INSTRUCTION msg_behavior`
`INSTRUCTION obj_behavior`
`FRAME first`
`FRAME last`

In this scenario we have just two instructions in the assembly that deal with the arrival of an object and of a message to a channel. `msg` w takes a channel as an argument and executes the channel behavior function for an incomming message. `obj` w takes a channel as an argument and executes the channel behavior function for an incomming object.

```
msg() { CC = w; *(CC->msg_behavior)(); }

obj() { CC = w; *(CC->obj_behavior)(); }
```

Each behavior function that is run changes the behavior pointers in the channel, at the end of its execution, to reflect the new state of the channel. The behavior transitions are described in detail below.

The possible behaviors of a channel can be visualized in the form of automata in which each state is an internal state of the channel and the transitions induced by incoming messages m or objects o. The final state of the automata reflects the state of the channel at the end of the program. The φs represent the C functions that implement the transitions.

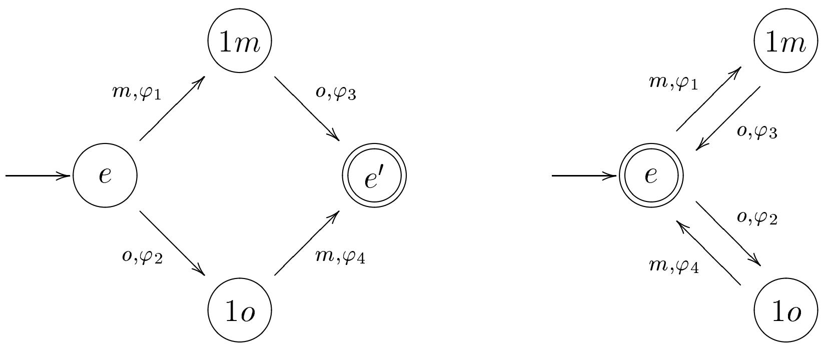

Figure 6.3: Automata for linear and quasi-linear channels.

For example, the first automata in figure 6.3 simulates the behavior of a linear channel. The states e and e' are both empty. e' is final and there is no transition from it. In the second example in figure 6.3, a quasi-linear channel, the empty state is the same as several linear interactions may occur at the channel.

The behavior transitions can be visualized easily in a table in which for each state and incoming symbol m or o, we assign a pair of φ functions that implement the new behavior. The symbol $\bullet$ stands for an undefined behavior. It is used in cases where a transition would result in an error, for example, the arrival of a second object to an uniform receptor channel. From an implementation point of view it can be quite useful, for example, for debugging. The example in table 6.1 is for a quasi-linear channel and the initial behavior of the empty channel is (φ_1,φ_2).

	m	o
e	$(\bullet, \varphi_3)$	$(\varphi_4, \bullet)$
$1m$	$(\bullet, \bullet)$	(φ_1, φ_2)
$1o$	(φ_1, φ_2)	$(\bullet, \bullet)$

Table 6.1: The changing behavior of a quasi-linear channel.

In the case of an uniform receptor any number of messages may be received until an object arrives ($m*$). At that point, the automata reaches a final state $1o$. Since the object is persistent, it will remain in the channel until the end of the program. Any incoming message is reduced and loops back to the final state. Only one object is allowed in the channel (figure 6.4).

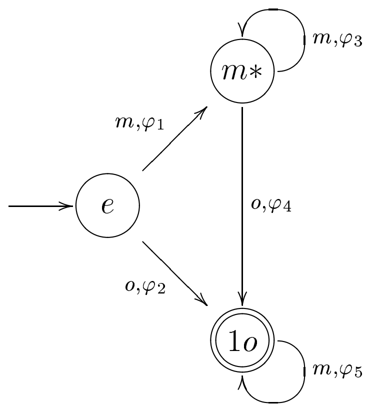

Figure 6.4: Automata for an uniform receptor channel.

The table of behavior transitions for an uniform receptor is built similarly and is presented in table 6.2. The initial behavior of the empty channel is (φ_1,φ_2).

	m	o
e	(φ_3,φ_4)	$(\varphi_5, \bullet)$
$m*$	(φ_3,φ_4)	$(\varphi_5, \bullet)$
$1o$	$(\varphi_5, \bullet)$	$(\bullet, \bullet)$

Table 6.2: The changing behavior of a uniform receptor.

Several usage patterns for channels may be analyzed in this way and optimized. Besides allowing channels to efficiently change their behavior dynamically this optimization also has another important advantage. It breaks down the reduction instructions into a set of faster C functions that implement a behavior separately. This improves the distribution of machine functionality over the instruction set and also contributes to homogenize the granularity of the instructions.

Chapter 7

Performance Evaluation

The current implementation of the TyCO programming language includes a source to assembly compiler, a byte-code assembler, and a byte-code emulator. We have chosen to separate the intermediate assembly code generation from the byte-code generation to allow us more flexibility namely in mapping other concurrent languages directly in the assembly and for the debugging and visualization of running programs. The emulator is a very compact implementation of the abstract machine with about 4000 lines of C code. Figure 7.1 illustrates the data-flow in the TyCO system. We used

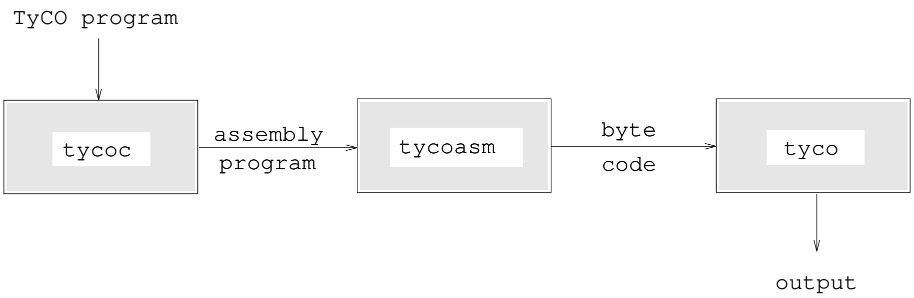

Figure 7.1: The TyCO system

a set of programs to measure the efficiency of the current implementation relative to the concurrent programming languages Pict [54], Oz [43] and JoCaml [32].

7.1 The Programs

The programs we used for the evaluation of the system include some typical functional programs such as `tak` and `queens`, as well as more complex programs such as `mirror` or `fourier` that manipulate large object-based data-structures. We describe the programs briefly and the parameters we used.

`tak` k n m computes the Takeuchi numbers using the deeply recursive functional definition presented in section 2.6. Here we take the values k=22, n=16 and m=8 for arguments.

`queens` n solves the n-queens problem for a square board of size n. The algorithm performs a search over the solution space, incrementally constructing configurations. A check is made on the legality of a configuration each time a new position is inserted. Illegal configurations are dropped. Here we take n=10.

`hanoi` n solves the problem of the Towers of Hanoi for n disks using three needles. The objective is to move the disks between two of the needles using the other as an intermediary in such a way that: a) only one disk can be moved at a time and; b) no disk can be placed on top of a smaller one. The program outputs the number of steps required to move the n disks. We take n=15.

`sieve` n computes all the prime numbers, up to the natural number n, using the Sieve of Erathostenes algorithm. The program generates a chain of prime numbers by keeping a chain of objects with all the prime numbers it finds. Each integer up to n is sent through this chain and is tested against each of the primes in it. If it fails a test it is discarded; otherwise it is inserted at the end of the chain. Here we take n=10240.

`mirror` n takes a tree with n nodes, leafs and buds, and builds its mirror image. It uses objects to construct the tree data structure and for pattern matching and is deeply recursive. We take n=10240.

`graph` n takes a connected graph, implemented as a list of list of nodes (objects with an integer attribute) and maps a function (in this case fib) on the graph. Each node of the graph is an object with an integer attribute. The computation time for each node is exponential on the integer attribute and, as the distribution of the integers in the graph is (pseudo-) random the workload in the graph is highly heterogeneous, with the calls to the function overlapping in the computation. We take n=128.

`fourier` n takes a signal of size n (a list of complex numbers implemented as objects) and a table of complex number coefficients of size n^2, and computes the Discrete Fourier Transform of the signal. The program uses floating point arithmetic and performs a large number of method invocations. Each component of the Fourier transform is computed in parallel. We take n=64.

We have implemented all programs in TyCO 0.1, Pict 4.1 [54], Oz 2.0.4 [43] and JoCaml [32] to compare the performance of these systems.

7.2 The Experiments

We have conducted a series of experiments to evaluate the performance of the system. The results of these experiments provide important information that we used to fine-tune the implementation and compare with the performance of other systems under similar conditions. Nevertheless, a full performance evaluation will have to use larger, *real world* applications [52]. Large applications commonly expose overheads in the virtual machine, namely in the way it interfaces with the underlying hardware (registers, cache, memory access) that are difficult to observe with small programs.

In the sequel we will refer to the heap size in units of kilo-words (kw) or mega-words (Mw). The current implementation uses 32-bit words. All the following results have been obtained, unless otherwise noted, with a default heap space of 256 kw, in a 233MHz Pentium II machine with 128Mb RAM, running Linux.

7.2.1 Raw Speed

Table 7.1 shows the smallest execution time for 10 consecutive runs for each of the programs. With Oz we used the switches `+optimize` and `-threadedqueries` to get full performance from the Oz code. Both Pict, JoCaml and the TyCO virtual machine were compiled with `-O3 -fommit-frame-pointer` optimization flags. The initial results show that TyCO's speed is clearly in the same order of magnitude as Pict and Oz, and indeed compares favorably considering it is emulated code. TyCO is clearly faster than JoCaml as can be seen in the rightmost portion of the table. These results were obtained for distinct problem sizes, relative to the first set, as for some programs JoCaml ran into some problems either compiling (for example, `graph`) or running them (for example, `queens` 10). The performance gap to Pict and Oz is higher than

Program	Pict	Oz	TyCO	Program	JoCaml	TyCO
10*`tak` 22,16,8	4.4	11.7	24.3	`tak` 22,16,8	7.0	3.0
10*`queens` 10	17.0	33.1	58.1	`queens` 8	2.5	0.4
10*`sieve` 10k	9.9	19.2	20.9	`sieve` 4k	26.4	0.6
10*`mirror` 10k	3.1	4.2	1.2	`mirror` 10k	2.5	0.2
10*`graph` 128	9.6	7.1	7.2	`graph` 128	—	—
10*`fourier` 64	2.3	2.6	0.9	`fourier` 64	0.9	0.2

Table 7.1: Execution time (in seconds)

average in the case of functional programs a fact that is explained by the optimized code generated by both Pict and Oz for functions. Further optimization, namely with information from the type system may allow this gap to diminish.

The performance ratio for applications that manipulate large numbers of objects (with more than one method), on the other hand, clearly favors TyCO. For example `mirror`, which uses objects to implement a large tree and to encode pattern matching, performs nearly three times faster than Pict and even more for Oz and JoCaml. Also notice that all the Oz programs required an increase in the heap size up to 3 Mw and once (`queens`) to 6 Mw to terminate. Compare this with the very conservative 256 kw used by both in TyCO and Pict. The exception for Pict is `fourier` where there is a lot of parallelism and method invocations. Pict required 1.5 Mw to run the program, as opposed to 256 kw in TyCO, and was about 2.3 times slower than TyCO. `fourier` shows that objects in both Pict and Oz are clearly less efficient than in TyCO. Pict showed lower performance on the object based benchmarks. This results from the fact that the encoding of objects into the base π-calculus is rather inefficient both in speed and heap usage. The `graph` program represents an intermediate situation where a part of the computation is object based and the other part is functional. The functional portion of the computation allowed Oz to perform slightly better than TyCO. Pict showed lower performance on the programs using large numbers of objects. The inefficiency results from the two messages required for each method invocation and more frequent garbage collections.

In summary, the results obtained show that on average the current performance of TyCO is comparable to that of Pict or Oz. Also a comparison with JoCaml, another byte-code implementation, reveals that TyCO is performing quite well at this stage and there is still a lot of work that can be done in optimizing the emulator and generated byte-code.

7.2.2 Size of Executables

Table 7.2 shows that byte-code files for the TyCO abstract machine are very compact except for cases where the data structures are included statically in the programs, such as in `mirror`, `graph` and `fourier`. For the moment TyCO lacks the functionality (for example, file I/O, module libraries) to allow these data structures to be generated by the program itself. If that were the case the byte-code in the last three examples would be considerably smaller, while the size of the Pict and Oz files would continue to be limited by the their run-time engines. In the remaining cases the byte-code

	executable-code (kb)		
	Pict	Oz	TyCO
`tak` 22,16,8	47	238	~ 1
`queens` 10	48	238	~ 2
`sieve` 10k	48	239	~ 1
`mirror` 10k	721	354	432
`graph` 128	383	312	123
`fourier` 64	1804	541	286

Table 7.2: Size of the executables.

files are rather compact, typically less than 5% the size of Pict or Oz binaries. The binary for the abstract machine itself uses only 39kb. Thus the entire system is very lightweight. An architecture independent, lightweight system running very compact byte code provides in our opinion the ideal starting point for the introduction of distribution and mobility of code, which is one of our initial requirements.

7.2.3 Heap Usage

Table 7.3 presents heap usage statistics for TyCO and Pict, for each program, using the default system configuration. The total amount of heap used for each program is given in machine-word units. The size of a word in both TyCO and Pict is 32 bits.

Table 7.3 shows that TyCO uses less heap space than Pict except for the highly recursive functional application – `tak`. This fact is particularly obvious for programs with object based data-structures. TyCO performs more *shift* garbage collections (see section 4.2) since it uses the run-queue to store the arguments of instances and some messages directly (see section 6.7) and, on average, TyCO uses one extra word per

	TyCO			**Pict**		
Program	heap(kw)	shift	full	heap(kw)	shift	full
`tak` 22,16,8	13487	406	18	11496	225	32
`queens` 10	12975	591	13	13933	390	55
`sieve` 10k	11110	218	31	11941	253	36
`mirror` 10k	505	4	0	1094	27	5
`graph` 128	3336	101	5	5769	139	25
`fourier` 64	538	18	1	4870	2	0

Table 7.3: Total heap usage and number of garbage collections.

heap frame. This increases the number of collisions with the top of the heap. On the other hand the number of *full* garbage collections performed is substantially smaller in TyCO. This is mostly due to the combined effect of the inlining of arguments in the run-queue and, especially, to the fact that TyCO does not produce run-time heap garbage in the form of unused channels or process closures. Pict produces significant amounts of heap garbage in applications where objects are pervasive since each object is modeled with a *server* channel for the object, one channel for each method and one process closure per method. Most of the times only a subset of these will actually be used. On the other hand an object in TyCO just requires a channel (*self*) and one closure for the method collection and the free variables. This effect is plainly visible in `mirror` for example. In `fourier` this effect is clearly visible as an apparent anomaly in the table. Despite performing just 2 shift garbage collections and no full garbage collections, Pict required nearly ten times the amount of heap to run the program.

This data shows that TyCO performs quite well in heap usage relative to Pict, the closest project to ours. Programs that map naturally in objects with more than one method perform significantly better.

7.2.4 Reductions, Performance and Optimizations.

Table 7.4 shows the effect of some optimizations on the execution time (in seconds) for the programs. The optimizations we explore in these examples relate to the way communication channels are used and have been previously discussed in chapter 6. The TyCO compiler uses data gathered by the type inference system on channel usage patterns to guide the code generation phase. Each optimization is referred to by a compiler switch:

`-s` enables the compiler to generate byte-code offsets directly, rather than method tables, for objects with a single method;

`-l` switches on the optimization for linear synchronization channels. Exactly a single message and object interact in these channels, and;

`-p` optimizes for persistent objects. These objects have exclusive use of a channel and any incoming message reduces immediately to a thread.

The last column shows the fraction of speed improvement of the programs compiled with the `-slp` switch relative to the unoptimized case. It is computed with the formula: $f = (t_{unopt} - t_{opt})/t_{unopt} \times 100$.

Experience showed that the optimization implemented with switch `-s`, enabling the compiler to shortcut method table lookup when an object has only one method, was not particularly effective. The additional indirection avoided by this optimization seems to be quite efficiently performed by the C compiler. Only the results for the other switches are presented.

	`-`	`-sl`	`-sp`	`-slp`	f
10*`tak` 22,16,8	29.45	24.34	29.40	24.31	17
10*`hanoi` 15	33.33	20.36	32.85	20.23	39
10*`queens` 10	62.36	58.17	62.24	58.09	7
10*`sieve` 10k	20.99	21.08	20.99	20.92	0
10*`mirror` 10k	2.10	1.83	1.43	1.18	44
10*`graph` 128	11.48	9.26	9.15	7.24	37
10*`fourier` 64	1.77	1.40	1.25	0.91	49

Table 7.4: The effect of optimizations on the execution time (in seconds).

Synchronization appears pervasively in programs, mostly in the form of function calls and pattern matching. In pure functional programs such as `tak` all synchronization is in the form of function calls. Other applications with object based data-structures have a combination of these synchronization patterns, with pattern matching being dominant. The results from table 7.4 show an improvement of up to 40% in the case of `hanoi` in the speed of functional programs. Programs using object based data-structures benefited heavily from the optimization for persistent objects as can easily be seen in the column for `-sp`. After the synchronizations in these programs were also optimized the combined effect of both optimizations was quite dramatic as can be observed in column `-slp` with gains in efficiency of up to 49% in `fourier`.

Table 7.5 shows the impact of the same optimizations on the number of abstract machine instructions executed per program. The numbers are the instruction counts in units of one thousand. The observed decrease in the number of instructions is due basically to two factors: a) some instructions are actually eliminated as a result of the optimizations for linearity and persistence, and; b) some sequences of instructions are merged together (see section 6.5), in the cases where this optimization is possible.

	-	-sl	-sp	-slp
`tak` 22,16,8	22868	20151	22868	20151
`hanoi` 15	14745	12975	14745	12975
`queens` 10	47529	46378	47529	46378
`sieve` 10k	13075	13075	13075	13075
`mirror` 10k	7966	7044	6302	5379
`graph` 128	66396	59324	58104	51032
`fourier` 64	9434	8321	7828	6714

Table 7.5: The effect of optimizations on the number of instructions.

Both factors help the increase in performance in two ways: a) there are less instructions to execute and so the emulator performs fewer unconditional jumps, and b) the new merged and simplified instructions allow a better use of heap resources, namely the transfer of some data directly to the run-queue. The emulator becomes faster and memory allocation is more efficient.

7.2.5 The instruction mix

The following tables provide some information about the relative use of each type of instruction in typical run-time conditions. This analysis is important to detect what kind of instruction gets executed more often. Optimizing these instructions for speed or trying short cuts to avoid them in the first place may lead to significant performance gains. Table 7.6 shows the relative weight of each kind of instruction at run-time for each program, running on a non-optimized virtual machine. For each program we present, in units of one thousand, the total number of machine instructions, and the total number of instructions of each kind. The percentage relative to the total number of instructions is given between parenthesis for each kind of instruction.

We choose to partition the instructions the same way as they are presented in section 4.1.7. `Alloc` instructions allocate frames in the heap for channels, messages, objects and instances. `Move` instructions copy data items between heap frames, push values from the heap to the operand stack, and pop values from the operand stack to the heap. `Flow` instructions control the program flow. We divide reduction instructions into instantiation instructions – `Inst`, and communication instructions – `Comm`. Finally, primitive instructions operate on builtin data-types and include IO.

	total	`Alloc`	`Move`	`Flow`	`Inst`	`Comm`	`Prim`
`tak`	22868	2717(12)	10642(46)	5208(23)	906(4)	1811(8)	1585(7)
`hanoi`	14745	1769(12)	6029(40)	3015(20)	590(5)	1180(9)	2163(14)
`queens`	47529	1151(2)	22288(49)	7264(15)	1546(3)	767(1)	14513(30)
`sieve`	13076	1619(13)	5733(43)	2457(18)	10(0)	1618(13)	1638(13)
`mirror`	797	215(28)	279(35)	123(15)	0(0)	154(19)	26(3)
`graph`	6639	1160(18)	2659(40)	1302(19)	19(0)	924(14)	575(9)
`fourier`	943	186(19)	404(42)	143(16)	6(0)	136(15)	68(8)

Table 7.6: The instruction mix (non-optimized virtual machine).

Table 7.6 shows that most of the instructions executed by programs move data between memory areas. Next come the control flow and heap allocation instructions. Finally, the instantiation and communication instructions have a comparatively small weight in the total count. The high number of move operations results mainly from the natural passage of variable bindings as message or instance arguments, or as object free variables. Each of these instructions moves a single data item from the original frame to the destination frame. The fact that builtin expressions are evaluated in an operand stack adds to the total amount of move instructions. The option to use an operand stack, rather than a set of scratch abstract machine registers, to perform expression evaluation was taken with a view to produce more compact byte-code.

A possible optimization for the current implementation may merge a group of `Move` instructions into a single more complex instruction that moves all the required data items. This has an obvious problem with a stack based implementation if some of the values to be moved result from evaluation of expressions. All the expressions would have to be computed first and kept at the top of the operand stack. Then we could move the values to the destination frame.

When we apply the optimizations for specific channel usages, we get a distinct instruction mix that is represented in table 7.7. Notice how in some of the programs the

	total	Alloc	Move	Flow	Inst	Comm	Prim
tak	20151	0(0)	10642(52)	5208(25)	906(6)	1811(9)	1585(8)
hanoi	12976	0(0)	6029(46)	3015(24)	590(5)	1180(9)	2163(16)
queens	46378	0(0)	22288(50)	7264(15)	1546(3)	767(1)	14513(31)
sieve	13076	1619(12)	5734(44)	2457(19)	10(0)	1618(12)	1638(13)
mirror	538	31(5)	236(44)	123(23)	0(0)	123(23)	26(5)
graph	5103	38(0)	2452(48)	1302(25)	19(0)	717(15)	575(12)
fourier	671	12(1)	330(49)	143(22)	6(0)	111(17)	68(11)

Table 7.7: The instruction mix (optimized virtual machine).

allocation instructions seem to have vanished or nearly so while the relative weight of the other instructions has remained relatively stable. This is not to say that these programs do not allocate heap memory. Rather, the optimizations for linear synchronization channels and persistent objects eliminate the need for much of the frame allocation seen in non-optimized code. The linear channel optimization for example eliminates completely the need for an allocation of linear channels in the heap. When this is combined with the inlining of arguments in the run-queue, we see that most of the allocation takes place directly in the run-queue, rather than in the heap itself. This is even more obvious in functional examples were most of the reductions are instantiations. Moreover, when the above cited optimizations are applied the frame allocation and reduction instructions are merged into simplified communication instructions thus contributing to the observed effect.

7.2.6 Average Thread Size

Table 7.8 presents in the second and third columns, respectively, the total number of instructions and reductions performed by TyCO when fully optimized. The number of instructions executed by the virtual machine is exactly the number of calls made by the code `instruction = (*instruction)()` in the emulator loop. The number of reductions performed by the machine is also easily computed since it coincides with the the number of vm_threads scheduled from the run-queue to execute a program. The ratio of these two values gives us the average number of virtual machine instructions per vm_thread. This value is presented in the last column.

From table 7.8 it can be easily observed that in most programs the average granularity of the threads is about 10 virtual machine instructions, with `queens` and `sieve` with

	instructions	reductions	ratio
tak 22,16,8	20151491	1811370	11.1
hanoi 15	12976105	1179646	11.0
queens 10	46378386	1929622	24.0
sieve 10k	13075839	818533	15.9
mirror 10k	538000	61461	8.8
graph 128	5103248	481039	10.6
fourier 64	671485	68133	9.9

Table 7.8: Instructions, reductions, and average thread size.

clearly higher averages. Nevertheless the average vm_thread is quite fine grained.

7.2.7 Average Emulation Overhead

The next step in this analysis of system performance tries to establish some bounds on the overhead resulting from the emulation itself. We tried to measure the amount of overhead generated by this emulation strategy using a small test program that has a similar emulation cycle but always invokes the same function. An artificial addition was introduced in the body of the function as a way to simulate the overhead of adjusting the program counter for the next instruction, as is done in the actual machine. Running this small program with optimizations (`-O3 -fomit-frame-pointer`), we found that the emulation overhead is approximately 34 nano-seconds per machine instruction.

	instructions	time(s)	cost %
10*tak 22,16,8	201514910	24.31	28%
10*hanoi 15	129761050	20.23	22%
10*queens 10	463783860	58.09	27%
10*sieve 10k	130758390	20.92	22%
10*mirror 10k	5380000	1.18	15%
10*graph 128	51032480	7.24	24%
10*fourier 64	6714850	0.91	25%

Table 7.9: The average emulation overheads

When we translate this overhead to the times we got for each program, and using the total number of instructions for each program we evaluate the average percentage of overhead (last column) induced by the emulation. This is shown in table 7.9 where for each program we give the total number of instructions, the execution time in seconds and finally, the estimated cost of the emulation.

As the table clearly shows, the emulation accounts for about 20% of the overhead observed in the programs with a slight tendency for higher values in functional programs (for example, `tak` or `queens`).

7.2.8 Performance vs. Problem Size

The relation between the performance of a system and the size of the problems it is handling is an important measure of both the robustness of the implementation and also of its expected efficiency running real world applications.

A system should be reliable, capable of running large problems over extended periods of time without problems. Also, the system should consistently perform well even with increasing problem size. By this we mean not only the time it takes to complete the computation but also its memory behavior. A robust system should be able to run large problems over a large time span, with minimal memory requirements.

	Pict		**TyCO**	
`sieve` n	heap(kw)	time(s)	heap(kw)	time(s)
n = 8k	256	0.6	256	1.4
n = 32k	256	11.7	256	17.4
n = 128k	256	–	256	261.9
n = 128k	512	277.7	512	243.0
n = 512k	512	–	512	4698.9
n = 512k	2048	3720.9	2048	3479.9

Table 7.10: Performance vs. Problem Size

To test the TyCO run-time system we chose the program that most closely matched its own version in Pict in overall performance – `sieve`. We executed `sieve` with an increasing problem size and measured how much time it took each system to complete and how large the initial heap was required to be in order to successfully run the example. The results we got are presented in table 7.10. Pict is faster for small problem sizes. However, as we increase the problem size Pict not only performs worse than TyCO in terms of speed, but also requires a significantly larger heap to perform the task. TyCO performs quite well in long computations and is more conservative in heap usage.

Chapter 8

Distribution and Mobility in TyCO

In this chapter we extend the TyCO calculus with the objective of supporting distributed computations and code mobility. Introducing distribution, code mobility, and failure detection and recovery into π-computations is a fast growing research field, with immediate applications in mobile computing, *web* languages, cryptography, to name a few [2, 12, 21, 73]. We are interested in using TyCO, extended with support for distributed computations and code mobility, to experiment with new programming idioms combining code mobility with concurrency or parallelism. We are especially interested in evaluating the potential of such languages in specific hardware platforms such as clusters of *off-the-shelf* PCs connected by high performance networks such as Myrinet, FastEthernet or ATM [17, 18]. These systems may provide an affordable, high performance alternative to current virtual shared memory or distributed memory parallel computing systems.

8.1 Overview of the Model

From the very beginning of the design of TyCO we had the objective of providing a flexible, portable and scalable emulator architecture that would easily fit in a distributed environment. In particular the choice of an architecture independent virtual machine design, running byte-code format executables was made with distribution and mobility in mind. Moreover, the virtual machine shows promising results in both byte-code compactness and very conservative heap usage (see chapter 7). This is an important feature for a system supporting distributed computations since less network bandwidth is spent moving code or data-structures. Heap usage is a major concern since it allows a greater level of concurrency/parallelism with a modest use of system resources. This is also relevant for computations triggered by code that migrated from remote nodes since the availability of system resources in a network may be quite heterogeneous. Starting from the TyCO calculus, as defined in chapter 2, we built a simple model of distribution for mobile processes [73]. The following major constraints guided its design:

1. the model should be a simple extension of the calculi we have today;
2. it should be independent of the base calculus chosen;
3. it must meet realistic expectations of current distributed systems;
4. it must be efficiently implementable in current hardware.

The extension to TyCO is orthogonal meaning that the semantics of the base calculus (see section 2.2) is unchanged. Actually, the model is applicable to any calculi satisfying a set of very mild pre-conditions:

1. the base calculus may incorporate values in general, and should provide for names in particular;
2. it should allow to create a new name visible only in a given process, obeying the lexical scoping convention;
3. it should have processes prefixed on some name. Examples are output and input prefixes ($\overline{a}vP$, $a(x)P$, $!a(x).P$) in the π-calculus [46], messages and objects ($a!l[v]$, $a!M$) in TyCO [74], requests for session initiation (**accept** $a(k)$ **in** P, **request** $a(k)$ **in** P) in Structured Communication-Based Programming [28], and names and resources (a, $a \Leftarrow P$, $a = P$) in the Blue calculus [10]. All the above examples are prefixed at name a;
4. it should have a parallel composition operator and the corresponding neutral element;
5. it should incorporate a notion of substitution of names by values in a process, avoiding the capture of the names substituted. If P is a process and σ a total function from names to values, we denote by $P\sigma$ the process resulting from applying σ to P.

We find these requirements mild; most calculi to date [10, 27, 28, 46, 74] fulfill the constraints. A possible exception is the Join calculus [21] and item 3 above. Here we focus on applying the model to TyCO and provide the basis for a distributed TyCO programming language with support for code mobility. We call this extended language DiTyCO. The model builds on the notion of *site* (or location) where conventional (name-passing, in this case) computation happens. We also have site identifiers, distinct from the usual names. Our processes are *network aware*: names can be local or remote; the distinction is explicit in the syntax. Local names are those of the base calculus; *located names* are pairs site-name. Sites are composed of *located processes*, that is, processes paired with site identifiers. They denote the execution of the process at the site, similarly to most proposals to date [6, 21, 24, 58, 63]. Processes at sites can be put to run in parallel. Furthermore, since name-passing calculi are capable of extruding the scope of a (local) name, our networks are equipped with a site-name restriction operation.

In summary, networks are sites equipped with a composition and a restriction operator. This yields a *flat organization of sites*, that reflects the architectures of current implementations of high performance networks [17, 18]. Our target hardware architectures are low-cost, off-the-shelf, clusters of PCs interconnected with a high speed network. We feel that, the site organization should map the low level hardware architecture as closely as possible to allow an efficient implementation of the model. From a formal point of view our site organization is quite close to Distributed-π [58] and contrasts with the tree structure of Mobile Join [20] and the nested structure of Ambients [12]. The model has two logical levels: processes and networks (similar to other proposals [6, 24, 58]). Local computations happen at sites, as prescribed by the semantics of the base calculus. Remote computations occur between prefixed processes at different sites. In TyCO [74] such prefixed processes are either remote message invocation or the migration of objects. The migration of prefixed processes is deterministic, point to point, and asynchronous; synchronization only happens locally, at reduction time. We adhere to the *lexical scoping in a distributed context* of Obliq [11]. The free names of any "piece of code" transmitted over the network are bound to the original location. Network transmission implies the translation of the free names in the code in order to reflect the new site where the code is to be executed.

An important design decision related to points 3 and 4 above is the incapacity of the model to create remote names and the inability to spawn processes at remote sites, thus providing for site protection against arbitrary uploads. Processes may be spawned remotely only through the collaboration of what we call friend processes [73].

This contrasts with other proposals such as Sewell et al. [63], Amadio [6] and Hennessy and Riely's [24, 58]. Finally, as a first approach, our site identifiers are not first class objects: they cannot be sent in messages. Also we deliberately eschew the following issues: checking whether a site is alive; killing a site [6, 21, 58]; checking whether two remote names reside at the same site [63]; comparing site identifiers, and; dynamically constructing a located name given a name and a site identifier.

8.2 The Syntax of Networks

In this section we describe a model that supports distribution and mobility in the TyCO calculus [73]. This model serves as the basis for the ongoing implementation of the DiTyCO distributed system [39]. The model has two levels: on the first level we have the *processes* in the base calculus – TyCO; on the second level we build *networks*. We start with TyCO as our base calculus (see chapter 2). Despite the support for distribution and mobility of processes, all reductions are performed locally and obey to the semantics of TyCO. We now proceed to build the second level of the model on top of TyCO.

We introduce a new class of identifiers, *sites*, distinct from names or any other class of identifiers in the base calculus. We let r and s range over the set Site of sites. *Located names*, are site-name pairs. We let $s{\cdot}a$, denoting a name a located at site s, range over the set Site $\times$ Name of located names. Similarly, *located template variables*, are elements of the set Site $\times$ TVar ranged over by $s \cdot X$, denoting the template variable X located at site s. Names and template variables are collectively known as *network identifiers*, elements of the set Id and ranged over by w. To summarize, besides the syntactic categories of the base calculus we have:

$$\begin{array}{rlr}
r, s \in & \text{Site} & \text{Sites} \\
s{\cdot}a \in & \text{Site} \times \text{Name} & \text{Located Names} \\
s{\cdot}X \in & \text{Site} \times \text{TVar} & \text{Located Templates} \\
w \in & \text{Id} = \text{Site} \times (\text{Name} \cup \text{TVar}) & \text{Network Identifiers}
\end{array}$$

We then allow network identifiers to occur in any position in the base calculus where (non-binding occurrences of) identifiers can. The calculus thus obtained constitutes the *first level* of the model. Since site identifiers are introduced anew, there must be no provision in the base calculus for binding located identifiers. As such, at this level, a located identifier behaves as any other constant in the base calculus. The *second level* is composed of site-process pairs called *located processes*. They are elements of the set Site × Proc and are ranged over by $s : P$. Located processes may be composed via conventional parallel, restriction, and definition operators. The set Net of networks, ranged over by N, is given by the following grammar.

$$N \quad ::= \quad s : P \quad | \quad N \parallel N \quad | \quad \textbf{new}\ s{\cdot}aN \quad | \quad \textbf{def}\ s{\cdot}D\ \textbf{in}\ N \quad | \quad \mathbf{0}$$

The bindings in networks are as expected: a located name $s{\cdot}a$ occurs *free* in a network if $s{\cdot}a$ is not in the scope of a **new** $s{\cdot}aN$; otherwise $s{\cdot}a$ occurs *bound*. The sets $\mathrm{fn}(N)$ and $\mathrm{bn}(N)$, respectively, of free and bound located names in a network N are defined accordingly. Similarly, a located template variable $s{\cdot}X$ occurs free in a network N if $s{\cdot}X$ is not in the scope of a **def** $s{\cdot}D$ **in** N, that is, X is not one of the X_i bound in a binding $X_i = (\tilde{x}_i)P_i$ in D. The sets $\mathrm{ft}(N)$ and $\mathrm{bt}(N)$ of, respectively, the free and bound located template variables in networks are defined accordingly.

8.3 Semantics of Networks

Structural congruence allows us to abstract from the static structure of networks; it is defined as the least relation closed over composition and restriction, that satisfies the monoid laws for parallel composition, as well as the following rules:

$$
\begin{array}{rl}
(\textsc{Nil}) & s : \mathbf{0} \equiv \mathbf{0} \\
(\textsc{Split}) & s : P_1 \parallel s : P_2 \equiv s : (P_1 \mid P_2) \\
(\textsc{New}) & s : \textbf{new}\ aP \equiv \textbf{new}\ s{\cdot}a(s : P) \\
(\textsc{GcNew}) & \textbf{new}\ s{\cdot}a\ \mathbf{0} \equiv \mathbf{0} \\
(\textsc{Def}) & s : \textbf{def}\ D\ \textbf{in}\ P \equiv \textbf{def}\ s{\cdot}D\ \textbf{in}\ s : P \\
(\textsc{GcDef}) & \textbf{def}\ s{\cdot}D\ \textbf{in}\ \mathbf{0} \equiv \mathbf{0} \\
(\textsc{Extr}) & N_1 \parallel \textbf{new}\ s{\cdot}aN_2 \equiv \textbf{new}\ s{\cdot}a(N_1 \parallel N_2) \quad \text{if } s{\cdot}a \notin \mathrm{fn}(N_1) \\
(\textsc{ExtrD}) & N_1 \parallel \textbf{def}\ s{\cdot}D\ \textbf{in}\ N_2 \equiv \textbf{def}\ s{\cdot}D\ \textbf{in}\ (N_1 \parallel N_2) \quad \text{if } \mathrm{bt}(D) \cap \mathrm{ft}(N_1) = \emptyset \\
(\textsc{Swap}) & \textbf{new}\ r{\cdot}a(\textbf{def}\ s{\cdot}D\ \textbf{in}\ N) \equiv \textbf{def}\ s{\cdot}D\ \textbf{in}\ \textbf{new}\ r{\cdot}aN \quad \text{if } r{\cdot}a \notin \mathrm{fn}(s{\cdot}D) \\
(\textsc{Merge}) & \textbf{def}\ s{\cdot}D\ \textbf{in}\ \textbf{def}\ r{\cdot}D'\ \textbf{in}\ N \equiv \textbf{def}\ s{\cdot}D\ \textbf{and}\ r{\cdot}D'\ \textbf{in}\ N \quad \mathrm{dom}(s{\cdot}D) \cap \mathrm{ft}(r{\cdot}D') = \emptyset
\end{array}
$$

Rules Nil, Split, New and Extr are taken from Hennessy-Riely [24]; rules Nil, Split, and Extr are present in Sewell et al. [63] as well; the remaining rules are adapted from [70]. Rule Nil garbage collects terminated located processes, whereas rules GcNew and GcDef garbage collect unused names and definitions, respectively. When used from left to right, the rule Split gathers processes under the same location, allowing reduction to happen; the right to left usage is for isolating prefixed processes to be transported over the network (see rule Move in the reduction relation below). The remaining rules allow the scope of a name or a template definition local to a process: a) to become visible to the network (rules New and Def); b) to extrude and encompass a network with several located processes (rules Extr and ExtrD), and; to aggregate template definitions (rule Merge) and to interchange **new**s and **def**s (rule Swap).

Non-located identifiers in processes are implicitly located at the site the process occurs at: an identifier w occurring in a network $r : P$ is implicitly located at site r. When sending identifiers over the network, the implicit locations of identifiers need to be preserved, if we are to abide by the lexical scoping convention. As such, an identifier w moving from site r to any other site must become $r{\cdot}w$. Similarly a located identifier $s{\cdot}w$ arriving at site s may drop its explicit location. The remaining identifiers need no translation. A *translation of identifiers* from site r to site s is a total function σ_{rs} defined as follows (the last rule should be applied last).

$$\sigma_{rs}(w) \stackrel{\text{def}}{=} r{\cdot}w \qquad \sigma_{rs}(s{\cdot}w) \stackrel{\text{def}}{=} w \qquad \sigma_{rs}(w) \stackrel{\text{def}}{=} w$$

For the purpose of defining the reduction relation over networks we introduce a new syntactic category G, of object and message bodies:

$$G \quad ::= \quad !l[\tilde{v}] \quad | \quad ?\{l_1 = (\tilde{x}_1)P_1, \cdots, l_k = (\tilde{x}_k)P_k\}$$

Processes prefixed at located names play a crucial role in the model, by moving towards the location of the located name: a process $s{\cdot}aG$ is meant to move to site s. If aG is a message then $s{\cdot}aG$ has the form $s{\cdot}a!l[\tilde{v}]$ and denotes a remote message send; if, on the other hand, aG is an object then $s{\cdot}aG$ has the form $s{\cdot}a?\{l_1 = (\tilde{x}_1)P_1, \ldots, l_k = (\tilde{x}_k)P_k\}$ and denotes an object migration operation. We thus see that conceptually there is not much difference between a remote message send and a process migration. In fact from an implementation point of view the difference is not abysmal either. Given the above definition, the reduction relation for networks is given by the following rule:

$$(\text{Local}) \qquad \frac{P \to Q}{s : P \to s : Q}$$

that allows the switching from the network reduction to local TyCO reduction. The other rules control local reduction and are unchanged from the TyCO semantics (see section 2.2). Finally, and most importantly we define the two axioms that allow code mobility in network computations. The first axiom supports the mobility of prefixed processes such as messages and objects.

$$(\textsc{Move}) \qquad r : s{\cdot}aG \to s : a(G\sigma_{rs})$$

If prefix $s{\cdot}aG$ is located at site r then, in order to keep the lexical scope of names, the free names in G must be translated according to $G\sigma_{rs}$. Hence, when sending $s{\cdot}aG$ from r to s we actually transmit $(s{\cdot}aG)\sigma_{rs} = aG\sigma_{rs}$. This is the essence of the axiom Move. Rule Local then allows processes in sites to evolve locally. As an example let us try a remote procedure call in TyCO. The client at site s invokes the procedure b at site r with a local argument v, waits for the reply and continues with P. The procedure accepts a request and answers a local name u (somewhere in the body Q of the procedure).

$$
\begin{array}{lr}
s : \textsf{\textbf{new}}\ a(r{\cdot}b!\,[va] \mid a?(y)P) \ \parallel\ r : b?(xr)Q \equiv & (\textsc{New},\textsc{Extr})\\
\textsf{\textbf{new}}\ s{\cdot}a(s : r{\cdot}b!\,[va] \ \parallel\ s : a?(y)P \ \parallel\ r : b?(xr)Q) \to & (\textsc{Move})\\
\textsf{\textbf{new}}\ s{\cdot}a(r : b!\,[s{\cdot}v\ s{\cdot}a] \ \parallel\ s : a?(y)P \ \parallel\ r : b?(xr)Q) \equiv & (\textsc{Split})\\
\textsf{\textbf{new}}\ s{\cdot}a(s : a?(y)P \ \parallel\ r : (b!\,[s{\cdot}v\ s{\cdot}a] \mid b?(xr)Q)) \to & (\textsc{Local})\\
\textsf{\textbf{new}}\ s{\cdot}a(s : a?(y)P \ \parallel\ r : Q\{s{\cdot}v\ s{\cdot}a/xr\}) \to^{*} & \\
\textsf{\textbf{new}}\ s{\cdot}a(s : a?(y)P \ \parallel\ r : s{\cdot}a!\,[u]) \to\to & (\textsc{Move},\textsc{Split},\textsc{Local})\\
\textsf{\textbf{new}}\ s{\cdot}a(s : P\{r{\cdot}u/y\}) \equiv s : \textsf{\textbf{new}}\ aP\{r{\cdot}u/y\} & (\textsc{New})
\end{array}
$$

We thus see that a remote communication involves two reduction steps: one to get the message/object to the target site and the other to consume the message/object at the target (see for example [21]); the former is an asynchronous operation, the latter requires a rendez-vous. This reflects actual implementations.

Another kind of remote interaction, which we call *download*, occurs when a site r finds an instantiation of the form $s{\cdot}X[\tilde{v}]$. A template variable X prefixed with a site name s indicates that X was originally defined at site s. An instantiation $s{\cdot}X[\tilde{v}]$ at a site r means that the definition for X is to be downloaded from the network. Once it arrives at r the Local rule enables the instantiation to occur.

$$\frac{X_i \in \mathrm{dom}(D) \qquad X_i = (\tilde{x}_i)P_i}{\textsf{\textbf{def}}\ s{\cdot}D\ \textsf{\textbf{in}}\ (r : s{\cdot}X_i[\tilde{v}] \ \parallel\ N) \to \textsf{\textbf{def}}\ s{\cdot}D\ \textsf{\textbf{in}}\ (r : \textsf{\textbf{def}}\ X_i(\tilde{x}_i) = P_i\sigma_{sr}\ \textsf{\textbf{in}}\ X_i[\tilde{v}] \ \parallel\ N)} \qquad (\textsc{Download})$$

Note that when moving P_i to r we must keep the lexical bindings for the free variables of P_i, so we actually download $P_i\sigma_{sr}$. Also, Download does not cache the downloaded template, and so other instantiations of $s{\cdot}X$ will again download P_i. The lexical scope of the template variables prevents a complete solution since processes with references to $s \cdot X$ may be downloaded at any time during a computation. Below we present an example of this kind of interaction. We move a piece of code that contains a template variable X. The code for X is downloaded thereafter.

$$
\begin{array}{ll}
r : \textsf{\textbf{def}}\ X = (x)P\ \textsf{\textbf{in}}\ s{\cdot}a?()X[b] \ \parallel\ s : a![] \equiv & (\textsc{Def},\textsc{ExtrD}) \\
\textsf{\textbf{def}}\ r{\cdot}X = (x)P\ \textsf{\textbf{in}}\ (r : s{\cdot}a?()X[b] \ \parallel\ s : a![]) \rightarrow & (\textsc{Move}) \\
\textsf{\textbf{def}}\ r{\cdot}X = (x)P\ \textsf{\textbf{in}}\ (s : a?()r{\cdot}X[r{\cdot}b] \ \parallel\ s : a![]) \rightarrow & (\textsc{Split},\textsc{Local}) \\
\textsf{\textbf{def}}\ r{\cdot}X = (x)P\ \textsf{\textbf{in}}\ s : r{\cdot}X[r{\cdot}b] \rightarrow & (\textsc{Download}) \\
\textsf{\textbf{def}}\ r{\cdot}X = (x)P\ \textsf{\textbf{in}}\ s : \textsf{\textbf{def}}\ X = (x)P\sigma_{rs}\ \textsf{\textbf{in}}\ X[r{\cdot}b] \rightarrow & (\textsc{Local}) \\
\textsf{\textbf{def}}\ r{\cdot}X = (x)P\ \textsf{\textbf{in}}\ s : P\sigma_{rs}\{r{\cdot}b/x\} &
\end{array}
$$

Note that with DOWNLOAD the calculus closely mimics the way class definitions are downloaded in languages such as Java.

8.4 Programming

The programming model associated with the framework described in the previous section is rather simple requiring just two new top-level declarations to be added to the set of processes defined in section 2.1. We define the category of *network processes*, ranged over by J and define their syntax as:

$$
\begin{array}{lll}
J ::= & \textsf{\textbf{export}}\ w\ \textsf{\textbf{in}}\ J & \text{Export Interface} \\
\mid & \textsf{\textbf{import}}\ w\ \textsf{\textbf{from}}\ s\ \textsf{\textbf{in}}\ J & \text{Import Interface} \\
\mid & P & \text{TyCO process}
\end{array}
$$

export is used to raise a local name or template definition to the network level of the model, making it visible to other sites. **import** lowers one of these names or template definitions to the context of a site so that it may be used locally. We call the language just defined DiTyCO. It is an orthogonal extension of TyCO and its semantics is that of the networks we have studied in the previous section. Notice that we never write located identifiers explicitly. The translation of the **export**/**import** declarations into the base calculus takes care of this conversion.

$$
\begin{aligned}
[\![\textbf{export } w \textbf{ in } J]\!] &\stackrel{\text{def}}{=} [\![J]\!] \\
[\![\textbf{import } w \textbf{ from } s \textbf{ in } J]\!] &\stackrel{\text{def}}{=} [\![J\{s{\cdot}w/w\}]\!] \\
[\![P]\!] &\stackrel{\text{def}}{=} P
\end{aligned}
$$

We see that the **export** declaration is really unnecessary. As programs are to be closed, we could take the view that every free identifier in a program is to be exported. From a programming point of view we, however, feel that the dual **import/export** declarations impose a more disciplined programming style, avoiding, for example, the automatic exporting of names that the programmer forgot to protect with a **new**. The remainder of this section is devoted to programming examples in DiTyCO, to attest the simplicity and flexibility of the model.

8.4.1 Applet Server

The example illustrates code transmission over the network. The idea is from Fournet et al. [21], but we have taken advantage of objects in TyCO to allow for the downloading of different applets.

An applet server provides for the downloading of k different applets through the k methods of an object. The server locates applet P_j at name p provided with the invocation of the method applet_j. Here is the code to be run at site sumatra.

```
def AppletServer = (self) self ? {
        applet_1 = (p) p?(x)P_1  |  AppletServer[self],
         ...
        applet_k = (p) p?(x)P_k  |  AppletServer[self]
}
in export appletserver
in AppletServer[appletserver]
```

Each client creates a fresh name where the applet server is supposed to locate the applet, then invokes the server with this name, and, in parallel, triggers the applet.

```
import appletserver from sumatra
in new p appletserver!applet_j[p]  |  p![v]
```

Let us try to understand how the server and the client interact. We start by translating the **import**/**export** clauses to obtain

sumatra: **def** ... **in** AppletServer[appletserver] ||
client: **new** p sumatra·appletserver!*applet*$_j$[p] | p![v]

Then, the message sumatra·appletserver!*applet*$_j$[p] moves to the server (yielding the message appletserver!*applet*$_j$[client·p]) with one MOVE reduction step, one local reduction at the server invokes the *applet*$_j$ method, and one final MOVE step migrates the applet client·p?(x)P$_j$ back to the client, yielding the process:

sumatra: **def** ... **in** AppletServer[appletserver] ||
client: **new** p p?(x)$P_j\sigma_{\text{sumatra client}}$ | p![v]

Incidentally, three is the number of reduction steps that Mobile Join [20] takes to perform the same operation. Notice how the structural congruence rules NEW and EXTR are used (from left to right) to allow name p at client to encompass both sites, and then (from right to left) to bring p local to the client again. Notice also that the applet body gets translated to reflect its new site: if P refers to a name a local to the applet server, then $P_j\sigma_{\text{sumatra client}}$ refers to the remote name sumatra·a.

It should be obvious that a client does not need to have the applet explicitly downloaded to its site; a message appletserver!*applet*$_j$[site·p] will migrate the code for the applet to site.

8.4.2 Compute server

The next example, from Cardelli [11], distinguishes local from remote computation. A compute server provides two operations, *lexec* and *rexec*, allowing the execution of a given parameterless procedure P at the client site and at the server site, respectively. Here is the code to be run at site borneo.

```
def ComputeServer = (self replay) self ? {
    lexec= (p) p![] | ComputeServer[self p],
    rexec= (replyTo) new p replyTo![p] | p![] | ComputeServer[self p]
}
in export computeserver
in ComputeServer[computeserver _]
```

The method for local execution triggers the procedure located at name p. Once again, exactly where the procedure runs depends on where p is located, and that is in the hands of the client. The method for remote execution provides for the migration of the procedure to the server by creating a new name p (local to the server) and by sending it back to the client. The client is then supposed to locate the procedure at this name while, as in the applet example, the server triggers the procedure. As in the original example [11], the server cheats on clients by storing the latest client procedure in (this time) a local variable. However since the variable replay is local to the server, there is not much use to it. We could however add a *replay* method to ComputeServer: *replay*= replay![] | ComputeServer[self replay]. Here is a possible client:

import computeserver **from** borneo
in new p p?P | computeserver!*lexec*[p] | – local execution
  let p = computeserver!*rexec*[] **in** p?P – remote execution

For the local execution, the client creates a new name p where it locates the procedure, and invokes *lexec* with argument p. For the remote execution the client waits for a name from the server and locates the procedure at this name. In both cases the triggering is done by the server. We can see that the difference between the two kinds of execution is centered on where the procedure identifier p is located.

While in the previous example, the applet server defines the procedures (applets) and provides for the uploading, in this example it is the client that defines the procedures to be run. The local execution takes three reduction steps until P is ready to be triggered; the remote execution takes five steps to accomplish the same. The two extra steps are needed to ask for and to get a name p, local to the server where the procedure will be located.

8.4.3 The Applet Server Revisited

An applet server provides for the downloading of k different applets, possibly mutually recursive, provided as template definitions. The applet server is now a site where the template definitions reside. The client program asks for a particular applet definition from the site, say sulawesi, and the code migrates from that site to the client where it is instantiated.

def $\mathsf{Applet}_1 = (\mathsf{x})\mathsf{P}_1$ **and** ... **and** $\mathsf{Applet}_k = (\mathsf{x})\mathsf{P}_k$ **in** ...

The client when creating an instance of, say, Applet_1, an imported identifier, triggers the code movement from server site to the client site. Once the code resides in the local (client) site the instantiation is performed as in TyCO. The code for the client would be:

import Applet_1 **from** sulawesi **in** Applet_1[v] | ...

8.5 Spawning Arbitrary Processes

An important design decision is that "*remote channel creation is only possible with a remote friend*". Hence **new** a **at** s is something we cannot write. We stick to the idea of not writing located names explicitly. The counterpart in the (extended) base calculus would be **new** $s{\cdot}a$ P (Sewell et al. [63] write (**new** $a@s)P$). The knowledge of a site name must not award the possibility of directly accessing the site's memory. The consequences would be far reaching. In particular, such a construct would allow the spawning of arbitrary processes regardless of the willingness of the server to accept the processes. Spawning a process P at site s without s's consent could be easily written as:

new a **at** s a?P | a![]

Other notation is used by Amadio [6] who writes **spawn**(s, P); Hennessy-Riely [24, 58] write $s :: P$ and also **goto**(s, P). Instead, to model arbitrary migration, we require the collaboration of some friend in the remote location to provide a remote name. Friends can be written as follows.

def Friend = (self) self ? *newName*= (replyTo) **new** a replyTo![a] | Friend[self]

Thus, spawning a process P in a location where we have aFriend can be modeled as:

spawn P **at** aFriend $\stackrel{\mathsf{def}}{=}$ **let** p = aFriend![] **in** p?P | p![]

This involves the exchange of three remote messages. However optimizing compilers could send p?P | p![] on the same message thus saving one remote message. We have already used this technique in the remote execution method of the compute server (section 8.4.2) but in that example the migrating process is triggered by the server.

An immediate application of this technique allows us to send a computation to a remote server and to get the results (see [21]). Here the client defines the request, the request moves to the server, runs there, and sends the result back to the client. Suppose that R is a request that eventually issues a message replyTo![v] with the result v, and s·b is the location of a friend at the server in site s. Then, we may send R to the remote server, get the result in x and continue with P, by simply writing:

new replyTo **spawn** R **at** s·b | replyTo?(x)P

We can specialize remote friends. Here is one that accepts the migration of arbitrary processes (with the necessary collaboration of the client; see method *rexec* of the compute server, section 8.4.2), and invokes them.

def Friend = (self) self ? *migrate*= (replyTo) **new** a replyTo![a] | a![] | Friend[self]

Since the procedure is triggered at the server, this version saves one remote message passing when compared to the method *newName*. We could go one step forward and stipulate a gateway for each site providing for all the services we could anticipate for the site, as in Amadio [6]. The gateway name would then represent the site itself and we could work with gateways as if we were dealing directly with sites.

There is also an implementation related reason why we do not want remote name creation. All our remote primitives (but **export**/**import**) are accomplished with a single (asynchronous) remote message passing. To implement remote name creation we would need two remote messages (one asking for the creation, the other replying the name created). Meanwhile the sender would have to wait a considerable amount of time (when compared to the time it takes for local message passing). To speed up things, we could put the continuation process aside while waiting for the reply from the server, but that would imply major changes in the design of sites.

8.6 Implementation Issues

An implementation of DiTyCO involves changes in the semantics of some abstract machine instructions and some care in the compilation process. Here we focus on local changes that have to be made at the level of the virtual machine and language compiler. These changes must interface with another software layer that supports the actual remote communication and code mobility. The design and implementation of this layer and DiTyCO in general is an ongoing team project [39] aiming at experimenting with concurrent distributed languages in the context of process calculi. As we write these lines a first stripped down working version of the system is only a few weeks away. There is however still a lot of programming and research that needs to be done in order to have a robust and efficient system.

8.6.1 Reduction

To implement reduction we must change the semantics of three virtual machine instructions: `redobj` , `redmsg` and `instof` . This is where the *mobility* (see section 8.3) rules are implemented. In the case of the first two instructions we must check whether the address of the current channel (in `CC`) is in the local heap or whether it is a reference to a channel in a remote heap. Local reduction is performed as in the TyCO virtual machine. In the case of remote reduction a message or an object closure must be sent through the network (this is the MOVE rule) to allow reduction to proceed locally at the remote site (applying the LOCAL rule). Similarly, in an `instof` instruction, we must check whether the argument label is local (refers to byte-code in the current program) or if it is a reference for byte-code that exists at a remote site. In the first case, instantiation occurs as usual in the virtual machine. However, in the second, the byte-code block for the template and its free variable bindings must be fetched across the network towards the local site (this is rule DOWNLOAD). After the byte-code is installed locally, the instantiation is performed as usual (as a result of the LOCAL rule).

8.6.2 Compilation Issues

As we have seen remote communication originating in a site r occurs when we find messages, $s{\cdot}a\,!\,l[\tilde{v}]$, or objects, $s{\cdot}a?M$, located at a name $s{\cdot}a$ that has been imported from the network layer, more precisely, from another site s. Another possibility for remote interaction occurs when we instantiate templates, such as $s{\cdot}X[\tilde{v}]$, which are defined at remote sites.

In the case of a message $s{\cdot}a\,!\,l[\tilde{v}]$, the only information that needs to be sent across sites is the label l of the method to be invoked and the arguments $\tilde{v}$ with the substitution σ_{rs} applied. In this case there is no byte-code being moved. In the case of an object $s{\cdot}a?M$, the information that needs to be sent from site r to site s consists of the free variables of M with σ_{rs} applied plus the complete byte-code for the object's method collection. This fact imposes some discipline in the way the DiTyCO source code is compiled into the intermediate language representation and later the byte-code. Finally, in the case of a template definition $X = (\tilde{x})P$ the byte-code for the template body P with σ_{rs} applied must be sent over the network.

An important observation is that if the nested structure of processes in the source code is preserved in the byte-code then an efficient way of extracting pieces of byte-code at run-time can be devised. Consider the AppletServer example from section 8.4. As we have seen the applet P_j is selected and downloaded from a server by sending a message with label *applet*$_j$ and a name local to the client requesting the applet. When the code reaches the server, it is *placed* in the argument p. The lexical scope of p forces the code P_j to be downloaded to the client site. Notice that process P_j can have an arbitrary nested structure. When we observe the intermediate assembly (that maps almost one to one with the byte-code) for this particular example:

```
main = {
   def AppletServer = {                        (5)
      objf       1
      cput       p0
      trobj      p0 = {
         { applet_1,...,applet_k }
         applet_1 = {                          (1)
            objf       0
            trobj      p0 = {                  (2)
               { val }
               val = { % code for P_1 }
            }                                  (3)
            instof     1,AppletServer
            cput       f0
         }
         ... ... ...
         applet_k = {
            objf       0
            trobj      p0 = {
               { val }
               val = { % code for P_k }
            }
            instof     1,AppletServer
            cput       f0
         }
      }
   }                                           (6)
   export     c0
   instof     1,AppletServer
   cput       c0
}
```

we see that if we preserve the nested structure of the original source code we can easily extract the piece of byte-code corresponding to P_j. We simply need to identify its starting point and extract all the code at that particular nesting level and deeper. So, in response to a client invocation of the form:

import appletserver **from** sumatra
in new p appletserver!*applet*$_j$[p] | p![v] (4)

The appletserver would run the code starting at (1) with `p0` bound to `p`. At (2) the server places the object with method table { `val` } and methods `val = { % code for P_1` } at the name `p` which is lexically bound to the client process. The server must then extract the block starting at (2) and ending at (3), a simple operation given that the nested structure of the original source process was preserved in the byte-code. Then it sends the byte-code back to the client, together with any existing free names. Upon arrival to the client site the downloaded code and the bindings are placed at the name `p`. The client runs the byte-code by sending a local message to `p` as illustrated in (4). We might also be interested in starting an applet-server at a given client. This could be achieved with the following code:

import AppletServer **from** sumatra
in new p AppletServer[p]

creating a server at the local name `p`. In this case, since we imported the definition of template AppletServer from site sumatra the complete code for the template must be downloaded from sumatra to the local site. At sumatra the selection of the byte-code to be sent is greatly simplified by its simple and nested layout. We just pack all the code inside the scope of the AppletServer definition, starting at (5) and ending at (6). The nested intermediate assembly simplifies considerably the compile-time task of computing object boundaries and the book-keeping information required by the emulator at run-time.

Chapter 9

Related Work

We overview the main features of related concurrent process-based programming languages and their foundations.

9.1 Process Calculi

The **CCS** [44] (Calculus of Communicating Systems), was first introduced by Robin Milner. It was the first major step towards a modern theory of concurrency. Milner introduced an algebra where the basic abstractions are processes that interact through synchronization on shared locations called names. The interactions do not involve the exchange of data. The CCS approach of looking at processes as terms in an algebra and their interactions as operations on these terms was particularly fruitful and gave way to what are now commonly called *process calculi*.

The π**-calculus** was introduced by Milner, Parrow and Walker [46] in the late eighties. As in CCS the basic abstractions are processes and names. Names are shared locations through which processes may interact. In its original form, the monadic π-calculus, a process could either send or receive a single value through a name. Almost immediately after it was shown that, through a simple encoding, an equivalent polyadic calculus could be obtained – the polyadic π-calculus. The early versions of the π-calculus were synchronous meaning that output processes (processes that send values to names) have continuation processes. Early in the nineties, Honda and Tokoro [27] and Boudol [9] independently showed that the polyadic calculus could be further simplified into an equivalent asynchronous formulation, where output processes do not carry continuations. Most of the current work on the π-calculus uses the asynchronous form.

The **γ-calculus** [64] is a specialization of a more general calculus that is obtained from the polyadic π-calculus by distinguishing between names and variables and making the variables logical. The basic abstractions are: names, (logical) variables, (named) procedural abstractions, equations and cells. Equations establish relations (simple equality) between variables and can be seen as constraints. Hence variables can be used without any specific information on the values they take, and using only the known constraints between them. Cells are primitive entities that maintain state and provide atomic read-write operations. Procedural abstraction allows functional processes to be modeled and be referred to by names. Objects are modeled from collections of procedures with an encapsulated state.

The **Join-calculus** [20] is based on the reflexive CHAM [8] model. Join collapses the creation of new names, input and replication into a single construct called a join pattern. This means that receptors are defined at the same time as the names they occur in and are therefore unique. As in other process calculi the basic abstractions are names and processes. Processes communicate by sending messages on names. In the chemical metaphor of CHAM processes are seen as molecules in a chemical soup, a program. The processes interact by first closing in on each other (structural congruence) and then reacting (reducing) producing new processes (molecules). Join patterns can be seen as catalysts in these reactions. They are places that attract sequences of molecules with certain signatures and induce the reaction. The reaction produces new molecules that continue the process.

The **Blue calculus** [10] is another variant of the asynchronous π-calculus. The motivation for introducing this variant was the observation that an indirect style of programming - where one has only an indirect access to values, via their names, and one has to explicitly manage "result channels" - is enforced in languages based on the π-calculus. The π-calculus is often viewed as an "assembly language" for concurrent computations and lacks the possibility of a direct usage of functions: π-terms are not functions (abstractions), and they cannot be applied. The Blue calculus provides a better programming notation for the functional subset of the π-calculus. Its main contribution is the fact that abstractions are also processes, and that processes may be applied to names. Blue splits communication from synchronization and this is enforced directly in the syntax: an input prefix on some name is a declaration that some value or process is available under that name, either once or in a replicated manner. The Blue calculus is a "continuation passing style calculus" and has been shown to contain both the λ-calculus and the π-calculus (without sum). The type system for the calculus combines Curry's type inference for the λ-calculus, and Milner's sorting for the π-calculus.

The **Fusion calculus** [51] simplifies and extends the π-calculus and inherits all its expressive power. The main innovation with Fusion is that it supports updates on a shared state, and these updates can be localized through a scope construct. The calculus is particularly useful in the representation of models for concurrent constraint computing. The Fusion calculus has only one binding operator as opposed to two in the π-calculus (input and restriction). On the other hand, it features a new kind of action, appropriately called *fusion*, that emulates updates on a shared state. Fusions are distinct from reductions in π in that names are not bound but rather unified as a side effect of the interaction between processes of opposite polarities. The names can be checked for equality by any process within the scope of these names. The absence of the binding operation in input processes has important consequences, the most notorious being the complete symmetry between input and output actions, a feature not present in the π-calculus.

Seal [75] is a process calculus suited for modeling Internet applications and programming languages. The mobile units of the calculus are called *seals*. The three main abstractions are: *locations* that model, for example, boundaries of networks, IP nodes, address spaces; *processes* modeling flow of control such as threads and OS processes, and; *resources* that model physical resources such as network, memory and peripherals. Names stand for channels, in which processes synchronize, and locations of seals. Processes are the computational units and are basically π processes plus Seals. Seals, denoted $n[P]$, are named, hierarchically-structured, locations. For example, a seal $n[h_1[P \mid a_1[Q]] \mid h_2[P' \mid a_2[Q']]]$ represents a network with hosts h_1 and h_2 with individual behaviors P and P', respectively. The hosts run applications with behaviors, Q and Q', respectively. There are only three patterns of interaction between processes: a) *local* – the usual π type reduction; b) *down-sync* – between a parent and a child seal, on a local name of the child, and; c) *up-sync* – between a child and a parent seal, on a local name of the parent. Each action corresponds to a distinct process prefix. Channel mobility is as in the π-calculus although with the above restrictions on the scope of interactions. Seal mobility is treated separately and involves moving entire computational subtrees (that protected inside the seal) to other locations. This operation can be viewed as a tree re-writing operation. Security is a main concern in Seal. The restrictions described on the communication and therefore in seal mobility already enforce some protection. The migration of seals into a local seal is always under the control of the later. The local seal must always perform a receive operation previous to the migration and, upon arrival, must choose a fresh name for the new seal so that it cannot interfere with its other services. Finally, inter-seal communication is monitored by way of *portals*, that is, linear channels provided by the receptor to the sender. Portals are ephemeral and thus do not survive synchronization. The semantics of Seal is quite forward from the π rules except for non-local, that is, crossing the boundaries of a seal, synchronization. In this case local names to the seal must be extruded via the reduction relation, a mechanism called *heating*.

In recent years there have been some efforts in trying to interpret object calculi such as Abadi and Cardelli's [1] in the context of typed π-calculus. Sangiorgi [61] presents an interpretation of the first-order ς-calculus into the π-calculus with Pierce and Sangiorgi's I/O types [53]. The encoding uses a subset of the π-calculus language with the following characteristics: a) all input names are either persistent or linear; b) a name appears in input subject position at most once, and; c) a name received in an input can only be used for output. This makes the interpretation quite amenable to optimization. The fragment of the π-calculus used includes a *case* construct that is very similar to TyCO's objects and replaces matching. In the author's own words: "Identifying combinations of the π-calculus operators which are useful for the interpretation of objects might lead to the definition of a higher level target calculus, capable of yielding more succinct and readable interpretations of objects." Variant types were introduced for the first time in the π-calculus and their typing and subtyping properties derived. Furthermore, the behavioral equivalences induced by these types in the π-calculus were investigated to study the possibility of using the typed π-calculus as a framework for reasoning about concurrent object-oriented languages.

Further work has been done in this direction by Nestmann and Ravara [55] in collaboration with Sangiorgi, trying to use the same basic approach of the previous work while using non-uniform TyCO [57] as the target calculus.

Another approach has been taken by Gordon and Hankin [23] who extend the ς-calculus with concurrent and synchronization operators. The calculus thus obtained is given a precise operational semantics in the form of an abstract machine. The type-system is derived from the first-order type-system for the imperative ς-calculus. Subject reduction is proven for the type system. The authors however do not study the behavior equivalences induced by this type-system in their calculus.

Honda, Vasconcelos, and Kubo [28] introduce a calculus with a set of basic language constructs and a typing discipline as a foundation for structured communication-based concurrent programming. The motivation for this proposal is based on the fact that most communication primitives offered by distributed programming languages support only "one-time" interactions between processes, for example RPC calls, and more complex behavior is described by combining these primitives. The authors idea is to introduce a set of communication primitives that allow the modeling of complex communication patterns between processes. The fundamental abstraction of the model is the "session", described as a concurrent composition of the communication primitives. The model is introduced in the form of a process calculus and its operational semantics is specified. A type discipline enforces the correct behaviors for the primitives. The type-system has the "subject reduction" property which makes the static verification of programs for run-time protocol errors feasible.

9.2 Programming Languages

Pict [54] is a pure concurrent programming language based on the asynchronous π-calculus. The run-time system is based on Turner's abstract machine [66] specification and the implementation borrows many features from the C programming language. The basic programming abstractions are processes and names (channels). Processes communicate by sending values along shared channels. Objects in Pict are persistent with each method implemented as an input process held in a distinct channel. The execution of methods in concurrent objects by a client process is achieved by first interacting with a server process (that serves requests to the object and ensures mutual exclusion), followed by the method invocation proper. This protocol [76] for method invocation involves two reductions as opposed to one in TyCO using branching structures. Pict uses replication to model recursion. In Turner's machine the use of replication is controlled by artificially changing the reduction rules so that replication only occurs in response to a message sent to a replicated input process. TyCO's branching structures allow a cleaner definition of reduction since recursion can only occur after communication. Also, replication (as opposed to recursion) generates significant amounts of computational garbage in the form of unused channels and process closures. This may result in frequent garbage collection and the consequent loss in performance.

Oz [43] 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 and its implementation is inspired in the AKL [31] machine. The basic abstractions are names, logical variables, 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. 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. 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 TyCO if we view the state held in logical variables as template parameter bindings. The basic message passing mechanism in Oz is the *port*. Ports hold queues of incoming messages that are explicitly manipulated by interested processes. These operations are atomic. Ports are run-time references to queues of messages and as such may be passed as arguments to other processes, just as names in the π-calculus for example.

Join [22] and **JoCaml** [32] implement the Join-calculus [20] which is itself based on the reflexive CHAM [8]. Names (also called ports in Join), expressions and processes are the basic abstractions. Join programs are composed of processes, communicating asynchronously on names and producing no values and, expressions evaluated synchronously and producing values. Join introduces a powerful extension of the notion of name – the *join-pattern*. Patterns combine names, input processes and replication into a single construct. A join-pattern defines a synchronization pattern between input processes waiting on a sequence of names. In the chemical metaphor they define the kind of reactions that may occur between molecules in the soup. Both Join and JoCaml have some fairly advanced tools for modular software development inherited from its development language Ocaml [37]. These include a source to byte-code object compiler, a linker to add compiled modules and a run-time byte-code interpreter.

9.3 Types and Optimizations

Programming languages based on process calculi require some kind of type discipline to ensure that programs are well formed. Since processes in these calculi interact through their free names, type systems assign to each free name of a process a type that describes the kind of data structures the name can carry and, its arity. This *type* ultimately defines how a process interacts with the surrounding environment. An error in communication can occur if an output process sends a value, say an integer, to a name where the input process is only expecting, for example, a boolean. In this case there is a type mismatch in the use of the output and input parts of the communication link. In the case of polyadic calculi, an error can also occur if the number of arguments sent by an output process to a name does not match the number of parameters of the input processes listening on the same name. These errors are commonly referred to as *protocol errors*. In TyCO, the existence of objects defined as collections of labeled methods held in a name, introduces the possibility of another protocol error, the so called *message not understood* error, in which a message invokes a non existing method in an object.

The first type-system for the π-calculus was introduced by Robin Milner [45, 46] . In this system, a type assignment for a process is commonly called a *sort*. The names in a π-term are partitioned into *subject sorts*. Each subject sort is associated with an *object sort* that describes how the names of a particular subject may interact. The type of a channel that carries k names is a k-tuple with the individual types for the names it carries. The type-system assigns a type to each free name in a process producing a sort. Milner's system also supports recursive types in a very natural way. The only information carried by this type-system is the shape of names and their arity. This early system has been substantially refined and enriched.

Polymorphism can be introduced in process calculi basically at two levels. One approach is to allow polymorphism directly in the types of names. This is for example the approach taken in the type system for Pict [66]. The problem with this approach is that typically these type-systems are undecidable so the compiler must be guided by user annotations. Moreover, the complexity of type-checking is typically exponential. Another approach, the one followed in TyCO, uses a form of predicative (or Damas-Milner) polymorphism [74] over process templates rather than names. In this case there are decidable type-inference algorithms with almost linear complexity that apply. Moreover, programming becomes much simpler since no explicit type annotations are required while advantages such as code re-use and information hiding are preserved.

Refining a type-system makes it more sensitive to the behavior of processes and allows processes exhibiting unwanted behaviors to be statically discarded (for example, using function names to perform input). Enriching the type-system allows extra information to be gathered such as the way channels are used. This can be used to guide a compiler in producing code that is both faster and uses heap space more efficiently at run-time.

Pierce and Sangiorgi [53] refined Milner's basic system to include information on the polarities of channels. In other words, the type of a channel is not only characterized by the shape of the names it carries and its arity, but also by the kind of interaction that it supports: (just) input, (just) output or both. Using polarity information the authors showed that β-equivalence was preserved in the simplest of the encodings of the λ-calculus in π. This could not be proved with earlier type-systems.

Recent advances have been focusing in making the type systems smarter, capable of detecting certain patterns in the usage of names. Kobayashi, Pierce and Turner [36], inspired in ideas from linear logic, proposed a type-system in which the type for a name is enriched with information about its *multiplicity*, that is, the number of times it is used for reduction. The authors considered only three basic multiplicity values: 0 (never used), 1 (linear, used exactly once) and ω (used an arbitrary number of times). Knowing that a given name is linear allows a compiler to produce faster code for reduction since the software representation of names can be made very compact and the amount of book keeping can be significantly reduced. Also, once a linear channel is used the space it occupies in the heap can be immediately reclaimed. In TyCO, for example, there are many cases in which no heap storage whatsoever is required for the linear channel.

Kobayashi [35] argues that the linearity property described above is too strong to be of much use in actual implementations. The relevant information according to the author is when a given value is last accessed in the heap, and not that it is accessed only once (or a fixed number of times). In quasi-linear types, the linearity condition is relaxed by extending types to include information on the evaluation order and data-flow of values. The technique is very promising. For example, a heap data-structure known to be linear at compile-time can be immediately reclaimed after the first (and only) access to it has been made. Moreover, this space can be used destructively rather than reclaimed by garbage collection. This means that, for example, if a value must be created after a linear one has been accessed, then it suffices to copy the new value over the old linear one (provided there is enough space for the copy).

Recently, Sumii and Kobayashi [65] proposed a type-system that guarantees that communication on names in a process-calculus is deadlock free. They use a type-annotated form of the asynchronous π-calculus. The kind of information added to the types of names and feed to the type-checker is in the form of *capabilities* and *obligations*. Capabilities are guarantees that an action (input or output) if performed will succeed. Obligation implies that a name must perform an action (input or output) regardless of whether it succeeds or not. In this framework, a deadlock is a situation where a capability and/or an obligation fails. Besides this usage data, types also carry a time tag that is used to check whether obligations are always fulfilled. A type-checker uses this information to assert that programs are deadlock-free.

Concurrent objects can offer services non-uniformly, that is, the availability of a service depends on the internal state of the object (which reflects the state of the system) [49]. Objects with methods enabled or disabled according to its internal state are very common in object-oriented programming (for example, stacks, finite buffers). To explore these richer non-uniform types Ravara and Vasconcelos [57] introduce a type-system that supports objects with non-uniform interfaces in the TyCO process calculus. The type-system is based on a definition of a run-time protocol error that is distinct from the one presented in section 2.1. In this definition of error, a pair object-message — a redex —is not an error if the message may be accepted at some time in the future. In the uniform case a redex that does not reduce is considered an error. However here redexes are not considered errors if the object may evolve to a state where it can accept the message. A communication error is a process with a persistent bad redex, either because the object is deadlocked, or because it will never have the right method for the message. The system ensures safety properties such as the absence of some deadlocks (called local), and the guarantee that typable processes will not run into communication errors. Currently we are studying how the virtual machine can be used to efficiently implement this flavor of TyCO [56].

9.4 Distribution and Code Mobility

Work in this area has focused mainly in extending the asynchronous π-calculus, or a variant form, to include primitives that support distribution and code mobility.

In **Mobile Join**, Fournet et. al. [21] , extend the join-calculus with the notion of *location* defined as a set of processes with a local environment surrounded by a membrane. They are located at sites and can be atomically moved across a web-wide tree of locations. Distribution and mobility are directly supported by explicit primitives in the calculus.

Distributed-π, by Riely and Hennessy [24, 58] introduces a two-step extension to the π-calculus that models distribution and failure. They separate basic (local) processes from *located* processes at a given location. They establish semantic equivalence results through bisimulation for processes in the presence of distribution and the possibility of failure.

Nomadic-π, by Sewell et.al. [63] introduces explicit primitives for mobility and distribution into the π-calculus. Processes are executed at sites (locations). They identify two fragments of the resulting calculus: the first lower-level with location dependent primitives while a second fragment is entirely location independent. The semantics of the location independent fragment is then defined based on the lower level fragment. The authors claim that this provides a clean model for implementation of prototype languages.

Amadio [6] uses the notion of location as the basic abstraction for his model of distribution and failure. Processes are explicitly distributed to locations and locations are also the basic unit for failures. Locations can be tested for failure and their computations can be stopped. His model allows the detection and recovery from location failures.

Sekigushi and Yonezawa [62] present a calculus based on the CBV λ-calculus with extensions to express units of code movement, locations, and mobility of code between locations. Within this framework many distribution mechanisms can be modeled formally. Mobility is supported explicitly and a semantics for splitting an agent's code is given. Each location has a local store for allocation of local resources. Each address in the local store has an associated data movement type that specifies the behavior of a stored value when it is moved. Several types are defined to model distinct semantics.

In **Obliq** [11] Cardelli introduces a *distributed* lexically scoped, prototype-based object-oriented language. The main feature of Obliq is its use of distributed lexical scoping in distributed computations: free identifiers in code moved between network nodes remain bound to their original locations, even though the computation may be proceeding in a distinct network site. Obliq moves computations in the form of closures through the network. The distributed semantics of Obliq is based on the notions of: site, location, thread and values. A site is an address space that contains locations. Locations contain values. Each location belongs to a unique site. Sites are implicit in the syntax when a new location is created, since it is bound to the current site. Threads are virtual sequential instruction processors. Constant identifiers denote values, while variable identifiers denote locations. A location containing a value may be updated by assignment to the variable denoting the location.

In **Ambients** Cardelli and Gordon [12] introduce a process calculus that models the notions of ambients, agents, actions and capabilities. The calculus is then extended with asynchronous communication and variable binding primitives to model interaction. Mobility is described in terms of the movement of agents and their associated ambients and capabilities through a network. Ambients are bounded places where agents compute. They are self contained. Ambients encounter obstacles while moving through a network. Ambients use capabilities to allow others to perform local operations without revealing their true identity. Each capability is connected to an action (for example, presenting a password) that acts as a guard for the execution of the operation. Communication is free within an ambient whereas non-local messages are restricted with the use of capabilities.

Distributed Oz [67] incorporates the notion of mobility between sites in a fundamental way, directly in its kernel language OPM (Oz Programming Model). The kernel language semantics is refined by introducing the notion of sites. The basic OPM abstractions: values, unbound variables, procedures, cells and threads – are given default mobility behaviors (for example, cells are mobile and threads are stationary) that may be overridden by programmers. Other abstractions such as objects, classes and ports may be built upon this kernel. The language provides a *distributed shared computation space* and is state-aware which is fundamental for its definition of *mobility control*. Latency tolerance is supported through the use of logic variables. Language security is supported by using lexical scopes in references. This insures data and computation integrity in a way reminiscent of passing capabilities. Fault tolerance mechanisms are also present.

Chapter 10

Conclusions

10.1 Summary

We started this work with the objective of using the TyCO process calculus to design and implement a small kernel language that allowed an efficient implementation of objects in the framework of process calculi. Moreover the language and its implementation should provide a good test-bed upon which higher level programming abstractions could be implemented both in the object-oriented paradigm and in distributed computing with code mobility.

Our first goal was to produce a fine-grained, polymorphic, object-based concurrent programming language. In this thesis we have presented a new programming language called TyCO based on the similarly named process calculus. The language features as main abstractions: objects, asynchronous labeled messages and template processes. The language is strongly, implicitly typed and supports parametric polymorphism. Support for other object-oriented abstractions is also present. For example, classes are readily obtained from generic process templates, whereas the type-system allows the definition of a kind of behavior inheritance, which however does not involve subtyping. Other type systems can be coupled with TyCO to provide more powerful abstractions such as objects with non-uniform interfaces that can, for example, adapt the set of available methods to their evolving internal state.

Next, we wanted the virtual machine for the language to have a clean, compact, portable implementation and be self-contained. The virtual machine design and implementation we presented is based on a formulation of the dynamic semantics for the calculus which we proved to be sound. The semantics presented grows from Turner's work on the π-calculus by introducing support for objects (labeled transitions), unbounded behavior using recursion rather than replication and explicitly defining the concurrency unit as the *thread* – method or template bodies, rather than the primitive process unit. Besides soundness, the dynamic semantics also guarantees important properties such fairness, impossibility of deadlocks in well typed programs and an important communication queue invariant for the implementation.

The design and implementation of the TyCO virtual machine follows closely the abstract machine specification. The virtual machine uses a compact byte-code emulator design with a heap for dynamic data-structures, a run-queue for scheduling virtual machine threads, an operand stack to evaluate expressions and a binding array to keep the bindings of local variables. The machine executes architecture independent byte-code programs assembled from an intermediate assembly language.

Another important point regards the type safety of the byte-code generated by the compiler. In other words, given a byte-code file to execute, the emulator should not produce run-time protocol errors. The TyCO compiler uses a type-inference algorithm that statically type-checks programs and guarantees that the generated byte-code is type safe. The TyCO compiler uses basic syntactic analysis and type-inference information to guide the code generation. We provided examples and implementations of optimizations that use both syntactic and type information to short cut some very pervasive interactions in programs, namely, synchronous communication and the interaction with persistent objects. The performance gains for these optimizations were quantified and indicate that further optimization is not only possible but likely to produce important speedups.

Through a set of experiments we have shown that TyCO is quite efficient even by comparison with systems that run native code. TyCO performs close to Pict and Oz in programs that are mainly functional whereas in programs with non-trivial object data structures it outperforms them clearly both in speed and minimum total heap usage. TyCO was systematically faster than JoCaml in every program we tried.

Finally, we wanted a kernel language to experiment with distribution and code mobility in the context of process calculi. For this purpose we extended the TyCO calculus orthogonally by introducing the notion of sites or locations where computations occur, and the possibility of extruding local names and template definitions from within sites to a network of located computations. Computations provide (export) names that can be used by other computations to interact with the local one. To accomplish this, a remote computation must (import) some name that has been exported. Interaction can occur in one of three ways: through remote message sending, moving objects to a remote location and, downloading code from the network. Code mobility is induced in a semi-transparent way by the lexical scope of names and template variables.

The implementation of this distributed TyCO – DiTyCO – is the scope of an ongoing team project.

10.2 Future work

Future work on the TyCO language will focus on performance evaluation and fine tuning of the virtual machine implementation and of the TyCO compiler. This work will comprise:

- using channel usage information from the type system to optimize both the byte-code and the abstract machine's data structures operating on such channels;
- analyzing the generated byte-code to identify sequences of instructions that are widespread in the programs and merge them into single virtual machine instructions;
- studying register allocation strategies for the machine registers and pre-fetching techniques;
- defining an external language call interface to provide TyCO with tools such as system or API libraries;
- developing debugging and visualization tools to help programming and system development;
- introducing a higher level concurrent programming language based on TyCO with constructs for abstract data-types, a functional core and common communication patterns;
- further testing with larger, *real world* applications [52] to assess the system's performance, and fine-tune it.

There is still a lot of work to be done in order to produce a robust distributed programming language based on the DiTyCO idiom. The implementation of the basic computational model is the focus of another project but several issues remain unexplored:

- a distributed type checking mechanism to ensure secure interactions between remote sites. The idea is to have a mixture of static and dynamic type-checking with the byte-code carrying some type-information;
- we need to introduce fault-tolerance and termination detection in the system. We want to be able to detect site failures, to reconfigure the computation topology and to cleanly terminate computations. There is quite a volume of work that has been done in this subject and we want to profit from it as much as possible;
- optimize the implementation so that it performs well with heterogenous network hardware and variable network loads. This implies not only fine-tuning the C code, but also looking for ways to reduce the overhead from the underlying physical network. For example, given that the data that is sent across networks in this model is usually very fine grained, strategies for packing buffers together before sending them may improve performance considerably.

10.3 Final Remarks

The TyCO software distribution, version 0.2(alpha) is available from the Internet at `http://www.ncc.up.pt/~lblopes/tyco.html`. The release includes: a) a TyCO to intermediate assembly compiler (`tycoc`); b) a byte-code assembler (`tycoasm`); c) a byte-code emulator (`tyco`); d) example applications, and; e) miscellaneous documentation on the system.

Bibliography

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

[2] M. Abadi and A. Gordon. A Calculus for Cryptographic Protocols: the Spi-Calculus. In *Computer and Communications Security (CCS'97)*, pages 36–47. The ACM Press, April 1997.

[3] P. Achten and R. Plasmeijer. The ins and outs of Clean I/O. *Journal of Functional Programming*, 5(1):81–110, January 1995.

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

[5] G. Agha and C. Hewitt. Actors: A Conceptual Foundation for Concurrent Object-Oriented Programming. *Research Directions on Object-Oriented Programming*, 1981.

[6] R. Amadio. An Asynchronous Model of Locality, Failure, and Process Mobility. In *International Conference on Coordination Models and Languages (COORDINATION'97)*, volume 1282 of *LNCS*, pages 374–391. Springer-Verlag, 1997.

[7] A. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, 1998.

[8] G. Berry and G. Boudol. The Chemical Abstract Machine. *Theoretical Computer Science*, 96:217–248, 1992.

[9] G. Boudol. Asynchrony and the π-calculus (Note). Rapport de Recherche 1702, INRIA Sophia-Antipolis, May 1992.

[10] G. Boudol. The π-calculus in Direct Style. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97)*. Springer-Verlag, January 1997.

[11] L. Cardelli. A Language with Distributed Scope. *Computing Systems*, 8(1):27–59, January 1995.

[12] L. Cardelli and A. Gordon. Mobile Ambients. In *Foundations of Software Science and Computation Structures (FoSSaCS'98)*, volume 1378 of *LNCS*, pages 140–155. Springer-Verlag, 1998.

[13] F. Cardonne and M. Coppo. Two Extensions of Curry's Type Inference System. *Logic and Computer Science*, pages 19–75, 1990.

[14] L. Damas and R. Milner. Principal Type-Schemes for Functional Programs. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'82)*, pages 207–212, 1982.

[15] E. Dijkstra. Co-Operating Sequential Processes. *Programming Languages*, 1965.

[16] E. Dijkstra. Solution of a Problem in Concurrent Programming Control. *Communications of the ACM*, 8(9), 1965.

[17] A. Chien et al. High Performance Virtual Machines (HPVM): Clusters with Supercomputing APIs and Performance. In *Parallel Processing for Scientific Computing (PP'97)*, March 1997.

[18] N. Boden et al. Myrinet: A Gigabit per second Local Area Network. *IEEE-Micro*, 15(1):29–36, February 1995.

[19] F. Le Fessant and L. Maranget. Compiling Join-Patterns. In *Workshop on High Level Programming Languages (HLCL'98)*, volume 16(3) of *ENCS*, pages 35–54. Elsevier Science, 1998.

[20] C. Fournet and G. Gonthier. The Reflexive Chemical Abstract Machine and the Join-Calculus. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96)*, pages 372–385. ACM, 1996.

[21] C. Fournet, G. Gonthier, and et al. A Calculus of Mobile Agents. In *International Conference on Concurrency Theory (CONCUR'96)*, volume 1119 of *LNCS*, pages 406–421. Springer-Verlag, 1996.

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

[23] A. Gordon and P. Hankin. A Concurrent Object Calculus: Reduction and Typing. In *Workshop on High Level Programming Languages (HLCL'98)*, volume 16(3) of *ENTCS*, pages 79–95. Elsevier Science, 1998.

[24] M. Hennessy and J. Riely. Resource Access Control in Systems of Mobile Agents. Technical report, University of Sussex, February 1998.

[25] C. Hoare. Towards a Theory of Parallel Programming. In *Operating Systems Techniques*, number 9 in A.P.I.C. Studies in Data Processing, pages 61–71. Academic Press, 1972.

[26] C. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, 1978.

[27] K. Honda and M. Tokoro. An Object Calculus for Asynchronous Communication. In *European Conference on Object-Oriented Programming (ECOOP'91)*, volume 512 of *LNCS*, pages 141–162. Springer-Verlag, 1991.

[28] K. Honda, V. Vasconcelos, and M. Kubo. Language Primitives and Type Disciplines for Structured Communication-Based Programming. In *European Symposium on Programming (ESOP'98)*, volume 1381 of *LNCS*, pages 122–138. Springer-Verlag, 1998.

[29] K. Honda and N. Yoshida. Combinatory Representation of Mobile Processes. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'94)*, pages 348–360, January 1994.

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

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

[32] The *JC-team*. The JoCaml Home Page. `http://pauillac.inria.fr/jocaml`.

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

[34] D. Kafura and R. Lavender. Concurrent Object-Oriented Languages and the Inheritance Anomaly . In *International Summer Institute on Parallel Computer Architectures, Languages, and Algoritms (ISIPCALA93), Prague*, pages 183–213. Czech ACM Chapter, July 1993.

[35] N. Kobayashi. Quasi-Linear Types. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'99)*, pages pp. 29–42, January 1999.

[36] N. Kobayashi, B. Pierce, and D. Turner. Linearity and the π-calculus. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96)*, 1996.

[37] X. Leroy. Objective CAML Home Page. `http://pauillac.inria.fr/ocaml`.

[38] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1997.

[39] L. Lopes, F. Silva, A. Figueira, and V. Vasconcelos. DiTyCO: Implementing Mobile Objects in the Realm of Process Calculi. In 5^{th} *Mobile Object Systems Workshop (MOS'99)*, June 1999.

[40] L. Lopes, F. Silva, and V. Vasconcelos. A Programming Language based on a Process Calculus and its Virtual Machine. Submitted to *Theory and Practice of Object Systems.*

[41] L. Lopes, F. Silva, and V. Vasconcelos. A Virtual Machine for the TyCO Process Calculus. In *Principles and Practice of Declarative Programming (PPDP'99)*, volume 1702 of *LNCS*, pages 244–260. Springer-Verlag, September 1999.

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

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

[44] R. Milner. *Communication and Concurrency.* Preentice-Hall, 1989.

[45] R. Milner. Functions as Processes. *Mathematical Structures in Computer Science*, 2(2):119–141, 1992.

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

[47] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* The MIT Press, 1990.

[48] A. Mycroft. Polymorphic Type Schemes for Functional Programming. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'84).* The ACM Press, 1984.

[49] O. Nierstrasz. *Object-Oriented Software Composition.* Prentice Hall, 1995. pages 99–121.

[50] A. Ohori. A Compilation Method for ML-style Polymorphic Record Calculi. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'92)*, pages 154–165. The ACM Press, 1992.

[51] J. Parrow and B. Victor. The Fusion Calculus: Expressiveness and Symmetry in Mobile Processes. In *Symposium on Logic in Computer Science (LICS'98)*, pages 176–185. IEEE Computer Society Press, 1998.

[52] W. Partain. The nofib Benchmark Suite of Haskell Programs. In *Proceedings of Functional Programming Workshop*, Workshops in Computing, pages 195–202. Springer-Verlag, 1992.

[53] B. Pierce and D. Sangiorgi. Behavioral Equivalence in the Polymorphic Pi-Calculus. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97)*, 1997.

[54] B. Pierce and D. Turner. Pict: A Programming Language Based on the Pi-Calculus. Technical Report CSCI 476, Computer Science Department, Indiana University, 1997. To appear in *Proof, Language and Interaction: Essays in Honour of Robin Milner*, Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, MIT Press, 1999.

[55] A. Ravara. From π to TyCO. personal communication.

[56] A. Ravara and L. Lopes. Programming and Implementation Issues in Non-Uniform TyCO. In *Workshop on Object-Oriented Specification Techniques for Distributed Systems and Behaviors (OOSDS'99)*, 1999.

[57] A. Ravara and V. Vasconcelos. Behavioural Types for a Calculus of Concurrent Objects. In *International Conference on Parallel Processing (EURO-PAR'97)*, volume 1300 of *LNCS*, pages 554–561. Springer-Verlag, 1997.

[58] J. Riely and M. Hennessy. Distributed Processes and Location Failures. In *Automata, Languages and Programming (ICALP'97)*, volume 1256 of *LNCS*, pages 471–481. Springer-Verlag, 1997.

[59] S. Jones and T. Nordin and D. Oliva. C–: A Portable Assembly Language. In *Implementation of Functional Languages (IFL'97)*, 1997.

[60] D. Sangiorgi. The name discipline of receptiveness. In *Automata, Languages and Programming (ICALP'97)*, volume 1256 of *LNCS*. Springer-Verlag, 1997.

[61] D. Sangiorgi. An Interpretation of Typed Objects into Typed π-calculus. *Information and Computation*, 143(1):34–73, 1998.

[62] T. Sekiguchi and A. Yonezawa. A Calculus with Code Mobility. In *Formal Methods for Open Object-based Distributed Systems (FMOODS'97)*, 1997.

[63] P. Sewell, P. Wojciechowski, and B. Pierce. Location Independence for Mobile Agents. In *Workshop on Internet Programming Languages – International Conference on Computer Languages (ICCL'98)*, 1998.

[64] G. Smolka. A Fundation for Higher-Order Concurrent Constraint Programming. In *International Conference on Constraints in Computational Logics*, volume 845 of *LNCS*, pages 50–72. Springer-Verlag, September 1994.

[65] E. Sumii and N. Kobayashi. A Generalized Deadlock-Free Calculus. In *Workshop on High Level Programming Languages (HLCL'98)*, volume 16(3) of *ENTCS*, pages 55–78. Elsevier Science, 1998.

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

[67] P. van Roy, S. Haridi, and et al. Mobile Objects in Distributed Oz. *submitted to ACM TOPLAS*, 1997.

[68] V. Vasconcelos. *A Process Calculus Approach to Typed Concurrent Objects*. PhD thesis, Keio University, November 1994.

[69] V. Vasconcelos. Predicative Polymorphism in π-Calculus. In *Parallel Architectures and Languages Europe (PARLE'94)*, LNCS. Springer-Verlag, July 1994.

[70] V. Vasconcelos. Processes, Functions, Datatypes. *Theory and Practice of Object Systems*, 5(2):97–110, 1999.

[71] V. Vasconcelos and R. Bastos. Core-TyCO - The Language Definition. Technical Report TR-98-3, DI / FCUL, March 1998.

[72] V. Vasconcelos and K. Honda. Principal Typing Schemes in a Polyadic π-Calculus. In *International Conference on Concurrency Theory (CONCUR'93)*, volume 715 of *LNCS*, pages 524–538. Springer-Verlag, August 1993.

[73] V. Vasconcelos, L. Lopes, and F. Silva. Distribution and Mobility with Lexical Scoping in Process Calculi. In *Workshop on High Level Programming Languages (HLCL'98)*, volume 16(3) of *ENTCS*, pages 19–34. Elsevier Science, 1998.

[74] V. Vasconcelos and M. Tokoro. A Typing System for a Calculus of Objects. In *International Symposium on Object Technologies for Advanced Software (ISOTAS'93)*, volume 742 of *LNCS*, pages 460–474. Springer-Verlag, November 1993.

[75] J. Vitek and G. Castagna. Seal: A Framework for Secure Mobile Computations. In *Workshop on Internet Programming Languages*, 1999.

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

[77] D. Warren. An Abstract Prolog Instruction Set. Technical Report 309, SRI International, October 1983.

[78] A. Yonezawa. *ABCL, an Object Oriented Concurrent System*. The MIT Press, 1990.

[79] A. Yonezawa, E. Shibayama, and et. al. Modelling and Programming in an Object-Oriented Concurrent Language ABCL/1. In *Object-Oriented Programming*, pages 55–89. The MIT Press, 1987.

Index