On the Massively Parallel Execution of Declarative Programs

vorgelegt von
Diplom-Informatiker
Manuel M. T. Chakravarty

Vom Fachbereich 13 — Informatik —
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Naturwissenschaften
— Dr. rer. nat. —

genehmigte Dissertation

Promotionsausschuß:

Vorsitzender: Prof. Dr. Peter Pepper
Berichter: Prof. Dr.-Ing. Stefan Jähnichen
Berichter: Prof. Dr. John Darlington

Tag der mündlichen Prüfung: 14. Februar 1997

Berlin 1997

D 83

ON THE MASSIVELY PARALLEL EXECUTION OF DECLARATIVE PROGRAMS

Manuel M. T. Chakravarty

Technische Universität Berlin
Fachbereich Informatik
Institut für Kommunikations- und Softwaretechnik
Forschungsgruppe Softwaretechnik

and

GMD FIRST

chak@cs.tu-berlin.de
http://www.cs.tu-berlin.de/~chak/

Zusammenfassung

Funktionen und Relationen lassen sich zu einem Formalismus kombinieren, der zur deklarativen Beschreibung paralleler Programme geeignet ist. In diesem Formalismus beschreiben Funktionen schrittweise ausgeführte, sequentielle Berechnungen, wohingegen das parallele Zusammenspiel dieser Berechnungen durch Relationen spezifiziert wird. Wie in einer aktuellen Arbeit gezeigt wurde, kann in einem solchen Rahmen ein Typsystem die Integrität der Semantik der rein funktionalen Unterberechnungen garantieren. Daher kann solch eine Integration funktionaler und logischer Konzepte als eine konservative Erweiterung des rein funktionalen Berechnungsmodells angesehen werden.

Die in dieser Dissertation beschriebenen Forschungsergebnisse verallgemeinern Resultate des rein funktionalen Programmierens für das resultierende, neue Berechnungsmodell. Insbesondere wird eine Erweiterung des typisierten λ-Kalküls eingeführt, die die übliche Semantik der rein funktionalen Teilberechnungen erhält. Desweiteren wird eine bestehende abstrakte Maschine für kompilierte Graphreduktion um die explizite Behandlung massiv paralleler Berechnungen ergänzt, die speziell auch Optimierungen und die Codegenerierung für Parallelrechner mit verteiltem Speicher unterstützt.

Die Erweiterung des λ-Kalküls, im weiteren $\mathcal{D}$ genannt, beinhaltet ein einfaches Konzept von Agenten, die durch Relationen repräsentiert werden und die parallel und sequentiell zu komplexeren Agenten kombiniert werden können. Sie kommunizieren mit Hilfe von Gleichheitsconstraints über ungebundenen Variablen. Agenten können ein indeterministisches Verhalten aufweisen, was für eine Klasse paralleler Algorithmen wichtig ist. Weiterhin sind Agenten in dem Sinne vollwertig, daß sie Programme höherer Ordung unterstützen. Berechnungsorientierte Aufgaben werden innerhalb eines Agenten durch rein funktionale Teilberechnungen verwirklicht. $\mathcal{D}$ stellt einen deklarativen Rahmen zur parallelen Programmierung dar, denn es besitzt eine logische Semantik, beschrieben in linearer Logik, und eine von der logischen Semantik abgeleitete, parallele operationale Semantik.

In dem zweiten Teil der Arbeit wird die „Spineless Tagless G-Machine“, die eine der effizientesten Implementierungen kompilierter Graphreduktion darstellt, derart erweitert, daß sie als eine Verfeinerung der operationalen Semantik von $\mathcal{D}$ betrachtet werden kann. Von grösserer Bedeutung ist jedoch eine weitere Eigenschaft der neuen abstrakten Maschine: Bei Parallelrechnern mit verteiltem Speicher treten durch die Kommunikation unweigerlich Latenzzeiten auf. Diese stellen eines der wesentlichen Hindernisse für die effiziente Implementierung deklarativer Sprachen auf solchen Hochleistungsrechnern dar. Durch den Einsatz mehrer Kontrollflüsse in der abstrakten Maschine wird die Auswirkung dieser Latenzzeiten reduziert. Die operationale Semantik der abstrakten Maschine wird formal spezifiziert, und die Abbildung auf konkrete Parallelrechner diskutiert. Ergebnisse, die mit Hilfe eines Emulators gewonnen wurden, der die operationale Sematik der abstrakten Maschine implementiert, belegen die Wirksamkeit des Einsatzes mehrfacher Kontrollflüsse.

Abstract

Functions and relations can be combined into a declarative notation for describing parallel programs, where step-at-a-time computations being expressed purely functionally, whereas parallel behaviour is specified relationally. A recent proposal demonstrated that such an integration can use a type system to guarantee the semantic integrity of purely functional subcomputations—and it can therefore be regarded as a conservative extension of purely functional programming.

The research documented in this dissertation generalizes results from purely functional programming to this extended setting. In particular, this work extends the typed lambda calculus by co-ordinating relations, but in such a way that it preserves the conventional semantics for purely functional subcomputations. Furthermore, it describes an extension of an abstract machine for compiled graph-reduction, which provides explicit support for optimizations and code generation, targeted at distributed-memory parallel computers.

The extension of the lambda calculus, called $\mathcal{D}$, includes a basic notion of agents, which are represented by relations and may form parallel ensembles. Such agents communicate via equational constraints on unbound variables; and they can exhibit indeterminate behaviour, which is important for some parallel algorithms. Agents are first-class values in the sense that they support full higher-order programming. Computational tasks are realized within these agents as purely functional subcomputations. $\mathcal{D}$ is a declarative framework for parallel programming, which has a logic semantics based on an encoding into linear logic as well as a parallel operational semantics derived from the logic semantics.

This work also introduces an extension of the the Spineless Tagless G-machine, one of the most efficient implementations of compiled graph-reduction, that covers all features of $\mathcal{D}$—which can be seen as a refinement of the operational semantics of $\mathcal{D}$. More importantly, the abstract machine is revised to integrate multi-threading as a mechanism for tolerating communication latencies. Such latencies are inherent in distributed-memory parallel computers and represent one of the major obstacles to an efficient implementation of declarative languages on these computers. The operational semantics of the abstract machine is specified formally and its mapping to concrete parallel hardware is discussed. The effectiveness of the multi-threading extension is demonstrated by results obtained using an emulator, which implements the parallel operational semantics of the abstract machine.

To peace,
within us,
among us,
and with Nature.

PREFACE

ALMOST five years ago, Martin Köhler and I set out to investigate the use of declarative languages for parallel computing, focusing on the integration of equational constraints into functional programming. We were striving for a declarative language that includes an explicit notion of parallelism. The result was not unlike the dataflow language Id; and like Id, we lost referential transparency. After a while, Hendrik Lock and Yike Guo joined us in our endeavour; together we discovered the importance of differentiating between step-at-a-time computational activities from the co-ordination of parallel ensembles. Based on this insight, we developed ideas on how the type system might help to enforce a proper separation of the two, enabling the separation to be statically enforced by a compiler. The result was the constraint functional language GOFFIN, which is an extension of the purely functional Haskell.

It was clear to us that we would need a good implementation, and we started to think about a parallel extension of the JUMP machine, which was designed as an abstract machine for the efficient implementation of functional logic languages. It was at this time that I became interested in recent developments in the multi-threading technology of the dataflow community and began integrating these concepts into the JUMP machine. Unfortunately, this resulted in an approach that favoured a purely operational view, and thus was not satisfying. In contrast, the concise and formal specification of the Spineless Tagless G-machine (STGM) seemed very attractive, especially because the abstract machine is built around a purely functional intermediate language with a denotational semantics and a quite concrete operational semantics. This was the starting point for the work that culminated in the present dissertation.

I attempted to integrate multi-threading into the STGM, with the goal of exposing thread-based optimizations at the source level of the machine language. At the same time, I tried to extend the STG language, to make it support the relational features of GOFFIN. I firmly believed that I would not succeed in finding an elegant operational specification of the abstract machine until I understood properly the abstract, declarative meaning of the functional logic machine-language. Indeed, the fruitful interaction between the theoretical and practical results—presented in Chapters 3 and 4, respectively—was for me the most exciting aspect of this work.

Acknowledgements

It is my special pleasure to thank the many people who have contributed in some way to the work presented here. My supervisor, Stefan Jähnichen, initially drew my attention to functional and logic programming languages; he gave me the opportunity to work on declarative languages, and later on their parallel implementation. The work environment he provided

was highly stimulating, and his firm support was very helpful. And last but not least, I am indebted to him for his advice on conduct within the scientific community. I am grateful to John Darlington for his hospitality during my visits to the Advanced Languages and Architectures group at Imperial College, London, which proved very important for my research. I would also like to thank Peter Pepper for tolerating my stubbornness in fixing a date for my oral examination.

I was lucky enough to have three highly competent and courageous collaborators in my quest for a declarative description of parallel algorithms. Yike Guo, Martin Köhler, and Hendrik C. R. Lock not only worked with me on the invention of the language GOFFIN; they have also been good friends and pleasant company. Each one of them has been a great help to me while writing this dissertation. Yike repeatedly gave me valuable advice on the right literature to consult when I needed it most. He also helped me structure the work and cooked fabulous meals together with his wife Wu Qian. Martin Köhler helped me launch the whole project in Berlin; his move to London made our meetings less frequent, but no less fruitful or less cordial. He also provided me with helpful feedback on the dissertation. Hendrik Lock guided me as I took my first steps in the scientific community and I have always appreciated his advice. I am very grateful for his detailed and insightful comments on an earlier draft of the dissertation. Discussions with all three of them have helped shape my way of thinking.

Martin Simons shared an office with me for five years. Our many discussions on computing in general, and parallel computing in particular, had a subtle, but important influence on the course of my work and its presentation. Our friendship made the work much more pleasant, and it was great advantage to have the local TeX guru sitting in my office.

I am indebted to Simon L. Peyton Jones for discussions on the sequential and parallel implementation of declarative languages, and especially for his comments on my attempt to integrate multi-threading into the STGM. These comments had an indirect, but nevertheless important, influence on the structure of Chapter 4. I am also grateful to Silvia Breitinger and Rita Loogen for our discussions on parallel programming with declarative languages. My thanks also go to Dragan Macos for his helpful comments on a draft version of this dissertation.

I am grateful to Phil Bacon for answering all my questions on the English language with competence and friendliness, and for proofreading parts of this dissertation. I like to thank Wilhelm Köhler for patiently answering my questions and helping me with the local computing facilities. I am very grateful to the many people who wrote the free TeX and LaTeX typesetting software and accompanying packages. The Software Engineering Research Group headed by Stefan Jähnichen provided a pleasant work environment, and I would like to thank all past and present members of the group for the stimulating atmosphere I found there.

My warmest thanks go to Gabriele Keller. She helped me greatly in my work by listening to my ideas and by proofreading this dissertation. But, more important, she has been a caring friend and lover throughout, bearing with me through the ups and downs of my work. She must surely have found the writing of the dissertation as exhausting as I did, but still managed to keep in good spirits and offer help wherever possible.

I am indebted to my parents in many ways, not least for their support during my early and sometimes violent attempts to become a scientist. Finally, I would like to thank all my friends who didn't get to see me a lot over the past few months.

Berlin, February 1997 Manuel M. T. Chakravarty

CONTENTS

Chapter 1

INTRODUCTION

DECLARATIVE programming languages have long been seen as good candidates for programming parallel computers. Their level of abstraction promises portability, and their clean semantics makes them suitable for optimizing program transformations, on the source level and during compilation. However, researchers have found it difficult to present efficient parallel execution models for declarative languages. Generally speaking, it appears to be impossible to automatically identify the exact parallelism that leads to a reduction in execution time, and the irregular access to dynamic data structures can result in considerable overhead on distributed-memory machines.

While the main representatives of declarative programming, namely functional and logic programming, are usually investigated and applied separately, there is evidence that they are complementary and should be used together when it comes to expressing parallel programs at a high level of abstraction. These two facets of declarative programming naturally complement each other in describing the computational and co-ordinative aspects that constitute a parallel program—making a distinction between computation and co-ordination in parallel programming has been advocated by Carriero & Gelernter (1992). Chakravarty, Guo, Köhler & Lock (1997) have proposed the combined use of the functional and logic paradigm for parallel programming. Let us consider the basic idea here.

Functional programs concisely specify step-at-a-time computations; but even with a high degree of freedom in the evaluation order, the programmer has—without extra provisions—no way of specifying the subexpressions that are *usefully* evaluated in parallel. On the other hand, the paradigm of relational programming has a natural interpretation in a concurrent setting; a formula such as

$$\exists x . p\ x \wedge q\ x$$

can be regarded as two concurrent agents $p\ x$ and $q\ x$ that are communicating via a common, but local channel x. But without additional means for specifying sequential computations that are encapsulated in those agents, the model is too fine-grained.

The use of arbitrary functional expressions in the arguments of predicates allows the respective strengths of the functional and the relational paradigm to be combined. The formula

$$\exists x, y . (x = f\ a) \wedge (y = g\ b) \wedge p\ x\ y$$

specifies three parallel agents; the equational constraints communicate the results of evaluating the expressions $f\ a$ and $g\ b$ via x and y to the agent $p\ x\ y$.[1]

Where function evaluation expresses step-at-a-time computations and relational formulae specify parallel co-ordination, there is no need to automatically determine a decomposition of a declarative program into subcomputations that can usefully be evaluated in parallel. This decomposition is now the responsibility of the programmer, nevertheless a declarative programming style is preserved.

Some researchers have proposed combining—at least conceptually—the sets of programming languages commonly termed "functional languages" and "logic languages" into a wider set of declarative languages, which naturally includes combined functional logic languages. In this context, the criterion for calling a programming language *declarative* is the existence of a clear, mathematically established correspondence between the language and mathematical logic—such that a declarative semantics for the language can be based on the model or the proof theory (or both) of the logic. Accordingly, we term a programming language that is declarative and at the same time provides an abstract notion of parallelism a *declarative parallel language*. One way of providing this abstract notion of parallelism is to separate co-ordination and computation, as discussed above.[2]

Nevertheless, even with the separation of computation and co-ordination, a declarative program still operates on its data structures in a highly irregular way. The same data structure may be examined, or even jointly produced, by a number of agents, which may be spread over several processor elements of a distributed-memory computer. The result is rapid and unpredictable communication between these processor elements, and as a consequence, computations will repeatedly have to wait because of the latency induced by interprocessor communication.

The high degree of freedom in the evaluation order of the subexpressions—which constitute the functional description of the computations—is the obvious asset for alleviating communication latency. If the reduction of a subexpression stalls, other subexpressions may meanwhile be processed. Compiler-controlled multi-threading has been proposed for scheduling fine-grained computations efficiently (Culler, Goldstein, Schauser & von Eicken 1993, Engler, Andrews & Lowenthal 1993, Hum et al. 1995). I have suggested applying this technique for the implementation of lazy functional programming languages (Chakravarty 1996).

The present dissertation investigates declarative parallel languages that are based on the separation of computation and co-ordination, and more specifically, their implementation on massively parallel computers and the use of compiler-controlled multi-threading to increase their efficiency. To this end, we begin by defining a declarative semantics for functional-logic specifications of parallel computations, then refine this semantics into a parallel execution model, eventually obtaining an abstract machine that exploits multi-threading for latency hiding. We conclude with a discussion of the implementation of the abstract machine on concrete parallel hardware.

Goals

The research documented in this dissertation had two main goals.

---

[1] Following the notation of modern functional programming languages—such as Haskell (Hudak et al. 1992)—function (and predicate) application is denoted by juxtaposition of the function and its argument, assuming that all functions are curried.

[2] Note that, in principle, any declarative language can be implemented in parallel, but this does not make it a declarative parallel language—it merely makes it a declarative language with a parallel implementation.

- First, to develop a calculus that expresses fundamental principles of task-parallel computing (such as parallel composition, communication, indeterminate choice, but also sequential computation) in a declarative framework based on the separation of coordination and computation.

- Second, to design an implementation technology for declarative parallel languages that are covered by this calculus, paying special attention to the problem of latencies induced by remote access in distributed-memory computers.

A result of pursuing these goals should be a declarative core language for the parallel language GOFFIN (Chakravarty, Guo, Köhler & Lock 1997) together with an implementation of GOFFIN. The core language should have a logic and an operational semantics, the logic semantics capturing the concepts of indeterminate choice and deferred evaluation (i.e., the suspension of functional reduction on unbound variables). While providing a new semantic foundation for GOFFIN, the core language—and especially, its operational semantics—is meant to form a basis for the development of an implementation technology that covers GOFFIN. The implementation techniques should be geared to the code generation for massively parallel computers, which usually use distributed memory. Hence, latency due to remote-memory accesses is considered to be one of the main efficiency concerns of such an implementation. An attempt will be made to integrate multi-threading into the implementation, to limit the impact of such latencies.

Organization of the Dissertation

The rest of the dissertation is organized as follows.

Basics—Chapter 2. This chapter takes a more detailed look at the basic idea of declarative parallelism and presents the language GOFFIN as an example of a declarative parallel programming language. It also contains the mathematical notation and some basic definitions used throughout the dissertation.

Declarative Parallelism—Chapter 3. This introduces the language $\mathcal{D}$, which is essentially an extension of the lambda calculus by a notion of names and relations to specify parallelism. Its logic semantics is defined by a mapping to linear logic, and this semantics is subsequently used to derive a complete parallel operational semantics.

Execution Model—Chapter 4. The operational semantics of $\mathcal{D}$ is refined into an abstract machine model. The design of the abstract machine is based on the Spineless Tagless G-machine, which is extended to cover the relational aspects and to exploit multi-threading to tolerate long latency communication operations. This extension enables us to express optimizations concerning the thread scheduling as source-to-source transformation of the abstract machine language. Experiments with a prototypical implementation are reported, providing evidence for the claim that these optimizations increase the tolerance of the code for long latency operations.

Implementation—Chapter 5. This chapter outlines the mapping of the abstract machine to concrete distributed-memory parallel computers. In particular, it discusses the efficient implementation of compiler-controlled multi-threading.

Conclusion—Chapter 6. A summary of results and a discussion of future work conclude the dissertation.

Linear Logic—Appendix A. This appendix gives a brief introduction to linear logic.

Haskell Prelude—Appendix B. This appendix lists definitions of functions from Haskell's prelude, which are used in Chapter 4.

A glossary and bibliography can be found at the end of the dissertation.

Chapter 2

BASICS

BEFORE taking up the technical discussion in the next chapter, we look briefly in the following sections at declarative languages, their use for parallel computing, and their implementation. We also discuss the constraint functional language GOFFIN and list some mathematical definitions and notations that are used throughout.

2.1 Declarative Programming Languages

The distinguishing feature of declarative languages is a clean and concise mathematical semantics, which is usually provided by mathematical logic. In the case of relational logic programming, the semantics is traditionally provided by first-order predicate logic (see, e.g., Lloyd 1987); and in the case of functional programming, the foundation is the lambda calculus (see, e.g., Gunter 1992). The virtues of programming languages with a sound mathematical basis have been enumerated by many researchers, so I will not got into this further.

2.1.1 Parallel Programming

It is sometimes argued that a portable and efficient use of parallel computers can already be achieved by designing languages that do not introduce unnecessary sequential interdependencies between the subcomputations of a program. This view is especially common in the context of declarative languages, which often provide a high degree of freedom in the evaluation order of program components. However, the current research results strongly indicate that such a view is overly optimistic. Parallelism requires the explicit attention of the programmer, especially when dealing with distributed-memory architectures and where there is a substantial number of processors; the programmer has to identify those opportunities for parallel execution that are worthwhile with a view to improving execution time. The challenge here is to explicitly identify worthwhile parallelism in the source code, while preserving a declarative style of programming. This challenge clearly originates from the fact that parallel execution is an operational feature, while declarative programming tends to emphasize a denotational view. This apparent contradiction calls for a broader view of the semantics of declarative programming languages and, much in the spirit of Hoare (1994), requires the incorporation of denotational and operational characterizations into the definition of languages.

By way of example, we discuss below the constraint-functional language GOFFIN, which co-ordinates parallel functional computations by means of relations. Later on, the next chapter presents a foundation for declarative parallelism that is characterized by a logic and an operational semantics. The presented formalism combines concepts from functional and relational programming to express sequential step-at-a-time computations and their combination in parallel ensembles, respectively.

2.1.2 Implementation

The mathematical rigour at the basis of declarative languages is commonly justified by the advantages it has for program development and the design of programming languages. But it also has a strong impact on the implementation—and not only when considering the correctness of an implementation.

Declarative languages offer the programmer a high level of abstraction, which makes life hard for the language implementor. There is a big gap between the abstractions of the source language and the details of an actual implementation, and in case of a parallel implementation the situation is aggravated by the additional complexities of asynchronous message passing and multiple threads of control. A clear semantics, denotational and operational, helps to structure the translation process and to break it down into subtasks that can be understood by themselves. Furthermore, the quest for a clear and concise semantics leads to orthogonal languages, which are generally easier to implement. In addition, the strong mathematical results that are available for declarative languages support the development of aggressive optimization and transformation techniques in compilers (Peyton Jones, Hall, Hammond, Partain & Wadler 1993, Didrich, Fett, Gerke, Grieskamp & Pepper 1994, Van Roy 1990, Somogyi, Henderson & Conway 1996, Peyton Jones 1996, Peyton Jones, Partain & Santos 1996).

Abstract machines have always played an important rôle in the implementation of declarative programming languages (Warren 1983, Augustsson 1987, Johnsson 1987, Peyton Jones 1987, Aït-Kaci 1991). They help to decompose the translation process into substeps and to build models of the execution process. And there is evidence that abstract machines with a high-level (i.e., block-structured and, preferably, even declarative) machine language are particularly helpful (Van Roy 1990, Peyton Jones 1992, Sestoft 1997, Chakravarty & Lock 1997)—they simplify code optimization and facilitate reasoning about their correctness, both formally and informally.

Chapter 4 introduces an abstract machine designed to implement parallel declarative languages that are covered by the semantic framework of the next chapter. Chapter 5 goes on to outline the mapping of the abstract machine model to concrete parallel architectures with distributed memory.

2.2 Declarative Parallelism in Functions and Relations

We have introduced the idea of declarative parallel programming by a combination of functions and relations (Chakravarty, Guo, Köhler & Lock 1994), subsequently refining this idea into the programming language GOFFIN (Chakaravarty, Guo, Köhler & Lock 1995, Chakravarty, Guo, Köhler & Lock 1997). The latter is an extension of the de facto standard for non-strict, purely functional programming, namely Haskell (Hudak et al. 1992). This section illustrates the idea of declarative parallel programming by introducing the language GOFFIN and discussing some small examples.

2.2.1 Functions in Haskell

Haskell is a strongly typed, polymorphic language that supports algebraic data types and emphasizes the use of higher-order functions. Haskell forms the functional kernel of Goffin. We briefly discuss the Haskell style of programming using the function *map* as an example. Given a function that maps an argument of type α to a result of type β (written $\alpha \to \beta$), and a list of elements of type α (written $[\alpha]$), *map* returns a list of elements of type β.

$$\begin{array}{lll}
map & :: (\alpha \to \beta) \to [\alpha] \to [\beta] & \text{— type assertion} \\
map\ f\ [] & = [] & \text{— [] is the empty list} \\
map\ f\ (x : xs) & = f\ x : map\ f\ xs & \text{— : is the infix list constructor}
\end{array}$$

The selection between the two different cases proceeds by comparing the patterns in the definition with the arguments supplied in an application of *map*. Here, only the top-level constructor of the second argument is inspected. In Haskell, function application is denoted by juxtaposition, a very convenient notation in the presence of curried higher-order functions. The expression $map\ (+\,1)\,[1, 2, 3]$, for example, evaluates to $[2, 3, 4]$. The expression $(+\,1)$ is the addition operation $+$ applied to 1, but with an argument still missing—i.e., it evaluates to the increment function. Similarly, all functions in Haskell are curried, and the expression $map\ (+\,1)$ is itself a function, which maps a list of numbers to a list where each number is incremented by one.

2.2.2 Co-ordination by Relations

As explained in the introduction, a declarative programming language, such as Haskell, becomes a declarative *parallel* language when an abstract notion of parallelism is added. In the case of Goffin, we achieve this transformation by enriching the purely functional language with relations that have a co-ordinative meaning. The essential idea here is conveyed by the slogan *functions compute, relations co-ordinate.* The dualism of computation and co-ordination has been succinctly described by Carriero & Gelernter (1992):

> We can build a complete programming model out of two separate pieces: the *computation model* and the *co-ordination model.* The computation model allows programmers to build a single computational activity: a single-threaded, step-at-a-time computation. The co-ordination model is the glue that binds separate activities into an ensemble. An ordinary computation language (e.g., Fortran) embodies some computation model. A co-ordination language embodies a co-ordination model; it provides operations to *create* computational activities and to support *communication* among them.

The co-ordination model has to provide (i) *glue*, (ii) operations to *create* computational activities, and (iii) support for *communication* among such activities. All this is accomplished by relations in Goffin.

The Glue

Given separate activities, we combine them by conjunction—in the sense that the ensemble succeeds if and only if all parts succeed. Given activities a_1 up to a_n, we write

$$\{a_1, \ldots, a_n\}$$

to combine them by conjunction, and mean that a_1 to a_n are *parallelly composed.* The special case of a conjunction over zero arguments, denoted by $\{\}$, means instant success. It is the neutral of general conjunction, i.e.,

$$\{a_1, \ldots, a_{i-1}, \{\}, a_i, \ldots, a_n\} = \{a_1, \ldots, a_{i-1}, a_i, \ldots, a_n\}$$

Creation of Activity

Haskell is strongly typed; so we need a type for the components and the result of conjunction. In GOFFIN, this is O—a Boolean type, since an activity succeeds or fails. We call expressions of type O *agents.* Functions yielding agents, such as

$$\begin{aligned} &\mathit{and} && :: O \to O \to O \\ &\mathit{and}\ x\ y && = \{x, y\} \end{aligned}$$

are called *relations*; generally, each function of type $\tau \to O$ is a relation.

Communication

GOFFIN realizes communication by *unbound (existentially quantified) variables* and *equational constraints*, unbound variables defining communication channels, and equational constraints providing information on these channels. Given two relations $p, q :: \tau \to O$, we conjoin them and link them over a common channel $x :: \tau$ by the following composite agent:

$$\{x\ \mathbf{in}\ p\ x, q\ x\} \qquad \text{, which means } \exists x.(p\ x \wedge q\ x)$$

In front of the keyword **in**, we allow a list of new, unbound variables separated by commas, which are visible up to the closing curly brace.

If one of the agents, say $p\ x$, wants to communicate a value a of type τ to q, it can do so by placing the equational constraint $x \leftarrow a$ on to x. This can be observed by q, and we can regard p as the sender and q as the receiver of a message a that has been sent over the channel x. More generally, an equational constraint $e_1 \leftarrow e_2$ demands equality between e_1 and e_2, which implies that any unbound variables occurring in e_1 or e_2 are appropriately restricted—or expressed in more operational terms, e_1 and e_2 are unified.[1]

A Small Example

Consider the following composite agent:

$$\{x, y, r\ \mathbf{in}\ x \leftarrow \mathit{foo}\ a, y \leftarrow \mathit{bar}\ a, r \leftarrow x + y\}$$

From a purely computational perspective, the above code constrains r to be equal to $(\mathit{foo}\ a + \mathit{bar}\ a)$. But the agent also has a co-ordination aspect, which involves the creation of parallel computations as well as the synchronization and communication between these parallel computations. In this example, there are three parallel computations, which are embodied by the three equational constraints that contain the evaluation of the expressions $\mathit{foo}\ a$, $\mathit{bar}\ a$, and $x + y$. As the computation of the sum depends on the values of x and y, the functional reduction of the expression $x + y$ is *suspended* until both x and y are instantiated by the other two constraints. The mechanism of suspending reduction on unbound variables is called *deferred evaluation.*

---

[1] The use of the symbol $\leftarrow$ instead of $=$ for equational constraints has purely syntactic reasons—extending a given language while demanding compatibility, the choice of symbols is strongly restricted.

2.2.3 A Simple Example: Parallelizing *map*

Reconsider the function *map*, which is purely computational. It applies a function f over a list, yielding a list of result values. When f has to perform a sufficient amount of work for each list element, it may be desirable to compute the elements of the resulting list in parallel. Such behaviour is specified by the relation *farm*:

$$\begin{array}{ll}
\mathit{farm} & :: (\alpha \rightarrow \beta) \rightarrow [\alpha] \rightarrow [\beta] \rightarrow O \\
\mathit{farm}\ f\ []\ r & = \{r \leftarrow []\} \\
\mathit{farm}\ f\ (x : xs)\ r & = \{ys\ \mathbf{in}\ \ r \leftarrow (f\ x) : ys, \\
 & \qquad\qquad\quad \mathit{farm}\ f\ xs\ ys\}
\end{array}$$

In essence, an agent of the form $\{r\ \mathbf{in}\ \mathit{farm}\ \mathit{foo}\ [v_1, \ldots, v_n]\ r\}$ unfolds to

$$\begin{array}{ll}
\{r, ys_1, \ldots, ys_n\ \mathbf{in} & r \leftarrow (\mathit{foo}\ a_1) : ys_1, \\
 & ys_1 \leftarrow (\mathit{foo}\ a_2) : ys_2, \\
 & \qquad\quad \vdots \\
 & ys_{n-1} \leftarrow (\mathit{foo}\ a_n) : ys_n, \\
 & ys_n \leftarrow []\}
\end{array}$$

Again, relations are used as a means of organizing the parallel reduction of a set of functional expressions, namely the $(\mathit{foo}\ a_i)$. Relations like *farm* capture recurring forms of co-ordination, which can be used in different contexts; this often relies on the ability to treat functions as values (for example, the first argument of *farm*). Higher-order functions and relations are valuable tools for decomposing problems and reusing code—in short, for modular program development (Hughes 1989).

Note that choosing between *map* and *farm* is the responsibility of the programmer, no automatic parallelization is assumed—nor is it feasible if we consider that the choice depends on the computational complexity of the function f that is passed as an argument.

This example is taken from Chakravarty, Guo, Köhler & Lock (1997), who present a number of other examples, including a parallel floorplan-optimization algorithm that is useful for optimal utilization of the area available on a VLSI chip. The algorithm is realized as an instantiation of a generic parallel branch-and-bound search using higher-order functions.

2.3 Definitions and Notation

Later on, particularly in the next chapter, we use some definitions and notations that are introduced below. This section is deliberately kept brief; a more detailed treatment of topics such as lambda terms, renaming, and substitutions is given in standard textbooks (see, e.g., Gunter 1992).

2.3.1 Definitions, Theorems, etc.

We denote defining equality by the symbol $\triangleq$. Thus, $M \triangleq N$ means that M is defined by N. Within a particular chapter, we use a consecutive numbering scheme for all numbered entities (such as definitions, lemmata, theorems, and so forth). When displayed equations (single or multiple) are numbered, this is done mostly in the same format used to display definitions, etc. Such displayed equations have the following form:

2.1 An Important Equation:

$$M \triangleq N$$

Whether such an equation constitutes a definition or lemma is always obvious from the relation used. If, as above, $\triangleq$ is used, we have a definition.

2.3.2 Kinds, Types, and Lambda Terms

We use *kinded polymorphic lambda terms* as a basic term structure, which is enriched by constants that possess a special meaning to form extended languages. The polymorphic lambda calculus (Huet 1990, Part II) extends the simply typed lambda calculus by abstraction and application of types, and it has been repeatedly used as a basis for studying polymorphically typed functional programming languages and their implementation (e.g., Harper & Mitchell 1993, Henglein & Jørgensen 1994, Hall, Peyton Jones & Sansom 1994). The following presentation of the polymorphic lambda calculus adds *kinds* to the classical definition as a means of imposing a simple structure of disjoint sets on the types.

2.2 Definition (Kinds)
Given a set of *kind names* $\mathcal{C}_{\mathrm{K}}$ that contains at least the element $\star$, we define the set of *kinds* $\mathcal{K}$ as the smallest set including

- $\kappa \in \mathcal{C}_{\mathrm{K}}$ and
- $\kappa_1 \to \kappa_2$ when $\kappa_1, \kappa_2 \in \mathcal{K}$. ∎

Kinds are used to classify types. Thus, the following definition of types associates a kind with each type.

2.3 Definition (Types)
Let $\mathcal{V}_{\mathrm{T}}$ be a set of *type variables* and $\mathcal{C}_{\mathrm{T}}$ a non-empty set of *type constructors*, where each type constructor c is associated with a kind κ, written c^κ. The set of *pre-types* is produced by the following rule:

$$\begin{array}{rcll}
\hat{\mathcal{S}} & \to & \mathcal{C}_{\mathrm{T}} & \text{(constant)} \\
 & | & \mathcal{V}_{\mathrm{T}} & \text{(variable)} \\
 & | & \hat{\mathcal{S}}_1\,\hat{\mathcal{S}}_2 & \text{(application)}
\end{array}$$

We denote the set of variables that occur in a pre-type $\sigma \in \hat{\mathcal{S}}$ by $\mathrm{Fr}\,\sigma$.

When a (pre) type σ is of kind κ, we write $\sigma :: \kappa$. A *kind assignment* is a set Γ_{K} of associations $\alpha_1 :: \kappa_1, \ldots, \alpha_n :: \kappa_n$, where all α_i are distinct. We define a pre-type σ as having kind κ under the kind assignment Γ_{K}, when $\sigma :: \kappa$ holds under the following inference rules:

$$\frac{c^\kappa \in \mathcal{C}_{\mathrm{t}}}{c^\kappa :: \kappa}\ (\text{tcon}) \qquad \frac{\alpha \in \mathcal{V}_{\mathrm{t}}}{\alpha :: \kappa \in \Gamma_{\mathrm{K}}}\ (\text{tvar})$$

$$\frac{\sigma_1 :: \kappa_1 \to \kappa_2 \qquad \sigma_2 :: \kappa_1}{\sigma_1\,\sigma_2 :: \kappa_2}\ (\text{tapp})$$

With respect to a given kind assignment, we call a pre-type a *simple type* if it has a kind according to the above rules—it is easy to verify that the kind of a simple type is unique with respect to a given assignment. Simple types are also called monotypes.

Given a kind assignment Γ_K, a *type scheme* has the form

$$\forall \alpha_1^{\kappa_1}, \ldots, \alpha_n^{\kappa_n}.\sigma \qquad \text{or, more concisely,} \qquad \forall\, \overline{\alpha^\kappa}.\sigma$$

where the α_i are distinct type variables and σ is a simple type with respect to the assignment $\{\alpha_1 :: \kappa_1, \ldots, \alpha_n :: \kappa_n\} \triangleright \Gamma_\mathrm{K}$—where $\triangleright$ denotes set union and occurrences in the left assignment override those in the right. The special case where n is zero is permissible, and we often omit the symbol $\forall$ in this case. The free variables $\mathrm{Fr}\,\tau$ of a type scheme are defined as $\mathrm{Fr}\,\sigma \setminus \{\alpha_1, \ldots, \alpha_n\}$. Type schemes are also called polytypes.

Assuming some kind assignment, we denote the set of simple types by $\mathcal{S}$ and the set of type schemes by $\mathcal{T}$. If a type does not contain any free variable, we call it a *closed type*, and in this case, we need not assume any kind assignment. ▮

A type of kind $\kappa \in \mathcal{C}_\mathrm{K}$ (i.e., where κ is a kind name) is called a *manifest type*—the other types are usually called type constructors. In the following treatment, we restrict ourselves to type variables that range over manifest types. Jones (1995) has shown that without this restriction we get a system that elegantly handles higher-order polymorphism. While the following presentation can be generalized to quantification over type constructors, the restriction keeps the presentation slightly simpler.

2.4 Notation

This text uses $\kappa, \kappa', \kappa_1, \kappa_2, \ldots$ as meta-level variables for kinds; $\rho, \rho', \rho_1, \rho_2, \ldots$ as well as $\sigma, \sigma', \sigma_1, \sigma_2, \ldots$ for simple types, and $\tau, \tau', \tau_1, \tau_2, \ldots$ for type schemes. A type substitution $[\sigma_1/\alpha_1, \ldots, \sigma_n/\alpha_n]$ (or, more concisely, $[\overline{\sigma}/\overline{\alpha}]$), when applied to some type ρ, replaces any occurrence of α_i with σ_i. ▮

Lambda terms are explicitly typed. This implies that every variable introduced by an abstraction is annotated with a simple type. In addition to the usual abstraction and application, a non-recursive **let** construct is used to introduce variables that are typed by a type scheme—Harper & Mitchell (1993) comment on the importance of this distinction. Instead of traditional, explicit type abstractions and type applications, we use the information contained in the type annotations together with annotations on constants and variables to denote the instantiation of a type scheme.

2.5 Definition (Lambda Terms)

Let $\mathcal{V}_\mathrm{L}$ be a set of variables and $\mathcal{C}_\mathrm{L}$ a set of constants, where each constant a is associated with a closed type scheme τ, written a^τ. Furthermore, let the set of type-constructor constants $\mathcal{C}_\mathrm{T}$ contain a symbol $\overset{\kappa_1 \kappa_2}{\to}$ for every pair of kinds $\kappa_1, \kappa_2 \in \mathcal{K}$. The set of *pre-lambda terms* is defined by the following production rule:

$$\begin{array}{rcll}
\hat{\mathcal{L}} & \to & \mathcal{C}_\mathrm{L}^{\overline{\mathcal{S}}} & \text{(constant)} \\
& \mid & \mathcal{V}_\mathrm{L}^{\overline{\mathcal{S}}} & \text{(variable)} \\
& \mid & \lambda \mathcal{V}_\mathrm{L}^{\mathcal{S}}.\hat{\mathcal{L}} & \text{(abstraction)} \\
& \mid & \hat{\mathcal{L}}_1\, \hat{\mathcal{L}}_2 & \text{(application)} \\
& \mid & \mathbf{let}\ \mathcal{V}_\mathrm{L}^{\mathcal{T}} = \hat{\mathcal{L}}_1\ \mathbf{in}\ \hat{\mathcal{L}}_2 & \text{(local binding)}
\end{array}$$

We denote instances of $\mathcal{C}_{\mathrm{L}}^{\overline{\mathcal{S}}}$ as $a^{\tau,\overline{\sigma}}$ when $a^{\tau} \in \mathcal{C}_{\mathrm{L}}$ and the elements of the tuple $\overline{\sigma}$ are from $\mathcal{S}$; similarly, we denote instances of $\mathcal{V}_{\mathrm{L}}^{\overline{\mathcal{S}}}$ as $v^{\overline{\sigma}}$ when $v \in \mathcal{V}_{\mathrm{L}}$ and the elements of the tuple $\overline{\sigma}$ are from $\mathcal{S}$.

When a (pre) lambda term M is of type τ, we write $M :: \tau$. A *type assignment* is a set Γ of associations $v_1 :: \tau_1, \ldots, v_n :: \tau_n$, where all v_i are distinct and the τ_i are closed types. We define that a pre-lambda term $M \in \hat{\mathcal{L}}$ has type τ under the type assignment Γ when $\Gamma \vdash M :: \tau$ holds under the following typing rules:

$$\frac{a^{\tau} \in \mathcal{C}_{\mathrm{L}} \qquad \tau = \forall \overline{\alpha_I^{\kappa_I}}.\sigma \qquad \text{for all } i \in I, \rho_i :: \kappa_i}{\Gamma \vdash a^{\tau,\overline{\rho_I}} :: [\overline{\rho_I}/\overline{\alpha_I}]\sigma}\ \text{(con)}$$

$$\frac{(v :: \forall \overline{\alpha_I^{\kappa_I}}.\sigma) \in \Gamma \qquad \text{for all } i \in I, \rho_i :: \kappa_i}{\Gamma \vdash v^{\overline{\rho_I}} :: [\overline{\rho_I}/\overline{\alpha_I}]\sigma}\ \text{(var)}$$

$$\frac{\{v :: \forall.\sigma\} \triangleright \Gamma \vdash M :: \sigma' \qquad \sigma :: \kappa \text{ and } \sigma' :: \kappa'}{\Gamma \vdash \lambda v^{\sigma}.M :: \sigma \overset{\kappa\kappa'}{\to} \sigma'}\ \text{(abs)}$$

$$\frac{\Gamma \vdash M :: \sigma \overset{\kappa\kappa'}{\to} \sigma' \qquad \Gamma \vdash N :: \sigma \qquad \sigma :: \kappa \text{ and } \sigma' :: \kappa'}{\Gamma \vdash M\ N :: \sigma'}\ \text{(app)}$$

$$\frac{\Gamma \vdash M :: \sigma' \qquad \{v :: \forall \overline{\alpha^{\kappa}}.\sigma'\} \triangleright \Gamma \vdash N :: \sigma \qquad \overline{\alpha} \cap \mathrm{Fr}\,\Gamma = \emptyset}{\Gamma \vdash \mathbf{let}\ v^{\forall \overline{\alpha^{\kappa}}.\sigma'} = M\ \mathbf{in}\ N :: \sigma}\ \text{(let)}$$

In the (let) rule, $\mathrm{Fr}\,\Gamma$ denotes the set of free variables in Γ; the condition $\overline{\alpha} \cap \mathrm{Fr}\,\Gamma = \emptyset$ requires that the generalized type variables are not constrained by the type assignment. Note that a type assignment implicitly specifies a kind assignment, because all the types it contains are closed, and that the rules (con) and (abs) together require that the tuple of type instantiations $\overline{\sigma}$ at a variable introduced by a lambda abstraction be empty.

Assuming some type assignment, we define a *lambda term* as a pre-lambda term that has a type according to the above rules. Lambda terms are contained in the set $\mathcal{L}$. In particular, the implicit type assignment can be omitted in the case of closed lambda terms. ∎

Assume that there is only a single kind name in $\mathcal{C}_{\mathrm{K}}$ and no constant in $\mathcal{C}_{\mathrm{L}}$. Then, due to the restriction of type abstraction to the **let** construct, and type application to the use of variables and constants, the terms in $\mathcal{L}$ correspond exactly to the type-normalized Core-XML expressions of Henglein & Jørgensen (1994).

In an abstraction $\lambda v^{\sigma}.M$, we call v the *bound* variable, in contrast to *free* variables which occur in a position that is not enclosed by an abstraction naming that variable. Formally, we define as follows, for a lambda term M, the free variables $\mathrm{Fr}\,M$ and the bound variables $\mathrm{Bd}\,M$.

$$\begin{aligned}
\mathrm{Fr}\,a^{\tau,\overline{\sigma}} &= \{\} & \mathrm{Bd}\,a^{\tau,\overline{\sigma}} &= \{\}\\
\mathrm{Fr}\,v^{\overline{\sigma}} &= \{v\} & \mathrm{Bd}\,v^{\overline{\sigma}} &= \{\}\\
\mathrm{Fr}(\lambda v^{\sigma}.M) &= \mathrm{Fr}\,M \setminus \{v\} & \mathrm{Bd}(\lambda v^{\sigma}.M) &= \mathrm{Bd}\,M \cup \{v\}\\
\mathrm{Fr}(M\ N) &= \mathrm{Fr}\,M \cup \mathrm{Fr}\,N & \mathrm{Bd}(M\ N) &= \mathrm{Bd}\,M \cup \mathrm{Bd}\,N\\
\mathrm{Fr}(\mathbf{let}\ v^{\tau} = M\ \mathbf{in}\ N) &= \mathrm{Fr}\,M \cup (\mathrm{Fr}\,N \setminus \{v\}) & \mathrm{Bd}(\mathbf{let}\ v^{\tau} = M\ \mathbf{in}\ N) &= \mathrm{Bd}\,M \cup \mathrm{Bd}\,N \cup \{v\}
\end{aligned}$$

When we speak of a *fresh variable* for a term M, we mean a variable name that is contained in neither $\mathrm{Fr}\,M$ nor $\mathrm{Bd}\,M$.

2.6 NOTATION
In the following, we usually omit the type annotations at lambda terms (including the type abstractions and instantiations). However, the unannotated terms are meant to be representatives of their explicitly typed counterparts, and it is well known that there exists an algorithm that reconstructs the type information mechanically (cf. Harper & Mitchell 1993).

Furthermore, we take the liberty of writing binary operators in an infix notation. Generally, application binds strongest, and lambda abstractions extend their scope as far as possible to the right—for example, $f\ x + \lambda v.v \circ w$ is read as $(f\ x) + (\lambda v.(v \circ w))$. ▮

In the following chapters, it will frequently be useful to combine an ordered set of terms into one object; in these cases, we use tuples.

2.7 DEFINITION (TUPLES)
We denote a *tuple* of terms with arity n as $\langle M_1, \ldots, M_n \rangle$, and abbreviate it to $\overline{M_I}$ with the index tuple $I = \langle 1, \ldots, n \rangle$, or simply $\overline{M}$ where convenient. Given a tuple $\overline{M}$, we denote its arity by $|\overline{M}|$. ▮

Such tuples include the special case of a unit $\langle\rangle$ and a singleton tuple $\langle M \rangle$, where the latter is isomorphic to M.

2.8 NOTATION
We use tuples as data terms in lambda terms. Formally, we regard them as elements of a product type $\sigma_1 \times \sigma_2$, where we use the isomorphism, which is specified by $(\sigma_1 \times \sigma_2) \times \sigma_3 \cong \sigma_1 \times (\sigma_2 \times \sigma_3)$, to justify writing $\langle \sigma_1, \ldots, \sigma_n \rangle$, or more concisely, $\langle \overline{\sigma} \rangle$. Furthermore, we use currying to regard $\lambda v_1. \cdots \lambda v_n.M$ as isomorphic to $\lambda \langle v_1, \ldots, v_n \rangle.M$, or shorter, $\lambda \overline{v}.M$. We extend this notation for variable bindings to (**let** $\cdot$ **in** $\cdot$) expressions, allowing tuples to occur on the left-hand side of a binding. (Typing guarantees that the arity of the left- and right-hand side always matches.) Gunter (1992, Chapter 3) provides a formal justification for this treatment of product types. ▮

2.3.3 Contexts

We use the notion of a context to denote positions within a lambda term.

2.9 DEFINITION (CONTEXT)
A *context (term)* is a lambda term that contains a hole, which is denoted by $\langle\cdot\rangle$. Context terms obey the following syntactic structure:

$$
\begin{array}{lcll}
\mathcal{O} & \to & \langle\cdot\rangle & \text{(hole)} \\
 & | & \lambda \mathcal{V}^{\mathcal{S}}_{\mathrm{L}}.\mathcal{O} & \text{(abstraction body)} \\
 & | & \mathcal{O}\,\hat{\mathcal{L}} & \text{(function)} \\
 & | & \hat{\mathcal{L}}\,\mathcal{O} & \text{(argument)} \\
 & | & \mathbf{let}\ \mathcal{V}^{\mathcal{T}}_{\mathrm{L}} = \mathcal{O}\ \mathbf{in}\ \hat{\mathcal{L}} & \text{(bound)} \\
 & | & \mathbf{let}\ \mathcal{V}^{\mathcal{T}}_{\mathrm{L}} = \hat{\mathcal{L}}\ \mathbf{in}\ \mathcal{O} & \text{(let body)}
\end{array}
$$

The *instantiation* of a context $O \in \mathcal{O}$ with a lambda term M, denoted by $O\langle M\rangle$, replaces the hole in O with the term M (which may include variable capture).

The free and bound variables of a context $O \in \mathcal{O}$ are given by the intersection of the sets $\mathrm{Fr}(O\langle M\rangle)$ and $\mathrm{Bd}(O\langle M\rangle)$, respectively, for all $M \in \mathcal{L}$. ∎

2.3.4 Normal Lambda Terms

A *renaming* of a lambda term M is a term N, such that M and N are distinguished only by the names of their bound variables; in this case, we write $M \overset{\alpha}{\equiv} N$.

A *substitution* is a function from names to lambda terms, denoted by $[M_1/v_1, \ldots, M_n/v_n]$ or, more concisely, $[\overline{M_I}/\overline{v_I}]$, which maps the name v_i to the term M_i. Substitutions are extended to terms in the usual way, renaming bound variables of the term when this is necessary to avoid variable capture.

2.10 DEFINITION (BETA REDUCTION)
The *beta reduction relation* $\overset{\beta}{\rightarrow}$ over lambda terms is defined as the smallest relation, such that for any lambda term $M \in \mathcal{L}$, when there is a context $O \in \mathcal{O}$ with $M = O\langle(\lambda v.N_1)\ N_2\rangle$, we have

$$M = O\langle(\lambda v.N_1)\ N_2\rangle \overset{\beta}{\rightarrow} O\langle[N_2/v]N_1\rangle$$

and when there is a context $O \in \mathcal{O}$ with $M = O\langle\mathbf{let}\ v^{\tau} = N_2\ \mathbf{in}\ N_1\rangle$, we have

$$M = O\langle\mathbf{let}\ v^{\tau} = N_2\ \mathbf{in}\ N_1\rangle \overset{\beta}{\rightarrow} O\langle[N_2/v]N_1\rangle.$$

A lambda term $M \in \mathcal{L}$ is in *beta normal form* when there is no $N \in \mathcal{L}$, such that $M \overset{\beta}{\rightarrow} N$. ∎

An important and well known property of the polymorphic lambda calculus is the fact that every term has a normal form.

2.11 THEOREM (STRONG NORMALIZATION)
Every term $M \in \mathcal{L}$ has a beta normal form $N \in \mathcal{L}$. ∎

A more detailed treatment of these notions can be found, for example, in the textbook by Gunter (1992).

2.12 NOTATION
We assume that a lambda term is in beta normal form, without explicitly saying so. In particular, we assume that, after applying a substitution to some term, the resulting term is implicitly reduced to normal form. ∎

2.3.5 Equality and Refinement

Reasoning about programs, we state equalities and reduction relations. We perform derivations in the calculational proof format of Dijkstra & Scholten (1990).

$$\begin{array}{ll} & M \\ = & \quad \{\langle\text{first hint}\rangle\} \\ & N \end{array}$$

$$
\begin{array}{ll}
= & \{\langle \text{second hint} \rangle\} \\
 & R
\end{array}
$$

The above is a short form for stating that $M = N$ due to $\langle$first hint$\rangle$ and $N = R$ due to $\langle$second hint$\rangle$, and so, overall, $M = R$. If a hint contains an assertion that is prefixed by $\bullet$, this means that the calculation is only valid if the assertion is satisfied. This proof format can be used for any transitive relation.

Assume the following equalities:

$$x + 0 = x \tag{add.1}$$

$$x + s(y) = s(x) + y \tag{add.2}$$

$$x * 0 = 0 \tag{mult.1}$$

$$x * s(y) = x + (x * y) \tag{mult.2}$$

and that $s^n(0)$ is an abbreviation for

$$\underbrace{s(\cdots s}_{n \text{ times}}(0)\cdots).$$

Now we calculate as follows by using fold-unfold transformations (Burstall & Darlington 1977).

$$
\begin{array}{ll}
 & x * s(s(0)) = s^6(0) \\
\equiv & \quad \{\text{mult.2}\} \\
 & x + (x * s(0)) = s^6(0) \\
\equiv & \quad \{\text{mult.2}\} \\
 & x + (x + (x * 0)) = s^6(0) \\
\equiv & \quad \{\text{mult.1}\} \\
 & x + (x + 0) = s^6(0) \\
\equiv & \quad \{\text{add.1}\} \\
 & x + x = s^6(0) \\
\equiv & \quad \{\text{add.1}\} \\
 & x + x = s^6(0) + 0 \\
\equiv & \quad \{\text{three times add.2}\} \\
 & x + x = s^3(0) + s^3(0) \\
\equiv & \quad \{\bullet x = s^3(0)\} \\
 & s^3(0) + s^3(0) = s^3(0) + s^3(0) \\
\equiv & \quad \{\text{equality is reflexive}\} \\
 & \mathbf{true}
\end{array}
$$

This proves that $x * s(s(0)) = s^6(0)$ if $x = s^3(0)$.

2.3.6 Inference Rules

Defining inference systems, we use inference rules of the form

$$\frac{P_1 \quad \cdots \quad P_n}{C}\ (\langle \text{rule name} \rangle)$$

Such a rule means that the conjunction of the *premises* P_i implies the *conclusion* C. The special case

$$\frac{}{C}$$

without premises is called an axiom. Proof trees are constructed from inference rules in the usual way.

Chapter 3

DECLARATIVE PARALLELISM

THIS chapter introduces a formalism that enables the declarative specification of parallelism. The formalism is declarative because its semantics is defined using mathematical logic and the principle of "reduction as deduction". The formalism models parallelism because it has a parallel operational semantics, which we will derive from its logic semantics.

Central to the formalism is the idea that combinators that are meant to specify parallel behaviour co-ordinate functional computations, which are regarded as step-at-a-time sequential computations. We realize this idea by augmenting the polymorphic lambda calculus introduced in the last chapter with a recursive let construct (to obtain the full power of functional programming) and with logical connectives, such as conjunction and existential quantification (to specify parallel behaviour). This extension carefully preserves important properties, such as referential transparency, for purely functional subcomputations. The result is a higher-order functional logic language that is amenable to formal reasoning and that allows to distinguish between sequential and parallel computations; the latter include communication by equational constraints over infinite trees and committed-choice indeterminism. In other words, the formalism that is defined in the present chapter relates to GOFFIN in the same way as the polymorphic lambda calculus with recursion and data constructors relates to purely functional programming—see the Haskell kernel of Hudak et al. (1992).

Overall, the present chapter contains the theoretical foundation of the dissertation. The chapter starts with a presentation of the basic ideas of the language $\mathcal{D}$ and continues with a short introduction to the use of linear logic (Girard 1987) for modeling the semantics of concurrent systems. Afterwards, the syntactic structure of $\mathcal{D}$ is introduced in detail. The logic semantics of $\mathcal{D}$ is defined by a mapping into linear logic, and it is subsequently used to derive a parallel operational semantics. Finally, we discuss possible extensions and related work.

3.1 Basic Ideas

We know that logical connectives, such as conjunction, disjunction, and existential quantification, are useful as primitives for the co-ordination of concurrent systems—they are modeling parallel composition, alternatives, and hiding, respectively (Shapiro 1989, Hoare 1990, Saraswat 1993, Smolka 1995, Chakravarty, Guo, Köhler & Lock 1997). For example, we understand the conjunctive composition of two formulae, where each consists of the application of two predicates p and q to a variable x,

$$p\,x \wedge q\,x,$$

to be the parallel composition of two agents $p\ x$ and $q\ x$ that are concurrently operating on x. If one agent, say $q\ x$, restricts x to be of value a, we regard this as communicating a via a medium x, which can be observed by q. Furthermore, we interpret logical truth (denoted by *true*) as successful termination. This intuitive correlation is supported by the algebraic observation that the quotient of the set of logical formulae by equality constitutes wrt. conjunction and truth forms a symmetric monoid—just as the quotient of the set of agents by observational equivalence forms a symmetric monoid wrt. parallel composition and successful termination. Similar observations are possible for other logical connectives.

Unfortunately, the connectives of classical logic alone are too coarse to model all aspects of concurrent systems. This problem is overcome by Saraswat (1993) with his family of cc languages by introducing operators that are operationally motivated, but have no clear logical meaning; similarly proceed Franzén, Haridi & Janson (1992) with AKL and Smolka (1994) with the Oz calculus. In contrast to these approaches, Girard's (1987) linear logic—which previous work has shown to be suited for modeling concurrent systems (see Alexiev 1994, Guo 1994, Miller 1995)—lives in the tradition of mathematical logic. To see the deficiencies of classical logic for modeling certain forms of concurrent computation, consider committed-choice indeterminism. For example, the cc agent

$$(c \longrightarrow f)\ []\ (d \longrightarrow g)$$

may select the first alternative and reduce to f when the constraint store entails c. The computation is not able to undo this choice, even when d is entailed also and f fails. We cannot faithfully model such a behaviour using classical logic.

More generally, the concept of rewriting is difficult to model in classical logic because the rewritten term has to be consumed—but classical logic is not resource conscious. Further arguments in this direction are, for example, given by Guo (1994, Chapter 7). Before following sections define the language $\mathcal{D}$ formally, the reminder of this section introduces $\mathcal{D}$ by way of example.

3.1.1 Parallel Behaviour

In $\mathcal{D}$, the symbol $\mathbf{1}$ denotes *successful termination* and $\mathbf{0}$ denotes *failure*. Given two agents A_1 and A_2, we may compose them parallelly or sequentially, using the following notation:

$$\underset{\textit{(parallel composition)}}{A_1 \parallel A_2} \qquad \text{and} \qquad \underset{\textit{(sequential composition)}}{A_1 \Rightarrow A_2}$$

Although from a logic perspective, both combinators realize a form of conjunction, their operational intention is different. A parallel composition $A_1 \parallel A_2$ allows both agents A_1 and A_2 to proceed at the same time, but a sequential composition $A_1 \Rightarrow A_2$ forces the agent A_2 to wait until A_1 succeeds. We may read $A_1 \parallel A_2$ as "A_1 and A_2" and $A_1 \Rightarrow A_2$ as "A_1 and *then* A_2".

Communication, i.e., exchange of information between agents, is realized by *constraints*—just as proposed by Saraswat (1993). For the time being, a constraint is simply a predicate that relates some unbound variables, thus restricting the possible valuations of these variables—for example, a constraint $x \doteq 1$ asserts that x equals 1.[1] To *tell* a constraint, we compose it with an agent; for example, the composite agent

$$x \doteq 1 \parallel A \qquad \text{, which is read as } (x \doteq 1) \parallel A,$$

tells the constraint $x \doteq 1$, which can subsequently be observed by A. Conceptually, told constraints are accumulated in the *constraint store*, where they can be observed by agents. We restrict the scope of agents that may observe a constraint by the introduction of local variables. For example, in

$$A \parallel (\nu x.(x \doteq 1) \parallel B),$$

the fact that x equals 1 will only be known to B, but not to A. In the case of sequential composition, e.g., in

$$(x \doteq y \parallel A) \Rightarrow B,$$

B can observe $x \doteq y$, together with any constraints imposed by A, only after A succeeds.

Finally, we have *ask* agents of the form

$$\overline{v} : \overline{G} \rightarrow A,$$

where the $\overline{G}$ are equational constraints that act as guards. The agent A can only proceed when there is a valuation for the variables $\overline{v}$, such that the $\overline{G}$ are entailed by the constraints in the constraint store. Note the difference between

$$(x \doteq y) \Rightarrow A \qquad \text{and} \qquad \langle\rangle : \langle x \doteq y\rangle \rightarrow A.$$

The former tells $x \doteq y$ and immediately proceeds with A, while the latter waits until $x \doteq y$ was told by some other agent before A is executed. (Here, the angle brackets denote tuples.) Composite ask agents can specify multiple alternatives by combining ask agents with the operator $[]$. It expresses indeterminate choice—that is, any of the alternatives can be taken, but once a choice is made, it cannot be undone.

Ask agents can perform pattern matching; the agent

$$\langle v\rangle : \langle a\ v \doteq a\ 1\rangle \rightarrow A \qquad \text{(the notation } a\ x \text{ means, data constructor } a \text{ applied to } x)$$

reduces to $[1/v]A$. Multiple agents of such a form combined by $[]$ can be used in a similar way as the $(\mathsf{case} \cdot \mathsf{of} \cdot)$ construct that will be introduced below; but in contrast to the latter, the alternative ask agents may overlap, and thus introduce committed-choice indeterminism.

3.1.2 Co-ordination versus Computation

While many researchers recognize the usefulness of logical connectives as combinators for the specification of parallelism, the declarative programming community has only recently found interest in a clear separation of the concerns of co-ordination and computation[2] in the spirit of Carriero & Gelernter (1992). In particular, we (Chakravarty, Guo, Köhler &

---

[1] We use the symbol $\doteq$ for equality in constraints, avoiding confusion with the symbol $=$, which is used for the equality in our meta-language.

[2] See the work by Breitinger, Loogen & Ortega-Mallén (1995, 1996) and Chakravarty et al. (1994, 1995).

Lock 1997) argue for the co-ordination of purely functional computations by combinators that are resembling logical connectives—as stated in Section 2.2: *functions compute, relations co-ordinate*. Such a declarative framework is well suited for expressing parallel programs on a high-level of abstraction. $\mathcal{D}$ is based on this research, and as mentioned in the previous chapter, the separation between co-ordinative activities, on one hand, and computation, on the other hand, is achieved by strong typing, using an extension of the well-established Hindley/Milner type system (Hindley 1969, Damas & Milner 1982).

In $\mathcal{D}$ we use the type o to denote agents (i.e., relational computations). Thus, we type parallel composition as $(\|) :: \mathsf{o} \to \mathsf{o} \to \mathsf{o}$, and in general, expressions of type $\sigma \to \mathsf{o}$ are abstractions of co-ordinative behaviour, not unlike skeletons (Cole 1989, Darlington, Field, Harrison, Kelly, Sharp, Wu & While 1993).

The separation of the concerns of co-ordination and computation is a fundamental difference of $\mathcal{D}$ compared to other multi-paradigm approaches to parallel programming, such as HLcc (Saraswat & Lincoln 1992), AKL (Janson 1994), and Oz (Smolka 1994), which realize functions by relations—i.e., they diminish the distinction between co-ordination and computation.

It should be understood that a *naive* integration of features, such as committed-choice indeterminism, introduction of unbound variables, and instantiation of the these variables by constraints, into functional programs ruins the clean semantics of functional programming. But the type discipline employed in GOFFIN and $\mathcal{D}$ saves us from such calamity. It cleanly separates co-ordinating agents from purely computational expressions, and so allows a conservative extension of the latter.

3.1.3 The Common Core: Abstractions, Bindings, and Choice

The use of a type system, instead of different (sub)languages, for the separation between co-ordination and computation, allows sharing common language constructs, such as abstraction and case distinction, between the computational and co-ordinative part. $\mathcal{D}$ uniformly uses lambda abstractions to construct both co-ordinative and computational abstractions, which are distinguished only by their type.

Provided that *Cons* and *Nil* represent list construction and the empty list, respectively,

$$\lambda x.\mathit{Cons}\ x\ \mathit{Nil} \tag{3.1}$$

applied to 42, yields a singleton list containing 42; while

$$\lambda x.1 \tag{3.2}$$

applied to some argument reduces to an agent terminating successfully. The abstraction in (3.1) has type $\forall\alpha.\alpha \to \mathit{List}\ \alpha$, and that in (3.2) has type $\forall\alpha.\alpha \to \mathsf{o}$.

Overall, in $\mathcal{D}$, lambda abstractions provides the well known richness of higher-order functional programming, but also support the construction of abstractions (templates or skeletons if you like) encapsulating some fixed parallel behaviour.

The construct $\mathsf{let}\ \overline{B}\ \mathsf{in}\ E$ introduces recursion and local bindings. It specifies that the possibly mutually recursive bindings $\overline{B}$ can be used in E, where a *binding* has the form $x = E$ and associates the name x with the expression E. For example,

$$\begin{array}{l}\mathsf{let}\ \mathit{unit} = \lambda x.\mathit{Cons}\ x\ \mathit{Nil}\\ \mathsf{in}\ E\end{array}$$

associates in E the name *unit* with the abstraction previously displayed in (3.1). The value bound to a name is accessed by dereferencing the name, using the notation $(\cdot)\!\downarrow$. So

$$\begin{array}{l}\mathsf{let}\ unit = \lambda x.Cons\ x\ Nil\\ \mathsf{in}\ unit\!\downarrow 42\end{array}$$

dereferences *unit* and applies the resulting lambda abstraction to 42—this leaves us with *Cons* 42 *Nil*. We can pass a name as an argument without dereferencing it; consider that

$$\begin{array}{l}\mathsf{let}\ unit = \lambda x.Cons\ x\ Nil\\ \mathsf{in}\ unit\!\downarrow unit\end{array}$$

gives $\mathsf{let}\ unit = \lambda x.Cons\ x\ Nil\ \ \mathsf{in}\ \ Cons\ unit\ Nil$.

We can also bind co-ordinative forms, as in

$$\begin{array}{l}\mathsf{let}\ eqOne = \lambda x.(x \doteq 1)\\ \mathsf{in}\ (\nu y.(eqOne\!\downarrow y) \parallel E),\end{array}$$

which tells $y \doteq 1$ to E. In this way, we can pass co-ordinative forms to lambda abstraction as arguments:

$$\begin{array}{l}\mathsf{let}\ eqOne = \lambda x.(x \doteq 1)\\ \mathsf{in}\ (\nu y.(\lambda z.\lambda c.c\!\downarrow z)\ y\ eqOne)\end{array}$$

Overall, $\mathcal{D}$ allows higher-order programming on both the co-ordination and the computation level.

Similarly, the *case construct* of the general form $\mathsf{case}\ E\ \mathsf{of}\ \overline{X}$, where the $\overline{X}$ are *case alternatives* of the form $d\ \overline{v} \rightarrow F$, is available on both the co-ordination and computation level. It supports the deterministic selection of one of several alternatives, in dependence on the value of the scrutinized expression E. For example,

$$\begin{array}{lll}\mathsf{case}\ E\ \mathsf{of}\ \langle & Cons\ \langle h,t\rangle & \rightarrow E_1,\\ & Nil\ \langle\rangle & \rightarrow E_2\rangle\end{array}$$

reduces to E_1 if E is a list constructor (simultaneously binding h and t to the head and tail of the list); otherwise if E is *Nil*, E_2 is taken.

3.1.4 Constraints and Deferred Evaluation

We call an agent *suspended* when it is waiting for some information, which should be communicated by some other agent. As already mentioned, communication is realized using constraints in $\mathcal{D}$, and hence, suspension occurs when the already told constraints do not entail the constraint that would be needed for an agent to continue. The need for the availability of a constraint may arise due to dereferencing a name or due to an ask expression. The first case is illustrated by the next example:

$$\begin{array}{l}\mathsf{let}\ y = E\\ \mathsf{in}\ (\nu x.A\langle x\!\downarrow\rangle \parallel B\langle x \doteq y\rangle)\end{array}$$

The agent represented by the context A contains $x\!\downarrow$ and B contains $x \doteq y$. When A depends on the value of $x\!\downarrow$, it cannot proceed until B tells the alias constraint $x \doteq y$—because in the presence of the constraint $x \doteq y$, the binding for y can be used for x.

An ask expression allows to select alternatives depending on some constraints, e.g., in

$$(\overline{v}_1 : \overline{G}_1 \rightarrow A_1) \;[]\; (\overline{v}_2 : \overline{G}_2 \rightarrow A_2)$$

either A_1 or A_2 is selected depending on whether the $\overline{G}_1$ or $\overline{G}_2$ is entailed (if both are entailed, the choice is arbitrary). If the constraint store is characterized by the compound constraint Ψ, such that there exists a valuation Θ for the variables $\overline{v}_1$ with $\Psi \Vdash \Theta\overline{G}_1$ (which means that Ψ entails $\Theta\overline{G}_1$ in the constraint system), then the overall choice expression may reduce to ΘA_1. But until such entailment holds, the choice suspends.

We call this mechanism of suspending reduction until sufficient information is told, *deferred evaluation*. It is sometimes also called residuation (Aït-Kaci, Lincoln & Nasr 1987).

Although a more general treatment is certainly possible, we restrict constraints to equational constraints over possibly infinite trees. This constraint system allows a simple constraint consistency and entailment check, which is important for an efficient distributed implementation; and it is well known that many applications can be realized conveniently with this constraint system (Shapiro 1989, Jagadeesan & Pingali 1992, Saraswat 1993, Janson 1994).

3.1.5 Names, Terms, and Strictness

A peculiarity of $\mathcal{D}$ is the distinction between a variable x and the value $x{\downarrow}$ associated with x—it is so central that it is fixed in the type system, which is explained in detail later. The introduction of this distinction brings a number of immediate benefits.

- It becomes explicit whether an agent depends on the value bound to a variable and so may have to be suspended until a value is associated with this variable. This simplifies the treatment of constraints in the presence of reduction and also the presentation of the semantics.
- Higher-order unification can be avoided. Two lambda abstractions, even if they denote the same function, are not regarded as equal unless they are bound to the same name.
- Support for both eager and lazy evaluation can be integrated smoothly.

Furthermore, this distinction anticipates the separation of boxed and unboxed types, which is important for the implementation and is detailed in the next chapter.

The types used for terms (or values) are different from those that used for variables. If a term has *term type* τ, then a variable associated with that term (by a binding) has the *name type* $\lfloor\tau\rfloor$ (pronounced τ *name*). In a binding $x = E$, the right-hand side E must always have a term type and the left-hand side x must always have a name type. Thus if E is of term type τ, then $\mathsf{let}\ x = E\ \mathsf{in}\ x$ is of name type $\lfloor\tau\rfloor$. Furthermore, dereferencing gets an expression of a name type as an argument and results in a term type—if x is of type $\lfloor\tau\rfloor$, then $x{\downarrow}$ is of type τ.

A plain abstraction $\lambda x.E$ is of term type $\sigma \rightarrow \sigma'$; but we restrict the occurrence of functions as arguments to name types. Hence in $\mathcal{D}$, if a function is passed as an argument, we always pass a variable name that is associated with the function. This permits us to view named functions as data (or inactive functions), while function terms represent computation (or active functions). For example, in

$$\lambda f.f{\downarrow}\,42$$

a function name f is passed as an argument, which is dereferenced (and so, activated), and then, applied to its argument 42.

It has been noted that both eager and lazy evaluation should be supported in functional programming (Wadler 1996); the addition of strictness annotations into the standard for Haskell 1.3 (Peterson et al. 1996) is another indication for the importance to support both forms of evaluation. Reduction in $\mathcal{D}$ eagerly evaluates all arguments before they are substituted into the body of an abstraction, but when the argument is a variable name, its associated term expression is not necessarily evaluated; hence,

$$\mathsf{let}\ x = E\ \mathsf{in}\ f{\downarrow}\ x \qquad \text{and also} \qquad f{\downarrow}\ (\mathsf{let}\ x = E\ \mathsf{in}\ x)$$

pass E lazily to $f{\downarrow}$. Overall, the distinction of names versus terms allows to control the strictness of a computation (see also Peyton Jones & Launchbury 1991). This idea was also exploited by Launchbury (1993) for his semantics of lazy evaluation.

3.2 Linear Logic and the Semantics of Concurrent Systems

Linear logic (Girard 1987) refines classical logic by dropping the structural rules of contraction and weakening and by providing instead modalities for the explicit management of resources. Researchers have repeatedly shown that linear logic is well suited for the formalization of the semantics of concurrent languages—see the surveys of Alexiev (1994) and Miller (1995). In particular, Guo (1994, Chapter 7) introduced a linear logic framework to define and reason about the semantics of concurrent systems. Moreover, Miller (1996) introduced the specification logic Forum, which restructures the proof theory of linear logic in such a way that it constitutes an abstract logic programming language (Miller, Nadathur, Pfenning & Scedrov 1991). Based on this previous work, Chakravarty, Guo & Köhler (1997) identify a fragment of linear logic as a specification language for concurrent systems. The latter is summarized in this section, as it is subsequently used to define the semantics of $\mathcal{D}$.

The rest of the present chapter requires a basic familiarity with linear logic. Hence, Appendix A contains a short introduction to both plain linear logic and its reformulation Forum—in particular, it contains the inference system of linear logic in Figure A.1 and the system $\mathcal{F}$ of Forum in Figure A.2. Girard (1995) provides more details on linear logic in his summary of the syntax and semantics of linear logic and Miller (1996) defines Forum.

3.2.1 Linear Logic Formulae as Processes

Plain linear logic comes with a sequent-style proof system, where sequents are of the form

$$\underbrace{A_1, \ldots, A_n}_{\text{antecedent}} \longrightarrow \underbrace{B_1, \ldots, B_m}_{\text{succedent}}$$

Due to the duality of positive and negative formulae in linear logic, the formulae A_i in the antecedent may be replaced by their dual in the succedent and vice versa. This symmetry allows many syntactically different, but semantically equivalent proofs. Furthermore, any of the A_i and B_j may be permanent resources—they merely have to be prefixed with the modalities ! or ?.

In contrast, sequents in Forum have more structure: Their general form is either

$$\Sigma : \Psi; \Delta \longrightarrow \Gamma; \Upsilon \qquad \text{or} \qquad \Sigma : \Psi; \Delta \xrightarrow{A} \Gamma; \Upsilon.$$

The set Σ is a *signature* associating the names of the nonlogical constants with types. The formulae in the antecedent and the succedent are split into a *linear part*, Δ and Γ, and a *classical context*, Ψ and Υ. In the second form, the formula A is associated with the linear part of the antecedent; its prominent position in the notation emphasizes that it is currently proven using left introduction rules. In other words, the above two Forum sequents correspond to

$$!\Psi, \Delta \longrightarrow \Gamma, ?\Upsilon \qquad \text{and} \qquad !\Psi, \Delta, A \longrightarrow \Gamma, ?\Upsilon$$

in plain linear logic. The additional structure of Forum sequents enforces structure on the proofs, so that some of the redundancy in the set of possible proofs is avoided. There is even more structure because Ψ and Υ are regarded as sets, while Δ and Γ (being linear) are regarded as a multiset and a list, respectively. The inference system of Forum exploits the list character of Γ by the restriction that a right introduction rule may only be applied to the leftmost formula in Γ that is not an atom. Overall, the additional structure in Forum sequents restricts, together with the carefully selected set of inference rules (see Figure A.2), the structure of Forum proofs to *uniform proofs* (Miller et al. 1991). Uniform proofs are cut-free proofs where the outermost logical connectives determine the applied inference rules. This enables interpreting the inference rules as the transition rules of an abstract interpreter, which regards “formulae as programs”. Consequently, uniform proofs are particularly suited for an operational interpretation following the idea of “proof as computation”. Miller (1996) shows that the uniform proofs of Forum are complete with respect to full (higher-order) linear logic.

The rest of this section identifies a subset of Forum formulae whose operational interpretation can be regarded as the operational semantics of a specification language for concurrent systems. This subset is related to process calculi, such as Milner’s (1991) π-calculus, and it is even closer to Sangiorgi’s (1993) higher-order variant HOπ. But in addition to this operational interpretation, the linear logic foundation allows to define nonlogical operators by using an appropriate set of axioms. Furthermore, meta-level arguments about formulae, which are regarded as processes, may use the rich theory of linear logic.

As in Forum, we define the terms of linear logic as typed lambda terms. More specifically, we instantiate the polymorphic lambda terms of Definition 2.5 with a set of constants that includes the logical connectives of linear logic, or more specifically, of Forum.[3]

3.1 DEFINITION (LINEAR LOGIC PROCESSES)
Process terms are polymorphic lambda terms where $\mathcal{C}_{\mathrm{K}} = \{\star\}$ and where the special type constant o is contained in $\mathcal{C}_{\mathrm{T}}$. Furthermore, we have $\top^{o}$, $\bot^{o}$, $\&^{o \to o \to o}$, $\parr^{o \to o \to o}$, $\multimap^{o \to o \to o}$, $\Rightarrow^{o \to o \to o}$, $\Pi^{\forall \alpha^{\star}.(\alpha \to o) \to o}$, and $?^{o \to o}$ as constants in $\mathcal{C}_{\mathrm{L}}$. We call these constants *logical connectives.*

Linear logic processes are process terms obeying the syntactic rules displayed in Figure 3.1, where we define $\mathbf{V} = \mathcal{V}_{\mathrm{L}}$ and $\mathbf{C} = \mathcal{C}_{\mathrm{L}} \setminus \{\top, \bot, \&, \parr, \multimap, \Rightarrow, \Pi, ?\}$. As usual, $\forall v.A \triangleq \Pi \lambda v.A$, $A \multimapinv B \triangleq B \multimap A$, $A^{\bot} \triangleq A \multimap \bot$, $A \oplus B \triangleq (A^{\bot} \mathbin{\&} B^{\bot})^{\bot}$, and $1 \triangleq \top^{\bot}$. ∎

It may attract attention that the above definition does not provide methods prefixed by the exponential ! (of-course). In the present higher-order setting, the expressive power of the exponential is recovered by the quantification over formulae (Lafont 1996).

---

[3] This kind of definition of higher-order logics by using lambda terms is due to Church (1940).

$$
\begin{array}{llcll}
\text{Process} & \mathbf{P} & \rightarrow & \top & \text{(abort)} \\
 & & \mid & \bot & \text{(nil)} \\
 & & \mid & 1 & \text{(error)} \\
 & & \mid & ?(\mathbf{T} \doteq \mathbf{T}) & \text{(equation)} \\
 & & \mid & \mathbf{P}_1 \parr \mathbf{P}_2 & \text{(parallel composition)} \\
 & & \mid & \mathbf{P}_1 \oplus \mathbf{P}_2 & \text{(indeterministic choice)} \\
 & & \mid & \mathbf{M}^{\bot} & \text{(method)} \\
 & & \mid & \forall \mathbf{V}.\mathbf{P} & \text{(local channel)} \\
 & & \mid & \mathbf{T} & \\
 & & & & \\
\text{Method} & \mathbf{M} & \rightarrow & \forall \overline{\mathbf{V}}.\mathbf{V}\,\overline{\mathbf{V}} \multimapinv \mathbf{P} & \text{(receive)} \\
 & & \mid & \forall \overline{\mathbf{V}}.(\mathbf{T} \doteq \mathbf{T}) \multimapinv \mathbf{P} & \text{(ask)} \\
 & & & & \\
\text{Term} & \mathbf{T} & \rightarrow & \mathbf{V} & \text{(variable)} \\
 & & \mid & \mathbf{C} & \text{(constant)} \\
 & & \mid & \lambda \mathbf{V}.\mathbf{P} & \text{(abstraction)} \\
 & & \mid & \mathbf{T}\,\mathbf{P} & \text{(application)}
\end{array}
$$

Figure 3.1 The grammar of linear logic processes, where we have $\mathbf{V} = \mathcal{V}_{\mathrm{L}}$ and $\mathbf{C} = \mathcal{C}_{\mathrm{L}} \setminus \{\top, \bot, \&, \parr, \multimap, \Rightarrow, \Pi, ?\}$.

The operational interpretation for linear logic processes is a consequence of linear implication ($\multimap$). Hence, given two processes $P_1, P_2 \in \mathbf{P}$, we regard a proof for $P_1 \multimapinv P_2$ as a computation that *reduces* P_1 to P_2—*reduction-as-deduction* as Guo (1994) puts it.[4]

The following illustrates the behaviour of the various process constructs. Due to strong normalization (Theorem 2.11), we can assume that all terms are normalized.

3.2.2 Interpretation of the Process Constructs

The constant $\top$ specifies an aborting process, regardless of its context it terminates the proof and thus the computation. This is obvious from its right introduction rule,

$$\frac{}{\Sigma : \Psi; \Delta \longrightarrow \mathcal{A}, \top, \Gamma; \Upsilon}\,(\top\mathrm{R})$$

The variable $\mathcal{A}$ represents a list of arbitrary atoms; so, $\top$ is the first nonatomic formula.

In contrast, the constant $\bot$ disappears without a trace, i.e., it denotes successful termination. This behaviour is embodied by the rule

$$\frac{\Sigma : \Psi; \Delta \longrightarrow \Gamma; \Upsilon}{\Sigma : \Psi; \Delta \longrightarrow \mathcal{A}, \bot, \Gamma; \Upsilon}\,(\bot\mathrm{R})$$

The process 1, which is an abbreviation for $\top \multimap \bot$, stalls the whole computation—i.e., a sequent containing 1 in its succedent has no proof, because there is no left introduction rule

---

[4] As usual in logic programming, execution of a logic program is associated with backwards implication.

for $\top$. An attempt to proof 1 leaves us with a partial proof tree such as the following:

$$\cfrac{\cfrac{\cfrac{\Sigma : \Psi; \Delta \xrightarrow{\top} \Gamma; \Upsilon}{\Sigma : \Psi; \top, \Delta \longrightarrow \mathcal{A}, \Gamma; \Upsilon}\;(\text{decide})}{\Sigma : \Psi; \top, \Delta \longrightarrow \mathcal{A}, \bot, \Gamma; \Upsilon}\;(\bot\text{R})}{\Sigma : \Psi; \Delta \longrightarrow \mathcal{A}, \top \multimap \bot, \Gamma; \Upsilon}\;(\multimap\text{R})$$

Equations that are prefixed by the modality ? are put in the classical context Υ by the rule

$$\frac{\Sigma : \Psi; \Delta \longrightarrow \mathcal{A}, \Gamma; t_1 \doteq t_2, \Upsilon}{\Sigma : \Psi; \Delta \longrightarrow \mathcal{A}, ?(t_1 \doteq t_2), \Gamma; \Upsilon}\;(?\text{R})$$

Classical equations are interpreted by the following theory **Eq**.

3.2 DEFINITION (EQUALITY THEORY **Eq**)
The theory **Eq** contains the following axioms:

$$\forall x. \bot \multimapinv ?(x \doteq x) \qquad \text{(reflexivity)}$$
$$\forall x, y. ?(x \doteq y) \multimapinv ?(y \doteq x) \qquad \text{(symmetry)}$$
$$\forall x, y, z. (?(x \doteq y) \parr ?(y \doteq z)) \multimapinv ?(x \doteq z) \qquad \text{(transitivity)}$$
$$\forall x, y, t. ?(x \doteq y) \parr (t\, x) \multimapinv (t\, y) \qquad \text{(substitutivity)}$$

(These are general Forum formulae not linear logic processes.) ▮

All proofs of reduction of linear logic processes are accompanied by the theory **Eq** in the classical context Ψ, in the antecedent of the sequent. Hence, when a processes states an equation $?(a \doteq b)$, **Eq** allows to regard a and b as equal. In the specification of the semantics of $\mathcal{D}$, we shall employ **Eq** to realize equational constraints.

Two parallel processes P_1 and P_2 reduce in parallel, competing for resources. Although it seems that the inference rule

$$\frac{\Sigma : \Psi; \Delta \longrightarrow \mathcal{A}, P_1, P_2, \Gamma; \Upsilon}{\Sigma : \Psi; \Delta \longrightarrow \mathcal{A}, P_1 \parr P_2, \Gamma; \Upsilon}\;(\parr\text{R})$$

prefers P_1 (remember that Γ is a list), Miller (1996, Corollary 4) proves that this is not the case. In contrast to a parallel composition $P_1 \parr P_2$, indeterministic choice $P_1 \oplus P_2$ commits reduction to either P_1 or P_2. The following proof tree displays the selection of P_1 (remember that we have by definition $P_1 \oplus P_2 = ((P_1 \multimap \bot) \parr (P_2 \multimap \bot)) \multimap \bot$).

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\Sigma : \Psi; \Delta \longrightarrow \mathcal{A}, P_1; \Upsilon \qquad \cfrac{}{\Sigma : \Psi; \cdot \xrightarrow{\bot} \cdot; \Upsilon}\;(\bot\text{L})}{\Sigma : \Psi; \Delta \xrightarrow{P_1 \multimap \bot} \mathcal{A}; \Upsilon}\;(\multimap\text{L})}{\Sigma : \Psi; \Delta \xrightarrow{(P_1 \multimap \bot) \parr (P_2 \multimap \bot)} \mathcal{A}; \Upsilon}\;(\&\text{L}_1)}{\Sigma : \Psi; (P_1 \multimap \bot) \parr (P_2 \multimap \bot), \Delta \longrightarrow \mathcal{A}; \Upsilon}\;(\text{decide})}{\Sigma : \Psi; (P_1 \multimap \bot) \parr (P_2 \multimap \bot), \Delta \longrightarrow \mathcal{A}, \bot; \Upsilon}\;(\bot\text{R})}{\Sigma : \Psi; \Delta \longrightarrow \mathcal{A}, ((P_1 \multimap \bot) \parr (P_2 \multimap \bot)) \multimap \bot; \Upsilon}\;(\multimap\text{R})$$

Similarly, P_2 may be selected using ($\&$L$_2$).

Local channel names are introduced by processes of the form $\forall v.P$. The inference rule

$$\dfrac{x :: \tau, \Sigma : \Psi; \Delta \longrightarrow \mathcal{A}, [x/v]P, \Gamma; \Upsilon}{\Sigma : \Psi; \Delta \longrightarrow \mathcal{A}, \forall v.P, \Gamma; \Upsilon}\;(\forall\text{R}) \qquad x \notin \Sigma$$

introduces an *eigen-variable* x, which we regard as a channel name. The proviso that x may not be declared in the signature Σ ensures that x does not occur in P. The application of a channel name to some term, as in $x\,t$, is interpreted as sending the term t as a message on the channel x. Such messages can be consumed by a receive method $(\forall w.x\,w \mathbin{\circ\!\!-} Q)^{\perp}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{x :: \tau, \Sigma : \Psi; \Delta \longrightarrow \mathcal{A}, [t/w]Q; \Upsilon \qquad \dfrac{}{x :: \tau, \Sigma : \Psi; \cdot \xrightarrow{x\,t} x\,t; \Upsilon}\;(\text{initial})}{x :: \tau, \Sigma : \Psi; \Delta \xrightarrow{[t/w](x\,w \mathbin{\circ\!\!-} Q)} \mathcal{A}, x\,t; \Upsilon}\;(\multimap\text{L})}{x :: \tau, \Sigma : \Psi; \Delta \xrightarrow{\forall w.x\,w \mathbin{\circ\!\!-} Q} \mathcal{A}, x\,t; \Upsilon}\;(\forall\text{L})}{x :: \tau, \Sigma : \Psi; (\forall w.x\,w \mathbin{\circ\!\!-} Q), \Delta \longrightarrow \mathcal{A}, x\,t; \Upsilon}\;(\text{decide})}{x :: \tau, \Sigma : \Psi; \Delta \longrightarrow \mathcal{A}, x\,t \parr (\forall w.x\,w \mathbin{\circ\!\!-} Q)^{\perp}; \Upsilon}\;(\parr\text{R},\multimap\text{R},\perp\text{R})}{\Sigma : \Psi; \Delta \longrightarrow \mathcal{A}, \forall v.(v\,t \parr (\forall w.v\,w \mathbin{\circ\!\!-} Q)^{\perp}); \Upsilon}\;(\forall\text{R})$$

The operational interpretation is that the value denoted by t is received via the local channel v by the process Q contained in the method. In a similar way, ask methods are triggered by equations of the correct form.

3.3 Syntax of $\mathcal{D}$

After the preliminaries, we now proceed to the first step in the formal definition of declarative parallelism, namely the introduction of the syntax of the language $\mathcal{D}$. Formally, $\mathcal{D}$ expressions are polymorphic lambda terms as introduced in Section 2.3.2, albeit they contain a number of constants carrying a special meaning. However, to improve readability, we enrich the plain grammar of lambda terms with the syntactic forms that were already introduced informally in the beginning of this chapter. We start this section by discussing the contents of the set $\mathcal{C}_{\text{K}}$ of kind names, the set $\mathcal{C}_{\text{T}}$ of type constants, and the set $\mathcal{C}_{\text{L}}$ of value constants. Afterwards, the enriched syntax is defined.

3.3.1 Constants

$\mathcal{D}$ contains two kinds of types, namely types of terms and types of names.

3.3 DEFINITION ($\mathcal{D}$ KINDS)
The set of kind names is for $\mathcal{D}$ defined to be $\mathcal{C}_{\text{K}} = \{\star, \sharp\}$. ▮

As previously discussed, the kind $\star$ represents *types of terms*, whereas $\sharp$ contains *types of names*. The idea here being that a name that is associated with a term of type $\tau :: \star$ has type $\lfloor \tau \rfloor$, where the type constructor $\lfloor \cdot \rfloor$ has the kind $\star \rightarrow \sharp$.

Constant	Type	Description
$(\cdot)\downarrow$	$\forall\alpha^\star.\lfloor\alpha\rfloor \to \alpha$	dereferencing (the term of a name)
$\lfloor\cdot\rfloor$	$\forall\alpha^\star.\alpha \to \lfloor\alpha\rfloor$	naming (give a name to a term)
fix	$\forall\overline{\alpha^\star}.(\lfloor\overline{\alpha}\rfloor \to \overline{\alpha}) \to \overline{\alpha}$	fixpoint combinator
$\mathsf{0}$	o	failure
$\mathsf{1}$	o	success
$\top$	o	error
$\overset{\sharp}{\doteq}, \overset{\star}{\doteq}$	$\forall\alpha^\star.\lfloor\alpha\rfloor \to \lfloor\alpha\rfloor \to \mathsf{o}, \forall\alpha^\star.\alpha \to \alpha \to \mathsf{o}$	equational constraint
$\parallel$	$\mathsf{o} \to \mathsf{o} \to \mathsf{o}$	parallel composition
$\Rightarrow$	$\mathsf{o} \to \lfloor\mathsf{o}\rfloor \to \mathsf{o}$	sequential composition
$[\,]$	$\mathsf{o} \to \mathsf{o} \to \mathsf{o}$	committed-choice alternatives
Σ	$\forall\alpha^\star.(\lfloor\alpha\rfloor \to \mathsf{o}) \to \mathsf{o}$	local name
ask	$\forall\overline{\alpha^\star}.(\overline{\alpha} \to \mathsf{o}) \to (\overline{\alpha} \to \mathsf{o}) \to \mathsf{o}$	check constraint entailment

Table 3.1 The special constants of $\mathcal{D}$. The kind annotations at the equational constraints are omitted when no confusion arises.

3.4 DEFINITION ($\mathcal{D}$ TYPES)

The set $\mathcal{C}_{\mathrm{T}}$ of type constants is for $\mathcal{D}$ composed out of special type constants and data type constructors. The special type constants are the function space constructor $\to$, which is always required for typed lambda terms, and additionally $\lfloor\cdot\rfloor$ and o. The kind of $\lfloor\cdot\rfloor$ is $\star \to \sharp$ and the kind of o is $\star$.

The data type constructors may vary; we assume that they are contained in the set T. All elements of T must be of kind $\star$ or $\kappa \to \star$ for some kind κ. ∎

As already said, types of the form $\lfloor\tau\rfloor$ represent names for terms of type τ—in the special syntax for $\mathcal{D}$, names are introduced by let bindings. We lift $\lfloor\cdot\rfloor$ to tuples: Writing $\lfloor\overline{\tau}\rfloor$ with $|\overline{\tau}| = n$, we mean the type of n-tuples with component types $\lfloor\tau_i\rfloor$. Moreover, the type o is the types of co-ordinating agents.

3.5 DEFINITION ($\mathcal{D}$ EXPRESSIONS)

$\mathcal{D}$ expressions are lambda terms over the set $\mathcal{C}_{\mathrm{L}}$ of constants, which includes the constants listed in Table 3.1 together with a varying number of data constructors and destructors that are contained in the set D. For each constructor $a \in \mathsf{D}$ of type

$$\forall\overline{\alpha}.\overline{\sigma} \to c\,\overline{\alpha} \quad \text{, where } a \in \mathsf{T},$$

there is a destructor $\mathsf{destruct}_a$ of type

$$\forall\overline{\alpha},\beta.c\,\overline{\alpha} \to (\overline{\sigma} \to \beta) \to \beta \to \beta$$

∎

Naming $\lfloor\cdot\rfloor$ and dereferencing $(\cdot)\!\downarrow$ are dual.[5] They introduce names for terms and dereference a term from a name, respectively. As with the type constructor for types of names, we generalize the constant $\lfloor\cdot\rfloor$ to tuples: writing $\lfloor\overline{E}\rfloor$, with $|\overline{E}| = n$, we mean the n-tuple of names $\lfloor E_i\rfloor$.

The fixpoint combinator fix is used to introduce recursion in the usual way. It is characterized by the equation

$$\mathsf{fix}\ f = f\ \lfloor \mathsf{fix}\ f\rfloor.$$

Note that the constants fix and ask are defined over arbitrary tuples, so to be exact, they represent a family of functions. But that does not have any consequences for the following exposition; thus, we do not explicitly distinguish the members of the families. The use of the other constants is explained in more detail below.

3.3.2 Additional Syntax

We keep the syntax of polymorphic lambda terms for kinds and types, but for $\mathcal{D}$ expressions, we use a more convenient syntactic form. The introduction of the new syntax is accompanied by the description of the corresponding polymorphic lambda terms in their original syntax—we call the latter *plain lambda terms*. We typeset keywords and nonterminals of $\mathcal{D}$'s syntax in a sans serif typeface.

3.6 NOTATION (GENERAL $\mathcal{D}$ EXPRESSIONS)
The following rules define the syntactic structure of *general expressions*:

Expression	E	$\rightarrow$	V	(variable)
		$\mid$	$\mathsf{D}\ \overline{\mathsf{E}}$	(data constructor application)
		$\mid$	$\lambda\mathsf{V}.\mathsf{E}$	(abstraction)
		$\mid$	$\mathsf{E}_1\ \mathsf{E}_2$	(application)
		$\mid$	$\mathsf{E}\!\downarrow$	(dereferencing)
		$\mid$	$\mathsf{let}\ \overline{\mathsf{B}}\ \mathsf{in}\ \mathsf{E}$	(mutually recursive bindings)
		$\mid$	$\mathsf{case}\ \mathsf{E}\ \mathsf{of}\ \overline{\mathsf{X}}$	(case selection)
		$\mid$	A	(agent)
Binding	B	$\rightarrow$	$\mathsf{V} = \mathsf{E}$	
Case	X	$\rightarrow$	$\mathsf{D}\ \overline{\mathsf{V}} \rightarrow \mathsf{E}$	

We assume that function application binds tightest and that the body of abstractions and of ($\mathsf{let}\ \cdot\ \mathsf{in}\ \cdot$) extends as far to the right as possible.

The first four cases in the above grammar are directly covered by plain lambda terms as in Definition 2.5. Although, we require that a data constructor from the set D occurs only together with an argument tuple of the correct arity. If the type of a variable v in a lambda abstraction $\lambda v.E$ is of kind $\star$ (a type of terms), the type may neither be o nor a function type $\sigma \rightarrow \sigma'$.[6] The same restriction holds for the elements of the argument tuple of data

---

[5] We overload $\lfloor\cdot\rfloor$ to represent either a type constant or a value constant. The intended meaning is always obvious from the context.

[6] This means that we may pass an agent or function only by passing a name referencing such a term, but not the term itself.

constructors. Furthermore, we require that in $\mathcal{D}$ the body of a lambda abstraction is always a term—i.e., it must have a type of kind $\star$.

The fifth case in the grammar, the form $E{\downarrow}$, is just the postfix application of the constant $(\cdot){\downarrow}$ to its single argument, which is required to be of a type of names (kind $\sharp$). If E evaluates to a name x, the function of $E{\downarrow}$ is to obtain the term that is associated with x by virtue of a binding construct of the form $\mathsf{let}\ \overline{B}\ \mathsf{in}\ F$, where x is bound in $\overline{B}$. A $(\mathsf{let}\ \cdot\ \mathsf{in}\ \cdot)$ binding construct corresponds to a plain lambda term in the following way:

$$\begin{aligned}&\mathsf{let}\ v_1 = E_1, \ldots, v_n = E_n\ \mathsf{in}\ E\\ &\qquad = \mathbf{let}\ \langle v_1, \ldots, v_n\rangle = \mathsf{fix}\,(\lambda\langle v_1, \ldots, v_n\rangle.\langle E_1, \ldots, E_n\rangle)\ \mathbf{in}\ E\end{aligned}$$

where $n \geq 0$, and we require that the v_i are distinct. Note that the plain lambda calculus binding $\mathbf{let}\ v = M\ \mathbf{in}\ N$ is not directly used in $\mathcal{D}$, but only indirectly via the more general $\mathsf{let}\ \overline{B}\ \mathsf{in}\ E$. Both constructs are easily distinguished by the typeface. Moreover, the $(\mathsf{let}\cdot\mathsf{in}\ \cdot)$ construct is the only construct in $\mathcal{D}$ that is defined via $(\mathbf{let}\ \cdot\ \mathbf{in}\ \cdot)$, and thus, it is the only construct introducing polymorphism in $\mathcal{D}$. We define the notation $\mathrm{dom}\,\overline{B}$ to specify the variables bound by the bindings $\overline{B}$.

Case expressions of the form $\mathsf{case}\ E\ \mathsf{of}\ \overline{X}$ deterministically select one of several cases $\overline{X}$, in dependence on the value of the expression E; the type of the latter must have a data type constructor from the set T as its outermost constructor. Case expressions correspond to plain lambda terms as expressed in the next equation, which uses the $\mathsf{destruct}_a$ constants contained in D.

$$\begin{aligned}&\mathsf{case}\ E\ \mathsf{of}\ a_1\ \overline{v}_1 \to E_1, \ldots, a_n\ \overline{v}_n \to E_n\\ &\qquad = \mathsf{destruct}_{a_1}\ E\ (\lambda\overline{v}_1.E_1)\ (\ldots\mathsf{destruct}_{a_n}\ E\ (\lambda\overline{v}_n.E_n)\ \top)\ldots)\end{aligned}$$

We require that $n \geq 1$ and that each $\overline{v}_i$ contains distinct variables. ∎

General expressions are used for both computation and co-ordination—the actual use depends on the type of the expression. In contrast, agent expressions, which are defined next, are only used for co-ordination. (They are always of type o, because of the constants that they contain.)

3.7 NOTATION ($\mathcal{D}$ AGENTS)

The following grammar defines the structure of co-ordinating *agents*.

Agent	A	$\to$	$\mathsf{0}$	(failure)
		$\mid$	$\mathsf{1}$	(success)
		$\mid$	$\mathsf{E}_1 \doteq \mathsf{E}_2$	(equational constraint)
		$\mid$	$\mathsf{E}_1 \parallel \mathsf{E}_2$	(parallel composition)
		$\mid$	$\mathsf{E}_1 \Rightarrow \mathsf{E}_2$	(sequential composition)
		$\mid$	$\nu\mathsf{V}.\mathsf{E}$	(unbound variable)
		$\mid$	S	(committed choice)

The operator $\doteq$ binds stronger than $\parallel$ and $\Rightarrow$, and the body of $\nu x.E$ extends as far to the right as possible.

The mapping of agents, except $\nu v.E$, into plain lambda terms is immediate. The introduction of unbound variables[7] corresponds to the use of the constant Σ in the following way:

$$\nu v.E = \Sigma(\lambda v.E) \qquad \blacksquare$$

The beginning of this chapter already provided an informal description of the meaning of the various constructs. Remember that we call the collection of all equational constraints that have been told by agents the current *constraint store*. In our setting of equational constraints, the constraint store can be simplified to accumulate only name aliases of the form $x \doteq y$, which intuitively allows to regard x and y as different names for the same thing.

There is a subtle point in the typing of equational constraints that should not go unnoticed. The constant $\doteq$ comes in two flavours (according to Table 3.1):

- $E_1 \overset{\sharp}{\doteq} E_2$ equates two expressions of a type of names and
- $E_1 \overset{\star}{\doteq} E_2$ equates expressions of a type of terms.

Unbound variables are always of a type of terms (see the type of the constant Σ); hence, equational constraints can not directly associate an unbound variable with a term. Instead, an equational constraint may associate the name of an unbound variable with the name of a binding, which was introduced by a $(\mathsf{let} \cdot \mathsf{in} \cdot)$ construct. So, if an unbound variable x should be associated with an empty list (which is represented by the application of the nullary data constructor *Nil* to an empty tuple, i.e., by $\mathit{Nil}\ \langle\rangle$), the following $\mathcal{D}$ expression is appropriate:

$$\nu x.\ \mathsf{let}\ l = \mathit{Nil}\ \langle\rangle\ \mathsf{in}\ x \doteq l.$$

Equational constraints over terms are used to check the consistency of these terms, i.e.,

$$\mathit{Cons}\ \langle h, l\rangle \doteq \mathit{Cons}\ \langle h', l'\rangle$$

matches the data constructors *Cons* against each other and asserts the equality between h and h' as well as l and l', i.e., $h \doteq h' \parallel l \doteq l'$.

The last case of the agent grammar, the nonterminal S, allows committed-choice selection, which we define next. In a committed-choice (or just choice) expression, each alternative is prefixed by a guard tuple, which consists of equational constraints and controls the selection. If more than one alternative could be selected, the choice is arbitrary, but fixed.

3.8 NOTATION ($\mathcal{D}$ CHOICE EXPRESSIONS)

The grammar below defines the structure of *choice expressions*.

Choice	S	$\rightarrow$	$\top$	(no alternative)
		$\mid$	$\overline{\mathsf{V}} : \overline{\mathsf{P}} \doteq \overline{\mathsf{E}} \rightarrow \mathsf{E}$	(ask expression)
		$\mid$	$\mathsf{S}_1 \mathbin{[\,]} \mathsf{S}_2$	(excluding alternatives)
Pattern	P	$\rightarrow$	V	(variable pattern)
		$\mid$	$\mathsf{D}\,\overline{\mathsf{V}}$	(constructor pattern)

---

[7] *Unbound variable* means a variable without a $(\mathsf{let} \cdot \mathsf{in} \cdot)$ binding, not a free variable.

We require that variables introduced in an ask expression occur within the guards only in the patterns (i.e., the left-hand side) of the equational constraints.

The correspondence of the first and last syntactic variant to plain lambda terms is obvious. Ask expressions match lambda terms as determined by the following equation:

$$\overline{v} : \overline{P} \doteq \overline{E} \to E' = \mathsf{ask}\ (\lambda\overline{v}.\,||\,\overline{P} \doteq \overline{E})\ (\lambda\overline{v}.E')$$

where we require that all variables in $\overline{v}$ are distinct and understand $\overline{P} \doteq \overline{E}$ a a tuple of equations $\langle P_1 \doteq E_1, \ldots, P_n \doteq E_n\rangle$. Such a tuple prefixed by $||$ represents the parallel composition of the equational constraints. (We require that the tuple contains at least one element.) ∎

A choice of the form $\top$ symbolizes an exhausted choice expression—one where no applicable choice exists. It is the neutral of $[]$, which combines choices in such a way that only one alternative may be taken. Ask expressions are intuitively interpreted in the following way. If there is a unique tuple of closed expressions $\overline{F}$, such that $[\overline{F}/\overline{v}]\overline{G}$ is entailed by the current contents of the constraint store, then the ask expression reduces to $[\overline{F}/\overline{v}]E$—it is, of course, assumed that the types of the $\overline{v}$ and $\overline{F}$ match.

Subsequently, we always assume that $\mathcal{D}$ expressions are constructed according to the definition of the nonterminal **E**, instead of being lambda terms freely build over the constants of Table 3.1. Furthermore, we assume that all expressions are well-typed, which means that the corresponding plain lambda terms are well-typed.[8] In this sprit, when we are speaking of the type of a $\mathcal{D}$ expression, we mean the type of the corresponding plain lambda term.

Finally, it is worthwhile to note the dualism in $\mathcal{D}$ between names and terms as well as between bindings and dereferencing. The kinds guarantee that we never get names of names, i.e., types of the form $\lfloor\lfloor\sigma\rfloor\rfloor$. Hence, dereferencing a name always yields a term.

3.4 A Linear Logic Semantics

Now that the syntax of $\mathcal{D}$ is formally defined, we proceed to the semantics. It is given by an encoding $\mathcal{E}[\![\cdot]\!]$ that maps $\mathcal{D}$ expressions to linear logic formulae. By virtue of the logic semantics of the formulae, this gives a logic semantics to $\mathcal{D}$.

More precisely, we first define a set of *process combinators*, which are lambda terms over linear logic processes. The encoding of $\mathcal{D}$ expressions is, then, defined by using the process combinators. This indirection in the definition of the semantics is purely syntactic, because lambda terms are part of linear logic processes as introduced in Definition 3.1; the indirection achieves some degree of modularization of the encoding, which allows a clearer exposition and simplifies reasoning about the encoding (as we will see in the next section). Simple unfolding and reduction of the process combinator definitions gives a direct encoding from $\mathcal{D}$ expressions to linear logic processes. (Strong normalization of typed lambda terms, Theorem 2.11, guarantees the termination of the unfolding and reduction process.)

An expression is expected to denote a value, which is the result of the evaluation of the expression. Hence, the translation of expressions into linear logic processes has to provide a canonical way of relating an expression and its result value. Guo (1994, Chapter 7) proposes the use of a result channel, which is, in his work, passed as an argument to the encoding function. In his work, $[\![E]\!]_r$ means that the encoding of the expression E produces its result via the channel r—i.e., it eventually sends a message $r\,t$, where t is the result of the expression

---

[8] This makes sense, even so $\mathcal{D}$ expression do not contain type annotations—see Notation 2.6.

E. Using higher-order linear logic, we can spare the additional argument to the encoding function, and instead, map each expression to a lambda term in the encoding; this lambda term expects to be applied to the channel name, which is used to return the result. In other words, we have the relation

$$\mathcal{E}[\![E]\!]\,r = [\![E]\!]_r$$

between the style of encoding that is used here and that used by Guo.

A channel maps a message to a formula; so, for a message of type σ, the channel is of type $\sigma \rightarrow \mathrm{o}$. Consequently, lambda terms produced by the mapping $\mathcal{E}[\![\cdot]\!]$ are of type $(\sigma \rightarrow \mathrm{o}) \rightarrow \mathrm{o}$. As types of this form occur repeatedly in the following, we define the synonym $\mathit{proc}\,\alpha$ for $(\alpha \rightarrow \mathrm{o}) \rightarrow \mathrm{o}$. In other words, we use $\mathit{proc}\,\sigma$ for linear logic processes that produce a result of type σ on a channel that they get as an argument. Before we discuss the actual encoding of $\mathcal{D}$ expression, we first define the process combinators over processes $\mathit{proc}\,\sigma$.

3.4.1 Process Combinators

Process combinators and the encoding of lambda terms is inspired by the monadic call-by-value encoding of the lambda calculus as proposed by Wadler (1992). Hence, it may helpful to view the following combinator definitions as an exercise in functional programming using a monadic style, where $\mathit{proc}\,\sigma$ is a monad that represents a process computation with result σ.

We start with the definition of the unit and bind operations of the monad, which are denoted by the symbols $\uparrow$ and $\succcurlyeq$, respectively.

$$\begin{array}{ll}
\mathit{proc}\,\alpha & \triangleq (\alpha \rightarrow \mathrm{o}) \rightarrow \mathrm{o} \\[1ex]
(\uparrow) & :: \forall\alpha.\alpha \rightarrow \mathit{proc}\,\alpha \\
\uparrow M & \triangleq \lambda r.r\,M \\[1ex]
(\succcurlyeq) & :: \forall\alpha,\beta.\,\mathit{proc}\,\alpha \rightarrow (\alpha \rightarrow \mathit{proc}\,\beta) \rightarrow \mathit{proc}\,\beta \\
M \succcurlyeq N & \triangleq \lambda r.\forall s.M\,s \parr (\forall v.s\,v \multimapinv N\,v\,r)^{\perp}
\end{array}$$

(In these and the following definitions, we always assume that all newly introduced variable names are fresh.) The intuition is that $\uparrow v$ is a process that merely returns v. The second operation $\succcurlyeq$ allows to combine two processes into a compound process where the result of the first one is consumed by the second. Thus, $M \succcurlyeq \lambda w.M'$ executes M and binds the result of M to w before executing M'. On the level of linear logic processes as produced by the above definitions, passing a result from M to $\lambda w.M'$ involves the creation of a local communication channel s, which is passed to M by application. Furthermore, $\lambda w.M'$ is wrapped into a receive method $(\forall v.s\,v \multimapinv (\lambda w.M')\,v\,r)^{\perp}$, which waits for the communication of the result of M on the channel s. This result is bound to v and passed to $\lambda w.M'$ as an argument.

To ensure that the triple $\langle \mathit{proc}, \uparrow, \succcurlyeq \rangle$ qualifies as a monad in the categorial sense, we would have to prove that certain laws hold between the combinators.[9] But, we do not explore this formal monadic interpretation further—although, it would certainly be an interesting investigation.

---

[9] To be precise, $\langle \mathit{proc}, \uparrow, \succcurlyeq \rangle$ is (assuming that the laws hold) a Kleisli triple, but it is well known that there is a one-to-one correspondence between Kleisli triples and monads. The latter just use a different, but isomorphic set of combinators.

But let us return to the process combinators. At times, we merely want to ensure that the execution of two processes is sequentialized, without any actual exchange of information between them. (We will realize $\mathcal{D}$'s sequential composition $\Rightarrow$ in this way.) Therefore, we introduce the combinator $\succ$, which builds on $\succsim$.

$$\begin{aligned}(\succ) \quad & :: \forall\alpha,\beta.\, \mathit{proc}\,\alpha \rightarrow \mathit{proc}\,\beta \rightarrow \mathit{proc}\,\beta \\ M \succ N \quad & \triangleq M \succsim \lambda v.N\end{aligned}$$

The combinator $\upuparrows$ composes two processes in parallel, such that the result of the overall computation is the pair of the results of the two sub-processes. Thus, in contrast to $\succsim$ and $\succ$, $\upuparrows$ is commutative (modulo the ordering of the results). A second commutative operator is committed-choice $\Lsh\!\Rsh$, which reduces to either of its two arguments.

$$\begin{aligned}(\upuparrows) \quad & :: \forall\alpha,\beta.\, \mathit{proc}\,\alpha \rightarrow \mathit{proc}\,\beta \rightarrow \mathit{proc}\,\langle\alpha,\beta\rangle \\ M \upuparrows N \quad & \triangleq \lambda r.\forall s_1, s_2.(M\,s_1 \parr N\,s_2) \parr (\forall v_1, v_2.(s_1\,v_1 \parr s_2\,v_2) \multimapinv r\,\langle v_1, v_2\rangle)^{\perp}\end{aligned}$$

$$\begin{aligned}(\Lsh\!\Rsh) \quad & :: \forall\alpha.\, \mathit{proc}\,\alpha \rightarrow \mathit{proc}\,\alpha \rightarrow \mathit{proc}\,\alpha \\ M \Lsh\!\Rsh N \quad & \triangleq \lambda r.M\,r \oplus N\,r\end{aligned}$$

Hence, the combinators $\upuparrows$ and $\Lsh\!\Rsh$ lift the parallel composition $\parr$ and the indeterministic choice $\oplus$ from plain linear logic processes **P** into the process monad *proc*. Similarly, failure and error are lifted by *fail* and *err*.

$$\begin{aligned}\mathit{fail} \quad & :: \forall\alpha.\, \mathit{proc}\,\alpha \\ \mathit{fail} \quad & \triangleq \lambda r.\top\end{aligned}$$

$$\begin{aligned}\mathit{err} \quad & :: \forall\alpha.\, \mathit{proc}\,\alpha \\ \mathit{err} \quad & \triangleq \lambda r.1\end{aligned}$$

There is no combinator that lifts $\perp$, because quiet termination has to fulfill the synchronization requirement that is introduced in *proc*. Assume the definition of a combinator $\lambda r.\perp$; placement of the combinator to the left of sequential composition, $(\lambda r.\perp) \succ M$, would block M forever. On the other hand, $\uparrow$ yielding an arbitrary constant, as in $\uparrow a \succ M$, immediately reduces to M. For this reason, the encoding in the next section uses $\uparrow\langle\rangle$ ($\langle\rangle$ is the nullary tuple) to encode the successful termination, **1**, of $\mathcal{D}$. (Inspecting the meaning of $\top$ and 1 as described in Section 3.2.2, it becomes clear, why such a provision of synchronization is not necessary for theses constants—they terminate the computation immediately.)

For the same reason, we have to be careful when a plain linear logic process of type o (without any immediate result) should be used as a combinator. The combinator $\upharpoonright$ is used in this case. The form $P \upharpoonright M$ executes P with M, such that it returns the result of M.

$$\begin{aligned}(\upharpoonright) \quad & :: \forall\alpha.o \rightarrow \mathit{proc}\,\alpha \rightarrow \mathit{proc}\,\alpha \\ P \upharpoonright M \quad & \triangleq \lambda r.P \parr M\,r\end{aligned}$$

For example, $(?(x \doteq y)) \upharpoonright M$ establishes the equation $x \doteq y$ and executes M.

We lift linear logic ask methods into *proc* by using *ask*.

$$\begin{aligned}\mathit{ask} \quad & :: \forall\overline{\alpha},\beta.(\overline{\alpha} \rightarrow o) \rightarrow (\overline{\alpha} \rightarrow \mathit{proc}\,\beta) \rightarrow \mathit{proc}\,\beta \\ \mathit{ask}\,G\,M \quad & \triangleq \lambda r.(\forall\overline{v}.G\,\overline{v} \multimapinv M\,\overline{v}\,r)^{\perp}\end{aligned}$$

Its arguments are a guard function G and a process M that should be executed when the method is triggered.

Finally, we have a set of combinators that allow to introduce names of type *name* and to store and retrieve values using these names as abstract references. Storage is realized by messages on a special channel *ref*, where $\mathit{ref}\,x\,v$ means that v is stored under the name x—we call such a message a *reference*.

$$\begin{array}{ll}
\mathit{new} & ::\ \mathit{proc}\ \mathit{name} \\
\mathit{new} & \triangleq \lambda r.\forall x.r\,x \\
 & \\
\mathit{put} & ::\ \forall\alpha.\mathit{name} \to \alpha \to \mathit{proc}\,\langle\rangle \\
\mathit{put}\,x\,M & \triangleq \lambda r.\mathit{ref}\,x\,M \parr r\,\langle\rangle \\
 & \\
\mathit{get} & ::\ \forall\alpha.\mathit{name} \to \mathit{proc}\,\alpha \\
\mathit{get}\,x & \triangleq (\lambda r.\forall v.\mathit{ref}\,x\,v \multimapinv r\,v)^{\perp}
\end{array}$$

The combinator *new* creates a new name using an eigen-variable produced by $\forall$ in linear logic. Furthermore, *put* and *get* store and retrieve values, respectively. An application of *get* suspends if no value is available for the given name. Below, we use this mechanism of references to realize unbound variables, the names introduced by ($\mathsf{let}\ \cdot\ \mathsf{in}\ \cdot$) constructs, and indirectly, deferred evaluation.

3.4.2 Encoding General Expressions

We define the encoding $\mathcal{E}[\![\cdot]\!] :: \mathsf{E} \to \mathbf{P}$ of $\mathcal{D}$ expressions by linear logic processes based on the process combinators that we just discussed. The encoding of variables, data constructors, abstraction, and application is exactly Wadler's (1992, Section 8) call-by-value encoding of the lambda calculus.

3.9 Encoding of Variables and Constructors:

$$\begin{array}{lrl}
\text{(a)} & \mathcal{E}[\![x]\!] \triangleq \uparrow x & \text{if } x \in \mathsf{V} \\
\text{(b)} & \mathcal{E}[\![a\ \overline{E_I}]\!] \triangleq \mathcal{E}[\![\overline{E}]\!] \succcurlyeq \lambda\overline{v}.\uparrow(a\,\overline{v}) & \text{if } a \in \mathsf{D}
\end{array}$$

In this encoding, variables represent themselves (Rule 3.9a), and the results of evaluating the argument expressions $\overline{E}$ of a constructor form the arguments of the resultant term (Rule 3.9b). The notation $\mathcal{E}[\![\overline{E}]\!]$ is a shorthand for the parallel composition of the encodings of the individual expressions (i.e., for $\mathcal{E}[\![E_1]\!] \upuparrows \cdots \upuparrows \mathcal{E}[\![E_n]\!]$).

$\mathcal{D}$ lambda terms are mapped to lambda terms of linear logic that abstract the encoding of the body expression (Rule 3.10a). Hence, an application $E_1\ E_2$ obtains the result of E_1 and E_2, and applies the former result, which, in type correct programs, is guaranteed to be a lambda term, to the latter (Rule 3.10b).

3.10 Encoding of Abstraction and Application:

$$\begin{array}{ll}
\text{(a)} & \mathcal{E}[\![\lambda v.E]\!] \triangleq \uparrow(\lambda v.\mathcal{E}[\![E]\!]) \\
\text{(b)} & \mathcal{E}[\![E_1\ E_2]\!] \triangleq (\mathcal{E}[\![E_1]\!] \upuparrows \mathcal{E}[\![E_2]\!]) \succcurlyeq \lambda\langle f, v\rangle.f\,v
\end{array}$$

In the last rule, the standard encoding would have been $\mathcal{E}[\![E_1]\!] \succcurlyeq \lambda f.(\mathcal{E}[\![E_2]\!] \succcurlyeq \lambda v.f\,v)$. In contrast, the displayed encoding does not fix the order of reducing E_1 and E_2, but still encodes call-by-value as the actual application synchronizes on the availability of v. It is more in the style of the rest of the encoding than the standard encoding would be.

So far, there was no surprise. More interesting is the encoding of the ($\mathsf{let} \cdot \mathsf{in} \cdot$) construct and of dereferencing. $\mathcal{D}$ names (i.e., values of kind $\sharp$) are encoded by values of the type *name*, which has been introduced with the process combinators. More precisely, we create, for every binding $x = E$, a new name with the process combinator *new* and associate it with the variable x. Then, the encoding $\mathcal{E}\llbracket E \rrbracket$ of the right-hand side of the binding is stored under the name of x (using the process combinator *put*). Finally, the body of the ($\mathsf{let} \cdot \mathsf{in} \cdot$) is executed after all bindings have been stored in this way—see Rules 3.11b and 3.11c below.

Now consider Rule 3.11a. When a name of a binding x is dereferenced in the body of the ($\mathsf{let} \cdot \mathsf{in} \cdot$), the encoding of the right-hand side of the corresponding binding is retrieved using *get*. Then, this encoding is executed (i.e., the right-hand side is evaluated) and the result is returned, after it is stored with the name x.

3.11 ENCODING OF BINDINGS AND DEREFERENCING:

(a) $\mathcal{E}\llbracket E{\downarrow} \rrbracket \triangleq \mathcal{E}\llbracket E \rrbracket \succsim \lambda x.\mathit{get}\, x \succsim \lambda e.e \succsim \lambda v.\mathit{put}\, x\,(\uparrow v) \succ \uparrow v$

(b) $\mathcal{E}\llbracket \mathsf{let}\ \overline{B_I}\ \mathsf{in}\ E \rrbracket \triangleq (\upuparrows_{i \in I} \mathit{new}) \succsim \lambda \overline{v_I}.\mathcal{B}\llbracket \overline{B_I} \rrbracket \succ \mathcal{E}\llbracket E \rrbracket \qquad \text{if } \overline{v_I} = \operatorname{dom} \overline{B_I}$

(c) $\mathcal{B}\llbracket x = E \rrbracket \triangleq \mathit{put}\, x\,(\mathcal{E}\llbracket E \rrbracket)$

There are some subtle points in the treatment of bindings. First, remember that $\mathit{put}\, x\,(\mathcal{E}\llbracket E \rrbracket)$ produces a single, linear reference $\mathit{ref}\, x\,(\mathcal{E}\llbracket E \rrbracket)$. Thus, a $\mathit{get}\, x$ (produced by dereferencing) consumes this single reference, which represents the binding $x = E$. As a substitute, the encoding of dereferencing (Rule 3.11a) establishes a new reference. The new reference has the form $\mathit{ref}\, x\,(\uparrow t)$, where t is the encoding of the result of E (i.e., $\mathcal{E}\llbracket E \rrbracket\, r$ produces a result message $r\, t$). Consequently, the next time when x is dereferenced, $\mathit{get}\, x$ obtains the process combinator $\uparrow t$—instead of $\mathcal{E}\llbracket E \rrbracket$—which directly yields t. Effectively, the first evaluation of E (i.e., the execution of $\mathcal{E}\llbracket E \rrbracket$) is shared by all subsequent accesses to x. Hence, the encoding of bindings realizes lazy evaluation for ($\mathsf{let} \cdot \mathsf{in} \cdot$) bound expressions.

Another interesting point is the implicit encoding of the fixpoint combinator fix, which is part of the ($\mathsf{let} \cdot \mathsf{in} \cdot$) construct according to Notation 3.6. In the fragment $\lambda e.e \succsim \lambda v.\mathit{put}\, x\,(\uparrow v) \succ \uparrow v$ of Rule 3.11a, the term that is bound to v is duplicated. When in the dereferenced binding $x = E$, the expression E is a function that uses itself, then $\mathcal{E}\llbracket E \rrbracket$ contains the encoding of the $\mathcal{D}$ expression $x{\downarrow}$—that means, it contains the duplication code $\lambda e.e \succsim \lambda v.\mathit{put}\, x\,(\uparrow v) \succ \uparrow v$. This implies that on execution, the right-hand side of the binding duplicates itself, which realizes recursion. Note that this is only possible due to the availability of higher-order quantification in our linear logic processes.[10]

The encoding of the $\mathsf{case}\ E\ \mathsf{of}\ \overline{X}$ construct is straight forward. An ask method is used for each alternative in $\overline{X}$, where an equational constraint compares the result of E with the pattern in the alternative (Rule 3.12b). The encodings of the alternatives are composed using the process combinator for indeterministic choice $\Lsh\!\Rsh$ (Rule 3.12a).

3.12 ENCODING OF CASE:

(a) $\mathcal{E}\llbracket \mathsf{case}\ E\ \mathsf{of}\ \overline{X_I} \rrbracket \triangleq \mathcal{E}\llbracket E \rrbracket \succsim \lambda v.\, \Lsh\!\Rsh_{i \in I}\, \mathcal{X}\llbracket X_i \rrbracket_v$

(b) $\mathcal{X}\llbracket a\, \overline{v} \rightarrow E \rrbracket_w \triangleq \mathit{ask}\,(\lambda \overline{v}.a\, \overline{v} \doteq w)\,(\lambda \overline{v}.\mathcal{E}\llbracket E \rrbracket)$

---

[10] In this case, the $\forall$ in the definition of *get* quantifies over linear logic processes—see also the reconstruction of contraction by means of second-order quantification that is used by Lafont (1996).

3.4.3 Encoding Agents

Agents are encoded by the scheme $\mathcal{A}[\![\cdot]\!] :: \mathsf{A} \to \mathbf{P}$. Hence, we assume $\mathcal{E}[\![A]\!] = \mathcal{A}[\![A]\!]$ for $A \in \mathsf{A}$. We start with the encoding of failure and success.

3.13 ENCODING OF FAILURE AND SUCCESS:

(a) $\mathcal{A}[\![\mathsf{0}]\!] \triangleq \mathit{fail}$

(b) $\mathcal{A}[\![\mathsf{1}]\!] \triangleq \uparrow\langle\rangle$

The justification for encoding success by $\uparrow\langle\rangle$ was already given in the discussion of the process combinators.

Equational constraints come in two flavours, either the arguments are of kind $\sharp$ or of kind $\star$. In the first case, the result of evaluating the arguments are variable names; while in the second case, the arguments are data constructors or lambda terms. We handle both cases by passing the results of evaluating the arguments to a predicate *tell* (which is discussed below).

3.14 ENCODING OF EQUATIONAL CONSTRAINTS:

(a) $\mathcal{A}[\![E_1 \doteq E_2]\!] \triangleq (\mathcal{E}[\![E_1]\!] \upuparrows \mathcal{E}[\![E_2]\!]) \succsim \lambda\langle v_1, v_2\rangle.(\mathit{tell}\, v_1\, v_2) \upharpoonright \uparrow\langle\rangle$

The behaviour of tell is defined by a set of axioms (in a similar way as **Eq** in Definition 3.2).

3.15 DEFINITION (AXIOMS OF *tell*)
The axioms of *tell* are as follows, where we require a (decompose) axiom for every constructor $a \in \mathsf{D}$ and an axiom (mismatch) for each pair of distinct constructors $a, b \in \mathsf{D}$.

$$\forall x, y^{name}.\mathit{tell}\; x\; y \multimapinv\ ?(x \doteq y) \qquad \text{(assert name alias)}$$

$$\forall \overline{t_I}, \overline{u_I}.\mathit{tell}\; (a\,\overline{t_I})\,(a\,\overline{u_I}) \multimapinv \textstyle\parr_{i \in I}\, \mathit{tell}\; t_i\; u_i \qquad \text{(decompose)}$$

$$\forall \overline{t}, \overline{u}.\mathit{tell}\; (a\,\overline{t})\; (b\,\overline{u}) \multimapinv \top \qquad \text{(mismatch)}$$

The are no axioms for the equality of lambda terms, which implies that lambda terms are considered as incomparable. ∎

The axiom (assert name alias) stores an equation of the form $?(x \doteq y)$ where x and y are of type *name*. This has, together with the equality theory **Eq** (Definition 3.2), an important implication for the references generated by the encoding of ($\mathsf{let} \cdot \mathsf{in} \cdot$) (Encoding 3.11). Assume that there is a references $\mathit{ref}\, x_1\, (\mathcal{E}[\![E_1]\!])$ and $\mathit{ref}\, x_2\, (\mathcal{E}[\![E_2]\!])$ and we assert $?(x_1 \doteq x_2)$. Then, any subsequent *get* on x_1 or x_2 can, by the (substitutivity) axiom of Definition 3.2, retrieve any of the two references. The two bindings became interchangeable. From the point of view of constraint programming, this requires to check the *consistency* between these two bindings.

3.16 DEFINITION (CONSISTENCY CHECKER)
A permanent consistency check is ensured by adding the following axiom.

$$\begin{aligned} &\forall x^{name}.(\mathit{ref}\, x\, e_1 \parr \mathit{ref}\, x\, e_2) \multimapinv \\ &\qquad (\forall s_1, s_2.e_1\, s_1 \parr e_2\, s_2 \parr (\forall v_1, v_2.(s_1\, v_1 \parr s_2\, v_2) \multimapinv \mathit{ref}\, x\, (\uparrow v_1) \parr \mathit{tell}\, v_1\, v_2)^{\perp}) \end{aligned}$$

∎

Whenever there exist two references for the same name x, the consistency checker obtains the process combinators e_1 and e_2 that are stored under this names, executes them by application to the channels s_1 and s_2, and passes the results v_1 and v_2 to *tell*. Furthermore, one reference is put back, to allow subsequent dereferencing of the name x.

The encoding of parallel composition (Rule 3.17a) and sequential composition (Rule 3.17b) directly uses the corresponding process combinators $\upuparrows$ and $\succ$.

3.17 Encoding of Parallel and Sequential Composition:

(a) $\mathcal{A}[\![E_1 \parallel E_2]\!] \triangleq \mathcal{E}[\![E_1]\!] \upuparrows \mathcal{E}[\![E_2]\!]$

(b) $\mathcal{A}[\![E_1 \Rightarrow E_2]\!]_r \triangleq \mathcal{E}[\![E_1]\!] \succ \mathcal{E}[\![E_2]\!]$

The introduction of an unbound variable generates a new *name* in the same way as the bindings of a $(\mathsf{let} \cdot \mathsf{in} \cdot)$ construct, but no reference is established. Consequently, dereferencing an unbound variable blocks. But, when *tell* asserts an equation that includes the unbound variable, then any binding whose name is aliased with the unbound variable can be used instead.

3.18 Encoding of Unbound Variables:

(a) $\mathcal{A}[\![\nu x.E]\!] \triangleq \mathit{new} \succcurlyeq (\lambda x.\mathcal{E}[\![E]\!])$

Now that we fixed the behaviour of unbound variables and equational constraints, the claim that referential transparency is preserved in $\mathcal{D}$ for purely functional subcomputations is discussed in more detail. Whenever a computation attempts to dereference an unbound variable, it blocks (no matching reference is available). Only when subsequently an equational constraint aliases the unbound variable with a binding of a $(\mathsf{let} \cdot \mathsf{in} \cdot)$, this binding is used, as if it had been dereferenced instead of the unbound variable in the first place. Consequently, the computation that performs the dereferencing is not able to observe the fact that the variable has been unbound previously, and the process of binding remains transparent. If more than one binding is associated with an unbound variable, the consistency checker of Definition 3.16 ensures that either the two bindings provide the same value or the whole computation fails—the latter is the case when the precondition of the (mismatch) axiom of Definition 3.15 is satisfied.

Finally, we have to encode choice expressions—that is, expressions from S (Notation 3.8). Ask expressions are realized by the *ask* process combinator, where the right-hand sides of the guards are evaluated before the guards are combined into a compound guard using $\parr$ (Rule 3.19c, below). Although linear logic process according to Definition 3.1 do not directly support compound guards, they can be regarded as a shorthand, because we have $(A \parr B) \multimapinv C = A \multimapinv (B \multimapinv C)$ in linear logic. Indeterministic choice $\Lsh\!\Rsh$, in Rule 3.19d, combines ask expressions.

3.19 Encoding of Choice:

(a) $\mathcal{A}[\![S]\!] \triangleq \mathcal{S}[\![S]\!]$ if $S \in \mathsf{S}$

(b) $\mathcal{S}[\![\top]\!] \triangleq \mathit{err}$

(c)
$$\mathcal{S}[\![\overline{v} : \overline{P_I} \doteq \overline{E_I} \to E]\!] \triangleq \Big(\upuparrows_{i \in I} \mathcal{E}[\![E_i]\!]\Big) \succcurlyeq \lambda\overline{w_I}.\mathit{ask}\,(\lambda\overline{v}.\, \parr_{i \in I} P_i \doteq w_i)\,(\lambda\overline{v}.\mathcal{E}[\![E]\!])$$

(d) $\mathcal{S}[\![S_1 \,[\!]\, S_2]\!] \triangleq \mathcal{S}[\![S_1]\!] \Lsh\!\Rsh \mathcal{S}[\![S_2]\!]$

3.4.4 Reduction by Proof

Now we may assemble the components of the logic semantics. The logic semantics of $\mathcal{D}$ expressions is accompanied by three sets of axioms, namely **Eq** (Definition 3.2), the axioms of *tell* (Definition 3.15), and the consistency checker (Definition 3.16); collectively, we call them **D**.

3.20 DEFINITION (LOGIC SEMANTICS OF $\mathcal{D}$ EXPRESSIONS)
The logic semantics of an expression $E \in \mathsf{E}$ is $\mathcal{E}[\![E]\!]$ under the theory $\mathbf{D}$. ∎

Remember from Section 3.2.1 that reversed linear implication ($\multimapinv$) represents reduction of linear logic processes. So, by virtue of the logic semantics of $\mathcal{D}$, we can utilize Forum proofs to reason about the reduction of $\mathcal{D}$ expressions. In particular, if there is an $\mathcal{F}$ proof for the Forum sequent

$$\Sigma : \mathbf{D}; \cdot \longrightarrow \forall r.\mathcal{E}[\![E]\!]\, r \multimapinv R; \cdot, \tag{3.3}$$

then the encoding of E reduces to R. In the next section, we will discuss relations where R is the encoding of another $\mathcal{D}$ expression E'; in this case, we may say that E reduces to E'. To be less verbose, we will just write $\mathcal{E}[\![E]\!]\, r \multimapinv R$ instead of the complete Equation (3.3), when reasoning about the reduction of $\mathcal{D}$ encodings.

It is interesting to ask which kind of structures are produced on the result channel for the encoding of the various types of $\mathcal{D}$ expressions. The encoding of the ($\mathsf{let} \cdot \mathsf{in} \cdot$) construct (Rule 3.11b) obviously realizes $\mathcal{D}$ names by values of the type *name* in linear logic. So, we have

$$\mathcal{E}[\![\mathsf{let}\ v = E\ \mathsf{in}\ v]\!]\, r \multimapinv (r\, x \parr \mathit{ref}\, x\, (\mathcal{E}[\![E]\!])),$$

where x is the *name* representing v in the encoding, and the reference stores the association of v with E.

Expressions of a type of terms are handled differently. In particular, data terms are represented by data terms with the same constructor in the encoding. By Rule 3.9b we see that

$$\mathcal{E}[\![\mathit{Nil}\ \langle\rangle]\!]\, r \multimapinv r\, (\mathit{Nil}\ \langle\rangle) \qquad \text{and} \qquad \mathcal{E}[\![\mathit{Cons}\ \langle E_1, E_2\rangle]\!]\, r \multimapinv r\, (\mathit{Cons}\ \langle t_1, t_2\rangle)$$

where t_1 and t_2 are the results of executing the encoding of E_1 and E_2 (these may be *names* or terms depending on the type of E_1 and E_2).

3.5 Deriving an Operational Semantics

The encoding of $\mathcal{D}$ into Forum formulae provides an operational interpretation for $\mathcal{D}$ via the operational interpretation of Forum. Below, we derive a transition system that operates directly on $\mathcal{D}$ expressions from the linear logic encoding. The transition system consists of two components, a structural congruence and a transition relation. Following Milner (1991)—and indirectly Berry & Boudol (1990)—the structural congruence identifies expressions that have the same computational behaviour, and the transition relation is defined modulo the structural congruence.

3.5.1 Normal Forms

Before we proceed to the structural congruence and the transition relation, we first define the normal forms of $\mathcal{D}$.

3.21 DEFINITION (NORMAL FORMS OF $\mathcal{D}$ EXPRESSIONS)
The following grammar defines the *normal forms* of $\mathcal{D}$ expressions.

$$\begin{array}{llll} \text{Normal form} & \mathsf{N} \to & \mathsf{V} & \text{(variable)} \\ & \mid & \mathsf{D}\,\overline{\mathsf{N}} & \text{(data term)} \\ & \mid & \lambda\mathsf{V}.\mathsf{E} & \text{(lambda term)} \\ & \mid & \mathsf{0} & \text{(failure)} \\ & \mid & \mathsf{1} & \text{(success)} \end{array}$$

Normal forms are constructor or lambda terms, or they consist of one of the two terminating agents. ∎

Section 3.3.2 (Definition 3.6) states that types of arguments must either be of a name type or, when they are of a term type, they are neither allowed to be of type o nor of a function type. This restriction guarantees that a normal form in an argument position is either of a type of names, in which case it is a variable name, or it is a data term (this holds recursively for the arguments of a data term). The subset of normal forms that may occur in argument positions is named $\mathsf{N_0}$, and it easy to see that for every $N \in \mathsf{N_0}$,

$$\forall r.\mathcal{E}[\![N]\!]\,r \multimapinv r\,N.$$

This justifies the following special case in the encoding, which we add explicitly to the encoding rules for technical reasons.

3.22 ENCODING OF NORMAL FORMS:

$$\mathcal{E}[\![N]\!] \triangleq \uparrow N \qquad \text{if } N \in \mathsf{N_0}$$

Furthermore, we call equational constraints that range over variables normal constraints.

3.23 DEFINITION (NORMAL CONSTRAINTS)
The set C of *normal $\mathcal{D}$ constraints* consists of constraints $x \doteq y$ where $x, y \in \mathsf{V}$. Given a set Ψ of normal constraints, we define the *constraint entailment relation* $\Vdash$ as the smallest relation satisfying the following rules, where $N, N_1, N_2, M_1, M_2 \in \mathsf{N_0}$.

$$\frac{x \doteq y \in \Psi}{\Psi \Vdash x \doteq y}\,\text{(store)} \qquad \frac{}{\Psi \Vdash N \doteq N}\,\text{(refl)}$$

$$\frac{\Psi \Vdash N_2 \doteq N_1}{\Psi \Vdash N_1 \doteq N_2}\,\text{(sym)} \qquad \frac{\Psi \Vdash N_1 \doteq N_2 \qquad \Psi \Vdash N_2 \doteq N_3}{\Psi \Vdash N_1 \doteq N_3}\,\text{(trans)}$$

$$\frac{\Psi \Vdash N_1 \doteq N_2 \qquad \Psi \Vdash M_1 \doteq M_2}{\Psi \Vdash (\lambda x.N_1)\,M_1 \doteq (\lambda x.N_2)\,M_2}\,\text{(subst)}$$

∎

Considering that the equality theory **Eq** (Definition 3.2) is part of all reductions of $\mathcal{D}$ expressions (see Section 3.4.4), we have the following law of constraint entailment, where $?\{x_1 \doteq y_1, \ldots, x_n \doteq y_n\}$ means $?(x_1 \doteq y_1) \parr \cdots \parr ?(x_n \doteq y_n)$.

3.24 LAW OF CONSTRAINT ENTAILMENT:

$$\Psi \Vdash N_1 \doteq N_2 \quad \text{iff} \quad ?\Psi \multimapinv N_1 \doteq N_2$$

3.5.2 Structural Congruence

The congruence $\equiv$ does not identify all $\mathcal{D}$ expressions that intuitively could be regarded as congruent—for example, $\nu x.\nu y.E \not\equiv \nu y.\nu x.E$.[11] But it identifies expressions that otherwise would have required additional transition rules.

3.25 DEFINITION (STRUCTURAL CONGRUENCE $\equiv$ ON $\mathcal{D}$ EXPRESSIONS)
The structural congruence $\equiv$ is defined to be the smallest congruence relation satisfying the following rules:

- $E \equiv E'$ if $E \overset{\alpha}{\equiv} E'$;
- $(\mathsf{E}/\equiv, \|, \mathsf{1})$ is a symmetric monoid and $\mathsf{0} \parallel E \equiv \mathsf{0}$;
- $\mathsf{0} \Rightarrow E \equiv \mathsf{0}$; and
- $(\mathsf{S}/\equiv, [], \top)$ is a symmetric monoid. ∎

By the encoding of $\|$, $\mathsf{1}$, $\mathsf{0}$, $\Rightarrow$, $[]$, and $\top$, it is easy to show that $\equiv$ is indeed a congruence.

3.5.3 The Transition Relation

The transition relation $\rightrightarrows$ is defined by linear implication. This is in spirit similar to Hoare, He & Sampaio (1995), who define a transition relation in terms of a refinement relation; so they can derive the transition rules from the refinement relation—instead of postulating the rules, and then, proving soundness. Similarly, we shall derive the transition rules from linear implication. But first, we lift linear implication, which ranges over encodings of $\mathcal{D}$ expressions, to a relation over plain $\mathcal{D}$ expressions.

3.26 DEFINITION (TRANSITION RELATION)
We lift linear implication to lambda functions over linear formulae as follows: For all linear logic lambda terms A and B of type $\sigma_1 \to \cdots \to \sigma_n \to o$, $n \geq 0$, we define

$$A \sqsubseteq B \quad \text{iff} \quad \forall x_1 \ldots \forall x_n . A\, x_1 \cdots x_n \multimapinv B\, x_1 \cdots x_n .$$

(Remember that the theory **D** is implicitly available—see Section 3.4.4.) We define the *transition relation* $\rightrightarrows$ between $\mathcal{D}$ expressions by means of the linear logic encoding as

$$\text{for all } E_1, E_2 \in \mathsf{E}, \quad E_1 \rightrightarrows E_2 \quad \text{iff} \quad \mathcal{E}[\![E_1]\!] \sqsubseteq \mathcal{E}[\![E_2]\!]$$ ∎

---

[11] The congruence could be extended, of course, but this wouldn't help us in the derivation of the operational semantics.

By virtue of this definition, $\sqsubseteq$ and $\rightrightarrows$ immediately inherit some properties from linear implication.

3.27 LEMMA (TRANSITIVITY OF $\sqsubseteq$ AND $\rightrightarrows$)
The relations $\sqsubseteq$ and $\rightrightarrows$ are reflexive and transitive. ▮

PROOF.
It is sufficient to show that

- $\forall x.A\,x \multimapinv A\,x$ and
- $((\forall x.A\,x \multimapinv B\,x) \otimes (\forall x.B\,x \multimapinv C\,x)) \multimap (\forall x.A\,x \multimapinv C\,x)$

are tautologous in Forum.[12] ▮

Subsequently, we derive a set of transition rules that characterizes $\rightrightarrows$ completely. We do this by reasoning about the encodings of the participating $\mathcal{D}$ expressions. As the encodings are based on the process combinators, we start by deriving the basic properties of these combinators.

Preliminaries

The following rule is important to reason about the use of receive methods (see Definition 3.1).

3.28 COMMUNICATION RULE:

$$\forall s.s\,M \parr (\forall v.s\,v \multimapinv P)^{\perp} \sqsubseteq [M/v]P$$

PROOF.
It is sufficient to see that

$$(\forall s.s\,M \parr (\forall v.s\,v \multimapinv P)^{\perp}) \multimapinv [M/v]P$$

is a tautology in Forum. ▮

Using the communication rule, we can calculate a basic relationship between the combinators $\uparrow$ (unit) and $\succcurlyeq$ (bind). Assume some arbitrary, but fixed r, then[13]

$$\begin{array}{ll} & (\uparrow M \succcurlyeq \lambda x.N)\,r \\ = & \quad \{\text{Definition of } (\uparrow) \text{ and } (\succcurlyeq)\text{; normalization}\} \\ & \forall s.s\,M \parr \forall v.s\,v \multimapinv (\lambda x.N)\,v\,r \\ \sqsubseteq & \quad \{\text{Communication rule}\} \\ & (\lambda x.N)\,M\,r \\ = & \quad \{\text{Normalization}\} \\ & ([M/x]N)\,r \end{array}$$

Hence, we have established the $\uparrow/\succcurlyeq$ law.

---

[12] Here and below, we refrain from explicitly constructing large Forum proof trees to prove such tautologies. Especially as these proofs are constructed rather mechanical due to Forum's restricted proof structure.

[13] The calculation with an arbitrary, but fixed, r to justify the universal quantification in Definition 3.26 is sound by the right introduction rule for $\forall$ in Forum.

3.29 LAW OF $\uparrow/\succcurlyeq$:

$$\uparrow M \succcurlyeq \lambda x.N \sqsubseteq [M/x]N$$

For the case where the communicated value is not used by the body of the method, we can do better than the communication rule—we can derive the symmetric synchronization rule. We use $A \cong B$ as an abbreviation for $A \multimap B \parr B \multimap A$.

3.30 SYNCHRONIZATION RULE:

$$\forall s.s\,M \parr (\forall v.s\,v \multimapinv P)^{\bot} \cong P \qquad \text{if } v \notin \mathrm{Fr}\,P$$

PROOF.
The direction from left-to-right is an instance of the communication rule. For the opposite direction, we have to show that

$$(\forall s.s\,M \parr (\forall v.s\,v \multimapinv P)^{\bot}) \multimap P$$

is a tautology in Forum. As the corresponding Forum proof requires one tricky step, consider the following proof tree:

$$
\dfrac{
 \dfrac{
  \dfrac{
   \dfrac{}{\Sigma : \Psi; \cdot \xrightarrow{\bot} \cdot; \Upsilon}\,(\bot\text{L})
   \qquad
   \dfrac{
    \dfrac{
     \begin{array}{c}\vdots\\ \Sigma : \Psi; \cdot \longrightarrow P, \bot \multimapinv P; \Upsilon\end{array}
    }{\Sigma : \Psi; \cdot \longrightarrow P, \forall v.\bot \multimapinv P; \Upsilon}\,(\forall\text{R}) \quad \text{it is } v \notin \mathrm{Fr}\,P
   }{\Sigma : \Psi; \cdot \xrightarrow{(\forall v.\bot \multimapinv P)^{\bot}} P; \Upsilon}\,(\multimap\text{L}, \bot\text{L})
  }{\Sigma : \Psi; \cdot \xrightarrow{\bot \parr (\forall v.\bot \multimapinv P)^{\bot}} P; \Upsilon}\,(\parr\text{L})
 }{\Sigma : \Psi; \cdot \xrightarrow{\forall s.s\,M \parr (\forall v.s\,v \multimapinv P)^{\bot}} P; \Upsilon}\,(\forall\text{L}) \quad \text{where we choose } s = \lambda x.\bot
}{
 \dfrac{\Sigma : \Psi; \forall s.s\,M \parr (\forall v.s\,v \multimapinv P)^{\bot} \longrightarrow P; \Upsilon}{\Sigma : \Psi; \cdot \longrightarrow (\forall s.s\,M \parr (\forall v.s\,v \multimapinv P)^{\bot}) \multimap P; \Upsilon}\,(\multimap\text{R})
}\,(\text{decide})
$$

Notice the choice of s in ($\forall$L). ∎

By this result together with the fact that, in $M \succ N = M \succcurlyeq \lambda v.N$, the variable v does not occur free in N, we easily adapt the calculation for the law of $\uparrow/\succcurlyeq$ to yield the following stronger law for $\succ$.

3.31 LAW OF $\uparrow/\succ$:

$$\uparrow M \succ N = N$$

By using the communication rule two times, we can derive the following law governing the interaction between $\uparrow$ and $\upuparrows$. (Remember that $(A \parr B) \multimapinv C \cong A \multimapinv (B \multimapinv C)$ in linear logic.)

3.32 LAW OF $\uparrow/\upuparrows$:

$$(\uparrow M_1) \upuparrows (\uparrow M_2) \sqsubseteq \uparrow\langle M_1, M_2\rangle$$

To reason about ask methods, we use the following ask rule.

3.33 ASK RULE:

$$(M_1 \doteq M_2) \parr \forall \overline{v}.(N_1 \doteq N_2) \multimapinv P \sqsubseteq [\overline{N'}/\overline{v}]P \qquad \text{if } M_i = [\overline{N'}/\overline{v}]N_i \text{ for } i \in \{1, 2\}$$

Similar to the communication and synchronization rules, this rule is proved by constructing the corresponding Forum proof tree. By using the ask rule and the law of constraint entailment (Property 3.24) repeatedly, together with the fact that $(A \parr B) \multimapinv C \cong A \multimapinv (B \multimapinv C)$, we can show that, for a given set of normal constraints $\Psi \subseteq \mathsf{C}$, the following law holds.

3.34 LAW OF *ask*:

$$\mathit{ask}\left(\lambda\overline{v}.\textstyle\bigparr_{j\in J} N_j^1 \doteq N_j^2\right)(\lambda\overline{v}.M) \sqsubseteq [\overline{N'}/\overline{v}]M \quad \text{if for all } j \in J,\ \Psi \Vdash [\overline{N'}/\overline{v}](N_j^1 \doteq N_j^2)$$

The next calculation derives the expected interaction between *put* and *get*.

$$\begin{array}{ll}
 & (\mathit{put}\, v\, M \succ \mathit{get}\, v)\, r \\
= & \quad \{\text{Definitions of } \succ,\ \mathit{get}, \text{ and } \mathit{put}\} \\
 & \forall s.\mathit{ref}\, v\, M \parr s\,\langle\rangle \parr (\forall w.s\, w \multimapinv (\forall w'.\mathit{ref}\, v\, w' \multimapinv r\, w')^{\perp})^{\perp} \\
\sqsubseteq & \quad \{\text{Twice the communication rule (Property 3.28)}\} \\
 & r\, M \\
= & \quad \{\text{Definition of } \uparrow\} \\
 & \uparrow M
\end{array}$$

3.35 LAW OF *put*/*get*:

$$\mathit{put}\, v\, M \succ \mathit{get}\, v \sqsubseteq \uparrow M$$

Generally in linear logic, we have $A \parr (B \multimap C) \cong B \multimap (A \parr C)$ and thus $P \parr (G \multimapinv Q)^{\perp} \cong (G \multimapinv (P \parr Q))^{\perp}$. We use this correspondence to prove the associativity of $\succcurlyeq$.[14]

$$\begin{array}{ll}
 & ((M_1 \succcurlyeq \lambda v.M_2) \succcurlyeq \lambda w.M_3)\, r \\
= & \quad \{\text{Definition of } (\succcurlyeq)\} \\
 & \forall s.(\forall s'.M_1\, s' \parr (\forall v.s'\, v \multimapinv M_2\, s)^{\perp}) \parr (\forall w.s\, w \multimapinv M_3\, r)^{\perp} \\
\cong & \quad \{\bullet s' \notin \mathrm{Fr}\, M_3\} \\
 & \forall s'.\forall s.M_1\, s' \parr (\forall v.s'\, v \multimapinv M_2\, s)^{\perp} \parr (\forall w.s\, w \multimapinv M_3\, r)^{\perp} \\
\cong & \quad \{P \parr (G \multimapinv Q)^{\perp} \cong (G \multimapinv (P \parr Q))^{\perp};\ \bullet v \notin \mathrm{Fr}\, M_3\} \\
 & \forall s'.\forall s.M_1\, s' \parr (\forall v.s'\, v \multimapinv (M_2\, s \parr (\forall w.s\, w \multimapinv M_3\, r)^{\perp}))^{\perp} \\
= & \quad \{\text{Definition of } (\succcurlyeq)\} \\
 & (M_1 \succcurlyeq \lambda v.(M_2 \succcurlyeq \lambda w.M_3))\, r
\end{array}$$

As $\succ$ is defined in terms of $\succcurlyeq$, we get associativity for $\succ$, too.

---

[14] We also implicitly use associativity of $\parr$ and similarly obvious properties of the linear logic.

3.36 Associativity of $\succcurlyeq$ and $\succ$:

(a) $(M_1 \succcurlyeq \lambda v.M_2) \succcurlyeq \lambda w.M_3 = M_1 \succcurlyeq \lambda v.(M_2 \succcurlyeq \lambda w.M_3)$ if $v \notin \mathrm{Fr}\, M_3$

(b) $(M_1 \succ M_2) \succ M_3 = M_1 \succ (M_2 \succ M_3)$

We can also move $\succcurlyeq$ into and out of the arguments of $\upuparrows$ as proved by the next calculation for the left argument of $\upuparrows$.

$$\begin{aligned}
& ((M_1 \succcurlyeq \lambda v.M_2) \upuparrows N)\, r \\
= \quad & \{\text{Definition of } (\succcurlyeq) \text{ and } (\upuparrows); \\
& \ \text{let } \langle \upuparrows \text{ method}\rangle = (\forall w_1, w_2.(s_1\, w_1 \parr s_2\, w_2) \multimapinv r\,\langle w_1, w_2\rangle)^{\perp}\} \\
& \forall s_1, s_2.(\forall t.M_1\, t \parr (\forall v.t\, v \multimapinv M_2\, s_1)^{\perp}) \parr N\, s_2 \parr \langle \upuparrows \text{ method}\rangle \\
\cong \quad & \{\bullet t \notin \mathrm{Fr}\, N\} \\
& \forall s_1, s_2.(\forall t.M_1\, t \parr (\forall v.t\, v \multimapinv M_2\, s_1)^{\perp} \parr N\, s_2 \parr \langle \upuparrows \text{ method}\rangle) \\
\cong \quad & \{P \parr (G \multimapinv Q)^{\perp} \cong (G \multimapinv (P \parr Q))^{\perp};\ \bullet v \notin \mathrm{Fr}\, N\} \\
& \forall s_1, s_2.(\forall t.M_1\, t \parr (\forall v.t\, v \multimapinv (M_2\, s_1 \parr N\, s_2 \parr \langle \upuparrows \text{ method}\rangle))^{\perp}) \\
\cong \quad & \{\text{Rearranging the quantifiers}\} \\
& \forall t.M_1\, t \parr (\forall v.t\, v \multimapinv (\forall s_1, s_2.M_2\, s_1 \parr N\, s_2 \parr \langle \upuparrows \text{ method}\rangle))^{\perp} \\
= \quad & \{\text{Definition of } (\succcurlyeq) \text{ and } (\upuparrows)\} \\
& (M_1 \succcurlyeq \lambda v.(M_2 \upuparrows N))\, r
\end{aligned}$$

The same calculation works with the right argument of $\upuparrows$, and again, we can transfer the result from $\succcurlyeq$ to $\succ$.

3.37 Law of $\succcurlyeq/\upuparrows$:

(a) $(M_1 \succcurlyeq \lambda v.M_2) \upuparrows N = M_1 \succcurlyeq \lambda v.(M_2 \upuparrows N)$

(b) $M \upuparrows (N_1 \succcurlyeq \lambda v.N_2) = N_1 \succcurlyeq \lambda v.(M \upuparrows N_2)$

(c) $(M_1 \succ M_2) \upuparrows N = M_1 \succ (M_2 \upuparrows N)$

(d) $M \upuparrows (N_1 \succ N_2) = N_1 \succ (M \upuparrows N_2)$

A further simple calculation can be used to show that $\succcurlyeq$ may be pushed into the right argument of $\upharpoonright$, which gives us the following law.

3.38 Law of $\upharpoonright/\succcurlyeq$:

$$(P \upharpoonright M) \succcurlyeq N = P \upharpoonright (M \succcurlyeq N)$$

Considering the definition of $\Lsh\!\Rsh$ by the linear logic connective $\oplus$, we can immediately deduce the next laws.

3.39 Laws of $\Lsh\!\Rsh$:

(a) $M_1 \Lsh\!\Rsh M_2 \sqsubseteq M_1$

(b) $M_1 \Lsh\!\Rsh M_2 \sqsubseteq M_2$

Furthermore, the process combinators $\succcurlyeq$, $\succ$, $\upuparrows$, and $\Lsh\!\Rsh$ are monotonic with respect to the relation $\sqsubseteq$.

3.40 Lemma (Monotonicity of $\succcurlyeq$, $\succ$, $\upuparrows$, and $\Lsh\!\Rsh$)
If $M \sqsubseteq M'$, $N_1 \sqsubseteq N_1'$, $N_2 \sqsubseteq N_2'$, *and* $N_3 \sqsubseteq N_3'$ *(where* M *and the* N_i *are of the appropriate types), we have*

- $(M \succcurlyeq N_1) \sqsubseteq (M' \succcurlyeq N_1')$,
- $(M \succ N_2) \sqsubseteq (M' \succ N_2')$,
- $(M \upuparrows N_2) \sqsubseteq (M' \upuparrows N_2')$, *and*
- $(M \Lsh\!\Rsh N_3) \sqsubseteq (M' \Lsh\!\Rsh N_3')$. ∎

Proof.
The Forum formula

$$((\forall v, r.N\,v\,r \multimapinv N'\,v\,r) \otimes (\forall s.M\,s \multimapinv M'\,s)) \multimap ((\forall r.(M \succcurlyeq N)\,r) \multimapinv (\forall r.(M' \succcurlyeq N')\,r))$$

is a tautology, which implies the monotonicity of $\succcurlyeq$. The latter, in turn, immediately leads to the monotonicity of $\succ$. The monotonicity of $\upuparrows$ and $\Lsh\!\Rsh$ is proved by two corresponding tautologies. ∎

Reduction Context

Reduction contexts are useful to determine the next *redex* (reducible expression) in an expression (Felleisen, Friedman & Kohlbecker 1987), while saving on the use of structure rules in the transition system.

3.41 Definition ($\mathcal{D}$ Reduction Context)
The *reduction context* of a $\mathcal{D}$ expression is defined as follows.

$$\begin{array}{llll}
\text{Reduction context} & \mathsf{L} & \rightarrow & \langle\cdot\rangle & \text{(hole)} \\
 & & \mid & \mathsf{D}\ (\overline{\mathsf{N}}, \mathsf{L}, \overline{\mathsf{E}}) & \text{(constructor argument)} \\
 & & \mid & \mathsf{L}\ \mathsf{E} & \text{(function)} \\
 & & \mid & \mathsf{N}\ \mathsf{L} & \text{(argument)} \\
 & & \mid & \mathsf{L}{\downarrow} & \text{(designator)} \\
 & & \mid & \mathsf{case}\ \mathsf{L}\ \mathsf{of}\ \overline{\mathsf{X}} & \text{(scrutinee)} \\
 & & \mid & \mathsf{L} \doteq \mathsf{E} & \text{(left in equation)} \\
 & & \mid & \mathsf{N} \doteq \mathsf{L} & \text{(right in equation)} \\
 & & \mid & \mathsf{L} \Rightarrow \mathsf{E} & \text{(precondition)} \\
 & & \mid & \overline{\mathsf{V}} : \overline{\mathsf{P}} \doteq (\overline{\mathsf{N}}, \mathsf{L}, \overline{\mathsf{E}}) \rightarrow \mathsf{E} & \text{(ask component)}
\end{array}$$

The free variables $\mathrm{Fr}\,\Lambda$ of a reduction context $\Lambda \in \mathsf{L}$ are the variables that are free in $\Lambda\langle E\rangle$ for any E. Bound variables are defined analogously. ∎

Note that arguments in function applications can be reduced (albeit only after the function is normal form) as we implement function application with a call-by-value semantics.[15] Argument tuples in data constructor applications and ask expressions are reduced from left-to-right, and similarly the left argument in an equational constraint is reduced before the right one.

---

[15] As explained in Section 3.1.5, $\mathcal{D}$ supports lazy evaluation due to the distinction between terms and names.

There is no provision for reduction below lambda abstractions and within bindings. The former indicates that we regard lambda abstractions as normal forms, and the latter will be allowed by the reduction rules only when the expression in the left-hand side of the binding is actually needed—this is important to faithfully realize lazy evaluation.

Anyway, the definition of reduction contexts leaves the treatment of many constructs open (e.g., reduction in the body of a $(\mathsf{let}\ \cdot\ \mathsf{in}\ \cdot)$ or in the arguments of a parallel composition). These constructs are explicitly handled by the transition rules.

For the use of reduction contexts in the transition system, it is important that we can move $(\mathsf{let}\ \cdot\ \mathsf{in}\ \cdot)$ constructs, unbound variables, and solved constraints out of a reduction context. This is possible by the following lemma.

3.42 LEMMA

Given a reduction context $\Lambda \in \mathsf{L}$, *the following distribution laws hold:*

$$\begin{array}{rll} \Lambda\langle \mathsf{let}\ \overline{B}\ \mathsf{in}\ E\rangle = \mathsf{let}\ \overline{B}\ \mathsf{in}\ \Lambda\langle E\rangle & & \textit{if}\ \mathrm{dom}\, B \cap \mathrm{Fr}\, \Lambda = \emptyset \\ \Lambda\langle \nu v.E\rangle = \nu v.\Lambda\langle E\rangle & & \textit{if}\ v \notin \mathrm{Fr}\, \Lambda \\ \Lambda\langle x \doteq y\rangle = (x \doteq y) \Rightarrow \Lambda\langle 1\rangle & & \textit{if}\ x, y \notin \mathrm{Bd}\, \Lambda \end{array} \quad ▮$$

PROOF.

The proof for the distribution of $(\mathsf{let}\ \cdot\ \mathsf{in}\ \cdot)$ and $(\nu v.\cdot)$ goes by structural induction over the syntactic variants of L with a simple calculation for each variant. These calculations rest on the associativity of $\succcurlyeq$ and $\succ$ (Property 3.36) and the law of $\succcurlyeq/\upuparrows$ (Property 3.37). The calculation for $(\mathsf{let}\ \cdot\ \mathsf{in}\ \cdot)$ and functions (third case in the definition of L) serves as an example.

$$\begin{array}{ll} & \mathcal{E}[\![(\mathsf{let}\ \overline{B_I}\ \mathsf{in}\ E)\ E']\!] \\ = & \quad \{\text{Encoding of application (Rule 3.10b) and bindings (Rule 3.11b); } \mathrm{dom}\,\overline{B_I} = \overline{v_I}\} \\ & (((\mathop{\upuparrows}_{i\in I} \mathit{new}) \succcurlyeq \lambda\overline{v_I}.\mathcal{B}[\![\overline{B_I}]\!] \succ \mathcal{E}[\![E]\!]) \upuparrows \mathcal{E}[\![E']\!]) \succcurlyeq \lambda\langle f, v\rangle.f\,v \\ = & \quad \{\text{Two times the law of } \succcurlyeq/\upuparrows \text{ (Property 3.37); } \bullet\overline{v_I} \cap \mathrm{Fr}\, E' = \emptyset\} \\ & ((\mathop{\upuparrows}_{i\in I} \mathit{new}) \succcurlyeq \lambda\overline{v_I}.\mathcal{B}[\![\overline{B_I}]\!] \succ (\mathcal{E}[\![E]\!] \upuparrows \mathcal{E}[\![E']\!])) \succcurlyeq \lambda\langle f, v\rangle.f\,v \\ = & \quad \{\text{The associativity of both } \succcurlyeq \text{ and } \succ \text{ (Property 3.36)}\} \\ & (\mathop{\upuparrows}_{i\in I} \mathit{new}) \succcurlyeq \lambda\overline{v_I}.\mathcal{B}[\![\overline{B_I}]\!] \succ ((\mathcal{E}[\![E]\!] \upuparrows \mathcal{E}[\![E']\!]) \succcurlyeq \lambda\langle f, v\rangle.f\,v) \\ = & \quad \{\text{Encoding of applications (Rule 3.10b) and bindings (Rule 3.11b)}\} \\ & \mathcal{E}[\![\mathsf{let}\ \overline{B_I}\ \mathsf{in}\ (E\ E')]\!] \end{array}$$

The distribution of equational constraints $x \doteq y$ needs, for type correct $\mathcal{D}$ expressions, to consider only two cases, namely

- the base case $(x \doteq y) = (x \doteq y) \Rightarrow \mathbf{1}$, which requires the use of the law of $\upharpoonleft/\succcurlyeq$ (Property 3.38), and
- the induction step, $\Lambda\langle x \doteq y\rangle \Rightarrow E = (x \doteq y) \Rightarrow (\Lambda\langle 1\rangle \Rightarrow E)$, which rests on the associativity of $\succ$. ▮

3.43 LEMMA (MONOTONICITY OF REDUCTION CONTEXTS)

If $E \rightrightarrows E'$, *we have for every reduction context* $\Lambda \in \mathsf{L}$, $\Lambda\langle E\rangle \rightrightarrows \Lambda\langle E'\rangle$. ▮

PROOF.
Considering the encodings of the expressions that can form reduction contexts according to Definition 3.41, we see that Lemma 3.40 implies the monotonicity of reduction contexts. ∎

Transition Context

While the reduction context specifies the position in an expression where a reduction may take place, we use additionally a transition context, which is used to collect and maintain some information about the context of the reduced expression. In other words, the transition rules gather the unbound variables, bindings, and normalized equational constraints from the expression and maintain them in the transition context.

3.44 DEFINITION (TRANSITION CONTEXT FOR $\mathcal{D}$ EXPRESSIONS)
A *transition context* is a quadruple $\langle \xi, \Gamma, \Psi, \Lambda \rangle$ of a set of variables $\xi \subseteq \mathbf{V}$, a set of bindings $\Gamma \subseteq \mathbf{B}$, a set of normal constraints $\Psi \subseteq \mathbf{C}$, and a reduction context $\Lambda \in \mathbf{L}$. We denote the transition context of an expression $E \in \mathbf{E}$ as

$$\xi : \Gamma; \Psi \,\bullet\, \Lambda\langle E \rangle$$

which is equivalent to the $\mathcal{D}$ expression

$$\nu\xi'.\,\mathsf{let}\ \Gamma\ \mathsf{in}\ \left(||\Psi\right) \Rightarrow \Lambda\langle E \rangle \quad \text{with } \xi' = \xi \setminus \operatorname{dom}\Gamma,$$

where $\nu\xi'.F$ denotes the successive introduction of all variables in ξ', and $||\Psi$ denotes the parallel composition of the elements of Ψ, which is $\mathbf{1}$ in the case where Ψ is empty. ∎

We regard Γ as the binding environment and Ψ as the constraint store in whose context E is reduced. The use of a special syntax for a transition context instead of the equivalent $\mathcal{D}$ expression is important in the transition rules because the reduction of bindings is nonlocal. Moreover, it is necessary to pass information about the bindings and the constraint store into the two components of a parallel composition, which are reduced simultaneously in the rules below. Note that ξ collects all variables, those that have been introduced as unbound variables and those that are bound in a $(\mathsf{let} \cdot \mathsf{in} \cdot)$ construct.

The appropriateness of the definition of transition contexts depends on the fact that the transition of an expression can be lifted to a transition of the expression in an arbitrary transition context.

3.45 THEOREM (MONOTONICITY OF TRANSITION CONTEXTS)
Let $\langle \xi, \Gamma, \Psi, \Lambda \rangle$ *be a transition context. If* $E \rightrightarrows E'$*, then*

$$\xi : \Gamma; \Psi \,\bullet\, \Lambda\langle E \rangle \rightrightarrows \xi : \Gamma; \Psi \,\bullet\, \Lambda\langle E' \rangle$$ ∎

PROOF.
Given a transition context $\xi : \Gamma; \Psi \,\bullet\, \Lambda\langle E \rangle$, its equivalent $\mathcal{D}$ expressions is

$$\nu\xi'.\,\mathsf{let}\ \Gamma\ \mathsf{in}\ \left(||\Psi\right) \Rightarrow \Lambda\langle E \rangle \quad \text{with } \xi' = \xi \setminus \operatorname{dom}\Gamma, \tag{3.4}$$

due to Definition 3.44. Considering the Rules 3.18a, 3.11b, 3.17a, and 3.17b, it becomes clear that the encoding of the expression displayed in Equation (3.4) is of the form $\cdots \succ (\cdots \succ \mathcal{E}[\![\Lambda\langle E \rangle]\!])$. Now by the monotonicity of $\succ$ (Lemma 3.40), the monotonicity of transition contexts follows from the monotonicity of reduction contexts (Lemma 3.43). ∎

Structural Congruence

As mentioned, the structural congruence identifies expressions with identical behaviour. By considering the encodings of the various forms of congruent expressions, we see that for all $E_1, E_2 \in \mathbf{E}$ with $E_1 \equiv E_2$, we have $\mathcal{E}[\![E_1]\!] \sqsubseteq \mathcal{E}[\![E_2]\!]$ and also $\mathcal{E}[\![E_2]\!] \sqsubseteq \mathcal{E}[\![E_1]\!]$. For the case of alpha equivalence, it is sufficient to note that bound variables of $\mathcal{D}$ expressions are mapped to lambda bound variables by the encoding $\mathcal{E}[\![\cdot]\!]$. The remaining cases can be put down to the associativity of $\upuparrows$, the synchronization rule (3.30), and the properties of the linear logic connectives $\top$, $\oplus$, and 1.

3.46 Transition Modulo Structural Congruence:

$$\frac{E_1 \equiv E_1' \qquad \xi : \Gamma; \Psi \centerdot \Lambda\langle E_1' \rangle \rightrightarrows \xi : \Gamma; \Psi \centerdot \Lambda\langle E_2' \rangle \qquad E_2' \equiv E_2}{\xi : \Gamma; \Psi \centerdot \Lambda\langle E_1 \rangle \rightrightarrows \xi : \Gamma; \Psi \centerdot \Lambda\langle E_2 \rangle}$$

General Expressions

We start the discussion of the operational behaviour of general expressions with the application of lambda abstractions. Of course, we expect that the encoding preserves beta reduction. (Remember that, due to the typing constraints, a normal form in an argument position is always from $\mathbf{N_0}$.)

$$\begin{array}{ll}
 & \mathcal{E}[\![(\lambda v.E)\ N]\!] \\
= & \quad \{\text{Encoding of abstraction and application (3.10)}\} \\
 & (\uparrow(\lambda v.\mathcal{E}[\![E]\!]) \upuparrows \mathcal{E}[\![N]\!]) \succcurlyeq \lambda\langle f, v\rangle.f\ v \\
= & \quad \{\text{Encoding of normal forms (3.22); } \bullet N \in \mathbf{N_0}\} \\
 & (\uparrow(\lambda v.\mathcal{E}[\![E]\!]) \upuparrows \uparrow N) \succcurlyeq \lambda\langle f, v\rangle.f\ v \\
\sqsubseteq & \quad \{\text{Law of } \uparrow/\upuparrows\} \\
 & \uparrow\langle \lambda v.\mathcal{E}[\![E]\!], N\rangle \succcurlyeq \lambda\langle f, v\rangle.f\ v \\
\sqsubseteq & \quad \{\text{Law of } \uparrow/\succcurlyeq\} \\
 & [N/v]\mathcal{E}[\![E]\!] \\
= & \quad \{\text{Due to the encoding of normals forms (3.22)}\} \\
 & \mathcal{E}[\![[v/N]E]\!]
\end{array}$$

Thus, we get the following transition rule.

3.47 $\mathcal{D}$ Beta Reduction:

$$\xi : \Gamma; \Psi \centerdot \Lambda\langle(\lambda v.E)\ N\rangle \rightrightarrows \xi : \Gamma; \Psi \centerdot \Lambda\langle[v/N]E\rangle \qquad \text{if } N \in \mathbf{N_0}$$

Bindings occuring in a ($\mathsf{let} \cdot \mathsf{in} \cdot$) construct can be pulled out of the reduction context by Lemma 3.42. We introduce the pulled out bindings into the binding environment of the transition context.

3.48 Enlarging the Binding Environment:

$$\xi : \Gamma; \Psi \centerdot \Lambda\langle \mathsf{let}\ \overline{B}\ \mathsf{in}\ E\rangle \rightrightarrows \xi, \mathrm{dom}\,\overline{B} : \Gamma, \overline{B}; \Psi \centerdot \Lambda\langle E\rangle \qquad \text{if } \mathrm{dom}\,\overline{B} \text{ fresh}$$

(The proviso that the bound variables have to be fresh is no restriction as the structural congruence contains alpha conversion.)

From the point of view of the logic semantics, we can reduce the expression in the right-hand side of any binding, regardless of whether or not the binding is actually dereferenced. But operationally, this is not desirable, as it would compromise lazy evaluation. Consequently, we may only reduce bindings that are actually used, i.e., bindings that are dereferenced. The substitution of the dereferenced variable name by the right-hand side of the binding is delayed until the right-hand side is in normal form. Hence, multiple reference to the same binding share the reduction of the bound expression. The following rule allows the reduction of used bindings. The reduction occurs in the same context as the dereferencing expression.

3.49 DEMAND OF A BINDING:

$$\frac{\xi : \Gamma; \Psi \mathbin{\centerdot} \Lambda\langle E\rangle \rightrightarrows \xi' : \Gamma'; \Psi' \mathbin{\centerdot} \Lambda\langle E'\rangle}{\xi : \Gamma, x = E; \Psi \mathbin{\centerdot} \Lambda\langle y{\downarrow}\rangle \rightrightarrows \xi' : \Gamma', x = E'; \Psi' \mathbin{\centerdot} \Lambda\langle y{\downarrow}\rangle} \qquad \text{if } \Psi \Vdash (x \doteq y)$$

The treatment of bindings here is related to the formalization of lazy evaluation in Launchbury's (1993) natural semantics for lazy evaluation.

When the right-hand side of the binding is in normal form, it may substitute the dereferenced variable, as we see in the following calculation for the case where the normal form is in $\mathsf{N_0}$.

$$\mathcal{E}[\![\mathsf{let}\ x = N\ \mathsf{in}\ x{\downarrow}]\!]$$
$= \quad$ {Encoding of bindings and dereferencing (Property 3.11)}
$$\mathit{new} \succcurlyeq \lambda x.\mathit{put}\,x\,(\mathcal{E}[\![N]\!]) \succ (\uparrow x \succcurlyeq \lambda x.\mathit{get}\,x \succcurlyeq \lambda e.e \succcurlyeq \lambda v.\mathit{put}\,x\,(\uparrow v) \succ \uparrow v)$$
$\sqsubseteq \quad$ {Law of $\uparrow/\succcurlyeq$}
$$\mathit{new} \succcurlyeq \lambda x.\mathit{put}\,x\,(\mathcal{E}[\![N]\!]) \succ (\mathit{get}\,x \succcurlyeq \lambda e.e \succcurlyeq \lambda v.\mathit{put}\,x\,(\uparrow v) \succ \uparrow v)$$
$\sqsubseteq \quad$ {Associativity of $\succcurlyeq$ and $\succ$ (Property 3.36); Law of *put*/*get* (Property 3.35)}
$$\mathit{new} \succcurlyeq \lambda x.\uparrow\mathcal{E}[\![N]\!] \succcurlyeq \lambda e.e \succcurlyeq \lambda v.\mathit{put}\,x\,(\uparrow v) \succ \uparrow v$$
$\sqsubseteq \quad$ {Law of $\uparrow/\succcurlyeq$}
$$\mathit{new} \succcurlyeq \lambda x.\mathcal{E}[\![N]\!] \succcurlyeq \lambda v.\mathit{put}\,x\,(\uparrow v) \succ \uparrow v$$
$= \quad$ {Encoding of normal forms (Property 3.22); $\bullet N \in \mathsf{N_0}$}
$$\mathit{new} \succcurlyeq \lambda x.\uparrow N \succcurlyeq \lambda v.\mathit{put}\,x\,(\uparrow v) \succ \uparrow v$$
$\sqsubseteq \quad$ {Law of $\uparrow/\succcurlyeq$}
$$\mathit{new} \succcurlyeq \lambda x.\mathit{put}\,x\,(\uparrow N) \succ \uparrow N$$
$= \quad$ {Encoding of normal forms (Property 3.22); $N \in \mathsf{N_0}$}
$$\mathit{new} \succcurlyeq \lambda x.\mathit{put}\,x\,(\mathcal{E}[\![N]\!]) \succ \uparrow(\mathcal{E}[\![N]\!])$$
$= \quad$ {Encoding of bindings and dereferencing}
$$\mathcal{E}[\![\mathsf{let}\ x = N\ \mathsf{in}\ N]\!]$$

In the calculation, we simplified the context to a single ($\mathsf{let}\ \cdot\ \mathsf{in}\ \cdot$) binding to avoid clutter. The generalization uses the monotonicity of the transition contexts and the fact that $\mathit{get}\,x$ retrieves any association where $\Psi \Vdash x \doteq y$. The cases where the normal form is not from $\mathsf{N_0}$ run in a similar way; only in the case where the normal form is $\mathsf{0}$, the property of the linear logic constant $\top$ to accept any context is used.

3.50 DEREFERENCING A BINDING:

$$\xi : \Gamma, x = N; \Psi \,.\, \Lambda\langle y{\downarrow}\rangle \rightrightarrows \xi : \Gamma, x = N; \Psi \,.\, \Lambda\langle N\rangle \qquad \text{if } \Psi \Vdash (x \doteq y) \text{ and } N \in \mathsf{N}$$

If the scrutinee of a $(\mathsf{case} \cdot \mathsf{of} \cdot)$ construct is in normal form and there is a matching alternative, we can take the alternative, while substituting the arguments of the scrutinee for the variables in the case pattern.

$$\begin{array}{cl}
 & \mathcal{E}[\![\mathsf{case}\ a\ \overline{N}\ \mathsf{of}\ \overline{X}_1, a\ \overline{v} \to E, \overline{X}_2]\!] \\
= & \quad \{\text{Rule 3.12a and 3.12b}\} \\
 & \mathcal{E}[\![a\ \overline{N}]\!] \succcurlyeq \lambda w.\, \bigl(\mathop{\Lsh\!\!\Rsh} \mathcal{X}[\![\overline{X_1}]\!]\bigr) \mathbin{\Lsh\!\!\Rsh} \mathit{ask}\,(\lambda \overline{v}.a\,\overline{v} \doteq w)\,(\lambda \overline{v}.\mathcal{E}[\![E]\!]) \mathbin{\Lsh\!\!\Rsh} \bigl(\mathop{\Lsh\!\!\Rsh} \mathcal{X}[\![\overline{X_2}]\!]\bigr) \\
= & \quad \{\text{Encoding of normal forms (Property 3.22); } \bullet\overline{N} \subseteq \mathsf{N}_0\} \\
 & {\uparrow}(a\ \overline{N}) \succcurlyeq \lambda w.\, \bigl(\mathop{\Lsh\!\!\Rsh} \mathcal{X}[\![\overline{X_1}]\!]\bigr) \mathbin{\Lsh\!\!\Rsh} \mathit{ask}\,(\lambda \overline{v}.a\,\overline{v} \doteq w)\,(\lambda \overline{v}.\mathcal{E}[\![E]\!]) \mathbin{\Lsh\!\!\Rsh} \bigl(\mathop{\Lsh\!\!\Rsh} \mathcal{X}[\![\overline{X_2}]\!]\bigr) \\
\sqsubseteq & \quad \{\text{Law of } {\uparrow}/{\succcurlyeq}\} \\
 & \bigl(\mathop{\Lsh\!\!\Rsh} \mathcal{X}[\![\overline{X_1}]\!]\bigr) \mathbin{\Lsh\!\!\Rsh} \mathit{ask}\,(\lambda \overline{v}.a\,\overline{v} \doteq a\ \overline{N})\,(\lambda \overline{v}.\mathcal{E}[\![E]\!]) \mathbin{\Lsh\!\!\Rsh} \bigl(\mathop{\Lsh\!\!\Rsh} \mathcal{X}[\![\overline{X_2}]\!]\bigr) \\
\sqsubseteq & \quad \{\text{Repeatedly, the laws of } \mathbin{\Lsh\!\!\Rsh} \text{ (Property 3.39)}\} \\
 & \mathit{ask}\,(\lambda \overline{v}.a\,\overline{v} \doteq a\ \overline{N})\,(\lambda \overline{v}.\mathcal{E}[\![E]\!]) \\
\sqsubseteq & \quad \{\text{Law of } \mathit{ask} \text{ (Property 3.34), for any } \Psi\} \\
 & [\overline{N}/\overline{v}]\mathcal{E}[\![E]\!] \\
= & \quad \{\text{Due to the encoding of normal forms (Property 3.22)}\} \\
 & \mathcal{E}[\![[\overline{N}/\overline{v}]E]\!]
\end{array}$$

Together with the monotonicity of transition contexts, we have derived the following rule.

3.51 CASE SELECTION:

$$\xi : \Gamma; \Psi \,.\, \Lambda\langle \mathsf{case}\ a\ \overline{N}\ \mathsf{of}\ \overline{X}_1, a\ \overline{v} \to E, \overline{X_2}\rangle \rightrightarrows \xi : \Gamma; \Psi \,.\, \Lambda\langle [\overline{N}/\overline{v}]E\rangle \qquad \text{if } \overline{N} \subseteq \mathsf{N}_0$$

Agents

By Lemma 3.42, we may pull newly introduced unbound variables out of the reduction context and incorporate them into the variable list ξ of the transition context.

3.52 NEW UNBOUND VARIABLE:

$$\xi : \Gamma; \Psi \,.\, \Lambda\langle \nu x.E\rangle \rightrightarrows \xi, x : \Gamma; \Psi \,.\, \Lambda\langle E\rangle \qquad \text{if } x \text{ fresh}$$

Similarly, normal constraints—i.e., equational constraints over names—are moved out of the reduction context into the constraint store of the transition context.

3.53 ENRICHING THE CONSTRAINT STORE:

$$\xi : \Gamma; \Psi \,.\, \Lambda\langle x \doteq y\rangle \rightrightarrows \xi : \Gamma; \Psi, x \doteq y \,.\, \Lambda\langle \mathsf{1}\rangle \qquad \text{if } x, y \in \mathsf{V}$$

As explained by Miller (1996), deduction in Forum represents potential parallelism by requiring permutability for the application of some inference rules. This is certainly sufficient in a logic treatment of parallelism, but in the operational semantics of $\mathcal{D}$, the goal is to make parallel reduction explicit. In other words, reducing a compound agent $E_1 \parallel E_2$, we like to be able to reduce both E_1 and E_2 in one reduction step. The difficulty carried by such a treatment of parallel composition is that reduction of the subagents E_1 and E_2 may have consequences for the transition context, which is shared by them. In particular, both agents may, by virtue of demanding a binding (Rule 3.49), change the binding environment. Consequently, we have to require that both agents only change disjoint portions of the binding environment. The treatment of the unbound variables and the constraint store is less difficult as the agents can only add to these parts and never change existing members of these sets.

The following rule handles the simultaneous rewriting of bindings by partitioning the binding environment into two disjoint sets. Each of the two subagents gets one set, which it may modify freely. The two (potentially modified) sets, which result from the reductions in the premises, are combined into one binding environment for the compound agent. Due to the convention that the variables bound by the binding environment are also added to the variables in ξ (Definition 3.44), we are still handling closed expressions in the premises. Reduction steps that are following may partition the environment differently. So overall, the two agents operate concurrently on the composite binding environment, but noninterference is enforced within a single reduction step.

3.54 PARALLEL TRANSITIONS:

$$\frac{\xi : \Gamma_1; \Psi \centerdot \Lambda\langle E_1\rangle \rightrightarrows \xi_1 : \Gamma_1'; \Psi_1 \centerdot \Lambda\langle E_1'\rangle \qquad \xi : \Gamma_2; \Psi \centerdot \Lambda\langle E_2\rangle \rightrightarrows \xi_2 : \Gamma_2'; \Psi_2 \centerdot \Lambda\langle E_2'\rangle}{\xi : \Gamma_1, \Gamma_2; \Psi \centerdot \Lambda\langle E_1 \parallel E_2\rangle \rightrightarrows \xi_1, \xi_2 : \Gamma_1', \Gamma_2'; \Psi_1, \Psi_2 \centerdot \Lambda\langle E_1' \parallel E_2'\rangle}$$

The soundness of this parallel transition rule with respect to the logic semantics follows from the fact that we can sequentialize the premises in any order and let them operate on the full transition context of the compound context. Consider the following calculation, which is valid due to the transitivity of $\rightrightarrows$ (Lemma 3.27).

$$\begin{array}{ll}
 & \xi : \Gamma_1, \Gamma_2; \Psi \centerdot \Lambda\langle E_1 \parallel E_2\rangle \\
\rightrightarrows & \quad \{\text{First premise; encoding of parallel composition (3.17a);} \\
 & \quad\ \ \text{monotonicity of } \upuparrows \text{ (Lemma 3.40)}\} \\
 & \xi_1, \xi : \Gamma_1', \Gamma_2; \Psi_1, \Psi \centerdot \Lambda\langle E_1' \parallel E_2\rangle \\
\rightrightarrows & \quad \{\text{Second premise; encoding of parallel composition (3.17a);} \\
 & \quad\ \ \text{monotonicity of } \upuparrows \text{ (Lemma 3.40)}\} \\
 & \xi_1, \xi_2 : \Gamma_1', \Gamma_2'; \Psi_1, \Psi_2 \centerdot \Lambda\langle E_1' \parallel E_2'\rangle
\end{array}$$

(Remember that ξ and Ψ are sets and that the transitions merely add elements.)

An immediate consequence of the synchronization rule (Property 3.30) and the monotonicity of the transition contexts is that a sequential composition where the precondition succeeds proceeds with its second argument.

3.55 SUCCESSFUL SYNCHRONIZATION:

$$\xi : \Gamma; \Psi \centerdot \Lambda\langle \mathbf{1} \Rightarrow E\rangle \rightrightarrows \xi : \Gamma; \Psi \centerdot \Lambda\langle E\rangle$$

The reduction of equations over terms (not names) has to deal with two cases corresponding to the axioms (decompose) and (mismatch) of *tell* (Definition 3.15): First, when the outermost constructor matches, the arguments are recursively decomposed; and second, if they do not match, the computation fails.

3.56 TERM DECOMPOSITION:

$$\xi : \Gamma; \Psi \mathbin{\scriptscriptstyle\blacksquare} \Lambda\langle a\,\overline{N} \doteq a\,\overline{N'}\rangle \rightrightarrows \xi : \Gamma; \Psi \mathbin{\scriptscriptstyle\blacksquare} \Lambda\langle (N_1 \doteq N'_1) \parallel \cdots \parallel (N_m \doteq N'_m)\rangle \qquad \text{if } \overline{N}, \overline{N'} \subseteq \mathsf{N}$$

$$\xi : \Gamma; \Psi \mathbin{\scriptscriptstyle\blacksquare} \Lambda\langle a\,\overline{N} \doteq b\,\overline{N'}\rangle \rightrightarrows \xi : \Gamma; \Psi \mathbin{\scriptscriptstyle\blacksquare} \Lambda\langle \mathsf{0}\rangle \qquad \text{if } a \neq b \text{ and } \overline{N}, \overline{N'} \subseteq \mathsf{N}$$

Two alternative choice agents may reduce simultaneously; similar as in the case of the parallel composition of two agents. (The justification for this rule corresponds to that of parallel composition.)

3.57 PARALLEL ALTERNATIVES:

$$\frac{\xi : \Gamma_1; \Psi \mathbin{\scriptscriptstyle\blacksquare} \Lambda\langle S_1\rangle \rightrightarrows \xi_1 : \Gamma'_1; \Psi_1 \mathbin{\scriptscriptstyle\blacksquare} \Lambda\langle S'_1\rangle \qquad \xi : \Gamma_2; \Psi \mathbin{\scriptscriptstyle\blacksquare} \Lambda\langle S_2\rangle \rightrightarrows \xi_2 : \Gamma'_2; \Psi_2 \mathbin{\scriptscriptstyle\blacksquare} \Lambda\langle S'_2\rangle}{\xi : \Gamma_1, \Gamma_2; \Psi \mathbin{\scriptscriptstyle\blacksquare} \Lambda\langle S_1 \mathbin{[\,]} S_2\rangle \rightrightarrows \xi_1, \xi_2 : \Gamma'_1, \Gamma'_2; \Psi_1 \mathbin{\dot{\wedge}} \Psi_2 \mathbin{\scriptscriptstyle\blacksquare} \Lambda\langle S'_1 \mathbin{[\,]} S'_2\rangle}$$

The operational intention is that the right-hand sides in the guards of alternative ask expressions can reduce simultaneously. And as soon as one guard is satisfied, the corresponding ask expression is taken—while the others are discarded. This situation is formalized next. A derivation similar to the one that we have used for ($\mathsf{case} \cdot \mathsf{of} \cdot$) above, leads us to the following transition rule. In contrast to ($\mathsf{case} \cdot \mathsf{of} \cdot$), this time the constraint store Ψ is important.

3.58 SUCCESSFUL ASK:

$$\xi : \Gamma; \Psi \mathbin{\scriptscriptstyle\blacksquare} \Lambda\langle \overline{v} : \overline{P_I} \doteq \overline{N_I} \to E' \mathbin{[\,]} S\rangle \rightrightarrows \xi : \Gamma; \Psi \mathbin{\scriptscriptstyle\blacksquare} \Lambda\langle \Theta E\rangle \qquad \begin{array}{l} \text{if } \overline{N_I}, \overline{N'} \subseteq \mathsf{N_0}, \Theta = [\overline{N'}/\overline{v}], \\ \text{for all } i \in I, \Psi \Vdash \Theta(P_i \doteq N_i), \\ \text{and } \mathrm{Fr}\,\overline{N'} \subseteq \xi \end{array}$$

In the last transition rule, we restate the Forum axiom (Definition 3.16) realizing the consistency check of bindings that have been aliased by equational constraints.

3.59 CONSISTENCY CHECK:

$$\xi : \Gamma, x_1 = E_1, x_2 = E_2; \Psi \mathbin{\scriptscriptstyle\blacksquare} E \rightrightarrows \xi : \Gamma, x_1 = E_1; \Psi \mathbin{\scriptscriptstyle\blacksquare} (x_1{\downarrow} \doteq E_2{\downarrow}) \parallel E \qquad \text{if } \Psi \Vdash (x_1 \doteq x_2)$$

When two bindings are aliased by the constraint store, it is verified that their right-hand sides are compatible. Furthermore, one of the bindings is removed (to ensure that the consistency check for these two bindings is not repeated).

3.5.4 The Operational Semantics of $\mathcal{D}$ Expressions

We characterize agents by success or failure.

3.60 DEFINITION (SUCCESS AND FAILURE OF AGENTS)
We call a $\mathcal{D}$ expressions $E \in \mathsf{E}$ of type o *successful* (*failing*) if, for some ξ, Γ, and Ψ,

$$\{\} : \{\}; \{\} \mathbin{\scriptscriptstyle\blacksquare} E \rightrightarrows \xi : \Gamma; \Psi \mathbin{\scriptscriptstyle\blacksquare} \mathsf{1} \qquad (\{\} : \{\}; \{\} \mathbin{\scriptscriptstyle\blacksquare} E \rightrightarrows \xi : \Gamma; \Psi \mathbin{\scriptscriptstyle\blacksquare} \mathsf{0})$$ ∎

This characterization may seem weak, because no statement is made about the resultant constraint store or binding environment; and furthermore, only expressions of type o are considered. But in fact, the characterization allows a wide spectrum of observations for various

$\mathcal{D}$ expressions. Consider, for example, an expression E that results in a list (this fact is obvious from its type). If the agent

$$\mathsf{let}\ y = E\ \mathsf{in}\ y \doteq \mathit{Nil}\ \langle\rangle$$

is successful, we know that the value of E is the empty list. Due to the discriminative power of $\doteq$, the test on success or failure allows to characterize $\mathcal{D}$ expressions sufficiently.

The operational semantics of $\mathcal{D}$ is now defined by the transition rules derived previously.

3.61 DEFINITION (OPERATIONAL SEMANTICS OF $\mathcal{D}$)
The *one step transition relation* $\overset{\cdot}{\rightrightarrows}$ on transition contexts is defined to be the smallest relation that satisfies the Rules 3.46 to 3.59, where we allow that in Rules 3.54 and 3.57 only one agent takes a transition step. The reflexive and transitive closure of $\overset{\cdot}{\rightrightarrows}$ is denoted $\overset{\cdot}{\rightrightarrows\!\!\!\!\rightrightarrows}$. ∎

The one step transition relation is by definition sound with respect to the transition relation $\rightrightarrows$ induced by linear implication on the encodings of $\mathcal{D}$ expressions.

3.62 THEOREM (SOUNDNESS OF THE OPERATIONAL WRT. THE LOGIC SEMANTICS)
If, for two expressions $E, E' \in \mathsf{E}$ and a transition context $\langle \xi, \Gamma, \Psi, \Lambda \rangle$, we have

$$\xi : \Gamma; \Psi \mathbin{\scriptscriptstyle\blacksquare} \Lambda\langle E \rangle \overset{\cdot}{\rightrightarrows} \xi : \Gamma; \Psi \mathbin{\scriptscriptstyle\blacksquare} \Lambda\langle E' \rangle,$$

then also

$$\xi : \Gamma; \Psi \mathbin{\scriptscriptstyle\blacksquare} \Lambda\langle E \rangle \rightrightarrows \xi : \Gamma; \Psi \mathbin{\scriptscriptstyle\blacksquare} \Lambda\langle E' \rangle.$$

∎

As said, no proof is necessary, because $\overset{\cdot}{\rightrightarrows}$ is by construction a consequence of $\rightrightarrows$.

3.63 LEMMA
If $E \in \mathsf{E}$ is of type o and

$$\{\} : \{\}; \{\} \mathbin{\scriptscriptstyle\blacksquare} E \overset{\cdot}{\rightrightarrows\!\!\!\!\rightrightarrows} \xi : \Gamma; \Psi \mathbin{\scriptscriptstyle\blacksquare} \mathsf{1} \qquad (\{\} : \{\}; \{\} \mathbin{\scriptscriptstyle\blacksquare} E \overset{\cdot}{\rightrightarrows\!\!\!\!\rightrightarrows} \xi : \Gamma; \Psi \mathbin{\scriptscriptstyle\blacksquare} \mathsf{0}),$$

then E is successful (failing). ∎

PROOF.
Induction on the length of the reduction proves by virtue of Theorem 3.62 and Definition 3.60 this lemma. ∎

3.5.5 Completeness

I conjecture that the operational semantics as defined above is complete with respect to the linear logic semantics of $\mathcal{D}$—i.e., whenever an agent is successful according to the logic semantics, it can be reduced to $\mathsf{1}$ by the transition rules of the operational semantics.

3.64 CONJECTURE (COMPLETENESS OF OPERATIONAL WRT. THE LOGIC SEMANTICS)
Given an expression $E \in E$ of type o, if E is successful (failing), $\{\} : \{\}; \{\} \mathbin{\scriptscriptstyle\blacksquare} E$ can be reduced to $\xi : \Gamma; \Psi \mathbin{\scriptscriptstyle\blacksquare} \mathsf{1}$ ($\xi : \Gamma; \Psi \mathbin{\scriptscriptstyle\blacksquare} \mathsf{0}$), for some ξ, Γ, and Ψ by $\overset{\cdot}{\rightrightarrows\!\!\!\!\rightrightarrows}$. ∎

3.6 Discussion

We previously proposed the GOFFIN core (Chakravarty, Guo, Köhler & Lock 1997) as the kernel language and basis for a semantic framework for languages like GOFFIN (see Section 2.2). In fact, the starting point for the research documented in this chapter was the GOFFIN core and the dissatisfaction with the gap that exists between its logic and its operational semantics. The logic semantics for the GOFFIN core is built on Church's (1940) Simple Theory of Types, which is not able to capture concepts like committed-choice indeterminism and the strategy of deferred evaluation. With respect to this previous work, the present dissertation makes two central contributions. First, the presented linear logic framework can capture concepts like committed-choice indeterminism and the strategy of deferred evaluation, and the logic semantics induces an appropriate operational semantics. Second, this chapter demonstrated that the essential concepts of languages like GOFFIN can be integrated orthogonally into the typed lambda calculus. Therefore, $\mathcal{D}$, in comparison to the GOFFIN core, eases the adaption of results from purely functional programming. This is, for example, useful for the development of an execution model, as we shall see in the next chapter.

The choice of using linear logic as the basic vehicle for the definition of the semantics was mainly motivated by the desire to follow the tradition of logic programming or, more generally, declarative programming. An alternative development could start with the operational aspects and formalize them in a process calculus, such as Miller's (1992) π-calculus or in one of the subsequent developments, such as Sangiorgi's (1993) HOπ or Odersky's (1995) Applied π. Starting at the operational level, the abstract semantics can be derived using techniques like bisimulation.

3.6.1 Related Work

Numerous approaches to the formal treatment of concurrency and parallelism exist, and many attempts were made to design declarative languages that are suitable for parallel programming. But there are few approaches to parallel declarative programming that are founded on a strict separation of the concerns of co-ordination and computation. There is the work on co-ordinating functional programs with skeletons, started by Cole (1989) and, for example, continued by Darlington et al. (1993), Danelutto & Pelagatti (1993), and Südholt (1994). The distinguished feature of the skeleton approach is that the number of skeletons (i.e., co-ordinating forms) is fixed and cannot be extended within a program. In particular, most work on skeletons focuses on co-ordinating forms that support only data parallel programming. However, it should be noted that the additional flexibility provided by $\mathcal{D}$, namely the ability for user-defined co-ordinating forms, has a price—distribution of and communication between agents is more dynamic, and thus, it is substantially harder implement efficiently.

In the work on the language Eden (Breitinger, Loogen & Ortega-Mallén 1995, Breitinger, Loogen, Ortega-Mallén & Peña 1996), the functional language Haskell (Hudak et al. 1992) is extended by a notion of processes that co-ordinate functional computations, but recent development (Breitinger, Loogen, Ortega-Mallén & Peña 1996) has harmed the declarativity of the approach by allowing the creation of processes from within purely functional computations, which compromises referential transparency. The main differences between the approach taken in Eden compared to that taken in $\mathcal{D}$ are the following two: first, an Eden process is closer to the intuitive, operational notion of a process than an agent in $\mathcal{D}$; and second, the formal, and so, the declarative fundament for processes or agents is better developed in $\mathcal{D}$ than in Eden.

In other words, Eden pays for a stronger operationally oriented process concept, which has its advantage in the implementation, with a less developed abstract semantics.

Section 3.1 already commented on the relation of $\mathcal{D}$ to cc (Saraswat 1993), HLcc (Saraswat & Lincoln 1992), AKL (Janson 1994), and Oz (Smolka 1995). Although these programming models support concurrent computation, they are not primarily intended for programming parallel computers and lack the distinction between co-ordination and computation. Concurrent constraint programming as defined initially by Saraswat (1993) is purely relational and does not support higher-order programming. Later Saraswat & Lincoln (1992) extended the framework to a higher-order setting based on linear logic. The resulting HLcc framework is similar to the process combinators of Section 3.4.1, but quite different from $\mathcal{D}$. In particular, Saraswat & Lincoln (1992) show that functional computations can be embedded within HLcc, but realize this via an encoding, and thus, HLcc does not exhibit the clean separation between co-ordination and computation by means of relations and functions, which is central to $\mathcal{D}$. Both AKL (Janson 1994) and Oz (Smolka 1995) are untyped. Their integration of functional computations is based on an encoding of functions by relations and does not provide a clean separation between functional and relational computations. The resulting integration of functions into their relational framework is questionable as it does not prevent the use of side-effecting computations in functions, e.g., the cells of Oz. The semantics of AKL was partially characterized by a mapping into classical logic, with the previously discussed problems that this approach carries; furthermore, Janson (1994) provides an operational semantics. For Oz, there is until now only an operational semantics.

The functional language Id (Nikhil 1990) provides I-structures, which are effectively unbound variables (Arvind, Nikhil & Pingali 1989). They can be instantiated by a single assignment statement; instead of executing a consistency check, Id programs immediately abort when an I-structure is assigned twice. Id does not separate purely functional computations from co-ordinating activities, which use I-structures and single assignments. Consequently, referential transparency is compromised. Committed-choice indeterminism is introduced into Id by means of M-structures, a form of updatable data structure (Barth, Nikhil & Arvind 1991). While computing with logical variables (or I-structures) and single assignments can be interpreted as a monotonic refinement of a partial data structure, namely the constraint store, the use of M-structures spoils monotonicity. As a result, reasoning about programs becomes much harder. For Id programs using I-structures, but no M-structures, Jagadeesan & Pingali (1992) formulated a denotational semantics based on closure-operators.

The language HACL (Kobayashi & Yonezawa 1994) is to some extend a mixture of linear logic processes (Chakravarty, Guo & Köhler 1997) and $\mathcal{D}$. It does have a functional and a propositional part, but as all communication is performed by means of linear messages and not logical variables, the functional part can only depend on messages when this is explicitly coded using the propositional part—in other words, the ask operation, which is implicit in pattern-matching on logical variables in $\mathcal{D}$, must be made explicitly in HACL. HACL is a special syntax for a fragment of higher-order linear logic.

Mück, Streicher & Lock (1994) introduced a generic language that is an extension of the lambda calculus that can be parameterized by an arbitrary constraint system. A fundamental difference between this language and $\mathcal{D}$ is that the former is untyped, and thus, allows indeterminism in functional computations. Therefore, it cannot be used directly as a semantics framework for languages like GOFFIN. Their work does not consider a concurrent execution of constraint functional programs, and when instantiated to equational constraints, it can only handle first-order equational constraints. These comments apply also to Crossley, Mandel &

Wirsing's (1993) constrained lambda calculus.

3.6.2 Future Work

It is possible to argue that the present chapter provides just a basic fundament of $\mathcal{D}$. The two semantics are sufficient to justify the claim that $\mathcal{D}$ provides a declarative way to specify a wide range of parallel behaviour, but a detailed investigation of the relation of $\mathcal{D}$ to other calculi that have the same goal would definitely be worthwhile. Furthermore, a denotational and, especially, an algebraic characterization are desirable. One possible approach towards the former goal could be the instantiation of a denotational characterization of linear logic for the special case of linear logic processes **P** as defined in Figure 3.1. A possible candidate is the phase spaces semantics (Girard 1995)—it was successful in the case of a logic programming language derived from a subset of linear logic (Volpe 1994). An algebraic characterization of $\mathcal{D}$ is of special importance as it allows to calculate with programs (Hoare 1994). There are basically two routes to derive an algebraic semantics, one top-down and the other bottom-up. The top-down route derives algebraic laws for $\mathcal{D}$ from the linear logic semantics. This is possible on the basis of the Forum proof system, in a similar way as we derived the transition rules; or an algebraic characterization of linear logic (Hesselink 1990) could be used to reason directly about the encodings—the second approach is likely to be more elegant. The bottom-up route would calculate the algebraic laws from the operational semantics, possibly using a technique like barbed bisimulation; but, this would require to prove the completeness of the transition system first.

The completeness of the transition system as formalized in Conjecture 3.64 is, of course, important in itself. It ensures a close correspondence between the logic and the operational semantics that allows to reason about $\mathcal{D}$ programs in any of the two semantics with the assurance that the result is also valid in the other one.

Apart from the further study of $\mathcal{D}$ in the form presented here, several extensions to the basic language should be explored. The above presentation is restricted to equational constraints, which are sufficient for many applications, comparatively easy to handle semantically, and straight forward to implement even in a distributed setting—as we will see in the following chapters. Nevertheless, other constraint systems are interesting—a taste of the choices was, for example, given by Saraswat (1993)—and their integration into the formalism should be studied. Most interesting would be an extension to a generic framework that can be instantiated with a given constraint system, just as this is the case in CLP and cc, but the interaction with the functional component of $\mathcal{D}$ and the lazy semantics makes this a difficult undertaking. Equational constraints are particularly simple to handle, because, as we have seen, it suffices to alias names, no new bindings have to be generated.

A further interesting extension is the inclusion of general disjunctions—i.e., in an operational sense, search. In linear logic, we can model this behaviour using the connective $\&$. In the transition system, it requires the management of alternative constraint stores, which are representing alternative solutions for a single agent. Alternative constraint stores require a considerably more complex operational semantics. We will return to this point in Chapter 6.

Two other interesting extensions to $\mathcal{D}$ are the introduction of deep guards and of coordinating subsystems. Deep guards require the generalization of the constraints in guarded expressions to arbitrary $\mathcal{D}$ agents, which could be treated in a similar way as constraints in guards of AKL (Janson 1994). But the encoding of such deep guards into linear logic processes is not clear—partially reflecting a number of semantic uncertainties in the definition of deep guards. Still, they are a worthwhile topic for future work, and the higher-order features of Forum (linear logic) may help in a formal treatment.

Co-ordinating subsystems may provide a way to encapsulate complete co-ordination systems into a functional expression. The idea is a to define a construct that is related to the description operator, the connective ι, of higher-order logic (Andrews 1986), which essentially is the inverse of curried equality. To understand this, consider that the function $\lambda x.(x = c)$ for some fixed c has type $\imath \rightarrow \mathrm{o}$ ($\imath$ is the type of individuals and o is the Boolean type) and is the characteristic function of c—it is true exactly if its argument is c. Now, ι is characterized by the equality $(\iota\ (\lambda x.(x = c))) = c$, and the type of ι is $(\imath \rightarrow \mathrm{o}) \rightarrow \imath$. Integrating a connective similar to ι into $\mathcal{D}$ may provide an elegant way to extract the solution of an agent that has a *unique* solution and to integrate this solution into a functional computation. A problem arises when committed-choice indeterminism occurs in such subsystems. Repeated evaluation of the same expression may, then, yield each time a different, but still only one solution, which destroys referential transparency. Maybe the approach of Marriott & Odersky (1995) to committed-choice indeterminism can help here.

Chapter 4

A MULTI-THREADED EXECUTION MODEL

THE operational semantics of $\mathcal{D}$ presented in the previous chapter is a first step towards an implementation of declarative parallelism. The next step is an execution model that is more machine-oriented—i.e., an abstract machine model. Such a model consists of a machine architecture, a machine language, and a transition system that defines the behaviour of the abstract machine. The transition system specifies the execution of the various constructs of the machine language in terms of the machine architecture.

The machine language introduced below is called simplified KCODE and is strongly related to $\mathcal{D}$; but it is not exactly $\mathcal{D}$. The notation of $\mathcal{D}$ is optimized for mathematical arguments, whereas KCODE is designed for a straightforward generation of efficient code by a compiler. But more importantly, KCODE includes an additional construct, the `letpar` construct, which enables the use of compiler-controlled multi-threading. However, we establish a formal link between both languages by defining the semantics of KCODE by means of a mapping from KCODE to $\mathcal{D}$.

The present chapter starts with the basic ideas that guided the design of the abstract machine, and then, briefly introduces the fundamental concepts of compiled graph-reduction as they are realized in the Spineless Tagless G-machine (STGM). More details on compiled graph-reduction are contained in the textbook of Peyton Jones (1987); he also provides a presentation of the STGM (Peyton Jones 1992). After these introductory topics, the present chapter continues with the definition of the syntax and semantics of the machine language KCODE. Then, it discusses the operational interpretation of the various language features and discusses an optimization scheme for latency hiding. Finally, it introduces the machine architecture and presents a transition system that specifies the execution of KCODE on the basis of the given machine architecture. A discussion of experimental results obtained with a prototype, together with related and future work, closes the chapter.

4.1 Basic Ideas

The abstract machine presented below is called HIPAR and is based on the STGM, which was originally introduced by Peyton Jones & Salkild (1988) and was, subsequently, revised by Peyton Jones (1992). The STGM implements compiled graph-reduction for higher-order, lazy functional languages (Peyton Jones 1987); it is at the heart of the Glasgow Haskell Compiler (Peyton Jones et al. 1993). The following presentation extends the design of the STGM by two main features: it implements co-ordination of parallelism by relations (including unbound variables and equational constraints) and it uses multi-threading for the efficient implementation of graph reduction on distributed-memory multi-processors. The former extension draws from the work on the JUMP machine (Chakravarty & Lock 1991, Lock 1992, Chakravarty & Lock 1997), and the latter develops from my earlier work on the STGM$_{\mathrm{MT}}$ (Chakravarty 1996), which extends the vanilla STGM with explicit support for multi-threading. Before following sections introduce the design of HiPar in detail, the reminder of this section explains the basic ideas that led to the design.

4.1.1 Relational Programming and Compiled Graph-reduction

There exist many approaches to the implementation of programming languages that are combining features of relational and functional programming—a survey was written by Hanus (1994). Usually, an abstract machine invented originally for the implementation of a relational or a functional language is extended to cover features of the other paradigm. The advantages of starting from compiled graph-reduction are discussed elsewhere (Lock 1992, Mück 1992, Loogen 1993, Chakravarty & Lock 1997). The advantages of graph reduction are even more clearly cut in the implementation of a language that uses deferred evaluation, because the dominating amount of computations is certainly of the kind for which compiled graph-reduction is optimized.

Generally, three main language features that are not present in a purely functional language require special attention in an integrated implementation: unbound variables, equational constraints, and search. We do not discuss the implementation of search, but have a look at unbound variables and equational constraints.

Unbound Variables

Unbound variables (or existentially quantified variables) are not associated with a value when they are introduced. Under the strategy of deferred evaluation, a functional computation that depends on an unbound variable is delayed until the variable is bound. In operational terms, the functional computation synchronizes on the binding. This implies a form of concurrent evaluation. When a computation suspends due to a dependence on a yet unbound variable, another ready-to-run computation has to be scheduled. When eventually a concurrent computation instantiates the previously unbound variable, the suspended computation gets ready-to-run.

Operationally, this implies not only the management of concurrent tasks by a scheduling mechanism, but also frequent checks of the accessed data structures for the occurrence of unbound variables. The work on the JUMP-machine (Chakravarty & Lock 1997) demonstrated that it is possible to extend the closure-oriented architecture of the STGM such that it performs these additional checks on-the-fly without explicit checking code. This extension uses a special form of closures to represent unbound variables that—in the case of deferred evaluation—suspend any task that is attempting to evaluate such a closure.

Equational Constraints

Equational constraints restrict the domain of unbound variables. Hence, they reactivate computations that are deferred due to a dependence on an unbound variable. Operationally, they can be realized by unification, which *instantiates* (or *binds*) unbound variables. Such instantiation entails the reactivation of computations that formerly suspended due to a dependence on an unbound variable. But the unification code that realizes equational constraints has to be able to check whether a given closure corresponds to an unbound variable without suspending itself. Again, we can use the technology of the JUMP-machine: We provide closures with more than one code component. Therefore, closures have the standard evaluation code and, in addition, a second code component that is used in the course of unification—details are given later.

4.1.2 Why Multi-threading?

In addition to the support for relational computations, we extend the STGM with an explicit notion of multi-threading. This is important to efficiently implement the frequent, highly irregular remote access operations that occur in an implementation of graph reduction on distributed-memory computers.

Let us assume a distributed implementation of the original STGM where on each of the many processor elements (PEs) a number of *tasks* are running interleaved. When the active task of one of these processors attempts the evaluation of the expression $(a + b) * (c + d)$, it may start by accessing the value of the variable a. If the value of a is stored in the memory of another PE, a remote access is triggered by entering a FETCHME closure (see Peyton Jones 1992). When such a long-latency remote-access stalls the progress of a task, it is common practice to execute some other, ready task instead. Such a task switch implies an expensive context switch. The current values of processor registers must be saved and other values must be loaded. Furthermore, the data in the processor's cache is probably useless for the computation performed by the newly selected task.

But let us return to the above example of evaluating $(a + b) * (c + d)$. Observe that the evaluation of the subexpression $c + d$ is (i) independent from the result of the remote access induced by a and (ii) can be enabled with less context switching costs than continuing with an arbitrary ready task. The context of the evaluation of $c + d$ is similar to the context of $a + b$; in particular, both computations are part of the same function activation.

Across function boundaries, a similar observation can be made. The result of a function activation may depend on a long latency operation, while there are no further computations that are independent from the long latency operation in this function activation. In this case, it is usually less costly to switch to the execution of some independent computation in the *calling* function than to activate a completely unrelated task—consider the processor's cache.

Overall, the general rule for scheduling is as follows: A computation that is used to cover the delay of a long latency operation should have a context as close to the current context as possible. Due to locality, which computations usually exhibit, it is a good heuristic to choose a computation that is independent, but would have been executed as shortly after the current computation as possible.

The potential for fine-grain parallelism that is inherent in functional programs can be used to mechanically partition a program into *threads* realizing independent computations. Within a function activation, long latency operations can then be covered by switching to any of the ready threads. The abstract machine language KCODE, which is described below, provides the `letpar` construct, which explicitly displays independent computations on the level of the abstract machine language. We can implement these independent computations by light-weight threads (Culler et al. 1993, Engler et al. 1993, Blumofe, Joerg, Kuszmaul, Leiserson, Randall & Zhou 1995), which are generated by the compiler. There is evidence that the use of compiler-controlled multi-threading minimizes the thread switching time, while simultaneously

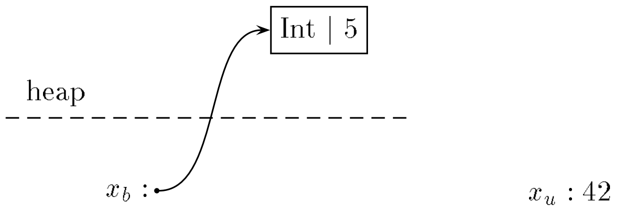

Figure 4.1 A boxed integer x_b versus an unboxed integer x_u. Boxed values are pointers to heap-located data, whereas unboxed values contain the data as is.

allowing the exploitation of the properties of the memory hierarchy (Culler et al. 1993, Hum et al. 1995, Culler, Goldstein, Schauser & von Eicken 1995).

4.1.3 Boxed and Unboxed Values

Implementations of declarative languages usually build their data structures in a memory area called the *heap* and realize user-level data structures by pointers to heap structures. Placement of data in the heap and retrieval from the heap can induce significant performance penalties when small data structures are manipulated, such as 32-bit integer values or fixed-precision floating-point numbers. Many researchers have proposed schemes that avoid these costs by keeping values out of the heap. Instead, these schemes store simple data structures within machine registers or on the runtime stack (for example, see the work of Peyton Jones 1992, Peyton Jones & Launchbury 1991, Leroy 1992, Hall et al. 1994, Henglein & Jørgensen 1994, Miniussi, Metzemakers, Sherman & Strandh 1994).

We call a datum *boxed* when it is stored in the heap (usually together with some additional administrative information); conversely, we call a datum *unboxed* when it is manipulated in its raw form. Figure 4.1 visualizes the distinction between boxed and unboxed values. Peyton Jones & Launchbury (1991) proposed a scheme were boxed and unboxed values are discriminated by their type and can be manipulated explicitly in a purely functional intermediate language. This scheme is realized in the STGM and is used, in a modified form, in the work described here.

We extend the distinction between name types and term types as introduced by $\mathcal{D}$ to the machine model. We identify name types with boxed types and term types with unboxed types. This is a natural correspondence, as names are the abstract equivalent of pointers and terms represent data without further indirections. This correspondence goes further.

- In $\mathcal{D}$, names have been introduced by bindings; in the abstract machine, bindings create closures (i.e., boxed values).

- In $\mathcal{D}$, dereferencing gets the term of a binding; in the abstract machine, evaluation of a closure yields the corresponding unboxed value.

- In $\mathcal{D}$, equational constraints create aliases between names; in the abstract machine, equational constraints identify closures that are representing unbound variables or bindings.

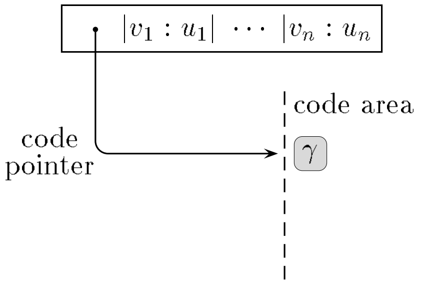

Figure 4.2 The general form of a closure. When execution enters such a closure, the referenced code γ is executed. It computes the value that is represented by the closure in dependence on the valuation u_i for its free variables v_i. This is expressed by an application of the form $(\gamma\ [v_1 \mapsto u_1, \ldots, v_n \mapsto u_n])$.

In the STGM, some data structures are available in both a boxed and an unboxed form, whereas some are always boxed. In contrast, the abstract machine HIPAR provides all first-order data structures in both unboxed and boxed form. This is a natural generalization of the scheme that is employed in the STGM, especially because the STGM internally realizes some structures in an unboxed form, where this is not explicit in the machine code (see Peyton Jones 1992, Section 9.4.4).

4.2 Concepts of Spineless Tagless Graph-reduction

Before we start with the details of KCODE, we have a look at the essential concepts of compiled graph-reduction as they are implemented in the STGM.

4.2.1 Tagless Closures

An important concept of the STGM is the uniform representation of all boxed values as *closures*, where a closure is a pair of code and *environment*. The environment of a closure is a valuation of the free variables of the code and may consist of boxed and unboxed data. The general structure of a closure is depicted in Figure 4.2, where the u_i constitute the environment. The code of a closure, on execution, takes some arguments from an argument stack (it may take none) and calculates a result in dependence of the values stored in its environment. We call the execution of the code of a closure *entering the closure*.

Depending on its code, a closure represents a data structure, an unevaluated expression, or a function. Consider the closure in Figure 4.3(i). When it is entered, it evaluates to the integer value stored in its environment. Such a closure realizes the boxed representation of an integer value. Similarly, the closure in Figure 4.3(ii) implements a data term consisting of an n-ary constructor c and its arguments.[1]

An implementation of a lazy language has to suspend the evaluation of function arguments until they are needed. So, it must maintain unevaluated expressions, which we call *thunks*. A closure can represent a thunk naturally by placing the valuation of the expression's free

---

[1] We represent the code of closures slightly different from the original STGM. For now just note that $(\mathit{Eval}\ e\ \rho)$ means to evaluate e in the context of an environment ρ; details are provided later.

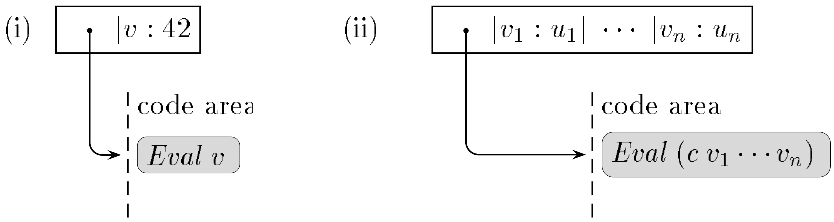

Figure 4.3 Closures representing (i) a boxed integer and (ii) a boxed data term.

variables in the environment and by using the code that evaluates the thunk as the code of the closure. This uniform representation of thunks and data structures has an interesting benefit: A function that needs the value of a non-strict argument just enters the associated closure. If it is a thunk, the suspended expression is evaluated and the result is returned in the same way as if the closure had already represented a normal form. This scheme avoids the explicit testing of tags, which is necessary in other abstract machines (Peyton Jones 1987, Chapter 10). The scheme is discussed in more detail by Peyton Jones (1992, Section 3.1.2).

4.2.2 The Push-Enter Model

Closures that expect to get additional arguments on entering represent functions in the STGM. Implementing a language that supports curried functions, such as Haskell (Hudak et al. 1992), a function application may be undersaturated or supersaturated—i.e., it may be applied to more or less arguments than its arity requires. Consider the following fragment of Haskell code:

$$\begin{aligned} &f\ a\ b\ c && = \ldots \\ &g\ x && = f\ x\ 3 \\ &h && = g\ 1\ 2 \end{aligned}$$

Here, f is applied to two arguments, although it expects three; and g is applied to two arguments, although it expects one. A convenient scheme to implement such function applications is the *push-enter model* (Peyton Jones 1992, Section 3.2). In this model, evaluation of h proceeds as follows: The application of g pushes 1 and 2 on the *argument stack*, and then, enters the closure for g—this is illustrated in Figure 4.4(i) and (ii). The code of g's closure takes as many arguments from the stack as it requires, i.e., it pops the value 1—the value 2 remains on the stack; see Figure 4.4(iii). Now, the application of f is processed, which pushes 3 and then again 1 on the stack—they are lying on top of 2 now—Figure 4.4(iv). Finally, the closure associated with f is entered. It takes three arguments from the stack, namely the values 1, 3, and 2. Note that this scheme allows to implement the application of g and f as a tail call, whereas the alternative of processing the application (g 1) first, and then, applying the result to 2 would require to return to the code implementing the body of h.

4.2.3 Updates

Lazy evaluation requires that thunks are evaluated at most once. Therefore, after being evaluated for the first time, a closure that represents a thunk is *updated* with the result of the

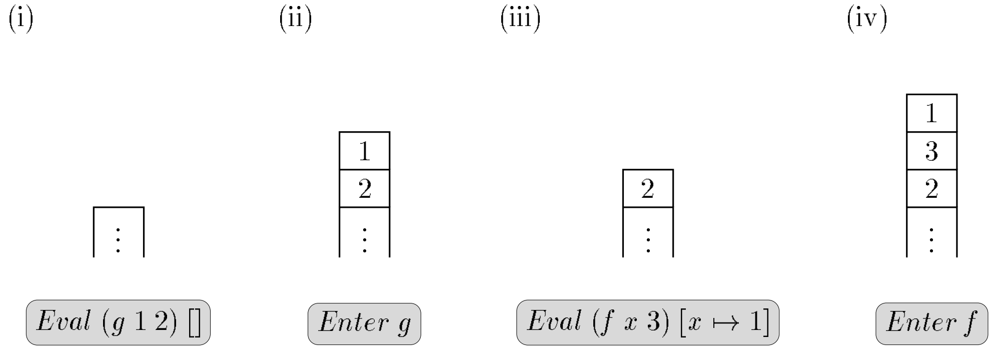

Figure 4.4 Illustration of the push-enter model. Applications push the arguments, starting with the rightmost, onto the argument stack, and then, enter the closure that is representing the applied function.

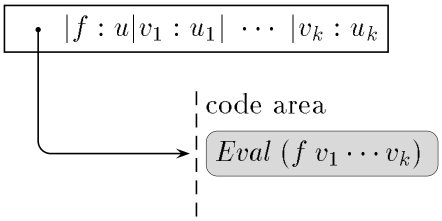

Figure 4.5 The scheme of closures that represents a partial application of the function that is implemented by the boxed value u. This function has an arity greater than k and was applied to the u_i.

evaluation—i.e., the closure is overwritten with a closure that represents the result. Every computation that maintains a reference to the updated closure has direct access to the result, after the update—the uniform closure representation of heap-located data makes the process of updating transparent to such computations.

Suppose an update is performed while a function is applied to less arguments than its arity requires. Then, we construct a closure that represents the application of the function to all available arguments. We call such a closure a *partial application*. The scheme for partial applications is depicted in Figure 4.5.

4.3 The Machine Language

Our next concern is the abstract machine language that is called simplified KCODE.[2] We start with its syntax. Then, we define its semantics by a mapping to $\mathcal{D}$. Finally, we discuss its operational interpretation on the basis of this mapping.

---

[2] Simplified KCODE is a subset of full KCODE (Chakravarty 1997), which is an intermediate language used in a compiler that I wrote while doing the research reported herein. The full language supports modules, makes types explicit, and includes first-class structures and a foreign language interface.

4.3.1 The Syntax

The grammar of simplified KCode is given in Figure 4.6, where nt^* denotes repetition of nt—i.e., a list of words where each is produced by nt. Types are not explicit in KCode, but the following exposition always assumes that KCode programs are well-typed. The notion of well-typedness is inherited from $\mathcal{D}$ by means of the mapping of simplified KCode to $\mathcal{D}$ that follows below. A number of type restrictions are important for the transition system, which comes later in this chapter. Although, they are induced as a consequence of the mentioned mapping to $\mathcal{D}$, we list them explicitly.

The types of all variables that are occuring in a KCode program are subject to the following restriction: Unboxed types may never be

- of functional type (i.e., $\sigma_1 \to \sigma_2$),
- of the agent type (i.e., `o`), or
- of a fully polymorphic type (i.e., the type may not consist of a type variable only, but it may contain type variables).

In other words, the type must have the form $c\,\overline{\tau}$, where c is any type constructor except $\to$, `o`, or `c`.

In a function binding f`=\`π vs`->`e, the variable f is boxed and the expression e is unboxed. In a `letpar` construct, `letpar` v_1 `=` $e_1, \ldots, v_n$ `=` e_n `in` e', the variables v_i and the expressions e_i are unboxed. Furthermore, while the v_i may occur in e', they may not occur in any e_i.

In a selection `case` v `of` *cases*, the variable v is unboxed. Furthermore, when v contains an integer, all the *cases* must have an integer pattern; otherwise, all *cases* must have a constructor pattern. Either way, the patterns of all alternatives must be distinct, and they may not contain a variable pattern.

In the introduction of an unbound variable $v : e$, the variable v is boxed and the expression e is of type `o`. In a choice p_1 `:=` $e_1, \ldots, p_n$ `:=` e_n `->` e, each expression e_i is unboxed and is neither an agent nor a functional. Furthermore, the variables bound in a p_i may not occur in any e_i. But they are allowed in the expression e that is forming the right-hand side of the choice. A choice is of type `o`.

In e_1 `||` e_2 and e_1 `=>` e_2, the expressions e_1 and e_2 have to be of type `o`. The composite expression is also of type `o`. In v_1 `:=:` v_2, the variable v_1 and v_2 must have the same type (in particular, both are boxed or both are unboxed). Their type may neither be `o` nor $\sigma_1 \to \sigma_2$. The composite expression is of type `o`.

In an application $(x\ vs)$, the name x must either denote a data constructor or it must denote a variable of boxed type. In contrast, a variable that stands alone (i.e., is not applied) must be unboxed. Constructor applications have to be saturated—i.e., they must be arity correct. Primitive functions, such as integer addition, must also be applied to the correct number of arguments, where the arguments are required to be unboxed—this guarantees that the application of a primitive does not introduce any hidden demands for evaluation.

4.3.2 The Semantics

We define the semantics of simplified KCode by a mapping $[\![\cdot]\!]$ into $\mathcal{D}$. A program consists of a list of bindings as defined by the nonterminal *Prog*, where we require that one binding defines `main`. The execution of a program evaluates `main`.

Program	*Prog*	$\rightarrow$	*FBind**	(recursive top-level bindings)
Function binding	*FBind*	$\rightarrow$	*Id* `=` *Mode* `[`*Id**`]` `->` *Exp*	
Value binding	*VBind*	$\rightarrow$	*Id* `=` *Exp*	
Update mode	*Mode*	$\rightarrow$	`u`	(update)
		$\mid$	`n`	(no update)
Expression	*Exp*	$\rightarrow$	*Agent*	(co-ordination form)
		$\mid$	*Caseof*	(selection)
		$\mid$	*Let*	(local binding)
		$\mid$	*Id* `[`*Id**`]`	(application)
		$\mid$	*Id*	(value)
		$\mid$	*Int*	(constant)
Co-ordination	*Agent*	$\rightarrow$	`choose` *Choice**	(committed choice)
		$\mid$	*Id* `:` *Exp*	(unbound variable)
		$\mid$	Exp_1 `||` Exp_2	(parallel composition)
		$\mid$	Exp_1 `=>` Exp_2	(sequential composition)
		$\mid$	Id_1 `:=:` Id_2	(equational constraint)
		$\mid$	`succ`	(success)
		$\mid$	`fail`	(failure)
Local binding	*Let*	$\rightarrow$	`letrec` *FBind** `in` *Exp*	(recursive)
		$\mid$	`letpar` *VBind** `in` *Exp*	(parallel)
Selection	*Caseof*	$\rightarrow$	`case` *Id* `of` *Case** [`dft ->`*Exp*]	
Case	*Case*	$\rightarrow$	*Pat* `->` *Exp*	
Choice	*Choice*	$\rightarrow$	*Match** `->` *Exp*	
Matching	*Match*	$\rightarrow$	*Pat* `:=` *Exp*	
Pattern	*Pat*	$\rightarrow$	*Id* `[`*Id**`]`	(constructor pattern)
		$\mid$	*Int*	(integer pattern)
		$\mid$	*Id*	(variable pattern)
Identifier	*Id*	:	⟨alphanumeric identifier⟩	
Integer	*Int*	:	⟨integer constant⟩	

Figure 4.6 The grammar of simplified KCODE. The grammatical form *nt** denotes the repetition of the nonterminal *nt*.

4.1 Semantics of Programs:

$$[\![fb_1,\ldots,fb_n]\!] \triangleq \mathsf{let}\ \langle [\![fb_1]\!],\ldots,[\![fb_n]\!]\rangle\ \mathsf{in}\ \texttt{main}\!\downarrow \qquad [\mathit{Prog}]$$

(The identifier that is enclosed in square brackets to the right of the equation denotes the syntactic class that contains the translated construct.)

$\mathcal{D}$ bindings, where the right-hand side is a lambda expression, represent KCode function bindings. The number of arguments n may be zero, in which case there is no abstraction.

4.2 Semantics of Function Bindings:

$$[\![f\ \texttt{=}\ \backslash\pi\ \texttt{[}v_1,\ldots,v_n\texttt{]}\ \texttt{->}\ e]\!] \triangleq f = (\lambda v_1.\ldots\lambda v_n.[\![e]\!]) \qquad [\mathit{Exp}]$$

A set of ask expressions is used to realize the committed-choice construct.

4.3 Semantics of Committed Choice:

$$\text{(a)}\quad [\![\texttt{choose}\ ch_1,\ldots,ch_n]\!] \triangleq ch_1\ [\,]\ \cdots\ [\,]\ ch_n \qquad [\mathit{Exp}]$$

$$\text{(b)}\quad [\![mat_1,\ldots,mat_n\ \texttt{->}\ e]\!] \triangleq \overline{v} : \langle [\![mat_1]\!],\ldots,[\![mat_n]\!]\rangle \rightarrow [\![e]\!] \qquad [\mathit{Choice}]$$

$$\text{(c)}\quad [\![c\ \texttt{[}v_1,\ldots,v_n\texttt{]}\ \texttt{:=}\ e]\!] \triangleq c\ \langle v_1,\ldots,v_n\rangle \doteq [\![e]\!] \qquad [\mathit{Match}]$$

$$\text{(d)}\quad [\![i\ \texttt{:=}\ e]\!] \triangleq i \doteq [\![e]\!] \qquad [\mathit{Match}]$$

$$\text{(e)}\quad [\![v\ \texttt{:=}\ e]\!] \triangleq v \doteq [\![e]\!] \qquad [\mathit{Match}]$$

In Equation 4.3b, the $\overline{v}$ on the right-hand side are the variables occuring in the left-hand sides of the mat_i.

We map the introduction of unbound variables, parallel composition, sequential composition, equational constraints, success, and failure to the corresponding constructs of $\mathcal{D}$.

4.4 Semantics of Composition and Constraints:

$$\text{(a)}\quad [\![v\ \texttt{:}\ e]\!] \triangleq \nu v.[\![e]\!] \qquad [\mathit{Exp}]$$

$$\text{(b)}\quad [\![e_1\ \texttt{||}\ e_2]\!] \triangleq [\![e_1]\!] \parallel [\![e_2]\!] \qquad [\mathit{Exp}]$$

$$\text{(c)}\quad [\![e_1\ \texttt{=>}\ e_2]\!] \triangleq [\![e_1]\!] \Rightarrow [\![e_2]\!] \qquad [\mathit{Exp}]$$

$$\text{(d)}\quad [\![v_1\ \texttt{:=:}\ v_2]\!] \triangleq [\![v_1]\!] \doteq [\![v_2]\!] \qquad [\mathit{Exp}]$$

$$\text{(e)}\quad [\![\texttt{succ}]\!] \triangleq \mathsf{1} \qquad [\mathit{Exp}]$$

$$\text{(f)}\quad [\![\texttt{fail}]\!] \triangleq \mathsf{0} \qquad [\mathit{Exp}]$$

We define the semantics of the two kinds of let expressions by the $\mathcal{D}$ letrec construct and value substitution by application, respectively.

4.5 Semantics of Lets:

$$\text{(a)}\quad [\![\texttt{letrec}\ fb_1,\ldots,fb_n\ \texttt{in}\ e]\!] \triangleq \mathsf{let}\ \langle [\![fb_1]\!],\ldots,[\![fb_n]\!]\rangle\ \mathsf{in}\ [\![e]\!] \qquad [\mathit{Exp}]$$

$$\text{(b)}\quad [\![\texttt{letpar}\ v_1\ \texttt{=}\ e_1,\ldots,v_n\ \texttt{=}\ e_n\ \texttt{in}\ e']\!] \triangleq (\lambda v_1.\ldots\lambda v_n.[\![e']\!])\ [\![e_1]\!]\cdots[\![e_n]\!] \qquad [\mathit{Exp}]$$

Due to the similarity of the letrec constructs of KCode and $\mathcal{D}$, the first equation is merely an adaption of syntax. The second equation is more involved. The variables bound by the `letpar` are introduced into the body expression $[\![e']\!]$ by means of a lambda abstraction. The bindings are established by applying the abstraction to the bound expressions. Note that this mapping implies that the e_i are independent of each other.

Next, we turn to the mapping of case expressions, which does not cover default cases. Instead, we assume that default cases are replaced by case completion. That is, we replace a default case by one ordinary case alternative for any constructor that is contained in the scrutinized type, but for which no case alternative exists yet. Each of these new case alternatives inherits its body expression from the expanded default case.[3] We can, then, map the resulting KCODE case expression without default cases to a $\mathcal{D}$ case expression.

4.6 SEMANTICS OF CASE:

(a) $[\![\texttt{case}\ v\ \texttt{of}\ case_1, \ldots, case_n]\!] \triangleq \mathsf{case}\ v\ \mathsf{of}\ \langle [\![case_1]\!], \ldots, [\![case_n]\!] \rangle \quad [Exp]$

(b) $[\![c\ \texttt{[}v_1, \ldots, v_n\texttt{]}\ \texttt{->}\ e]\!] \triangleq c\ \langle v_1, \ldots, v_n \rangle \rightarrow [\![e]\!] \quad [Case]$

(c) $[\![i\ \texttt{->}\ e]\!] \triangleq i \rightarrow [\![e]\!] \quad [Case]$

The semantics of application distinguishes two possible cases: Either a variable is applied, then it must be dereferenced first; or a data constructor is applied, then the mapping of the application just changes the syntax.

4.7 SEMANTICS OF APPLICATION:

(a) $[\![f\ \texttt{[}v_1, \ldots, v_n\texttt{]}]\!] \triangleq f\!\downarrow v_1 \cdots v_n \quad [Exp]$

(b) $[\![c\ \texttt{[}v_1, \ldots, v_n\texttt{]}]\!] \triangleq c\ \langle v_1, \ldots, v_n \rangle \quad [Exp]$

In both cases the number of arguments n may be zero. Value variables and integer constants remain variables and constants as formalized in the next display.

4.8 SEMANTICS OF VALUES:

(a) $[\![v]\!] \triangleq v \quad [Exp]$

(b) $[\![i]\!] \triangleq i \quad [Exp]$

Overall, the main differences between simplified KCODE and $\mathcal{D}$ is that the former (i) distinguishes global bindings (i.e., the function bindings constituting the program) from local bindings, (ii) introduces the `letpar` construct, and (iii) restricts arguments in applications to variables names.

First, we treat global bindings specially because they can be realized more efficiently than local ones. Second, we use the `letpar` construct to make independent subcomputations explicit and to expose them to transformational program optimization. In particular, optimization for multi-threading can be realized as a source-to-source transformation on KCODE, with the objective to cover latencies that are induced by remote access operations. And third, the restriction of arguments to variables supports the process of memory mapping—i.e., assigning memory locations to variables, in a compiler.

---

[3] Strictly speaking, the expansion of default cases is not possible for integer values—we would get infinitely many cases. But for the purpose of defining a semantics, we can represent integers by the Peano numerals, which leads to a finite representation of the expanded case expressions.

4.3.3 The Operational Interpretation

The mapping $[\![\cdot]\!]$ of KCODE to $\mathcal{D}$ provides KCODE indirectly with a logic and with a comparatively abstract operational meaning. This abstract operational meaning can be refined further to a machine-oriented operational interpretation of the KCODE constructs. We discuss the latter interpretation next; it is the basis for KCODE's use as an abstract machine language. Nevertheless, note that the provision of the abstract semantics is also important for an implementation: First, it guides the translation of a source language into the abstract machine language; and second, an algebraic semantics is the basis for a transformational approach to compilation—especially for declarative languages, a promising approach (Peyton Jones 1996, Appel 1992).

Function Bindings

Consider a general `letrec` expression and its mapping into $\mathcal{D}$:

$$\begin{aligned} & [\![\texttt{letrec } f_1 \texttt{ = \textbackslash}\pi_1\ xs_1 \texttt{ -> } e_1, \ldots, f_n \texttt{ = \textbackslash}\pi_n\ xs_n \texttt{ -> } e_n \texttt{ in } e]\!] \\ = \quad & \{\text{Semantics of } \texttt{letrec} \text{ (Property 4.5a) and function bindings (Property 4.2)}\} \\ & \mathsf{let}\ \langle f_1 = \lambda xs_1.[\![e_1]\!], \ldots, f_n = \lambda xs_n.[\![e_n]\!]\rangle\ \mathsf{in}\ [\![e]\!] \end{aligned}$$

Obviously, each function binding f_i=\\π_i xs_i->e_i—as its name suggests—implements a function $\lambda xs_i.[\![e_i]\!]$. The bindings give the names f_i to these functions. The mutual recursion admitted in $\mathcal{D}$ is carried over to KCODE. Because bindings introduce variables of name type in $\mathcal{D}$, which correspond to boxed values, the KCODE `letrec` introduces boxed values.

This matches well with a refined operational interpretation, where function bindings are an abstract notation of heap-allocated data structures—in other words, `letrec`s create closures. Consider the function binding

$$f \texttt{ = \textbackslash}\pi\ xs \texttt{ -> } e \tag{4.1}$$

Let us assume that vs are the free variables of e, not including the xs. Then, the closure that is denoted by this binding comprises a pointer to the code corresponding to the function together with a valuation of the free variables vs. The latter is called the environment of the function. As explained in Section 4.2, heap-allocated closures are a uniform representation of boxed data. In the transition system that we discuss later, we denote a closure for the function binding of Equation (4.1) by

$$\langle vs\ \backslash\pi\ xs \rightarrow \mathit{Eval}\ e\ , us, \mathit{Fun}\rangle$$

where the us are a valuation for the vs, and hence, they are of equal length ($|vs| = |us|$).[4] The expression *Eval* e specifies that, on entering this closure, the expression e is evaluated. The update mode π may be `u` or `n` and defines whether or not the closure must be updated after evaluation. The marker *Fun* indicates that this closure was created by a function binding—below, we will meet other markers that help to realize unbound variables.[5]

---

[4] We follow the functional programming tradition that appends an "s" at list-valued variables. The notation $|xs|$ means the length of the list xs. We shall define the latter more formally below.

[5] Although it may seem that we introduce tags into the tagless STGM by these markers, the next chapter shows in which way we preserve taglessness.

Following the terminology of Peyton Jones (1992, Section 4.2), we call the first component of a closure a *lambda-form* and distinguish the following classes of lambda-forms. (Closures are classified indirectly according to the lambda-forms that they contain.)

Manifest Functions. A lambda-form with a nonempty argument list (i.e., $|xs| \geq 1$ in the layout displayed above) is called a manifest function. They are never updated.

Partial applications. A lambda-form where the body evaluates an undersaturated function application is called a partial application. The general form is

$$vs \ \backslash\texttt{n}\ [\,] \rightarrow \mathit{Eval}\ (f\ xs)$$

where $|xs|$ is smaller than the arity of the function f.

Constructors. A lambda-form where the body evaluates the application of a data constructor is called a constructor; it has the following general form:

$$vs \ \backslash\texttt{n}\ [\,] \rightarrow \mathit{Eval}\ (c\ xs)$$

where c is the name of a data constructor.

Basic values. A lambda-form is called a basic value when the body returns an unboxed integer value that is stored in the environment. The form is

$$[v] \ \backslash\texttt{n}\ [\,] \rightarrow \mathit{Eval}\ v$$

(Note that this variant is not used in the STGM; we introduce it due to our modified treatment of unboxed values.)

Success. A lambda-form whose body contains the agent `succ`. It is used to update closures that represent values of type `o`, when these closures have been evaluated successfully. The form is

$$[\,] \ \backslash\texttt{n}\ [\,] \rightarrow \mathit{Eval}\ \texttt{succ}$$

(This variant is also new in HIPAR.)

Thunks. All other lambda-forms are called thunks. Thunks are the only kind of lambda-forms where the update mode may be `u`, because they are the only lambda-forms that are not in normal form.

Note that we could have a different representation for data constructors and for basic values. Data constructors could be represented in the same way as basic values, storing their unboxed representation in the environment. Furthermore, basic values could be represented as lambda-forms that contain the stored value as a constant (leaving the environment empty). Although on the current level of abstraction these alternative representations are feasible, we will see in the next chapter that they imply less efficient code on concrete hardware.

Value Bindings

A general `letpar` expression and its mapping into $\mathcal{D}$ have the following form:

$$\begin{array}{ll} & [\![\texttt{letpar } x_1 \texttt{ = } e_1, \ldots, x_n \texttt{ = } e_n \texttt{ in } e]\!] \\ = & \quad \{\text{Semantics of } \texttt{letpar} \text{ (Property 4.5b)}\} \\ & (\lambda x_1. \ldots \lambda x_n.[\![e]\!]) e_1 \cdots e_n \end{array}$$

Two important facts about `letpar` are immediately obvious from this semantic definition: (i) the evaluation of the different e_i is completely independent of each other and (ii) the body expression e cannot be evaluated before the evaluation of all e_i is completed—remember that beta reduction, in $\mathcal{D}$, is only possible when the arguments are in normal form.

In some sense, value bindings are the operational opposite of function bindings.

- Function bindings bind boxed values, whereas value bindings bind unboxed values.
- Function bindings suspend the evaluation of their right-hand side, whereas value bindings demand the evaluation of their right-hand sides.
- Function bindings are mutually recursive, whereas value bindings must be independent.

As a consequence, a `letrec` evaluates its body before it evaluates the right-hand side of any of its bindings, whereas a `letpar` evaluates all its right-hand sides before it evaluates its body. Overall, the operational concept associated with a `letpar` of the form

$$\texttt{letpar } v_1 \texttt{ = } e_1, \ldots, v_n \texttt{ = } e_n \texttt{ in } e$$

is that (i) the bindings v_i = e_i constitute a collection of independent threads and that (ii) the evaluation of e is synchronized on the termination of all of these threads. Referring to the discussion on multi-threading in Section 4.1.2, we see that if the evaluation of any e_i stalls due to a long latency operation, the evaluation of any other e_j is a good candidate to cover the latency—it is strongly related, but independent. Hence, by accumulating as many value bindings as possible into one `letpar`, we increase the ability of the code to cover long latency operations. Consequently, KCODE provides an abstract notion of thread independence, and thus, allows to realize optimizations as source-to-source transformations that otherwise had to be buried in the code generator. Interpreting the value bindings of a `letpar` as independent threads, we can regard the evaluation of the body e as a thread that depends on the threads implementing these bindings—i.e., we have an explicit, but abstract concept of thread synchronization.

Case Expressions

A general `case` expression and its mapping into $\mathcal{D}$ have the following form:

$$\begin{array}{ll} & [\![\texttt{case } x \texttt{ of } p_1 \texttt{ -> } e_1, \ldots, p_n \texttt{ -> } e_n]\!] \\ = & \quad \{\text{Semantics of case (Property 4.6)}\} \\ & \mathsf{case}\ x\ \mathsf{of}\ \langle p'_1 \rightarrow [\![e_1]\!], \ldots, p'_n \rightarrow [\![e_n]\!]\rangle \end{array}$$

where the p'_i have the form $c\ vs_i$ or are integer constants. As we require the alternatives of a $(\mathsf{case} \cdot \mathsf{of} \cdot)$ construct to be distinct, the construct implements a deterministic case selection by examination of the unboxed value that is identified by x. When a default case is present, evaluation takes this route only when it cannot select any other case alternative.

At this point, it is important to note a fundamental difference between the STG-language and KCODE. In the STG-language, a `case` demands the scrutinized expression and performs case selection; but in KCODE, a `case` just performs case selection, whereas the `letpar` construct issues demands. The reason for this separation of concerns is the wish to express the clustering of independent threads in KCODE. Where we use an expression

`case` e `of` $alt_1 ; \ldots ; alt_n ; dft$

in the STG-language, we use

`letpar` v `=` e `in`
`case` v `of` $alt_1 ; \ldots ; alt_n ; dft$

in KCODE, instead.

Function Application

The semantics for KCODE, Property (4.7a), provides us with the equality

$$[\![f\ [x_1, \ldots, x_n]]\!] = f{\downarrow}\, x_1\ \cdots\ x_n$$

Thus, after dereferencing f, which the typing constraints require to be of a type of names, we apply the resulting function to the x_i.

Using the push-enter model (as described in Section 4.2), we realize this behaviour by pushing the x_i onto the argument stack, and then, entering the closure identified by f. Entering a closure—i.e, a boxed value—retrieves the corresponding unboxed value. The latter is bound to be a function when $n \geq 1$, and this function pops the arguments that it requires from the argument stack. If $n = 0$, the whole mechanism reduces to unboxing a boxed value, which may include the evaluation of a thunk.

Overall, the duality of names and terms in $\mathcal{D}$ finds its counterpart in the duality of boxed and unboxed values in KCODE. Furthermore, the duality of binding and dereferencing in $\mathcal{D}$ is realized by the duality of creating closures and entering them in KCODE—conjoined with the duality of function abstraction and application.

Constructor Application and Constants

The operational interpretation of both the application of a data constructor to some arguments and the plain statement of an integer constant is straight-forward; both explicitly denote an unboxed value. But this interpretation lead us to a second important difference between KCODE and the STG-language.

Although the effect attained in KCODE by unboxed values is very similar to the effect attained in the STG-language, unboxed values are handled differently in KCODE. The main difference comes from the fact that, in the STG-language, boxed values are the default and unboxed values are specially marked. A data constructor in the STG-language is usually employed to represent boxed data. If an unboxed representation is desired, this is explicitly stated by appending a hash mark (`#`) at the name. Nevertheless, the implementation of the

STGM takes the freedom to temporarily represent some values unboxed, although this is not explicit in the STG-program.

In contrast, KCODE values and types are unboxed by default. If a value is required to be boxed, it is put into a closure by means of a function binding. Given that the unboxed value is of type τ, the type of the boxed value is $\lfloor\tau\rfloor$ (where we use the type notation of $\mathcal{D}$). Hence in KCODE, there is no boxed value without an unboxed one. Although this principle may also be true on the implementation level of the STGM, it is not visible on the level of the STG-language.

To illustrate the difference, let us have a look at the representation of boxed and unboxed integers in both languages. In the STG-language, we get a boxed integer from a unboxed one by wrapping it into a datatype. Consider the data type definition

```
data Int = MkInt Int#
```

where `Int#` is the type of unboxed integers and `Int` is a new data type whose values are composed by the application of the new data constructor `MkInt` to an unboxed integer. In KCODE, the type *Int* contains unboxed integers, whereas $\lfloor Int \rfloor$ represents boxed integers.

Turning our attention back to the STG-language, the creation of a boxed integer value that represents 1 implies the use of a lambda-form, where the unboxed integer `1#` is wrapped into the constructor `MkInt` as follows:

```
one = {} \n {} -> MkInt 1#
```

In KCODE, we can use a function binding that contains `1` verbatim (where `1` is the KCODE representant for an unboxed 1):

```
one = \n [] -> 1
```

Now, to unbox a boxed integer, the STG-language requires a `case` construct, which first evaluates a potential suspension, and then, strips the `MkInt`:

```
case e of
  MkInt x# -> ...
```

In KCODE, we use a `letpar` for the evaluation, and no stripping is necessary:

```
letpar
  x = e
in ...
```

The main reason for the change in the management of unboxed values is the separation of demand from case selection, which has been discussed in the context of the operational interpretation of `case` expressions. To understand the need for the change, we have to consider a low-level optimization of the STGM.

For example, a function that creates and returns a cons cell does return an unboxed form of the cons cell (a cons cell can be stored in three machine registers: one for the constructor, and one each for head and tail). This is not obvious on the level of the STG-language, but just a sort of optimization of the concrete implementation (Peyton Jones 1992, Section 9.4.4).

This optimization is beneficial, because a boxed version is often not needed. It would be decomposed by the calling `case` anyway.

Unfortunately, case expressions with a default alternative as well as updates, require that a boxed value is created from the returned unboxed value. Hence, whether a boxed value is needed depends on the calling context. The STGM solves this problem as follows: The demand for an evaluation always originates from a case expression; if required, this case expression creates a boxed version of the unboxed value on-the-fly.

Due to the separation of demand and selection into `letpar` and `case` in KCODE, it is no longer possible to realize this optimization behind the scenes. Therefore, KCODE employs the rule that all functions return their result unboxed. Whenever a boxed version is needed, it is explicitly constructed. Consequently, superfluous boxing must now be removed by employing a source-to-source optimization.

Committed Choice

The general form of a `choose` and its mapping into $\mathcal{D}$ are as follows:

$$\begin{array}{ll} & [\![\texttt{choose}\ (p_1^1\ \texttt{:=}\ e_1^1, \ldots, p_1^{m_1}\ \texttt{:=}\ e_1^{m_1})\ \texttt{->}\ e_1, \ldots, (p_n^1\ \texttt{:=}\ e_n^1, \ldots, p_n^{m_n}\ \texttt{:=}\ e_n^{m_n})\ \texttt{->}\ e_n]\!] \\ = & \quad \{\text{Semantics of committed choice (Property 4.3)}\} \\ & vs_1 : \langle {p'}_1^1 \doteq [\![e_1^1]\!], \ldots, {p'}_1^{m_1} \doteq [\![e_1^{m_1}]\!]\rangle \rightarrow [\![e_1]\!] \ [] \\ & \quad \ldots [] \ vs_n : \langle {p'}_n^1 \doteq [\![e_n^1]\!], \ldots, {p'}_n^{m_n} \doteq [\![e_n^{m_n}]\!]\rangle \rightarrow [\![e_n]\!] \end{array}$$

The alternative ask expressions may be overlapping—i.e, the `choose` construct allows to express don't care indeterminism. Before the i-th alternative can be taken, the e_i^j have to be completely evaluated. Hence, the selection blocks on the evaluation of these expressions, which realizes blocking ask.

Operationally, we regard such blocking as waiting for communication. Furthermore, the e_i^j of the i-th alternative are required to be independent (as the vs_i may not occur on the right-hand side of any $\doteq$). Consequently, we can treat these expressions similar to the right-hand sides of the value bindings contained in a `letpar` and utilize them to cover latency.

Moreover, the guards in the different alternatives of a `choose` can be checked concurrently. For reasons of efficiency, we require that whenever the guard of one alternative succeeds, this alternative is taken immediately.

Unbound Variables and Equational Constraints

Unbound variables are required to be of boxed type. Hence, they have to be represented as a closure. Furthermore, unbound variables of KCODE are mapped to $\mathcal{D}$ as follows:

$$[\![x\ :\ e]\!] = \nu x.[\![e]\!]$$

Thus, dereferencing x—i.e., entering the corresponding closure—has to suspend the current computation. Only when an equational constraint of the form

$$[\![x\ \texttt{:=:}\ y]\!] = x \doteq y$$

associates the unbound variable x with a function binding, dereferencing x may proceed. Hence, equational constraints update the closures of unbound variables.

This is operationally achieved by introducing two new forms of closures. First, the introduction of a new unbound variable creates an *unbound variable closure* of the form

$$\langle [t]\ \texttt{\textbackslash n}\ [\,] \to \mathit{Suspend}\ t\ x\ ,\ [u],\ \mathit{UB}\rangle$$

The special code form *Suspend* suspends any computation that attempts to evaluate it. The closure marker *UB* allows the unification mechanism, which realizes equational constraints, to recognize this kind of closures and to treat them specially. In particular, when an unbound variable is associated with another closure, it is updated with a reference to this closure. Note that, for two reasons, it may not be updated with a copy of the closure itself.

- When the other closure is a thunk, copying the thunk would break the sharing required for lazy evaluation—both copies of the thunk may be evaluated, duplicating the work of the evaluation.
- The other closure may also represent an unbound variable. Then, this other variable may be updated later by another equational constraints. This update also has to take effect on the first unbound variable, to realize the transitivity of $\doteq$.

Both cases can be handled by a second, new form of closures, the *forwarding closure*, which has the general form

$$\langle [v]\ \texttt{\textbackslash n}\ [\,] \to \mathit{Eval}\ (v\ \texttt{[]})\ ,\ [y],\ \mathit{Fwd}\rangle]$$

When a forwarding closure is entered, it simply enters the closure denoted by y—i.e., when the closure of x is updated with the above forwarding closure, entering x will be forwarded to y. Thus, when y is updated—either because it was a thunk that got evaluated or because it was an unbound variable that was associated with another closure—any subsequent evaluation of x will behave as if the updated y had been evaluated.

Parallel and Sequential Composition

The constructs `||` and `=>` correspond to parallel composition $\|$ and sequential composition $\Rightarrow$. The expression e_1 `||` e_2 evaluates the two subexpressions e_1 and e_2 in parallel; whereas e_1 `=>` e_2 defers the evaluation of e_2 until the evaluation of e_1 is successfully completed.

We use the concept of *machine instances* to express parallelism. Each machine instance contains its own, complete set of data structures (including heap, argument stacks, and so on), and evaluates some expression asynchronously to the other machine instances. When an instance i needs to enter a closure located in the heap of another instance j, it cannot do so directly; instead, i requests instance j to enter the closure and to pass the result back to i. In this way, machine instances are an abstract means to model distributed memory, and remote closure evaluation models the long latency operations associated with remote-memory access.

The concrete operational interpretation of e_1 `||` e_2 is the creation of a new machine instance for the evaluation of e_2, while e_1 is evaluated by the instance that executed the `||`. Note that at this level of abstraction, we are just modeling the *partitioning* of a program into functionally independent tasks, but neither their agglomeration nor the mapping onto a restricted number of real PEs (for a definition of these terms, see Foster 1995). We return to this topic in the next chapter.

It is important to understand the difference between the parallelism specified by `||` and that expressed by `letpar`. Expressions built from `||` specify potential for *true parallelism*, in the sense that different machine instances (which are created by `||`) may be executed on different PEs at the same time. In contrast, `letpar` merely states some freedom in the execution order that can be used to cover latencies induced by remote access, but the evaluation will always be *interleaved* on one PE. We will see that this interleaving can be realized with only small overhead, whereas true parallelism always carries significant costs (mainly due to inter-processor communication).

4.3.4 Transformational Optimization of KCODE Programs

An important feature of the STGM is the possibility to realize optimizations as source-to-source transformations on the abstract machine language (Peyton Jones 1996, Peyton Jones et al. 1996). These optimizations can be transferred from the STGM to HIPAR. Furthermore, the `letpar` construct allows new optimizations, which improve the ability of the code to cover latencies induced by remote data access. These new optimizations build on the operational interpretation of `letpar` as a notation to express thread partitioning and synchronization as discussed above.

$$\texttt{letpar } v_1 = e_1, \ldots, v_n = e_n \texttt{ in } e$$

The n threads corresponding to the n value bindings $v_i = e_i$ are independent and, if no remote access occurs, are executed from left to right. Only after all n threads completed may the evaluation of the body expression e begin.

Each of the e_i may initiate a remote access, which stalls the corresponding thread. In this case, any of the other threads may be selected to cover the remote access. If there is any static information on the probability that a thread issues a remote access, the hiding of latencies can be improved by placing threads with a high remote access probability at the beginning of the binding list. As a result, remote accesses are usually issued while there is still independent work left. In particular, there is one form of value bindings that should always be moved to the very end of the binding list. These are value bindings that do not contain an application of a defined function, although they may contain an application of a primitive. Such bindings can never initiate a remote access.

A second, more important transformation allows to combine multiple `letpar`s into one if the bindings are independent. We can apply the rule

$$\begin{array}{lcl} \texttt{letpar} & & \\ \quad v_1 = e_1, \ldots, v_n = e_n & & \texttt{letpar} \\ \texttt{in letpar} & \Longrightarrow & \quad v_1 = e_1, \ldots, v_n = e_n, v'_1 = e'_1, \ldots, v'_{n'} = e'_{n'} \\ \quad v'_1 = e'_1, \ldots, v'_{n'} = e'_{n'} & & \texttt{in } e \\ \texttt{in } e & & \end{array} \tag{4.2}$$

if none of the v_i occurs in the e'_j. Furthermore, if the v_i occur in one of the e'_j, but not in the body expression e, we can still apply the following rule:

$$\begin{array}{lcl} \texttt{letpar} & & \texttt{letpar} \\ \quad v_1 = e_1, \ldots, v_n = e_n & & \quad v'_1 = e'_1, \ldots, v'_{j-1} = e'_{j-1}, \\ \texttt{in letpar} & \Longrightarrow & \quad v'_j = \texttt{letpar } v_1 = e_1, \ldots, v_n = e_n \texttt{ in } e'_j, \\ \quad v'_1 = e'_1, \ldots, v'_{n'} = e'_{n'} & & \quad v'_{j+1} = e'_{j+1}, \ldots, v'_{n'} = e'_{n'} \\ \texttt{in } e & & \texttt{in } e \end{array} \tag{4.3}$$

Reconsidering the example of the evaluation of $(a + b) * (c + d)$ from Section 4.1.2, the naive translation of the expression results in the following KCODE expression (`addPrim` and `mulPrim` are the primitives for addition and multiplication):

```
letpar av = a []
in letpar bv = b []
in letpar ab = addPrim [av, bv]
in letpar cv = c []
in letpar dv = d []
in letpar cd = addPrim [cv, dv]
in mulPrim [ab, cd]
```

Although the code uses `letpar` instead of `case` to demand the evaluation of a, b, c, and d, the behaviour is similar to the corresponding STG-code (see also the comparison in the previous subsection). In particular, the access to the four variables as well as the execution of the three arithmetic operations is sequentialized. If the evaluation of a initiates a remote access, the whole computation stalls.

By applying the transformation rules displayed in the Equations (4.2) and (4.3), we can transform the expression into the following optimized expression, which computes the same result:

```
letpar
  ab = letpar
         av = a []
         bv = b []
       in addPrim [av, bv]
  cd = letpar
         cv = c []
         dv = d []
       in addPrim [cv, dv]
in mulPrim [ab, cd]
```

The optimized expression places the access to the four variables into four independent threads and introduces dependencies on the calculation of the three arithmetic expressions only as far as the dataflow requires. In particular, when the evaluation of a stalls, the access to the other three variables can be executed and the second addition can be used to cover the latency. In general, it is beneficial to collect as many bindings into a single `letpar` as possible, nest `letpar`s according to data dependencies, and put bindings with a high remote access probability towards the beginning of a binding list.

4.4 The Machine Architecture

The preceding discussion about the operational interpretation of the various KCODE constructs already introduced many important concepts of the machine architecture of HIPAR. Next, we discuss this architecture in more detail. To improve the precision of the definition of the data structures and the accompanying operations, the following informal descriptions are accompanied by Haskell type and function definitions. To further readability, we use some nonstandard syntax in the Haskell definitions, such as special operator symbols, subscripts, Greek letters, and some ad hoc operator overloading. Where the mapping of the additional syntactic constructs to plain Haskell is not obvious, we define it explicitly. The Haskell prelude functions, which are used in the following, are defined in Appendix B. More information on Haskell is provided by Hudak et al. (1992). Together the definitions of the machine architecture and the transition system define an executable interpreter for KCODE, which is optimized for clarity not efficiency.

4.4.1 Basic Definitions

Lists are predefined in Haskell. They are constructed from [], which denotes the empty list, and $x : xs$, which combines a list element x and a list xs into a new list. Instead of $x_1 : \cdots : x_n : []$, we may write $[x_1, \ldots, x_n]$. Haskell's standard library provides many operations on lists, we use a special syntax for some of them.

4.9 NOTATION
We use the following operations on lists with a special syntax:

$$
\begin{aligned}
|s| &\triangleq \mathit{length}\ s && \text{— length of a list}\\
x \in l &\triangleq x\ \text{`}\mathit{elem}\text{`}\ l && \text{— check element membership}\\
l_1 \setminus l_2 &\triangleq l_1\ \backslash\backslash\ l_2 && \text{— list difference}\\
l_1 +\!\!\!+\, l_2 &\triangleq l_1 \mathbin{++} l_2 && \text{— append a list at another one}
\end{aligned}
$$
∎

In the transition system, we repeatedly need finite maps, which we implement by lists.

4.10 DEFINITION (FINITE MAPS)
A *finite map* associates keys of type k with elements of type e. (We take the freedom to use infix type and data constructors in the following definition.)[6]

$$
\begin{aligned}
&\mathbf{data}\ (k \mapsto e) && = (k \mapsto e)\\
&\mathbf{type}\ [k \mapsto e] && = [k \mapsto e] && \text{— all keys are required to be distinct}
\end{aligned}
$$

The symbol $\mapsto$ is overloaded with a function associating the elements of two lists pairwise.

$$
\begin{aligned}
&(\mapsto) && :: [k] \to [e] \to [k \mapsto e]\\
&ks \mapsto es && = \mathit{zipWith}\ (\mapsto)\ ks\ es
\end{aligned}
$$

We use the combine operator $m_1 \triangleright m_2$, which prefers bindings contained in its left argument m_1, to ensure that the keys remain distinct when we combine finite maps. The definition of this operator uses the auxiliary function *addExcl*, which excludes duplicate keys (we use $(\doteqdot)$ to denote the equality test provided by the *Eq* class of Haskell's standard library[7]).[8]

$$
\begin{aligned}
&(\triangleright) && :: [k \mapsto e] \to [k \mapsto e] \to [k \mapsto e]\\
&m_1 \triangleright m_2 && = \mathit{foldl}\ m_2\ (\lambda m'\ (k \mapsto e) \to \mathit{addExcl}\ k\ e\ m')\ m_1)\\
& && \quad \mathbf{where}\\
& && \qquad \mathit{addExcl}\ k\ e\ m = (k \mapsto e) : [a \mid a@(k' \mapsto \_) \leftarrow l, k \doteqdot k']
\end{aligned}
$$

---

[6] Note that it is allowed in Haskell to have type and data constructors with the same names.

[7] Hence, $\doteqdot$ denotes ASCII ==.

[8] We omit type contexts in the following type assertions.

We have the following partial function to inquire an association.

$$\begin{array}{ll} (\uparrow) & :: [k \mapsto e] \to k \to e \\ m \uparrow k & = \mathit{head}\,[e \mid (k' \mapsto e) \leftarrow m, k \doteqdot k'] \end{array}$$

We overload this notation with inquiry of the associations of a list of keys:

$$\begin{array}{ll} (\uparrow) & :: [k \mapsto e] \to [k] \to [e] \\ m \uparrow \mathit{ks} & = \mathit{map}\,(m \uparrow)\,\mathit{ks} \end{array}$$

And the function *keys* obtains the keys of a finite map:

$$\begin{array}{ll} \mathit{keys} & :: [k \mapsto e] \to [k] \\ \mathit{keys}\ m & = [k \mid (k' \mapsto e) \leftarrow m] \end{array}$$ ∎

All nonterminals from Figure 4.6 are regarded as names of types, which contain the words produced by the nonterminal as values. The identifiers defined by the nonterminal *Id* from Figure 4.6 are also employed as a type for names that are used internally in the abstract machine. We add a subscript to all occurrences of *Id* that denote a specific kind of named objects—for example, $\mathit{Id}_{\text{inst}}$ denotes the names of machine instances. We regard these subscripted occurrences as type synonyms of *Id*.

4.11 DEFINITION (IDENTIFIERS)
The type *Id* contains identifiers. No assumption is made about their representation. We merely require that there exists an instance of the standard Haskell equality class *Eq* for identifiers, and that the following functions generate new, unique names—Augustsson, Rittri & Synek (1994) have discussed the details of these functions.

data *Supply* — this type is abstract

$$\begin{array}{ll} \mathit{ids} & :: \mathit{Supply} \to [\mathit{Id}] \\ \mathit{supplies} & :: \mathit{Supply} \to [\mathit{Supply}] \end{array}$$ ∎

The idea is that, given a name supply $\nu :: \mathit{Supply}$, we can split ν into arbitrary many supplies with the function *supplies*, or else, we can extract arbitrary many new identifiers with *ids*.

4.4.2 Values

We start the introduction of the machine specific data structures by defining the representation of values.

4.12 DEFINITION (VALUES)
We define *values* of three different forms, namely unboxed integers, unboxed terms, and global addresses—the latter are used to identify boxed values.

$$\begin{array}{lll} \textbf{data}\ \mathit{Value} = & \mathit{Int}\ \mathit{Int} & \text{— unboxed integer} \\ & \mid \mathit{Term}\ \mathit{Id}_{\text{constr}}\,[\mathit{Value}] & \text{— unboxed term} \\ & \mid \mathit{GAdr}\ \mathit{GId} & \text{— global address} \end{array}$$

Global address comprise an instance identifier and a local name—the latter is either the name of a closure (i.e., a heap location) or the name of a thread.

$$\textbf{data}\ \mathit{GId}_{kind} = \mathit{GId}\ \mathit{Id}_{\text{inst}}\ \mathit{Id}_{kind} \qquad \blacksquare$$

4.13 NOTATION
For global addresses, we use an improved syntax. (Note that the arguments are exchanged.)

$$x@i \triangleq \mathit{GId}\ i\ x \qquad \blacksquare$$

4.14 DEFINITION (ENVIRONMENTS)
We represent *environments* by finite maps from variable identifiers to values.

$$\textbf{type}\ \mathit{Env} = [\mathit{Id}_{\text{var}} \mapsto \mathit{Value}]$$

In the transition rules, we use a function *val* that, given two environments, looks up a variable. First, it considers the first environment, and if this is not successful, it takes the second.

$$\begin{array}{lll} \mathit{val} & & :: \mathit{Env} \to \mathit{Env} \to \mathit{Id}_{\text{var}} \to \mathit{Value} \\ \mathit{val}\ \rho\ \sigma\ x & \mid x \in \mathit{keys}\ \rho & = \rho \uparrow x \\ & \mid \mathit{otherwise} & = \sigma \uparrow x \end{array}$$

We overload this function for lists of variable names.

$$\begin{array}{ll} \mathit{val} & :: \mathit{Env} \to \mathit{Env} \to [\mathit{Id}_{\text{var}}] \to [\mathit{Value}] \\ \mathit{val}\ \rho\ \sigma\ xs & = \mathit{map}\ (\mathit{val}\ \rho\ \sigma)\ xs \end{array} \qquad \blacksquare$$

In the transition system, *val* is used to compute the valuation of a variable in the presence of a local and a global environment, where we want the local environment to take precedence.

4.4.3 Closures

As explained, boxed values are represented by closures that are stored in the heap. Therefore, we can model a heap as a finite map from closure names Id_{cls} to closures.

4.15 DEFINITION (CLOSURE)
Closures are triples consisting of a lambda-form, a valuation of the environment variables, and a marker.

$$\begin{array}{ll} \textbf{type}\ \mathit{Cls} = (\mathit{LF}, & \text{— lambda form (code component)} \\ \qquad\qquad [\mathit{Value}], & \text{— valuation of the free variables} \\ \qquad\qquad \mathit{Marker}) & \text{— closure marker} \end{array}$$

Lambda-forms consist of the following four components:

$$\begin{array}{ll} \textbf{data}\ \mathit{LF} = \mathit{LF}\ [\mathit{Id}_{\text{var}}] & \text{— free variables (environment)} \\ \qquad\qquad\quad \mathit{Mode} & \text{— update mode} \\ \qquad\qquad\quad [\mathit{Id}_{\text{var}}] & \text{— argument variables} \\ \qquad\qquad\quad (\mathit{Env} \to \mathit{Code}) & \text{— code form} \end{array}$$

where the update mode *Mode* is already defined by the KCODE grammar (Figure 4.6), and *markers* are defined as

$$\begin{array}{lll} \textbf{data}\ \mathit{Marker} = \mathit{Fun} & & \text{— function binding closure} \\ \qquad\qquad\quad \mid \mathit{UB} & & \text{— unbound variable closure} \\ \qquad\qquad\quad \mid \mathit{Fwd} & & \text{— forwarding closure} \end{array}$$ ∎

4.16 NOTATION
We use the following syntax for closures

$$\langle \mathit{lf}, \mathit{us}, \mu \rangle \triangleq (\mathit{lf}, \mathit{us}, \mu)$$

and for lambda-forms

$$\mathit{vs}\ \backslash \pi\ \mathit{xs} \to \gamma \triangleq \mathit{LF}\ \mathit{vs}\ \pi\ \mathit{xs}\ \gamma$$ ∎

Overall, we get the following general form of closures

$$\langle \mathit{vs}\ \backslash \pi\ \mathit{xs} \to \gamma, \mathit{us}, \mu \rangle$$

which is the form that was introduced in Section 4.3.3.

The code contained in a lambda-form has the type $(\mathit{Env} \to \mathit{Code})$—i.e., it is a function from environments to code forms. The idea is that, when a computation enters a closure, an environment is constructed by associating the names of the free variables with the valuation stored in the closure plus the association of the names of the arguments with values popped from the argument stack. Then, the code contained in the lambda-form is applied to this environment, to yield the code form that has to be executed.

4.4.4 Thread Activations

Thread activations are a generalization of the activation records that are used in sequential implementations of functional and imperative languages to store the data structures required for one function activation (see Waite & Goos 1985). The differences between thread activations and conventional activation records stem from the fact that threads may proceed asynchronously, whereas conventional function calls are synchronous. In the sense that a parent activation passes control to a child on calling it; suspends exactly until the child activation is finished; and finally gets control back from the child, while the latter ceases to exist. In other words, there are two new requirements in the management of threads: We want to be able to deviate from the depth-first traversal to cover latencies induced by remote accesses and, as a consequence, we have to maintain multiple activities that are executing in an interleaved fashion on a single machine instance.

A similar motivation led Culler et al. (1993) to the introduction of frames that are representing asynchronous function invocations in the TAM. In the TAM, the conventional depth-first traversal of sequential languages is completely abandoned. Nevertheless, recent results by Blelloch, Gibbons & Matias (1995) and Goldstein, Schauser & Culler (1996) show that it is preferable to stick as closely as possible to the conventional depth-first traversal and deviate only as far as necessary to exploit parallelism. The point made in Section 4.1.2, namely that it is preferable to use computations with a *close* context to cover latencies, also requires to follow the depth-first order as far as possible. Below, we use a stack, called the DFT stack, to keep track of the evaluation order that is induced by a depth-first traversal.

4.17 Definition (Thread)

A *thread* is composed from the code that it has to execute, an argument stack, and a target list that determines where the result of the current computation is needed.

$$\begin{array}{lll}\textbf{type } \mathit{Thread} = (\mathit{Code}, & & \text{— executed code}\\ \qquad\qquad\qquad [\mathit{Value}], & & \text{— argument stack}\\ \qquad\qquad\qquad [\mathit{Target}]) & & \text{— target list}\end{array}$$

The entries into the target list may assume the following three forms:

$$\begin{array}{lll}\textbf{data } \mathit{Target} = & \mathit{Upd}\ \mathit{GId}_{\text{cls}} & \text{— update the named closure}\\ & \mid \mathit{Store}\ \mathit{GId}_{\text{thread}}\ \mathit{Id}_{\text{var}} & \text{— signal completion to the named thread}\\ & \mid \mathit{Sig}\ \mathit{GId}_{\text{thread}} & \text{— store unboxed value and signal}\\ & \mid \mathit{Remote}\ \mathit{GId}_{\text{inst}}\ [\mathit{Target}] & \text{— targets on remote instance}\end{array}$$

∎

A thread (γ, as, ts) executes the code γ—we discuss the structure of code forms latter. The code may push and pop arguments to and from the stack as (on termination of the thread, the argument stack will be empty, since we always assume type correct programs). Finally, the result of the computation is communicated to the targets ts. The intuition that underlies the three forms of targets is as follows.

- *Upd* g_c: The thread updates the closure identified by g_c with the boxed version of the result.
- *Sig* g_t: The current thread notifies the thread named g_t about the completion of the computation (independent of the actually computed result). This is pure synchronization, where no actual communication occurs.
- *Store* g_t x: The thread stores an unboxed version of the result in the environment that is contained in the code form of thread g_t. The result is associated with the variable name x in this environment. Afterwards, the thread named g_t is notified, in the same way as a *Sig* target would notify it.
- *Remote* i ts: The thread forwards the result to the targets ts, which are located on the instance named i.

The last variant is not strictly necessary, because the global closure or thread name in the other three variants is sufficient to determine the instance where a target is located. Nevertheless, it simplifies the handling of remote targets and, in particular, it helps to guarantee that all entries in a list of remote targets is completely processed before any other operation on the remote instance can be executed.

4.18 Notation

Threads are usually associated with a name. Thus, we introduce named threads by

$$\textbf{type } \mathit{NamedThread} = (\mathit{Id}_{\text{thread}} \mapsto \mathit{Thread})$$

and use a special notation for them,

$$\langle \tau \bullet \gamma, as, ts \rangle \triangleq (\tau \mapsto (\gamma, as, ts))$$

∎

4.4.5 Machine Instances

As already mentioned, the abstract machine architecture is composed of machine instances—or short, instances.

4.19 DEFINITION (MACHINE INSTANCE)
A *machine instance* is a quadruple consisting of a finite map of named threads, two lists of thread names, and a finite map representing the heap of the instance.

$$\begin{array}{llr} \textbf{type}\ \mathit{Inst} = ([\mathit{NamedThread}], & \text{— thread pool} & \\ \qquad\qquad\quad [\mathit{Id}_{\text{thread}}], & \text{— DFT stack} & \\ \qquad\qquad\quad [\mathit{Id}_{\text{thread}}], & \text{— list of waiting threads} & \\ \qquad\qquad\quad [\mathit{Id}_{\text{cls}} \mapsto \mathit{Cls}]) & \text{— heap} & \blacksquare \end{array}$$

The idea is that all threads that have to be executed on an instance are contained in the thread pool of this instance. The thread pool is a list of named threads—i.e, a finite map associating thread names from Id_{thread} with threads. Each thread name is either contained in the DFT stack or in the list of waiting threads.

The *depth-first traversal stack*, which we abbreviate to DFT stack, has a similar function as the return stack in a sequential machine: For every thread, it records the predecessor of the thread. The *predecessor* or parent of a *child* thread τ is the thread τ', which is characterized as follows: It creates τ, expects to receive the result that is computed by τ, and it should be scheduled when τ terminates. Instead of a proper result value, the parent may only wait for a synchronization event from its child.

The thread whose name is on top of the DFT stack (i.e., at the front of the list) is called the *active thread*—it is currently executed by the instance. Assume that the DFT stack has the form $\delta : ds$, then the thread named δ is active. When δ needs to compute some intermediate result, which should be returned to the current computation, it will create a new thread named δ' and push its name on the DFT stack—which gives $\delta' : \delta : ds$ and makes δ' the active thread.

To cover latencies that are induced by some remote access, an instance may want to locally deviate from the standard depth-first evaluation order. This implies (i) that the current thread must be suspended (it is delayed by the remote access) and (ii) that another thread must be scheduled. To this end, we pop the active, but delayed thread from the DFT stack and add its name to the list of waiting threads (also called the *waiting list*). As a result, its predecessor will appear on top of the DFT stack and, thus, become active. Usually, this is a good choice (as already argued in Section 4.1.2). We call the threads that are occuring in the waiting list *waiting threads* and those on the DFT stack *running threads*.

But there is a second reason why a thread may enter the waiting list: It may be sent from another instance, which executes in parallel. (Actually, the sole form of communication required in HIPAR is the ability to send threads to the pool of other instances.) As a consequence, it is sometimes preferable to check for ready-to-run threads in the waiting list, instead of proceeding immediately to the predecessor thread, which is denoted by the DFT stack. This is for two reasons: First, a remote request should not have to wait too long, and second, the communication may provide a result that was previously requested from this instance. We shall return to the order of scheduling later.

4.20 NOTATION
To improve the presentation of the transition rules, we define the following notation to display instances.

$$\boxed{\Gamma \quad ds \quad ws \quad h} \triangleq (\Gamma, ds, ws, h)$$

Often, we want to represent the components of the active thread in the same display; we do this as follows:

$$\boxed{\langle \delta \bullet \gamma, as, ts \rangle \in \Gamma \quad [\delta, ds] \quad ws \quad h} \triangleq (\Gamma, \delta : ds, ws, h)$$
$$\textbf{where}$$
$$(\gamma, as, ts) = \Gamma \uparrow \delta$$

Note the choice of names for the components. The thread pool is denoted by Γ, the DFT stack by ds, the waiting list by ws, and the heap by h. Furthermore, we call the active thread δ, the currently executed code form γ, the argument stack as, and the list of targets ts. ∎

4.4.6 Configurations

Given a program, a configuration comprises the instances that are collectively evaluating the program together with the global environment defined by the functions of the program.

4.21 DEFINITION (CONFIGURATIONS)
Instances are usually associated with a name, giving *named instances*:

$$\textbf{type}\ \mathit{NamedInst} = (\mathit{Id}_{\text{inst}} \mapsto \mathit{Inst})$$

We get a *configuration* by pairing a set of named instances with a global environment:

$$\begin{array}{ll} \textbf{data}\ \mathit{Config} = \langle [\mathit{NamedInst}], & \text{— collection of instances} \\ \qquad\qquad\quad \mathit{Env} \rangle & \text{— global environment} \end{array}$$ ∎

The global environment associates the names of the variables bound by global function bindings with the closures representing these function bindings. The global environment does not change during computation and is the same for all machine instances—it represents the executed program.

4.4.7 Code Forms

The most diverse data structure in the abstract machine architecture of HIPAR is the code form. The various appearances of the code form identify different states that can be assumed by a machine instance. For example, the code form $(\mathit{Enter}\ g_c)$ signals that the instance is about to enter the closure referenced by g_c. It would be possible to represent this fact by evaluating a KCODE application of some variable, which is bound to g_c in the local environment. But this indirect representation for entering a closure—and similar representations for other actions—would disguise the essentials.

4.22 DEFINITION (CODE FORMS)

The following variants of code forms are used in the transition system.

data *Code* = *Eval Expr Env* — evaluate expression
Enter GId_{cls} — enter closure
Ret Value Cls — return value
Sync Int Env $(Env \to Code)$ — synchronize on *Sig* target
Suspend Id_{var} Id_{cls} *Env* — suspend on closure
Poll — check for ready threads
Eq Value Value — compare two values
Unif Bool GAdr GAdr — unify two boxed values
Choose Id_{var} *Env* — select an alternative
Try Choice [*Target*] *Env* — try one alternative
Match $[(Pat, Id_{\text{var}})]$ *Expr* [*Target*] *Env* — pairwise matching ∎

The precise behaviour of the code forms is defined by means of the transition system that follows in the next section, but their intuitive meaning is the following.

- *Eval* $e\ \rho$: Evaluate the KCODE expression e in the local environment ρ, which associates values with all variables that are not defined in the global environment.

- *Enter* g_c: Enter the closure referenced by g_c. (When the closure is a function, the needed number of arguments are taken from the argument stack of the thread executing the *Enter*.)

- *Ret* u *cls*: Return a value to all targets occuring in the target list of the executing thread, where u is an unboxed and *cls* a boxed version of the returned value—the latter is an actual closure, not only a reference to a closure. Sometimes, either the boxed or the unboxed version is not available, in which case we represent the missing argument with "_" for an undefined value. In this cases, there will never be a target that would need the value that is not available.

- *Sync* $n\ \rho\ \gamma$: Wait for n signals (from *Sig* targets) before applying γ to ρ. Note that *Store* targets may place additional bindings asynchronously into ρ. We call n the *synchronization counter* of the code form.

- *Suspend* $t\ g_c\ \rho$: Suspend the current thread on the closure named g_c until the thread named $\rho \uparrow t$ is completed. (This thread is always located in the executing instance.)

- *Poll*: Execute any ready-to-run thread that is mentioned in the waiting list of the executing instance before passing control to the predecessor, which is the thread that is next according to the DFT stack of the executing instance.

- *Eq* $u_1\ u_2$: Check two values for equality. (Due to type correctness, both are either boxed or unboxed values.)

- *Unif* $m\ g_1\ g_2$: Unify the two closures referenced by g_1 and g_2. When m is *True*, the closure referenced by g_2 contains the marker *Fun*.

- *Choose* $x\ \rho$: Select the alternative that is evaluated by thread $\rho\ x$.
- *Try ch ts* ρ: Evaluate the guard contained in the KCODE choice *ch*, while accumulating the variables bindings in ρ. If the guard succeeds, the targets *ts* are informed, by signaling the co-ordinating *Choose* code form.
- *Match qs e ts* ρ: Execute the pattern-matching specified by the *qs*, and accumulate the bindings in ρ. When all patterns match, *e* may be evaluated under ρ by notifying the targets *ts* about the successful matching.

Within one machine instance, we conceptually distinguish two kinds of threads in the waiting list: (i) waiting threads with *Sync* as their code form and a synchronization counter greater than zero—we call them *suspended*—and (ii) waiting threads with other code forms—we call them *ready-to-run* or simply *ready*.

4.23 DEFINITION
The function *runnable* distinguishes suspended from ready threads.

$$\begin{aligned} &\mathit{runnable} && :: \mathit{NamedThread} \rightarrow \mathit{Bool} \\ &\mathit{runnable}\ \langle \_ \bullet \mathit{Sync}\ n\ \_\ \_, \_, \_ \rangle && = n \div 0 \end{aligned}$$ ∎

4.4.8 Messages

Machine instances communicate solely by sending each other threads, which are executed in the receiving instance.

4.24 DEFINITION (MESSAGES)
A *message* that sends a named thread to a given instance is defined by

$$\mathbf{data}\ \mathit{Msg} = \mathit{NamedThread} \leadsto \mathit{Id}_{\mathrm{inst}}$$ ∎

4.5 The Transition System

The transition system that defines the concrete operational semantics of KCODE consists of three parts: (i) a function *trans* mapping a single machine instance to its successor state (if such a state exists; otherwise it is left as is), (ii) a function *partrans* mapping a configuration to its successor, and (iii) a definition of the initial and terminal states of the system.

The function

$$\mathit{trans} :: \mathit{Env} \rightarrow \mathit{NamedInst} \rightarrow \mathit{Supply} \rightarrow (\mathit{NamedInst}, [\mathit{Msg}])$$

when given a global environment, a named instance, and a name supply, produces the successor of the instance and a list of messages. The implementation of the function *trans* is discussed in detail below. For now it is sufficient to note that each transition is local to one instance and that the only way for two instances to interact is to send each other messages containing threads.

The function *partrans* :: *Config* → *Config* advances all instance in the configuration by one step, using *trans* on the individual instances. Moreover, it puts the sent threads into the receivers' thread pool. It is defined as

$$\begin{array}{ll}
\mathit{partrans} & :: \mathit{Config} \to \mathit{Supply} \to \mathit{Config} \\
\mathit{partrans}\ \langle \mathcal{I}, \sigma \rangle\ \nu & = \langle ((\mathit{foldl}\ \mathcal{I}'\ \mathit{deliver} \circ \mathit{concat})\ \mathit{msgs}), \sigma \rangle \\
& \quad \textbf{where} \\
& \qquad (\mathcal{I}', \mathit{msgs}) = \mathit{unzip}\ (\mathit{zipWith3}\ (\mathit{trans}\ \sigma)\ \mathcal{I}\ (\mathit{supplies}\ \nu))
\end{array}$$

and

$$\begin{array}{ll}
\mathit{deliver} & :: [\mathit{NamedInst}] \to \mathit{Msg} \to [\mathit{NamedInst}] \\
\mathit{deliver}\ \mathcal{I}\ (\mathit{thread} \rightsquigarrow i) \mid i \in \mathit{keys}\ \mathcal{I} & = [i \mapsto (\mathit{thread} : \Gamma, \mathit{ds}, \tau : \mathit{ws}, h)] \triangleright \mathcal{I} \\
\mid \mathit{otherwise} & = [i \mapsto ([\mathit{thread}], [], [\tau], [])] \triangleright \mathcal{I} \\
& \quad \textbf{where} \\
& \qquad \langle \tau \bullet \gamma, \mathit{as}, \mathit{ts} \rangle = \mathit{thread} \\
& \qquad (\Gamma, \mathit{ds}, \mathit{ws}, h) = \mathcal{I} \uparrow i
\end{array}$$

Note *deliver*'s treatment of messages directed to an instance that does not exist yet: It generates a new instance with the given instance name.

4.25 NOTATION
Below, when we define *trans* by specifying the transitions on individual instances, we use displays of the following form.

$$\begin{array}{ll}
 & \langle\text{original instance}\rangle \\
\text{when} & \langle\text{condition}\rangle \\
\overset{\sigma,i}{\Longrightarrow} & \langle\text{altered instance}\rangle \\
\text{and} & \langle\text{messages}\rangle
\end{array}$$

This corresponds to an equation

$$\mathit{trans}\ \sigma\ (i \mapsto \langle\text{original instance}\rangle)\ \nu \mid \langle\text{condition}\rangle = (i \mapsto \langle\text{altered instance}\rangle, \langle\text{messages}\rangle)$$

The supply ν is used to generate new names, in some rules. We avoid clutter by omitting the explicit application of *ids* at ν to generate new names. Instead, new names are simply marked by underlining in the transition rules. When the ⟨condition⟩ or the ⟨messages⟩ are empty, we omit the corresponding lines in the display completely.

To complete the definition of *trans*, we assume an additional catch-all rule of the form

$$\mathit{trans}\ \_\ \mathit{inst}\ \_ = (\mathit{inst}, [])$$

It is used when none of the transition rules that are defined in the following applies—this means, the machine instance cannot proceed. ▮

4.5.1 Initial and Terminal Configurations

Before we discuss the transition rules that determine the function *trans*, we define the initial configuration corresponding to a KCode program and have a look at the terminal configurations that may arise.

$$\mathit{initial} \qquad :: \mathit{Prog} \to \mathit{Supply} \to \mathit{Config}$$

$$\mathit{initial} \begin{bmatrix} f_1 \texttt{ = }\backslash\pi_1\ xs_1 \texttt{ -> } e_1 \\ \vdots \\ f_n \texttt{ = }\backslash\pi_n\ xs_n \texttt{ -> } e_n \end{bmatrix} \quad \nu = \langle [\underline{i} \mapsto ([\mathit{thread}], [\tau], [], h)], \sigma \rangle$$

$$\begin{array}{lll} \textbf{where} & & \\ \quad \mathit{thread} & = & \langle \underline{\tau} \bullet \mathit{Eval}\ (\texttt{main []})\ [], [], [] \rangle \\ \quad vs_1 & = & \mathrm{Fr}\ e_1 \setminus xs_1 \\ & & \vdots \\ \quad vs_n & = & \mathrm{Fr}\ e_n \setminus xs_n \\ \quad \sigma & = & [f_1 \mapsto o_1@i, \ldots, f_n \mapsto o_n@i] \\ \quad h & = & \begin{bmatrix} i@\underline{o_1} \mapsto \langle vs_1\ \backslash\pi_1\ xs_1 \to \mathit{Eval}\ e_1\ , (\sigma \uparrow vs_1), \mathit{Fun} \rangle \\ \vdots \\ i@\underline{o_n} \mapsto \langle vs_n\ \backslash\pi_n\ xs_n \to \mathit{Eval}\ e_n\ , (\sigma \uparrow vs_n), \mathit{Fun} \rangle \end{bmatrix} \end{array}$$

Figure 4.7 Initial configuration for a given KCODE program.

4.26 DEFINITION (INITIAL CONFIGURATION)
Given a KCODE program containing a global function binding that is defining the variable `main`, we get the initial configuration by the function displayed in Figure 4.7. As in the transition rules, the new variables that have been extracted from ν are underlined, as in $\underline{\tau}$. ▮

The abstract machine has *successfully terminated* when the following three conditions are met:

1. The DFT stack of all instances is empty,
2. there are no more ready threads in the waiting lists, and
3. all suspended threads contain an empty target list.

It will become clear in the discussion of the transition rules that a successfully terminating machine may include instances with suspended threads; but they will always have an empty target list. For example, a thread is created for each newly introduced unbound variable (Section 4.5.13). When no computation suspends on a unbound variable, this thread will have an empty target list and its existence does not prevent the machine from successful termination.

Apart from the fact that the machine terminated successful, no explicit result is provided by an execution. In real applications, the interaction of a program with the environment is usually performed via a library—for example, see the IO monad of Haskell 1.3 (Peterson et al. 1996). Alternatively, an interactive system may choose to regard the associations of a selected set of unbound variables as the output of a program—similar to the answer substitution in Prolog (Sterling & Shapiro 1986). In this case, the closures that are associated with these unbound variables on termination of the program represent the results. Due to lazy evaluation, it is usually necessary to demand the evaluation of these closures to obtain a printable form of the results. We refrain from a further discussion of possible user interaction models.

4.5.2 Values

A KCODE expression that only contains an integer constant, a constructor application, or a value variable denotes an unboxed value. A thread that evaluates any of these three kinds of expressions returns the unboxed value to its targets. The distribution of the values to the targets is realized by the code form *Ret*, which takes an unboxed and a boxed representation of the returned value as its arguments. This implies that a boxed representation has to be built from the unboxed value that is explicitly denoted by the program. Note that we can sometimes avoid this explicit construction of closures in the concrete implementation outlined in the next chapter—for example, when the machine currently evaluates a WHNF closure, this closure can be taken directly, instead of creating a copy.

First, we consider the evaluation of a variable x that references an unboxed integer value in the local environment ρ. (Note that the global environment σ does only store boxed values.) The unboxed integer value has the form $\mathit{Int}\ j$, and the corresponding boxed representation is a basic value closure that stores the integer value in its single environment entry.

$$\langle \delta \bullet \mathit{Eval}\ x\ \rho \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad , [], \mathit{ts}\rangle \in \Gamma \quad \cdots$$

when $\rho \uparrow x = \mathit{Int}\ j$

$$\overset{\sigma,i}{\Longrightarrow} \quad \langle \delta \bullet \mathit{Ret}\ (\rho \uparrow x)\ \langle [\underline{v}]\ \mathtt{\backslash n}\ [] \rightarrow \mathit{Eval}\ v\ , [\rho \uparrow x], \mathit{Fun}\rangle, [], \mathit{ts}\rangle \in \Gamma \quad \cdots$$

In the same way, the local environment can store an unboxed term value, which is denoted by $\mathit{Term}\ c\ \mathit{us}$. The corresponding boxed representation is a constructor closure that stores the us in its environment entries.

$$\langle \delta \bullet \mathit{Eval}\ x\ \rho \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad , [], \mathit{ts}\rangle \in \Gamma \quad \cdots$$

when $\rho \uparrow x = \mathit{Term}\ c\ \mathit{us}$ with $|\mathit{us}| \div n$

$$\overset{\sigma,i}{\Longrightarrow} \quad \langle \delta \bullet \mathit{Ret}\ (\rho \uparrow x)\ \langle [\underline{v_1}, \ldots, \underline{v_n}]\ \mathtt{\backslash n}\ [] \rightarrow \mathit{Eval}\ (c\ \mathtt{[}v_1\mathtt{,}\ \ldots\mathtt{,} v_n\mathtt{]})\ , \mathit{us}, \mathit{Fun}\rangle, [], \mathit{ts}\rangle \in \Gamma \quad \cdots$$

The second way of returning an unboxed integer is to denote an integer constant j. We utilize the above transition that handles the access to integers stored in the environment to generate the appropriate *Ret* form. To this end, the following rule stores the unboxed value in the local environment associated with a new name y.

$$\langle \delta \bullet \mathit{Eval}\ j\ \rho \qquad\qquad , [], \mathit{ts}\rangle \in \Gamma \quad \cdots$$

$$\overset{\sigma,i}{\Longrightarrow} \langle \delta \bullet \mathit{Eval}\ \underline{y}\ [y \mapsto \mathit{Int}\ j], [], \mathit{ts}\rangle \in \Gamma \quad \cdots$$

The application of a data constructor c to a list of variables xs denotes the unboxed value $\mathit{Term}\ c\ (\mathit{val}\ \rho\ \sigma\ \mathit{xs})$. We handle it by using the corresponding transition that extracts a value from the environment. The function val is as defined in Definition 4.14.

$$\langle \delta \bullet \mathit{Eval}\ (c\ \mathit{xs})\ \rho \qquad\qquad\qquad\qquad , [], \mathit{ts}\rangle \in \Gamma \quad \cdots$$

$$\overset{\sigma,i}{\Longrightarrow} \langle \delta \bullet \mathit{Eval}\ \underline{y}\ [y \mapsto \mathit{Term}\ c\ (\mathit{val}\ \rho\ \sigma\ \mathit{xs})], [], \mathit{ts}\rangle \in \Gamma \quad \cdots$$

In the above four transition rules, it is guaranteed that the argument stack is empty, for all type correct programs.

4.5.3 Application of a Primitive

Primitive functions, such as addition of integers, may only be applied to the correct number of unboxed values. We assume a function *prim* that, given the name of a primitive and its arguments, calculates the (unboxed) result. The implementation of the function *prim* depends on the supported primitives and is not further considered here.

$$\begin{aligned}
&\langle \delta \bullet \mathit{Eval}\ (p\ \mathit{xs})\ \rho \qquad\qquad\qquad\quad,\ [],\ \mathit{ts}\rangle \in \Gamma \quad \cdots \\
\overset{\sigma,i}{\Longrightarrow}\ &\langle \delta \bullet \mathit{Eval}\ \underline{y}\ [y \mapsto \mathit{prim}\ p\ (\rho \uparrow \mathit{xs})],\ [],\ \mathit{ts}\rangle \in \Gamma \quad \cdots
\end{aligned}$$

The result, which may be an unboxed integer or an unboxed term, is treated as in the previous rules.

4.5.4 Function Applications

We handle function application according to the the push-enter model, which we discussed in Section 4.2. Hence, the application of a function f to a set of argument identifiers *xs* has two duties: First, it must push the values that are associated with the arguments *xs* in the local environment ρ or global environment σ on the argument stack; and second, it has to enter the closure that is associated with f. Depending on the kind of the closure and the number of available arguments, the behaviour of the machine instance while entering a closure is manifold. Therefore, the dedicated code form *Enter* is employed to mark the state of a machine instance that is about to enter a closure. It gets the global address of the entered closure as its only argument.

$$\begin{aligned}
&\langle \delta \bullet \mathit{Eval}\ (f\ \mathit{xs})\ \rho \qquad,\qquad\qquad\qquad \mathit{as},\ \mathit{ts}\rangle \in \Gamma \quad \cdots \\
\overset{\sigma,i}{\Longrightarrow}\ &\langle \delta \bullet \mathit{Enter}\ (\mathit{val}\ \rho\ \sigma\ f),\ (\mathit{val}\ \rho\ \sigma\ \mathit{xs}) +\!\!+\ \mathit{as},\ \mathit{ts}\rangle \in \Gamma \quad \cdots
\end{aligned}$$

4.5.5 Local Bindings

The treatment of local function bindings is very similar to the handling of global bindings in constructing the initial configuration (see Figure 4.7). But in the case of local function bindings, the new associations have to be entered into the local environment. The corresponding transition rule is displayed in Figure 4.8.

As explained, value bindings specify independent computations that have to be completed before the body expression of the enclosing `letpar` can be executed. The following transition rule enforces this synchronization by placing the evaluation of the body expression into a *Sync* form that waits for as many signals as value bindings are present. Each of the value bindings is realized by an independent thread, which on completion stores the result in the local environment of the mentioned *Sync* form.

The threads that evaluate the value bindings are placed on top of the DFT stack. Thus, each of them is processed until the thread δ containing the *Sync* is activated again. When all bindings were processed to completion, the synchronization counter is zero, and the body expression can be executed. However, when a value binding is suspended, due to a long latency operation, the threads of the other bindings are still processed and may cover the latency. In this case, the previously suspended binding will be completed before the thread with the body

$$\begin{array}{llllll} \langle \delta \bullet \mathit{Eval} \begin{pmatrix} \texttt{letrec} \\ \quad f_1 \texttt{ = } \backslash\pi_1 \; xs_1 \texttt{ -> } e_1 \\ \quad \vdots \\ \quad f_n \texttt{ = } \backslash\pi_n \; xs_n \texttt{ -> } e_n \\ \texttt{in } e \end{pmatrix} & \rho,\ as,\ ts\rangle \in \Gamma & \delta : ds & ws & h \\ \overset{\sigma,i}{\Longrightarrow} \ \langle \delta \bullet \mathit{Eval}\ e\ \rho' & ,\ as,\ ts\rangle \in \Gamma & \delta : ds & ws & h' \end{array}$$

where

$$\begin{array}{lll} vs_1 & = & \mathrm{Fr}\, e_1 \setminus xs_1 \\ & \vdots & \\ vs_n & = & \mathrm{Fr}\, e_n \setminus xs_n \\ \rho' & = & [f_1 \mapsto i@o_1, \ldots, f_n \mapsto i@o_n] \rhd \rho \\ h' & = & \begin{bmatrix} i@\underline{o_1} \mapsto \langle vs_1 \ \backslash\pi_1 \ xs_1 \to \mathit{Eval}\ e_1 \,,\, (\rho\ vs_1),\, \mathit{Fun}\rangle \\ \vdots \\ i@\underline{o_n} \mapsto \langle vs_n \ \backslash\pi_n \ xs_n \to \mathit{Eval}\ e_n \,,\, (\rho\ vs_n),\, \mathit{Fun}\rangle \end{bmatrix} \rhd h \end{array}$$

Figure 4.8 Transition rule for local function bindings.

expression is tried—otherwise, the thread δ with the body expression also suspends. (In this rule, $\underline{ds'}$ represents a list of fresh names.)

$$\begin{array}{llllll} & \langle \delta \bullet \mathit{Eval}\ (\texttt{letpar}\ bs\ \texttt{in}\ e)\ \rho,\ as,\ ts\rangle \in \Gamma & & \delta : ds & ws & h \\ \overset{\sigma,i}{\Longrightarrow} & \langle \delta \bullet \mathit{Sync}\ |bs|\ \rho\ (\mathit{Eval}\ e\ ) & ,\ as,\ ts\rangle \in \Gamma' & ds' +\!\!+ (\delta : ds) & ws & h \end{array}$$

where

$$\Gamma' = [\langle \delta' \bullet \mathit{Eval}\ e\ \rho, [], [\mathit{Store}\ \delta@i\ x]\rangle \mid (x \texttt{ = } e, \delta') \leftarrow \mathit{zip}\ bs\ \underline{ds'}] \rhd \Gamma$$

4.5.6 Entering Local Closures

There are two kinds of closures, updatable and nonupdatable. When a machine instance enters a closure, its behaviour depends largely on the kind of this closure. When a nonupdatable closure is entered, we again discriminate two cases: either there is a sufficient number of arguments on the argument stack or not. (Such a distinction is not necessary for updatable closures, as they never expect any arguments.)

Nonupdatable Closures

The first, and simplest, case that we consider is a thread that enters a nonupdatable closure while there is a sufficient number of arguments on the stack. The code of the closure is evaluated in the context of a local environment, which is constructed from the arguments and the valuation of the free variables that is stored in the closure. More precisely, as many arguments as required by the closure are popped from the argument stack (below these are the as'); they are associated with the argument identifiers xs. Moreover, the identifiers of the free variables vs are associated with the valuation us that is stored in the environment section of the closure. The closure code γ is applied to the resulting local environment $(xs \mapsto as) +\!\!+ (vs \mapsto us)$.

$$\langle \delta \bullet \mathit{Enter}\ o@i \qquad\qquad\qquad ,\ as' +\!\!+ as,\ ts\rangle \in \Gamma \quad \delta : ds \quad ws \quad h$$

$$\text{when } h \uparrow o = \langle vs\ \texttt{\textbackslash n}\ xs \rightarrow \gamma, u, \mathit{Fun}\rangle \quad \text{with} \quad |as'| \div |xs|$$

$$\overset{\sigma,i}{\Longrightarrow} \quad \langle \delta \bullet \gamma\ ((xs \mapsto as') +\!\!+ (vs \mapsto us)), \qquad as,\ ts\rangle \in \Gamma \quad \delta : ds \quad ws \quad h$$

In the second case, matters are getting more involved. The fact that the arguments stack is not able to supply a sufficient number of arguments indicates the evaluation of a partial function application, which has to be communicated to the targets of the current thread. Similar to a thread that returns an unboxed value, the code form *Ret* is employed, but without any information in its first argument—we indicate this by the symbol "_". The boxed representation of a partial application is a closure—*cls* in the rule below—which stores the function (i.e., the currently entered closure) together with the already supplied arguments in its environment. Furthermore, it contains a lambda-form representing the partial application.

$$\langle \delta \bullet \mathit{Enter}\ o@i,\ as,\ ts\rangle \in \Gamma \quad \delta : ds \quad ws \quad h$$

$$\text{when } h \uparrow o = \langle vs\ \texttt{\textbackslash n}\ (xs +\!\!+ xs') \rightarrow \gamma, us, \mathit{Fun}\rangle \quad \text{with} \quad |as| \div |xs|, \text{ and } |as|, |xs'| \geq 1$$

$$\overset{\sigma,i}{\Longrightarrow} \quad \langle \delta \bullet \mathit{Ret}\ \_\ \mathit{cls} \quad ,\ [],\ ts\rangle \in \Gamma \quad \delta : ds \quad ws \quad h$$

where

$$\mathit{cls} = \langle (\underline{f} : xs)\ \texttt{\textbackslash n}\ [] \rightarrow \mathit{Eval}\ (f\ xs)\ , o@i : as, \mathit{Fun}\rangle$$

Note that the type correctness of the evaluated program guarantees that none of the targets *ts* expects to get an unboxed value.

Although the previous rule would correctly cover the case of an empty argument stack in the sequential case, it is not sufficient when the partial application is demanded by a remote instance. In fact, the closure that is named o has to be returned, instead of the new closure *cls*, which is a mere indirection whenever the argument stack is empty.

$$\langle \delta \bullet \mathit{Enter}\ o@i \quad ,\ [],\ ts\rangle \in \Gamma \quad \delta : ds \quad ws \quad h$$

$$\text{when } h \uparrow o = \langle vs\ \texttt{\textbackslash n}\ xs \rightarrow \gamma, us, \mathit{Fun}\rangle \quad \text{with} \quad |xs| \geq 1$$

$$\overset{\sigma,i}{\Longrightarrow} \quad \langle \delta \bullet \mathit{Ret}\ \_\ (h \uparrow o),\ [],\ ts\rangle \in \Gamma \quad \delta : ds \quad ws \quad h$$

Updatable Closures

Next we discuss the second kind of closures, namely updatable ones. The evaluation of such a closure is more costly, because it has to be overwritten with a closure representing the result after it is evaluated. To this end, a target of the form *Upd* $o@i$ is added to the target list, before the code of the closure is executed. When eventually a *Ret* form processes this target, the second argument of the *Ret* form is used to update the closure at address $o@i$; so that subsequent access to $o@i$ yields the result without repeated evaluation of the original, updatable closure. But there is an additional complication. While the closure is evaluated and before it is updated, concurrent threads may enter the closure and so redo some of the work. This would compromise the sharing that should be achieved by the updating mechanism. We avoid this effect by overwriting the closure with a closure that contains a *Suspend* code form when it is entered for the first time. The details of the *Suspend* code form are discussed later. The sketched scheme is directly applicable when there are no arguments on the stack.

$$\langle \delta \bullet \mathit{Enter}\ o@i \quad , [], \qquad\qquad ts\rangle \in \Gamma \quad \delta : ds \quad ws \quad h$$

when $h \uparrow o = \langle vs\ \texttt{\textbackslash u}\ [] \rightarrow \gamma, us, \mathit{Fun}\rangle$

$$\xRightarrow{\sigma,i} \quad \langle \delta \bullet (\gamma\ (vs \mapsto us)), [], (\mathit{Upd}\ o@i) : ts\rangle \in \Gamma \quad \delta : ds \quad ws \quad h'$$

where

$$h' = [o \mapsto \langle [\underline{t}]\ \texttt{\textbackslash n}\ [] \rightarrow \mathit{Suspend}\ t\ o@i\ , [\delta@i], \mathit{Fun}\rangle] \triangleright h$$

The behaviour is slightly more involved if arguments are on the stack when a thread enters an updatable closure. Assuming type correctness of the executed program, the closure that is just entered will turn out to be a function, which eventually consumes the arguments on the stack. But the current closure must not be updated with the result of applying that function to the arguments, but with the function itself. To prevent the function from consuming the arguments, a new thread δ' with an empty argument stack is created, and the closure is entered by this new thread. As a result, the original closure is updated with a closure constituting a partial application that represents the mentioned function.

As a consequence, the old thread δ has to re-enter the closure, with the arguments on the stack, after the update; so that the final result of applying the function to the arguments is computed and, eventually, distributed to the targets *ts*. To this end, the current thread assumes a *Sync* code form that re-executes an *Enter* $o@i$ as soon as it is signaled by the target *Sig* $\delta@i$, which is placed into the newly created thread δ'.

$$\langle\ \delta \bullet \mathit{Enter}\ o@i \qquad\qquad , as, \qquad ts\rangle \in \Gamma \qquad \delta : ds \quad ws \quad h$$

when $h \uparrow o = \langle vs\ \texttt{\textbackslash u}\ [] \rightarrow \gamma, us, \mathit{Fun}\rangle$ and $as \neq []$

$$\xRightarrow{\sigma,i} \quad \begin{array}{l} \langle\ \delta \bullet \mathit{Sync}\ 1\ []\ (\mathit{const}\ (\mathit{Enter}\ o@i)), as, \qquad ts\rangle \\ \langle \underline{\delta'} \bullet \mathit{Enter}\ o@i \qquad\qquad\qquad , \ [], [\mathit{Sig}\ \delta@i]\rangle \end{array} \in \Gamma \quad \delta' : \delta : ds \quad ws \quad h$$

4.5.7 Entering Remote Closures

When an instance i attempts to enter a closure o located on another machine instance j, it cannot do so directly. Instead, the instance i sends a message to j, which contains a thread with the code form *Enter* $o@j$.

Depending on the number of arguments that are on the return stack when the attempt to enter the remote closure is made, the behaviour is different. In the first case, when the argument stack is empty, the thread that is sent to j also includes the targets *ts* that are waiting for the result of evaluating the given closure.

$$\langle \delta \bullet \mathit{Enter}\ o@j, [], ts\rangle \in \Gamma \quad \cdots$$

$$\xRightarrow{\sigma,i} \langle \delta \bullet \mathit{Poll} \qquad , [], []\rangle \in \Gamma \quad \cdots$$

and $\langle \underline{\tau} \bullet \mathit{Enter}\ o@j, [], [\mathit{Remote}\ i\ ts]\rangle \rightsquigarrow j$

As will be discussed in the next chapter, it is not necessary to actually copy *ts* to another PE in a distributed implementation, instead a forwarding reference is used that transfers the result back to i, and then, distributes it to the targets. Hence, the *Remote* target can be regarded as abstract form of this remote reference.

In the second case, when there are some arguments on the stack, the remote instance evaluates to a function (this is related to the previous case where an updatable closure is entered with a nonempty argument stack). At this point, we may proceed in two different ways. Either we transfer the arguments from i to j and let the function application be evaluated on j. Or else, we keep the arguments on instance i and transmit the function—as a closure representing a partial application—back to i and, then, re-enter it, in the context of the arguments. We choose the second alternative, to avoid the transmission of the arguments.

We proceed similar as in the case of an updatable closure with arguments on the stack. The current thread δ assumes a *Sync* code form and waits for the result of the remote evaluation of the closure. The new thread δ', then, enters the closure without any arguments, but two targets. The first target is $\mathit{Upd}\ o'@i$, which stores the resulting partial application in the heap of instance i using a new heap location o'. And the second target is $\mathit{Sig}\ \delta@i$, which wakes the waiting thread δ, thereby initiating the evaluation of the partial application in the context of the arguments as.

$$\begin{array}{llr} \langle\, \delta \bullet \mathit{Enter}\ o@j & ,\ \mathit{as}, & \mathit{ts}\rangle \end{array} \in \Gamma \qquad \delta : \mathit{ds} \quad \mathit{ws} \quad h$$

when $\mathit{as} \neq []$

$$\xRightarrow{\sigma,i} \left.\begin{array}{llr} \langle\, \delta \bullet \mathit{Sync}\ 1\ []\ (\mathit{const}\ (\mathit{Enter}\ o'@i)) & ,\ \mathit{as}, & \mathit{ts}\rangle \\ \langle \underline{\delta'} \bullet \mathit{Enter}\ o@j & ,\ [], & [\mathit{Upd}\ \underline{o}'@i, \mathit{Sig}\ \delta@i]\rangle \end{array}\right. \in \Gamma \quad \delta' : \delta : \mathit{ds} \quad \mathit{ws} \quad h$$

4.5.8 Suspension

As previously discussed, updatable closures, on first entry, are overwritten with a closure containing a *Suspend* code form. Hence, any thread that subsequently enters the closure suspends this operation until the closure is, eventually, updated with the result, which is computed by the thread that initially entered the closure.

The current thread suspends by assuming a *Sync* code form, which waits until an appropriate *Sig* target is executed. The latter is entered into the list of targets of the thread that is currently evaluating the closure, which in turn is identified by the *Suspend* code form—by $\rho \uparrow t = \tau@i$ in the rule below. A new thread δ', which includes a *Poll* code form, is executed to check for responses from other instances, which may return information that is necessary to compute the closure g. The meaning of the *Poll* form will be explained later in detail.

$$\begin{array}{ll} \langle\, \delta \bullet \mathit{Suspend}\ t\ g\ \rho & ,\ \mathit{as},\ \mathit{ts}\rangle \end{array} \in \Gamma \qquad \delta : \mathit{ds} \quad \mathit{ws} \quad h$$

when $\rho \uparrow t = \tau@i$ with $\langle \tau \bullet \gamma, \mathit{as}_t, \mathit{ts}_t\rangle \in \Gamma$

$$\xRightarrow{\sigma,i} \left.\begin{array}{ll} \langle\, \delta \bullet \mathit{Sync}\ 1\ []\ (\mathit{const}\ (\mathit{Enter}\ g)) & ,\ \mathit{as},\ \mathit{ts}\rangle \\ \langle \delta' \bullet \mathit{Poll} & ,\ [],\ []\rangle \end{array}\right. \in \Gamma' \quad \delta' : \delta : \mathit{ds} \quad \mathit{ws} \quad h$$

where

$$\Gamma' = [\langle \tau \bullet \gamma, \mathit{as}_t, (\mathit{Sig}\ \delta@i) : \mathit{ts}_t\rangle] \triangleright \Gamma$$

Note that the thread specified by $(\rho \uparrow t)$ is always local—we use a global address to identify the thread to keep the values that are stored in the arguments of closures uniform.

4.5.9 Returning a Value

When the active thread returns a result, using the *Ret* code form, the result value is distributed to the targets mentioned in the target list of the thread. Afterwards, the thread terminates, which re-activates its predecessor—as determined by the DFT stack.

A target of the form *Upd* $o@i$ commands the executing thread to update the local closure named o with the boxed value that is contained in the *Ret* code form.

$$\begin{array}{lllll} & \langle \delta \bullet \mathit{Ret}\ u\ \mathit{cls},\ [],\ (\mathit{Upd}\ o@i) : \mathit{ts}\rangle \in \Gamma & \delta : \mathit{ds} & \mathit{ws} & h \\ \overset{\sigma,i}{\Longrightarrow} & \langle \delta \bullet \mathit{Ret}\ u\ \mathit{cls},\ [],\ \mathit{ts}\rangle \in \Gamma & \delta : \mathit{ds} & \mathit{ws} & [o \mapsto \mathit{cls}] \triangleright h \end{array}$$

A target of the form *Sig* $\tau@i$ requires that the local thread τ is signaled. The thread τ is bound to have a *Sync* code form. Signaling decrements the synchronization counter of the *Sync* form (i.e., its first argument) by one.

$$\begin{array}{lll} & \langle \delta \bullet \mathit{Ret}\ u\ \mathit{cls},\ [],\ (\mathit{Sig}\ \tau@i) : \mathit{ts}\rangle \in \Gamma & \cdots \\ \text{when} & \langle \tau \bullet \mathit{Sync}\ n\ \rho\ \gamma,\ \mathit{as}_s,\ \mathit{ts}_s\rangle \in \Gamma & \\ \overset{\sigma,i}{\Longrightarrow} & \langle \delta \bullet \mathit{Ret}\ u\ \mathit{cls},\ [],\ \mathit{ts}\rangle \in \Gamma' & \cdots \\ \text{where} & & \\ & \Gamma' = [\langle \tau \bullet \mathit{Sync}\ (n-1)\ \rho\ \gamma,\ \mathit{as}_s,\ \mathit{ts}_s\rangle] \triangleright \Gamma & \end{array}$$

A target of the form *Store* $\tau@i\ x$ implies that the returned unboxed value has to be stored in the local thread τ; as before, this thread will contain a *Sync* code form. The following rule, which handles this case, adds an association to the local environment ρ contained in the *Sync* form, such that this association maps x to the returned unboxed value. After the value has been stored, the target behaves as a *Sig* target.

$$\begin{array}{lll} & \langle \delta \bullet \mathit{Ret}\ u\ \mathit{cls},\ [],\ (\mathit{Store}\ \tau@i\ x) : \mathit{ts}\rangle \in \Gamma & \cdots \\ \text{when} & \langle \tau \bullet (\mathit{Sync}\ n\ \rho\ \gamma),\ \mathit{as}_t,\ \mathit{ts}_t\rangle \in \Gamma & \\ \overset{\sigma,i}{\Longrightarrow} & \langle \delta \bullet \mathit{Ret}\ u\ \mathit{cls},\ [],\ (\mathit{Sig}\ \tau@i) : \mathit{ts}\rangle \in \Gamma' & \cdots \\ \text{where} & & \\ & \Gamma' = [\langle \tau \bullet (\mathit{Sync}\ n\ ([x \mapsto u] \triangleright \rho)\ \gamma),\ \mathit{as}_t,\ \mathit{ts}_t\rangle \triangleright \Gamma & \end{array}$$

A target of the form *Remote* $j\ ts'$ sends a thread that contains the currently executed *Ret* form and the remote targets ts' to the remote instance j.

$$\begin{array}{lll} & \langle \delta \bullet \mathit{Ret}\ u\ \mathit{cls},\ [],\ (\mathit{Remote}\ j\ \mathit{ts}') : \mathit{ts}\rangle \in \Gamma & \cdots \\ \overset{\sigma,i}{\Longrightarrow} & \langle \delta \bullet \mathit{Ret}\ u\ \mathit{cls},\ [],\ \mathit{ts}\rangle \in \Gamma & \cdots \\ \text{and} & \langle \underline{\tau} \bullet \mathit{Ret}\ u\ \mathit{cls},\ [],\ \mathit{ts}'\rangle \rightsquigarrow j & \end{array}$$

Finally, after all targets have been processed, the current thread completed. But before it passes control to the predecessor of the completed thread, as specified by the DFT stack, the code form *Poll* is employed to look for ready threads in the waiting list.

$$\begin{array}{lll} & \langle \delta \bullet \mathit{Ret}\ u\ \mathit{cls},\ [],\ []\rangle \in \Gamma & \cdots \\ \overset{\sigma,i}{\Longrightarrow} & \langle \delta \bullet \mathit{Poll}\ \ \ ,\ [],\ []\rangle \in \Gamma & \cdots \end{array}$$

Polling is important at this point, because the predecessor may wait for some signal that is supplied by a ready thread, which is contained in the waiting list. In this case, returning to the predecessor before checking the waiting list would lead to a superfluous suspension. But even if this is not the case, the waiting list may contain a thread that obtains some local data for another instance—processing such threads as quickly as possible reduces the latency for other instances.

4.5.10 Thread Scheduling

A *Sync* code form waits until its synchronization counter is zero. Then, the code γ contained in the last argument of *Sync* is applied at the local environment ρ, and the resulting code form is executed.

$$\langle \delta \bullet \mathit{Sync}\; 0\; \rho\; \gamma,\; \mathit{as},\; \mathit{ts}\rangle \in \Gamma \quad \cdots$$

$$\overset{\sigma,i}{\Longrightarrow} \langle \delta \bullet \gamma\; \rho \qquad\quad ,\; \mathit{as},\; \mathit{ts}\rangle \in \Gamma \quad \cdots$$

If the synchronization counter is greater than zero, the machine instance suspends the active thread δ by placing it into the waiting list. Afterwards, it inspects the waiting list for ready-to-run threads by issuing a *Poll*.

$$\langle\, \delta \bullet \mathit{Sync}\; n\; \rho\; \gamma,\; \mathit{as},\; \mathit{ts}\rangle \in \Gamma \quad \delta : \mathit{ds} \qquad\quad \mathit{ws} \quad h$$

when $n \geq 1$

$$\overset{\sigma,i}{\Longrightarrow} \quad \langle \underline{\delta'} \bullet \mathit{Poll} \qquad\quad ,\;\; [],\;\; []\rangle \in \Gamma \quad \delta' : \mathit{ds} \quad \delta : \mathit{ws} \quad h$$

If all threads on the DFT stack have been processed, the waiting list may still contain work. In fact, it may asynchronously acquire work from other instances that send threads to its thread pool. Therefore, an instance looks for work in the waiting list by generating a new thread containing a *Poll* command, whenever its DFT stack is empty.

$$\Gamma \quad\; [] \quad \mathit{ws} \quad h$$

$$\overset{\sigma,i}{\Longrightarrow} \langle \underline{\delta} \bullet \mathit{Poll},\; [],\; []\rangle \in \Gamma \quad [\delta] \quad \mathit{ws} \quad h$$

The code form *Poll* copies the names of all ready threads from the waiting list on top of the DFT stack. As mentioned in Section 4.4.7, the threads in the waiting list are either ready-to-run or suspended; a *Sync* code form with a synchronization counter greater than zero identifies the suspended threads.

$$\langle \delta \bullet \mathit{Poll},\; [],\; []\rangle \in \Gamma \qquad\qquad \delta : \mathit{ds} \qquad\qquad \mathit{ws} \quad h$$

$$\overset{\sigma,i}{\Longrightarrow} \qquad\qquad \Gamma \quad (\mathit{keys}\; \mathit{rs}) \mathbin{+\!\!+} \mathit{ds} \quad (\mathit{keys}\; \mathit{ws}') \quad h$$

where

$$(\mathit{rs}, \mathit{ws}') = \mathit{partition}\; \mathit{runnable}\; (\mathit{ws} \mapsto (\Gamma \uparrow \mathit{ws}))$$

The function *runnable* was introduced in Definition 4.23.

4.5.11 Case Selection

The execution of a `case` expression selects a case alternative in dependence on the inspected unboxed value. The following rule handles `case` expressions that are scrutinizing integer values. It considers the case where a matching alternative is found.

$$\langle \delta \bullet \mathit{Eval}\ (\texttt{case}\ x\ \texttt{of}\ (cs_1 \mathbin{+\!\!+} [j\ \texttt{->}\ e] \mathbin{+\!\!+} cs_2))\ \rho,\ as,\ ts\rangle \in \Gamma \quad \cdots$$

when $\rho \uparrow x = \mathit{Int}\ j$

$$\xRightarrow{\sigma,i} \langle \delta \bullet \mathit{Eval}\ e\ \rho \qquad\qquad ,\ as,\ ts\rangle \in \Gamma \quad \cdots$$

The variable x cannot be in σ, as it is unboxed. If there is no appropriate alternative, the default case applies. If there is no default case, the instance is stuck, because there is no matching transition rule. This situation indicates a runtime error.

$$\langle \delta \bullet \mathit{Eval}\ (\texttt{case}\ x\ \texttt{of}\ [i_1\ \texttt{->}\ e_1, \ldots, i_n\ \texttt{->}\ e_n, \texttt{dft}\ \texttt{->}\ e])\ \rho,\ as,\ ts\rangle \in \Gamma \quad \cdots$$

when $\rho \uparrow x = \mathit{Int}\ j$ and $j \notin [i_1, \ldots, i_n]$

$$\xRightarrow{\sigma,i} \langle \delta \bullet \mathit{Eval}\ e\ \rho \qquad\qquad ,\ as,\ ts\rangle \in \Gamma \quad \cdots$$

The execution of `case` expressions that scrutinize terms proceeds similarly. But in the case of a matching alternative, the variables vs in the matching pattern $c\ vs$ have to be associated with the corresponding arguments us of the term.

$$\langle \delta \bullet \mathit{Eval}\ (\texttt{case}\ x\ \texttt{of}\ (cs_1 \mathbin{+\!\!+} [c\ vs\ \texttt{->}\ e] \mathbin{+\!\!+} cs_2))\ \rho,\ as,\ ts\rangle \in \Gamma \quad \cdots$$

when $\rho \uparrow x = \mathit{Term}\ c\ us$

$$\xRightarrow{\sigma,i} \langle \delta \bullet \mathit{Eval}\ e\ ((vs \mapsto us) \triangleright \rho) \qquad\qquad ,\ as,\ ts\rangle \in \Gamma \quad \cdots$$

The default case is, again, taken when no other alternative fits.

$$\langle \delta \bullet \mathit{Eval}\ (\texttt{case}\ x\ \texttt{of} \begin{bmatrix} c_1\ vs_1 & \texttt{->}\ e_1, \\ & \vdots \\ c_n\ vs_n & \texttt{->}\ e_n, \\ \texttt{dft} & \texttt{->}\ e \end{bmatrix})\ \rho,\ as,\ ts\rangle \in \Gamma \quad \cdots$$

when $\rho \uparrow x = \mathit{Term}\ d\ us$ and $d \notin [c_1, \ldots, c_n]$

$$\xRightarrow{\sigma,i} \langle \delta \bullet \mathit{Eval}\ e\ \rho \qquad\qquad ,\ as,\ ts\rangle \in \Gamma \quad \cdots$$

4.5.12 Success, Failure, and Parallel and Sequential Composition

We start the discussion of the transition rules for the co-ordinating forms with `succ`. It signals successful completion by returning with a *Ret* code form. In contrast to other expressions, there is no proper return value, but still a closure must be provided that can be used when an *Upd* target demands the updating of a closure with the returned value. We use a success closure, which contains `succ` in the code of its lambda-form.

$$\langle \delta \bullet \mathit{Eval}\ \texttt{succ}\ \rho \qquad\qquad ,\ [],\ ts\rangle \in \Gamma \quad \cdots$$

$$\xRightarrow{\sigma,i} \langle \delta \bullet \mathit{Ret}\ \_\ \langle []\ \texttt{\textbackslash n}\ [] \rightarrow \mathit{Eval}\ \texttt{succ}\ ,\ [],\ \mathit{Fun}\rangle,\ [],\ ts\rangle \in \Gamma \quad \cdots$$

Note that it is not sufficient just to terminate the thread executing a `succ`, because *ts* may contain *Sig* and *Upd* targets. There is no transition rule covering the evaluation of `fail`—i.e., the machine stalls on execution of this command, indicating failure.

The evaluation of a parallel composition e_1 `||` e_2 creates a new machine instance for the evaluation of e_2 by sending a suitable thread to a new instance named j. This message creates the new instance—see the definition of the function *deliver* at the beginning of this section. The expression e_1 is evaluated locally in a new thread δ'. The thread δ, which initiated the evaluation of the two subexpressions e_1 and e_2, synchronizes on the completion of the new threads, which evaluate the subexpressions—this is necessary to realize sequential composition, as we will see below.

$$\begin{array}{lllll}
 & \langle\, \delta \bullet \mathit{Eval}\ (e_1\ \texttt{||}\ e_2)\ \rho \quad , [], \quad ts\rangle \in \Gamma & \delta : ds & ws & h \\[1ex]
\overset{\sigma,i}{\Longrightarrow} & \begin{array}{l}\langle\, \delta \bullet \mathit{Sync}\ 2\ []\ (\mathit{Eval}\ \texttt{succ}\ ), [], \quad ts\rangle \\ \langle \underline{\delta'} \bullet \mathit{Eval}\ e_1\ \rho \quad , [], [\mathit{Sig}\ \delta @ i]\rangle\end{array} \in \Gamma_i & \delta' : \delta : ds & ws & h
\end{array}$$

and $\langle \underline{\tau} \bullet \mathit{Eval}\ e_2\ \rho, [], [\mathit{Remote}\ i\ [\mathit{Sig}\ \delta @ i]]\rangle \rightsquigarrow \underline{j}$

The co-domain of ρ are values, which are either unboxed values or global heap addresses (representing boxed values)—both can easily be copied to the new instance. When the new instance attempts to enter a closure referenced in ρ, it will trigger one of the two transition rules for entering remote closures.

In a sequential composition e_1 `=>` e_2, the evaluation of e_2 has to wait for the completion of e_1, which is realized with a *Sync* code form.

$$\begin{array}{lllll}
 & \langle\, \delta \bullet \mathit{Eval}\ (e_1\ \texttt{=>}\ e_2)\ \rho \quad , [], \quad ts\rangle \in \Gamma & \delta : ds & ws & h \\[1ex]
\overset{\sigma,i}{\Longrightarrow} & \begin{array}{l}\langle\, \delta \bullet \mathit{Sync}\ 1\ \rho\ (\mathit{Eval}\ e_2\ ), [], \quad ts\rangle \\ \langle \underline{\delta'} \bullet \mathit{Eval}\ e_1\ \rho \quad , [], [\mathit{Sig}\ \delta @ i]\rangle\end{array} \in \Gamma & \delta' : \delta : ds & ws & h
\end{array}$$

When considering the case where e_1 is a parallel composition, it becomes clear that the *Sync* code form is necessary in the transition rule for parallel composition. Note the similarity between this mechanism and the empty messages employed in the linear logic semantics of the co-ordination constructs in Section 3.4.

4.5.13 Unbound Variables

As described in Section 4.3.3, unbound variables are implemented by a special form of closure, which can be recognized by the marker *UB*. The lambda-form of such a closure suspends any thread that attempts to enter the closure—in a similar way as updatable closures that were already entered, but not yet updated with their normal form. Computations that suspend on entering such a closure have to be reactivated after the unbound variable is instantiated. Therefore, the following transition rule creates a new thread ω that is stored in the closure and records all threads that have to be reactivated in its target list—the *Suspend* code form performs the recording. The rules that realize unification and are discussed below signal ω when the unbound variable is instantiated. The execution of ω, in turn, reactivates all tasks that suspended while entering o.

$$\begin{array}{lllll}
 & \langle\, \delta \bullet \mathit{Eval}\ (x : e)\ \rho \qquad\qquad , [], \mathit{ts}\rangle \in \Gamma & \delta : \mathit{ds} & \mathit{ws} & h \\
\overset{\sigma,i}{\Longrightarrow} & \begin{array}{l}\langle \underline{\omega} \bullet \mathit{Sync}\ 1\ []\ (\mathit{const}\ (\mathit{Ret}\ \_\ \_)), [], []\rangle \\ \langle\, \delta \bullet \mathit{Eval}\ e\ ([x \mapsto o] \triangleright \rho) \quad , [], \mathit{ts}\rangle\end{array} \in \Gamma & \delta : \mathit{ds} & \omega : \mathit{ws} & h'
\end{array}$$

where

$$h' = [\underline{o} \mapsto \langle [\underline{t}]\ \backslash\mathtt{n}\ [] \rightarrow \mathit{Suspend}\ t\ o@i\ , [\omega@i], \mathit{UB}\rangle] \triangleright h$$

The target list of ω collects only *Sig* targets due to the transition rule for *Suspend*. This allows the use of the code $(\mathit{Ret}\ \_\ \_)$ in ω.

4.5.14 Equational Constraints

The arguments of equational constraints are variables; they reference either unboxed or boxed values. In the case of unboxed values, the values are compared. If they are not equal, the computation `fails`. If they are equal integers, the constraint is successfully solved; but if they are terms, the arguments of the terms have to be compared pairwise by equational constraints. In the case of boxed values, one or both arguments may be an unbound variable, which is instantiated by the constraint.

To implement this complex behaviour, we employ two further code forms, namely *Eq* and *Unif*. An equational constraint is first rewritten into an *Eq* code form. When it constraints boxed values, the unification procedure that is realized by *Unif* is invoked by *Eq*. Hence, the evaluation of an expression x_1 `:=:` x_2 first leads to an *Eq* code form.

$$\begin{array}{lllll}
 & \langle\, \delta \bullet \mathit{Eval}\ (x_1\ \mathtt{:=:}\ x_2)\ \rho \qquad , [], \mathit{ts}\rangle \in \Gamma & \delta : \mathit{ds} & \mathit{ws} & h \\
\overset{\sigma,i}{\Longrightarrow} & \begin{array}{l}\langle\, \delta \bullet \mathit{Eval}\ \mathtt{succ}\ [] \qquad\qquad , [], \mathit{ts}\rangle \\ \langle \delta' \bullet \mathit{Eq}\ (\mathit{val}\ \rho\ \sigma\ x_1)\ (\mathit{val}\ \rho\ \sigma\ x_2), [], []\rangle\end{array} \in \Gamma & \delta' : \delta : \mathit{ds} & \mathit{ws} & h
\end{array}$$

The thread δ reports success to the context of the computation, i.e., other threads that are waiting for the completion of this computation do not have to wait until the comparison that is initiated by δ' is finished (it may get suspended). This behaviour conforms to the operational semantics of $\mathcal{D}$, and realizes “eventual tell” (Saraswat 1993).

We first consider the comparison of unboxed values by *Eq*. In the case of two equal integers, the comparison succeeds, and the thread δ ceases.

$$\begin{array}{ll}
 & \langle \delta \bullet \mathit{Eq}\ (\mathit{Int}\ j_1)\ (\mathit{Int}\ j_2), [], []\rangle \in \Gamma \quad \cdots
\end{array}$$

when $j_1 \doteqdot j_2$

$$\overset{\sigma,i}{\Longrightarrow} \qquad\qquad\qquad \Gamma \quad \cdots$$

In the case of two matching terms, the constructors are equal and both terms have the same number of arguments, which have to be compared pairwise. Therefore, a new thread δ' is created for each pair of arguments u_1 and u_2. The new threads employ an *Eq* code form. (The where clause below uses a Haskell list comprehension.)

$$\begin{array}{lllll}
 & \langle \delta \bullet \mathit{Eq}\ \mathit{Term}\ c\ \mathit{us}_1\ \mathit{Term}\ c\ \mathit{us}_2, [], []\rangle \in \Gamma & \delta : \mathit{ds} & \mathit{ws} & h \\
\overset{\sigma,i}{\Longrightarrow} & \qquad\qquad\qquad\qquad \Gamma' & \mathit{ds}' +\!\!+\ \mathit{ds} & \mathit{ws} & h
\end{array}$$

where

$$\Gamma' = [\langle \delta' \bullet \mathit{Eq}\ u_1\ u_2, [], []\rangle \mid (u_1, u_2, \delta') \leftarrow \mathit{zip3}\ \mathit{us}_1\ \mathit{us}_2\ \underline{\mathit{ds}'}] \triangleright \Gamma$$

There are no transition rules that cover the case where two different unboxed values are compared. As with `fail`, the instance stalls in this case, which indicates failure.

The second group of transitions that are concerned with the code form *Eq* deals with boxed values. The treatment of boxed values is complicated by three facts.

- The two closures associated with the boxed values may be in the heap of two different instances.
- Forwarding closures (they contain the marker *Fwd*) have to be followed, to check whether a boxed value represents an unbound variable.
- When a thread identified a closure as an unbound variable, the same thread has to instantiate the variable, to avoid race conditions.

Consequently, *Eq* and *Unif* code forms move to the instance where the closure that is identified by their first argument is stored, and they chase forwarding closures. More precisely, *Eq* follows forwarding closures and moves to different instances until it finds the closure that is at the end of the forwarding chain referenced by its first argument. Then, it mutates into a *Unif* code form. The next rule forwards a thread with an *Eq* code form, which is not on the same instance as the closure referenced by its first argument $o_1@j_1$.

$$\langle \delta \bullet \mathit{Eq}\ o_1@j_1\ o_2@j_2,\ [],\ []\rangle \in \Gamma \quad \cdots$$

when $j_1 \neq i$

$$\overset{\sigma,i}{\Longrightarrow} \qquad \Gamma \quad \cdots$$

and $\langle \delta \bullet \mathit{Eq}\ o_1@j_1\ o_2@j_2,\ [],\ []\rangle \rightsquigarrow j_1$

A thread with an *Eq* code form chases along any forwarding closure that is referenced by its first argument—such a closure is identified by a *Fwd* marker. But when the closure represents an unbound variable or a function binding, the code form mutates into an *Unif* code form.

$$\langle \delta \bullet \mathit{Eq}\ o_1@i\ o_2@j_2,\ \ [],\ \ []\rangle \in \Gamma \quad \cdots$$

when $h \uparrow o_1 = \langle \mathit{lf}, \mathit{us}, \mu\rangle$

$$\overset{\sigma,i}{\Longrightarrow} \quad \langle \delta \bullet \gamma \qquad\qquad ,\ \mathit{as},\ \mathit{ts}\rangle \in \Gamma \quad \cdots$$

where

$$\gamma = \textbf{if}\ \mu = \mathit{Fwd}\ \textbf{then}\ \mathit{Eq}\ (\mathit{head}\ \mathit{us})\ o_2@j_2\ \textbf{else}\ \mathit{Unif}\ \mathit{False}\ o_2@j_2\ o_1@i$$

The resulting *Unif* code form gets the two arguments of *Eq* in reversed order. Furthermore, it gets a boolean value as its first argument; this value indicates whether *Unif*s second argument is guaranteed to refer to a closure with a *Fun* marker. Similar to *Eq*, a *Unif* form chases along any chain of forwarding closures that is referenced by its second argument. Thus, the exchange of the order of the arguments guarantees that forwarding chains of both boxed values are followed.

However, there remains another special case; both boxed values may reference the same unbound variable—we call this the *alias case*. In the alias case, unification must succeed without instantiating the unbound variable. We have to be careful to detect all occurrences of this case, because a concurrent thread may instantiate an unbound variable, while the thread in consideration is chasing along a forwarding chain. Fortunately, there is a solution to this problem that avoids expensive locking of data structures. The *Eq* code form has chased any forwarding chain referenced by its first argument until it either encountered a function binding or an unbound variable. So in the alias case, the other boxed value either references the same closure or a forwarding closure that points (maybe indirectly) to this closure. Together with the swapping of arguments in the previous rule, it is guaranteed that both arguments reference the same closure at some point during the chasing performed by the *Unif* form. This situation is spotted by the next rule, which terminates unification successfully.

$$\langle \delta \bullet \mathit{Unif}\ m\ o@j\ o@j),\ [],\ []\rangle \in \Gamma \quad \cdots$$

$$\overset{\sigma,i}{\Longrightarrow} \qquad \Gamma \quad \cdots$$

The second rule for *Unif* moves the code form to the instance where the closure is located and that is referenced by its second argument $o_1@j_1$.

$$\langle \delta \bullet \mathit{Unif}\ m\ o_1@j_1\ o_2@j_2),\ [],\ []\rangle \in \Gamma \quad \cdots$$

when $j_1 \neq i$

$$\overset{\sigma,i}{\Longrightarrow} \qquad \Gamma \quad \cdots$$

and $\langle \delta \bullet \mathit{Unif}\ m\ o_1@j_1\ o_2@j_2), [], []\rangle \rightsquigarrow j_1$

The third rule realizes the chasing along forwarding closures.

$$\langle \delta \bullet \mathit{Unif}\ m\ o_1@i\ o_2@j_2,\ [],\ []\rangle \in \Gamma \quad \cdots$$

when $h \uparrow o_1 = \langle \mathit{lf}, [o@j], \mathit{Fwd}\rangle$

$$\overset{\sigma,i}{\Longrightarrow} \quad \langle \delta \bullet \mathit{Unif}\ m\ o@j\ o_2@j_2\ ,\ [],\ []\rangle \in \Gamma \quad \cdots$$

It remains to consider threads with a *Unif* code form whose second argument references a closure that is marked by *Fun* or *UB*. When we have a *Fun* closure, the behaviour depends on the boolean flag in the first argument of *Unif*. If it is *True*, the closure referenced by the third argument is also a *Fun* closure; otherwise, we swap the second and third argument while setting the flag to *True*—because we know that, after the swapping, the third argument is a *Fun* closure. This is done by the next rule.

$$\langle \delta \bullet \mathit{Unif}\ \mathit{False}\ o_1@i\ o_2@j_2,\ [],\ []\rangle \in \Gamma \quad \cdots$$

when $h \uparrow o_1 = \langle \mathit{lf}, \mathit{us}, \mathit{Fun}\rangle$

$$\overset{\sigma,i}{\Longrightarrow} \quad \langle \delta \bullet \mathit{Unif}\ \mathit{True}\ o_2@j_2\ o_1@i\ ,\ [],\ []\rangle \in \Gamma \quad \cdots$$

But if the flag is already *True*, we know that both closures represent function bindings. We unify them by obtaining the unboxed values of the closures and using an appropriate equational constraint on these values. To this end, the instance creates two new threads δ_1 and δ_2 that are evaluating the two closures. The thread δ synchronizes on these evaluations and, when they return, compares the unboxed results.

$$\langle \delta \bullet \mathit{Unif}\ \mathit{True}\ o_1@i\ o_2@j_2\quad , [], []\rangle \in \Gamma \qquad\qquad \delta : ds \quad ws \quad h$$

when $h \uparrow o_1 = \langle \mathit{lf}, \mathit{us}, \mathit{Fun}\rangle$

$$\xRightarrow{\sigma,i} \quad \langle \delta \bullet \mathit{Sync}\ 2\ []\ \mathit{Eval}\ (y_1\ \texttt{:=:}\ y_2)\ , [], []\rangle \in \Gamma' \quad \delta_1 : \delta_2 : \delta : ds \quad ws \quad h$$

where

$$\Gamma' = \begin{bmatrix} \langle \delta_1 \bullet \mathit{Enter}\ o_1@i, [], [\mathit{Store}\ \delta@i\ \underline{y_1}]\rangle, \\ \langle \delta_2 \bullet \mathit{Enter}\ o_2@j_1, [], [\mathit{Store}\ \delta@i\ \underline{y_2}]\rangle \end{bmatrix} \triangleright \Gamma$$

It may appear odd to swap the two arguments again, because they have been swapped already by the transit from *Eq* to *Unif*. But consider the case where the first argument of the *Eq* form references an unbound variable, which is concurrently instantiated by some other thread, while the *Unif* form chases along a forwarding chain referenced by the other value.

Finally, when the second argument of *Unif* references a closure that represents an unbound variable (it is marked by *UB*), the closure is updated by a forwarding closures as described in Section 4.3.3.

$$\langle \delta \bullet \mathit{Unif}\ m\ o_1@i\ o_2@j_2, [], \qquad\qquad []\rangle \in \Gamma \quad \cdots$$

when $h \uparrow o_1 = \langle [t]\ \texttt{\textbackslash n}\ [] \to \gamma, [\tau@j], \mathit{UB}\rangle$

$$\xRightarrow{\sigma,i} \quad \langle \delta \bullet \mathit{Ret}\ \_\ \mathit{cls} \qquad\qquad , [], [\mathit{Upd}\ o_1@i, \mathit{Sig}\ \tau@j]\rangle \in \Gamma \quad \cdots$$

where

$$\mathit{cls} = \langle [\underline{y}]\ \texttt{\textbackslash n}\ [] \to \mathit{Eval}\ (y\ \texttt{[]})\ , [o_2@j_2], \mathit{Fwd}\rangle]$$

The *Ret* form is used to perform the update and to signal the task τ, which is associated with the unbound variable (see Section 4.5.13).

4.5.15 Committed Choice

The evaluation of a committed choice construct creates a new thread δ' for each of its alternatives *chs*. Each of these threads is responsible for checking the guards of one alternative; to this end, it employs the code form *Try*. The thread δ, which evaluates the committed choice, assumes a *Choose* code form, to co-ordinate the selection of one alternative. This co-ordination is important when the alternatives are overlapping, as only the body expression of the first alternative that successfully completes the evaluation of its guards may be taken.

$$\langle \delta \bullet \mathit{Eval}\ (\texttt{choose}\ \mathit{chs})\ \rho\ , [], ts\rangle \in \Gamma \qquad\qquad \delta : ds \quad ws \quad h$$

$$\xRightarrow{\sigma,i} \quad \langle \delta \bullet \mathit{Sync}\ 1\ []\ (\mathit{Choose}\ \underline{x}\ ), [], \ []\rangle \in \Gamma' \quad ds' \mathbin{+\!\!+} (\delta : ds) \quad ws \quad h$$

where

$$\Gamma' = [\langle \delta' \bullet \mathit{Try}\ ch\ [\mathit{Store}\ \delta@i\ x]\ \rho, [], ts\rangle \mid (ch, \delta') \leftarrow \mathit{zip}\ \mathit{chs}\ \underline{ds'}] \triangleright \Gamma$$

A unique selection is achieved as follows. A thread that succeeds in checking its guard does not directly proceed to the evaluation of the body expression of the corresponding alternative. Instead, it signals its success to the co-ordinating *Choose* code form, after placing the name of the thread that contains the body expression into the environment ρ of the co-ordinating thread. When the *Choose* is executed due to the signal, it allows the thread that contains the body expression to run by sending a signal. In this process, which is formalized in the next rule, the *Choose* disappears, so no further alternative can be taken.

$$\begin{array}{ll} & \langle \delta \bullet \mathit{Choose}\ x\ \rho,\ [],\ \qquad\qquad []\rangle \in \Gamma \quad \cdots \\ \overset{\sigma,i}{\Longrightarrow} & \langle \delta \bullet \mathit{Ret}\ \_\ \_\ \qquad,\ [],\ [\mathit{Sig}\ (\rho \uparrow x)]\rangle \in \Gamma \quad \cdots \end{array}$$

A *Try* code form is responsible for checking the guard of one alternative of a `choose`. The check comprises two phases: First the right-hand side expressions e' of all pattern-matching equations $p := e'$ are evaluated; and second, the results of this evaluations are matched against the patterns p. The evaluation is realized by new threads δ', one for each right-hand side expression. The results of the evaluations are communicated to a *Sync* code form that contains a *Match* form. The latter performs the pattern-matching after the *Sync* form has received the result of all expressions e'. This is very similar to the treatment of `letpar`; especially, the evaluation of the right-hand side expressions is independent of each other, and can cover latencies.

$$\begin{array}{llllll} & \langle \delta \bullet \mathit{Try}\ (\mathit{ms}\ \texttt{->}\ e)\ \mathit{ts}'\ \rho & ,\ [],\ \mathit{ts}\rangle \in \Gamma & \delta : \mathit{ds} & \mathit{ws} & h \\ \overset{\sigma,i}{\Longrightarrow} & \langle \delta \bullet \mathit{Sync}\ |\mathit{ms}|\ \rho\ (\mathit{Match}\ (\mathit{zip}\ \mathit{ps}\ \mathit{xs})\ e\ \mathit{ts}'\ ) & ,\ [],\ \mathit{ts}\rangle \in \Gamma' & \mathit{ds}' +\!\!+ (\delta : \mathit{ds}) & \mathit{ws} & h \end{array}$$

where

$$\mathit{ps} = [p \mid (p\ := \_) \leftarrow m]$$

$$\Gamma' = [\langle \delta' \bullet \mathit{Eval}\ e'\ \rho, [], [\mathit{Store}\ \delta@i\ x]\rangle \mid (\_ := e', \delta', x) \leftarrow \mathit{zip3}\ \mathit{ms}\ \underline{\mathit{ds}'}\ \underline{\mathit{xs}}] \triangleright \Gamma$$

The code form *Match* maintains a list of pattern-value pairs that have to be matched. When all matches have been successfully completed, the co-ordinating *Choose* is notified, as explained above. The body expression e of this `choose` alternative waits for a signal from the *Choose*, which will subsequently arrive when this alternative is the first one that signaled success to the *Choose*.

$$\begin{array}{lllll} & \langle\ \delta \bullet \mathit{Match}\ []\ e\ \mathit{ts}'\ \rho\ \quad,\ \mathit{as},\ \ \mathit{ts}\rangle \in \Gamma & \delta : \mathit{ds} & \mathit{ws} & h \\ \overset{\sigma,i}{\Longrightarrow} & \left.\begin{array}{l}\langle\ \delta \bullet \mathit{Sync}\ 1\ \rho\ (\mathit{Eval}\ e\ ),\ []\ ,\ \ \mathit{ts}\rangle \\ \langle \delta' \bullet \mathit{Ret}\ \delta'@i\ \_ \qquad\quad ,\ []\ ,\ \mathit{ts}'\rangle\end{array}\right. \in \Gamma & \delta' : \delta : \mathit{ds} & \mathit{ws} & h \end{array}$$

If an integer j is matched against an identical integer pattern, the pair can be removed from the list.

$$\begin{array}{ll} & \langle \delta \bullet \mathit{Match}\ [(\mathit{Int}\ j, x), \mathit{qs}]\ e\ \mathit{ts}'\ \rho,\ [],\ \mathit{ts}\rangle \in \Gamma \quad \cdots \\ \text{when } \rho \uparrow x = \mathit{Int}\ j & \\ \overset{\sigma,i}{\Longrightarrow} & \langle \delta \bullet \mathit{Match}\ \mathit{qs}\ e\ \mathit{ts}'\ \rho \qquad\qquad\quad ,\ [],\ \mathit{ts}\rangle \in \Gamma \quad \cdots \end{array}$$

Otherwise, if the integer pattern contains a different value, the alternative fails, and the active thread is removed.

$$\begin{array}{lllll} & \langle \delta \bullet \mathit{Match}\ [(\mathit{Int}\ i, x), \mathit{qs}]\ e\ \mathit{ts}'\ \rho,\ [],\ \mathit{ts}\rangle \in \Gamma & \delta : \mathit{ds} & \mathit{ws} & h \\ \text{when } \rho \uparrow x \neq \mathit{Int}\ i & & & & \\ \overset{\sigma,i}{\Longrightarrow} & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \Gamma & \mathit{ds} & \mathit{ws} & h \end{array}$$

If a term *Term c us* is matched against a constructor pattern *Term c vs*, the argument variables *us* are associated pairwise with the values *us* in the local environment ρ.

$$\langle \delta \bullet \mathit{Match}\ [(\mathit{Term}\ c\ \mathit{xs}, y), \mathit{qs}]\ e\ \mathit{ts}'\ \rho,\ [],\ \mathit{ts}\rangle \in \Gamma \quad \cdots$$

when $\rho \uparrow y = \mathit{Term}\ c\ \mathit{us}$

$$\overset{\sigma,i}{\Longrightarrow} \quad \langle \delta \bullet \mathit{Match}\ \mathit{qs}\ e\ \mathit{ts}'\ ((\mathit{xs} \mapsto \mathit{us}) \rhd \rho)\quad,\ [],\ \mathit{ts}\rangle \in \Gamma \quad \cdots$$

Again, if the term does not match the constructor pattern, the alternative fails and the active thread disappears.

$$\langle \delta \bullet \mathit{Match}\ [(\mathit{Term}\ c\ \mathit{xs}, y), \mathit{qs}]\ e\ \mathit{ts}'\ \rho,\ [],\ \mathit{ts}\rangle \in \Gamma \quad \delta : \mathit{ds} \quad \mathit{ws} \quad h$$

when $\rho \uparrow y = \mathit{Term}\ d\ \mathit{us}$ and $c \neq d$

$$\overset{\sigma,i}{\Longrightarrow} \qquad\qquad \Gamma \qquad \mathit{ds} \quad \mathit{ws} \quad h$$

Finally, a variable pattern always fits and places an association of the variable v with the value of x into the local environment.

$$\langle \delta \bullet \mathit{Match}\ [(y, x), \mathit{qs}]\ e\ \mathit{ts}'\ \rho \qquad\quad ,\ [],\ \mathit{ts}\rangle \in \Gamma \quad \cdots$$

$$\overset{\sigma,i}{\Longrightarrow} \langle \delta \bullet \mathit{Match}\ \mathit{qs}\ e\ \mathit{ts}'\ ([y \mapsto (\rho \uparrow x)] \rhd \rho),\ [],\ \mathit{ts}\rangle \in \Gamma \quad \cdots$$

4.6 Experimental Results

As mentioned above, the formal specification of the transition system of HIPAR constitutes a Haskell program. I have used the specification as the basis for a prototypical implementation of the abstract machine, where I have replaced the naive implementation of finite maps (Definition 4.10) by more efficient array- and tree-based variants. Furthermore, I have used a monadic coding style to modularize the implementation and to allow for updatable arrays and a flexible output of traces.

In this prototype, the amount of parallelism that the execution of a KCODE program allows is determined by calculating two measures, namely the *depth* and the *work* complexity. These measures are, for example, proposed by Blelloch (1996); they correspond to the critical path and the overall running time, respectively, of a program. We call one application of the function *partrans* a *parallel step*; it advances all machine instances that have an active task by one step. The number of parallel steps that are needed to execute a program gives the depth—the program cannot be faster, regardless of the present physical resources. The work complexity of a program is the sum of the number of instances that have work over all parallel steps. In the extreme case, where in each parallel step only a single instance can actually perform a transition, work and depth are equal, and we have a sequential program. The ratio between work and depth is a measure for the parallelism in a program.

When the ability of a program to hide latency is bad, the amount of parallelism that it exhibits will be low when remote access is frequent. Repeated remote access suspends all threads, and consequently, only few instances can proceed in a parallel step. The introduction of the `letpar` construct was justified by the claim that it is useful to tolerate the latency induced by remote accesses. Hence, it should be possible to observe the beneficial effect of `letpar`s by studying the ratio between the work and depth of KCODE programs.

As an example, consider the definition of *farm* from Section 2.2.3. (The following code is GOFFIN, not KCODE, as the former is easier to read.)

$$\begin{array}{ll} \mathit{farm} & :: (\alpha \to \beta) \to [\alpha] \to [\beta] \to O \\ \mathit{farm}\ f\ []\ r & = \{r \leftarrow []\} \\ \mathit{farm}\ f\ (x : xs)\ r & = \{ys\ \mathbf{in}\ \ r \leftarrow (f\ x) : ys, \\ & \qquad\qquad\quad \mathit{farm}\ f\ xs\ ys\} \end{array}$$

Consider that *farm* is used with $\mathit{farm}\ f\ l\ l'$. Due to the parallel composition, in the second equation of *farm*, and the operational interpretation of parallel composition in HIPAR, all applications of f to some element of the list l are performed on a different machine instance. Hence, they potentially execute in parallel.

Furthermore, assume the following function definition for *sum*, which adds all elements of a list of integers.

$$\begin{array}{ll} \mathit{sum} & :: [\mathit{Int}] \to \mathit{Int} \\ \mathit{sum}\ [] & = 0 \\ \mathit{sum}\ (x : xs) & = x + \mathit{sum}\ xs \end{array}$$

The evaluation of an expression $\mathit{sum}\ l$ traverses l sequentially. Hence, if we compose *farm* and *sum* into

$$\{\mathit{result}, l'\ \mathbf{in}\ \mathit{farm}\ f\ l\ l', \mathit{result} \leftarrow \mathit{sum}\ l'\}$$

we will not be able to observe much parallelism. Haskell, and so its extension GOFFIN, are lazy languages. Although *farm* unrolls, and imposes a constraint $r \leftarrow f\ x : ys$ for every x in l, the actual evaluation of the $f\ x$ will not be carried out until it the result is needed. When *sum* traverses its input list l', the remote access to the elements of l' triggers the evaluation of the $f\ x$, while *sum* is waiting.

Experiments with the implementation of the abstract machine confirmed this theoretical considerations. In a first experiment, all `letpar`s were restricted to a single value binding; consequently, no latency hiding by independent threads was possible. The observed work and depth complexity are almost identical in this case.

To see the potential for latency hiding, we have look at the realization of the GOFFIN + function in KCODE.

```
add = \n [a, b] -> letpar
                     au = a []
                   in letpar
                     bu = b []
                   in
                     addPrim [au, bu]
```

Here, `addPrim` is the primitive function that adds two unboxed integers. The remaining code unboxes the two boxed integers `a` and `b`, to be able to call the primitive `addPrim`. The attempt to unbox `a` is what causes the evaluation of the applications $f\ x$ in the *farm*/*sum* example. Both unboxing operations are realized by an independent thread in the abstract machine, but following the semantics of `letpar`, the second thread has to wait for the completion of the first.

By means of the source-to-source transformation displayed in Equation (4.2) (in Section 4.3.4), we may rewrite the definition of `add` to

```
add = \n [a, b] -> letpar
                     au = a []
                     bu = b []
                   in
                     addPrim [au, bu]
```

This simple transformation changes the behaviour of the *farm*/*sum* example drastically. The second argument to `add` is the recursive call to *sum*. Thus, the execution of `b []`, while waiting for the demand on `a`, triggers the demand for the next list element, and so forth. In effect, the evaluation of all elements of the list l' (the argument of *sum*) is triggered in rapid order. Given that the function f, which was passed to *farm*, is sufficiently work intensive (otherwise, the use of *farm* would have been unskillful in the first place), the evaluation of the applications execute largely in parallel. The results of a second experiment with the *farm*/*sum* example, using the transformed version of `add`, supported the above reasoning. The result was a work/depth ratio proportional to the length of the processed list—for example, a list with 32 elements produced a ratio of 31 when f takes around 3800 function calls to complete.

Experiments with slightly more complex examples showed the same result. The optimizations from Section 4.3.4 improve the ability of the code to hide latencies. However, it should be clear that these results are no guarantee for a good performance of this abstract machine on concrete parallel hardware, but they are an important precondition. Furthermore, they show that the design of the abstract machine, on this level of abstraction, leads to the expected behaviour.

4.7 Discussion

Compiled graph-reduction, the key technology for the efficient implementation of lazy functional languages, comes with considerable technical detail. The addition of multi-threading naturally increases this technicality. Consequently, it becomes difficult to overview the complete design of an abstract machine and the subtle interaction of all components. In this situation a rigorous, formal specification—as the one that has been provided here—comes in helpful. Although the formality may make the initial reading more strenuous, it answers by its very nature all questions that arise in a continuous study or, even more important, in an attempt to implement a compiler that is based on the abstract machine. Furthermore, when the specification formalizes the behaviour of the abstract machine in an executable language, the possibility to run the specification allows to test, debug, and understand it in every detail. This was very useful in the development of HIPAR, and I recommend it for future designs of abstract machines.

The main novelty of the abstract machine design presented here is the tight integration of multi-threading into the process of graph reduction. Function activations are generalized to thread activations, and thread scheduling is performed by the program itself, no external scheduler is necessary, as it is in other designs (for example, Trinder, Hammond, Mattson Jr, Partridge & Peyton Jones 1996). Scheduling without the need for an external scheduler is called *self-scheduling*. In connection with the ability of the compiler to control the scheduling by program transformations (see Section 4.6), we speak of *compiler-controlled* self-scheduling. This avoids the overhead that is introduced by an external scheduler and supports an adaption of the scheduling process to the executed program. The discussed approach distinguishes between threads, which are used to tolerate the latency of remote access, and tasks, which are created by parallel composition and which represent true parallelism. The result is a *scheduling hierarchy*, where light-weight threads are scheduled locally on a PE, and coarse-grain tasks may migrate to other PEs. This distinction reflects the need for fast contexts switches between threads and the wish to ameliorate the overhead of task migration over the work load induced by a task.

Furthermore, we saw that the management of unbound variables can be realized by the thread-scheduling mechanism. This includes the creation and instantiation of unbound variables as well as the suspension of reduction on unbound variables (i.e., deferred evaluation). Instead of managing explicit lists of suspended processes, as in other approaches, we associate an unbound variables with a thread. The target list of this thread collects signal targets for computations that suspend on the unbound variable (refer to Section 4.5.13). This integration of the management of unbound variables and threads is, to the best of my knowledge, new.

4.7.1 Related Work

There are many attempts to the implementation of graph reduction on distributed-memory parallel computers[9]; among these are the parallel implementation of the STGM (Peyton Jones, Clack & Salkild 1989, Trinder et al. 1996), the PAM (Loogen, Kuchen, Indermark & Damm 1989), the ABC-machine (Nöcker, Smetsers, van Eekelen & Plasmeijer 1991), and the HDG-machine (Kingdon, Lester & Burn 1991). None of them integrates multi-threading into the design of the abstract machine. Instead, the same tasks[10] that are used to express true parallelism—that is, those that may be executed by other processors—are used to cover the latency of remote operations. The price are high context switching costs. Nevertheless, it is interesting to note that the design of the parallel graph-reduction machine Alice (Darlington & Reeve 1981) already included the use of multi-threading for latency hiding. It was employed to hide long latency memory accesses.

Compiler-controlled multi-threading is central to the TAM (Culler et al. 1993). The TAM implements lenient evaluation, which means that functions are non-strict, but they are evaluated eagerly. This evaluation strategy originates from dataflow computing and exposes all parallelism contained in a program. The result are fine-grained computations that need dynamic scheduling. The compilation technology that comes with the TAM clusters these fine-grained computations, as far as possible, statically into threads (Schauser, Culler & Goldstein 1995). These threads are scheduled dynamically within function activations. The TAM reduces context switching times by register allocation across threads, a scheduling hierarchy, and the use of active messages (von Eicken, Culler, Goldstein & Schauser 1992). The results are encouraging (Culler et al. 1995), but there remain problems due to the inherently asynchronous model for function activations and the fact that every function activation may migrate to another PE. This precludes conventional techniques for passing arguments on a stack or in registers and induces high memory costs for the allocation of the activation frames. In particular, the TAM

---

[9] We do not consider implementations on shared-memory or on special hardware here. See Hammond's (1994) survey for a general overview.

[10] Sometimes these tasks are called threads, but this is just a matter of terminology.

completely abandons the depth-first traversal that is usually employed in the implementation of programming languages.

Acknowledging these problems, Goldstein et al. (1996) developed their techniques further. Independently, they arrived at a stack-based architecture that has many things in common with HIPAR. In particular, they use a conventional stack to traverse the call-graph in depth-first order and deviate from this order only when remote access or other latency makes this necessary. But there is an important reason that makes it difficult to use their approach for the implementation of graph reduction. They build on the assumption that the size of a function activation does not change after the activation record has been allocated. First, this presumes tail-call optimization, which is regarded as important in compiled graph-reduction. And second, they cannot employ the push-enter model (Section 4.2.2), as this model makes it impossible to determine statically the maximal number of arguments on the stack in the presence of polymorphic functions and separate compilation.

StackThreads are proposed by Taura, Matsuoka & Yonezawa (1994) as an abstract machine for programming languages that are based on fine-grained parallelism. As the proposal of Goldstein et al. (1996), *StackThreads* follow the sequential depth-first traversal as close as possible; only when necessary are synchronous procedure calls transformed into asynchronous calls. Both proposal correspond in their use of threads to realize not only tolerance to latency, but also true parallelism.

Blumofe et al. (1995) argue for the multi-threaded runtime-system Cilk, which deviates from the usual depth-first traversal of the call tree only as far as this is necessary to tolerate latency and to exploit parallelism. But they do not distinguish between threads that are meant to keep a PE busy while waiting for the completion of a remote access and true parallelism. An interesting aspect of their work is that it comes with a formal performance model, and they proved that their scheduler keeps within the bounds of this model for space, time, and communication complexity for an important class of programs.

A further formal model for scheduling was presented by Blelloch et al. (1995), who advocate p-DFT schedules for the implementation of fine-grained parallelism. These schedules are related to the schedules realized by HIPAR, in the sense that both stick as close to the sequential depth-first traversal as possible. But the objective of Blelloch et al. is different, as they consider this schedule for the exploitation of true parallelism, whereas HIPAR uses fine-grain parallelism to tolerate communication latency.

The integration of multi-threading with graph reduction as described here is more elegant than my earlier proposal of the multi-threaded STGM, called STGM$_{\text{MT}}$ (Chakravarty 1996). This becomes obvious when comparing the transition system of the STGM$_{\text{MT}}$ with that of Section 4.5. In particular, the message dissemination mechanism of the STGM$_{\text{MT}}$ is replaced by the simpler target lists, and the use of global addresses leads to a simpler representation and access of remote closures when compared with the *EnterOn* code form of the STGM$_{\text{MT}}$. Unlike KCODE, the machine language of the STGM$_{\text{MT}}$ does not support agents, but is purely functional and uses a special `letrem` construct to create new machine instances. It is an easy task to give a purely functional rendering of the improved mechanisms that were presented here, and so, to get an improved STGM$_{\text{MT}}$.

Starting from the STGM$_{\text{MT}}$, Horn (1996) proposed an alternative integration of multi-threading into the STGM. It avoids the separation of demand and selection (we discussed it in Section 4.3.3), and instead, introduces an additional `Delay` alternative in all `case` expressions that have only a single case. As a result, the stack management is closer to the original STGM, but opportunities for increasing the latency tolerance in `case` expressions with more than one alternative cannot be used.

4.7.2 Future Work

An obviously interesting task for future work is an attempt to establish the correctness of the KCODE transition system with respect to the semantics that was defined by the mapping of KCODE into $\mathcal{D}$. Although a direct comparison between the operational semantics of $\mathcal{D}$ and the transition system of KCODE is most certainly not feasible, the gap between both may be closed by stepwise refinement. An approach that has been taken by Sestoft (1997) for the purely functional case.

Another direction of future work is to extend the machine with support for the additions that have been discussed for $\mathcal{D}$ in Section 3.6.2. Most of the comment made there apply here also; the discussion is not repeated here.

Furthermore, additional experiments with the abstract machine and an indepth study of the possible optimizations by source-to-source transformations are definitely worthwhile. This should also include studying the interaction between the optimizations performed on KCODE and optimizations at later stages of the compilation process.

Chapter 5

CONCRETE IMPLEMENTATIONS

THE final step in the implementation of declarative parallelism is the mapping of the abstract machine model to concrete parallel hardware. The parallel architectures considered in the following can be characterized as follows: state-of-the-art RISC processors with large local memory coupled by a high-speed interconnection network that has a uniform processor-to-processor latency. Examples of such architectures are the Cray T3E and also the Fujitsu AP-3000 and IBM SP-2. The per processor computing power of such machines is comparable to high-end workstations. This is beneficial for the user because already a small number of processors provides impressing computing power—a 24 processor Cray T3E has a peak performance of 14.4 GFLOPS. But on the other hand, high-speed processors enlarge the gap between the per PE processing speed and the speed of the interconnection network; hence, it is hard to fully exploit the processing power of the machine in communication intensive applications. Consider, for example, that the processors of a Cray T3E (they are DEC alpha chips) operate at 300 MHz, whereas the point-to-point latency of an 8 byte message is around 2μs.[1]

A consequence of this discrepancy between PE speed and network latency is the need to overlap communication with computation as well as communication with other communication operations. This means that a PE after a remote request for data does not wait idly until the result is returned, but meanwhile performs some independent computation or communication. It is already difficult to amortize expensive communication operations in this way in applications with a regular and statically determined communication structure, but it can be achieved by careful program design. In the case of irregular and highly dynamic communication patterns, appropriate runtime mechanisms have to be employed. As already mentioned in previous chapters, the efficient implementation of declarative languages critically depends on effective ways to tolerate latency in largely unpredictable communication patterns.

As has been noted by Culler et al. (1995), the main benefit of dataflow computing on the implementation level is fast, data-driven scheduling. In other words, the technology of dataflow computing teaches us to select, in an efficient way, the appropriate local computations, in dependence on the data that is currently locally available. The move of the dataflow community away from pure dataflow hardware to hybrid architectures (for example, Nikhil, Papdopoulos & Arvind 1992, Ang, Arvind & Chiou 1995) or even stock-hardware (for example, Culler et al. 1995, Hum et al. 1995) has been focusing on this virtue. The integration of multi-threading into compiled graph-reduction that was discussed in the last chapter was

---

[1] This is when the low overhead, vendor supplied SHMEM library is used.

inspired by this development. The present chapter continues in this spirit towards an efficient implementation of HIPAR on distributed-memory multi-processors. In particular, the communication primitives are based on active messages (von Eicken et al. 1992) and multi-threading is employed to overlap communication and computation. The following covers the most important aspects of the mapping of the abstract machine to stock hardware, but many of the details are not discussed.

5.1 Concrete Instances

The machine architecture of HIPAR defines a configuration as a set of machine instances and a global environment (see Section 4.4). A program creates instances dynamically and usually in large numbers. Although this was a convenient abstraction in the last chapter, it does not fit well to the comparatively small and fixed number of PEs that a parallel computer contains. We introduce the concept of concrete instances to bridge this gap.

5.1.1 Virtual Instances Versus Concrete Instances

Henceforth, we call machine instances as they have been introduced in the last chapter *virtual instances*; they are an abstract concept, but do not appear as such in a concrete implementation. Instead, we implement *concrete instances*—one per PE—where each performs the work of arbitrarily many virtual instances.

The data structures of all those virtual instances mapped to a single concrete instance are merged into one set of structures. Consequently, virtual instances do not even exist conceptually in the implementation, but their identity is replaced by that of a concrete instance. In particular, an instance identifier references one concrete instance, but does not distinguish between the virtual instances that are realized by this concrete instance.

5.1.2 Lazy Task Creation

The actual assignment of virtual to concrete instances is not fixed, but determined at runtime. Virtual instances are created by the transition rule that processes parallel composition (Section 4.5.12)—one virtual instance is created for every right-hand side of a parallel composition. The one-to-one correspondence between concrete instances and physical PEs makes the distribution of virtual over concrete instances into a problem of proper load balance. Previous work on parallel implementations of dynamically-scheduled languages (Goldstein et al. 1996, Trinder et al. 1996) suggests a lazy work distribution scheme, where work packages are communicated to other PEs only on request. This strategy is often called *work stealing*—idle processors steal from processors that have excess work.

This scheme requires the maintenance of work packages that are processed locally or stolen, in dependence on the load distribution in the system. We call such work packages *tasks*; the execution of a parallel composition creates a task for the subexpression on the right-hand side. Mohr, Kranz & Halstead, Jr. (1991) have demonstrated that it is preferable to avoid the eager creation of tasks, and to defer task creation until another PE actually wants to steal a task. They have called this strategy *lazy task creation*. In this scheme, explicit task creation is substituted by a conventional function call along with the maintenance of a *task queue*. The potential task lies dormant on the stack until either (i) the function call returns and inlines the execution of the task or (ii) the task is stolen by an idle PE. In the first case, the creation of an independent task is avoided and the only overhead is the maintenance of the task queue. In the second case, the stealing operation has to ensure that the function call, which eventually returns, finds a stack layout that reflects the new situation. In fact, Mohr et al. (1991) have gone into considerable technical difficulty to ensure a correct manipulation of the stack.

Fortunately, the multi-threaded nature of HIPAR allows a comparatively simple implementation of lazily created tasks. The implementation is further simplified by the fact that a task always realizes an agent—which does not return a result, only synchronization with the context is important. Consider the execution of a KCODE expressions of the form $e_1 \mid\mid e_2$, according to the transitions in Section 4.5.12. The transition system creates two new threads, one for each e_i. The old thread assumes a *Sync* code form and waits for the completion of the new threads. According to the transition rules, the thread that is meant to evaluate e_2 is sent to a newly created virtual instance. As discussed, it represents a potential task in a concrete implementation.

We implement the idea of lazy task creation as follows. The execution of $e_1 \mid\mid e_2$ creates two new threads and places *both* on the DFT stack, with the thread for e_2 above that for e_1. In addition, the thread for e_2 is linked into the task queue. If e_2 is not stolen before e_1 returns, the thread for e_2 becomes active and is executed locally. Otherwise, if another concrete instance needs work, it finds the thread for e_2 via the task queue, steals the thread, and replaces it by a thread that implements `succ` and contains an empty target list. When the execution of e_1 returns, it finds the substitute and executes it, which has no effect apart from clearing the stack. The execution of e_2 proceeds asynchronously in the other instance.

It should be noted that this implementation handles the suspension of the execution of e_1 without any additional provisions. On suspension, the thread executing e_1 is placed into the waiting list as usual. If it is not stolen, the thread for e_2 becomes active; otherwise, the situation is the same as in the transition system that considers only virtual instances.

Finally, it remains to specify how work stealing is triggered. If the stack of a concrete instance becomes empty, the instance has to look for new work. In this case, the transition system executes a *Poll* form (see Section 4.5.10), which looks for ready tasks in the waiting list; but there may be no ready tasks in the waiting list—the instance is idle. An idle instance looks for work on other concrete instances by sending a work stealing message. To avoid the explicit check for an empty stack, the last element on the stack is always a special thread that acquires new work when it becomes active.

5.1.3 Global Function Bindings

The code of the executed program is available on all concrete instances, and, as in the STGM, we allocate a static closure for each global function binding (Peyton Jones 1992, Section 9.1). In HIPAR, this happens once for each concrete instance, which leads to a problem with constant applicative forms. Constant applicative forms (CAFs) are global closures that do not expect any arguments. When they are updatable, they may not be duplicated before they are evaluated; otherwise, sharing is broken.

So for each CAF, we only allocate a closure on a single concrete instance, the other instances reference this unique representant by a remote reference. For efficiency reasons, we do not treat the other global closures, which expect arguments, in this way. Access to global closures is frequent, and even with the caching mechanism that is discussed below, we would introduce serious overhead.

5.2 Closures

An implementation associates a heap area with each concrete instance. This heap occupies most of the memory of the PE that hosts the concrete instance and stores the closures of all virtual instances that are mapped to this concrete instance. Next, we discuss the concrete layout of closures and the realization of remote references.

5.2.1 Generalized Closures

In Section 4.4.3, closures were introduced as triples that consist of a lambda form, a valuation of the free variables, and a marker. The lambda form can be regarded as the code of the closure, the valuation as its environment, and the marker distinguishes between three different kinds of closures. Central to the work on the STGM (Peyton Jones 1992) and the JUMP-machine (Lock 1992, Chakravarty & Lock 1997) is a particularly efficient representation and handling of closures. We adopt the same techniques for HIPAR.

Allocation

We realize the heap as a uniform block of memory. The program allocates closures contiguously. It maintains a dedicated register that points to the borderline between used and unused heap, by advancing the register accordingly whenever it allocates a closure. This model supports no explicit deallocation of closures, but rather requires a garbage collection routine that is invoked when the size of the unused heap falls below a fixed threshold. There exists a lot of work on the implementation of such garbage collection routines (Sankaran 1994), and we will not discuss this topic here. But it should be noted that HIPAR requires a compacting garbage collector—i.e, the used closures have to be packed into a contiguous memory area, to provide one big block of memory for the allocation of further closures.

The Info Table

Instead of a single code pointer per closures, it is advantageous to associate a set of code pointers with every closure. The additional code pointers can be used for garbage collection and unification (Lock 1992, Peyton Jones 1992, Lock, Keller & Chakravarty 1995, Chakravarty & Lock 1997). To save space in the heap, each closure only contains a single pointer into the code area. This pointer references an *info table*[2], which contains all code pointers that are associated with the closure.

Marking Closures Without Explicit Tags

It remains to discuss the implementation of the marker that is the third component of every closure. The marker distinguishes three types of closures: function closures, unbound variable closures, and forwarding closures. The behaviour of the transition rules in Section 4.5.14 depends on the types of the inspected closures. We can avoid an extension of the closure layout by storing the marker in the info table of the closure. But we can do even better: Lock (1992, Section 6.4) has demonstrated that we can avoid the explicit comparison of closures by providing additional code pointers. The unification code of a closure realizes unification with respect to the term that is represented by that closure, and the boolean value that is

---

[2] The info table is called method table in the JUMP-machine.

contained in the *Unif* code form is realized by using two different entry points. The result is a fully-compiled unification-procedure without markers and without any interpretative steps.

5.2.2 Global Addresses

In the transition system, virtual instances reference closures or tasks that are located on other instances by means of the global addresses of Section 4.4.2. Boxed values contain a global address, which comprises the identifier of an instance together with the local identifier of a closure or task. This scheme is convenient in the abstract machine architecture, but it would lead to some problems in a concrete implementation. In particular, it requires the following procedure whenever a closure is entered: The instance identifier of the closure must be compared against the identifier of the executing instance and, if they are different, a message has to be sent to the remote instance. But this is costly as entering a closure is one of the most frequent operations in HIPAR; furthermore, the whole procedure contradicts the spirit of taglessness. Therefore, we represent references to remote closures and remote tasks by using a different scheme.

The idea is to represent boxed values always by instance-local addresses and to provide a new kind of closures—which we call *shadow closures*—for remote objects. A global reference is then replaced by a local reference to a shadow closure that contains the information that is necessary to identify the remote closure. As a result, no extra test is necessary when a closure is entered; if a shadow closure is entered, it initiates the remote access. This scheme has been used successfully in the HDG-machine (Kingdon et al. 1991) and parallel implementations of the STGM (Trinder et al. 1996).

The use of shadow objects has the additional advantage that it comes with a free, on-the-fly caching mechanism. We mark shadow closures with an update mode of `u`; so when the remote access is finished, the closure is updated with the result of the remote access. Future access to the closure finds the result without any additional remote access.

As mentioned, HIPAR relies on a garbage collector, to reclaim space that is no longer used in the heap. In a distributive setting, the costs induced by halting the reduction on all PEs, only to perform a garbage collection, are prohibitive. Fortunately, Lester (1989) has demonstrated that purely local garbage collection can be used to reclaim most of the unused heap space. He proposes to replace direct global references to closures by an indirection via an *entry table*. In a local garbage collection, the entry table is considered to be part of the root set, and it is not moved, but it is updated such that its components point to the new locations of the local objects. Hence, global references into the entry table are not directly affected by a local garbage collection, but are indirectly updated by means of the entry table.

5.3 Thread Management

Threads as the central computational resource have to be implemented carefully to achieve sufficient efficiency. As a consequence of the self-scheduling nature of HIPAR, there is no need for a dedicated scheduling mechanism, which is needed in many other approaches (for example, Augustsson & Johnsson 1989, Loogen et al. 1989, Kingdon et al. 1991, Trinder et al. 1996). Thus, the main concerns are the storage areas that are used for the various components of a thread and the efficiency of thread switching. In particular, the uniform storage of all threads in a thread pool, as used in the abstract machine architecture, does not lead to an efficient implementation. It is important that the most frequent case, which are threads that are located on the DFT stack and that do never suspend, is handled as efficient as function calls in a sequential implementation; otherwise, it will be impossible to gain absolute speedup. Threads occur in HIPAR in three different states: active, running (i.e., on the DFT stack, but not active), and suspended. Active threads correspond to a currently executing function activation. Moreover, running threads correspond to a function activation that is buried on the stack below other function activations. Finally, suspended threads are new in the concurrent setting; they are the most costly variant in terms of memory and time. It is the whole purpose of the latency tolerance mechanisms of HIPAR to avoid the last kind of threads as far as possible.

5.3.1 The Active Thread

There is exactly one active thread per concrete machine instance. The active thread occupies the processor that is associated with the instance, in terms of both memory and processing time. The state of the active thread is kept in processor registers as far as possible; all standard technology for code generation applies here (Waite & Goos 1985, Aho, Sethi & Ullman 1986).

5.3.2 Stacks

The argument stack of the active thread is implemented as in the STGM (Peyton Jones 1992, Section 8). Hence, we use two physical stacks per instance to store pointer and nonpointer values, respectively. Furthermore, we can adapt a technique from the STGM, to realize running threads that a buried below the data of other threads, in the following way.

In the STGM, the creation of an update frame requires to empty the argument stack. In the same way, the creation of a new thread that is placed on top of the DFT stack requires a new argument stack in HIPAR. Instead of copying the old contents of the argument stacks to a physically distinct save area, we maintain two stack base-pointers—one for each stack. Then, the stacks are emptied by adjusting the base pointers. Furthermore, the state of the active thread is spilled on the stacks before the base pointers are adjusted. This correspond to the process of "saving the local environment" in the STGM (Peyton Jones 1992, Section 9.4.1).

5.3.3 Synchronization Frames

When a thread assumes a *Sync* code form, it has to maintain a synchronization counter and an environment, where *Store* targets can insert unboxed return values. Both the counter and the environment cannot be placed into registers or on the stack, because they must be accessible in a uniform way, independent of whether the thread is active, running, or suspended. Other threads have to be able to decrement the counter and to store values in the environment asynchronously via appropriate targets. The costs of updating the targets, whenever the location of these structures moves, would be prohibitive. To solve this problem, we introduce *synchronization frames* (*sync frames*, for short). A sync frame is a heap-allocated data structure that contains a synchronization counter and one slot for each unboxed value that may be returned by a *Store* target. Most *Sync* forms are generated by the execution of a `letpar`; in this case, the sync frame initializes the counter to the number of value bindings in the `letpar` and contains one slot for each value binding.

A sync frame is always associated with a thread. In fact, we use pointers to sync frames instead of the thread identifiers that are used in the transition system. This allows threads to move from the stack into the heap if necessary—we return to this point later.

5.3.4 Target Lists

HIPAR target lists are a substitute for the return addresses and update frames of the STGM. In contrast to the STGM, where these structures are stack-allocated, target lists are partially heap-allocated. One reason for this change is the fact that *Sig* targets are added asynchronously by other threads (see the rules in Section 4.5.8).

We represent all targets by two pointers plus one pointer to provide the link structure.

- A *Store* target consists of a pointer to the sync frame where the result has to be stored plus a code pointer that points to some code that executes the store.
- A *Sig* target has the same layout, but its code simply decrements the counter in the sync frame.
- An *Upd* target contains a pointer to the closure that has to be updated plus a code pointer.

The processing of a target list is realized as a chain of tail calls via the code pointers in the targets. The last element in a target list is always a *Store* or a *Sig* target, and we handle it in a special way. If it is created together with the thread, it is not allocated in the heap, but on the stack—this is the case, for example, in the transition rule for `letpar`. As a result, a `letpar` thread, which does not enter an updatable closure, can proceed without the need to allocate any of its structures on the heap.

The active thread maintains a pointer to its target list in a register—in a similar way as a function activation maintains a return address to its parent in a sequential implementation. Whenever the active thread places another thread on top of the DFT stack, a sync frame is created, so the pointer to the target list can be stored in the sync frame. This is important to allow concurrently executing threads to extend the target list (which is necessary to realize the *Suspend* code form).

5.3.5 Thread Suspension

As mentioned above, the most expensive operation of the thread management is suspension. The transition system of HIPAR defines that a thread containing a *Sync* code form where the synchronization counter is greater than zero has to be suspended when it becomes active (Section 4.5.10). This means that its identifier is taken from the DFT stack and placed into the waiting list. In an concrete implementation, all data has to be taken from the stacks and copied into a *thread closure*, which is allocated in the heap.

In the case of suspension, a thread always has a sync frame, which is used to store a reference to the thread closure. When eventually the counter in the sync frame becomes zero and the thread is re-activated, the thread closure is entered. The closure's code copies the contents of the closure back on the stacks and into the registers, and then, runs the thread as if it had never been suspended.

The main costs obviously are the memory area that is used by the thread closure as well as the time consumed by copying data from the stacks into the heap and back. I expect that the memory consumption will not be a problem, as I estimate that the rapid allocation of thunks in the implementation of a lazy language exceeds the costs for thread closures. The time spent by copying is the main cost that is induced by the thread management, but it is paid only when there is not enough local work to cover a long latency access.

It is an interesting consequence of the excessive use of tail calls in HIPAR that the thread, which triggers a remote access does not need to be suspended—simply because it ceases to exist in the moment of the remote access (see Section 4.5.7). In other words, the tail call continues on the remote instance, and as it is usual with tail calls, execution never returns to the caller. The result of the remote call is returned via the target list to the instance that issued the remote access. However, shortage of local work may lead to the suspension of other threads on the DFT stack, and thus, a final evaluation of the technique will have to wait until a concrete implementation is completed and tested in detail.

5.3.6 The Ready List

In the abstract architecture, suspended threads are entered into the waiting lists. They are removed after they become ready, and when a *Poll* form is executed. Thus, there are two kinds of threads in the waiting lists: suspended threads and ready threads. Both kinds are stored in a thread closure. For suspended threads, their closure is accessible via the *Sig* and *Store* targets that will eventually enable them (indirectly via the sync frame of the thread). Hence, there is no need to link them into a further structure. The situation is different for ready threads. They are not necessarily referenced from somewhere apart from the waiting list; furthermore, the efficient implementation of *Poll* requires cheap access to all ready threads. As a consequence, each concrete instance maintains a list of ready threads, which we call the *ready list*, but we renounce an explicit waiting list.

The ready list of an instance is extended at two occasions: (i) when a suspended thread becomes ready and (ii) when a remote instance sends a thread. In the first case, a *Sig* or *Store* target decreased the counter in the sync frame of a thread to zero. Thus, the code that implements the target adds the thread to the ready list. In the second case, a message that contains the thread was received. Then, the message handling code places the thread into the ready list.

As mentioned, entering a thread closure activates the thread. Thus, the code for *Poll* simply has to the enter the first closure in the ready list. The ready list never becomes empty, as it contains, at its end, a dedicated closure that returns to the topmost thread on the stack.

5.4 Returning and Updating Closures

In the transitions system, a *Ret* code form contains an unboxed and a boxed representation of the result. This is different in the implementation, which again uses ideas from the STGM. In particular, we often just return unboxed values and create boxed values on-the-fly only where necessary. To avoid the repetitive creation of boxed values, for example, in the case of multiple *Upd* targets, we use vectored returns as the STGM.

The STGM introduced the idea that the compiler may decide on a datatype for datatype basis whether values are returned boxed or unboxed, we can use the same technique in HIPAR. This allows to return big data objects, such as arrays, as heap structures.

5.5 Communication

The transition system of HIPAR uses an asymmetric communication protocol. On receipt, a message stores a thread into the thread pool of the receiving virtual instance. No explicit receive operation is necessary. In a dynamic environment where the overlap of communication and computation is important, such asymmetric communication has repeatedly proven to be valuable (Culler et al. 1993, Krishnamurthy, Culler, Dusseau, Goldstein, Lumetta, von Eicken & Yelick 1993, Foster, Kesselman & Tuecke 1996).

The concept of *active messages* was introduced by von Eicken et al. (1992) as a low-level communication layer for asymmetric communication. An active message consists of a *message handler* and a variable-sized *message body*. The message handler is a code pointer that identifies a routine for processing the message on the receiving PE. The message body contains the transfered data. On receipt of an active message, its message handler is executed; it has access to the message body and may perform application specific operations.

In the case of HIPAR, a message that transports a thread contains a message handler linking the thread into the ready list of the receiving instance. In addition, the flexibility of active messages can be exploited to handle some messages more efficiently—i.e., the overhead induced by the creation of a thread closure can be avoided. This is the case in two situations: messages with a *Enter* thread and messages with a *Ret* thread.

If a thread contains a *Enter* form that will enter a closure that is already in normal form, then a message that contains the code for an appropriate *Ret* form can be returned directly. This does not only save effort on the side of the instance that processes the request, but also leads to a faster response, and thus, shorter latency on the requesting side. In general, a message that contains an *Enter* thread does not directly place the thread into the ready list; instead, it executes a dedicated code that is accessible via the info table of the closure that should be entered. This code differs in dependence on the evaluation status of the closure: If the closure is a normal form, it directly replies to the request; otherwise, it places the thread into the ready list.

When a thread contains a *Ret* code, we can always avoid the explicit creation of a thread on the receiving instance. The work that is necessary to handle the entries in the target list is strictly bound.

5.6 Discussion

Threads are managed in HIPAR much like ordinary function activations in sequential languages; only on suspension, additional costs are incurred—mainly for copying data between the stack and the heap. Threads that issue a remote access need not be suspended as they cease to exist after they sent an appropriate message; we can view the active message as the continuation of the tail call. Other threads suspend when there is not sufficient local work to tolerate the latency of the remote access. The asymmetric communication mechanism of active messages can save the creation of a thread on the receiving instance in some cases, which further decreases the cost of communication. Anyway, a final performance evaluation has to wait until a code generator on the basis of these techniques is finished.

The mechanisms that were presented in this chapter draw heavily from previous work that has been referenced at the appropriate places. Most notably, lazy task creation (Mohr et al. 1991) and active messages (von Eicken et al. 1992) are used. Many ideas from the implementation of the STGM (Peyton Jones 1992) are transferred to HIPAR, which is not surprising as HIPAR is an extension of the STGM. It is noteworthy that the parallel graph-reduction machine Alice (Darlington & Reeve 1981) not only employed multi-threading to hide latencies, but also introduced a message passing mechanism that resembles active messages.

Thus in a sense, HIPAR reintroduces both concepts, which Alice already employed for parallel graph reduction, into modern compiled graph-reduction.

I defer a detailed comparison with the TAM (Culler et al. 1993, Goldstein et al. 1996) and GUM (Trinder et al. 1996) until a performance evaluation of my implementation is available.

Chapter 6

CONCLUSION

THE polymorphic lambda calculus enriched with the fixpoint combinator, data constructors, and case expressions is usually regarded as the essence of functional programming languages. The language definition of Haskell calls such an extended lambda calculus the *Haskell kernel*; it defines the semantics of the full language by a straight forward translation into the Haskell kernel (Hudak et al. 1992, Section 1.2). Such simplicity at the heart of a mature programming language is certainly one of the main virtues of Haskell—if not *the* main virtue.

Chapter 3 attempted to follow this example with $\mathcal{D}$, which is a conservative extension of the Haskell kernel; it adds combinators for the co-ordination of parallel activities. $\mathcal{D}$ includes a notion of agents, supporting parallel and sequential composition, communication by equational constraints over unbound variables, and indeterminate ask expressions. This approach opens up the full flexibility of higher-order programming for agents, while leaving the semantics of the Haskell kernel intact. Lambda abstractions and case expressions can be used to define complex co-ordination forms from the basic agent combinators. Still, the type system guarantees the proper separation of co-ordinative expressions and purely functional, computational expressions. Furthermore, $\mathcal{D}$'s linear logic semantics, together with the parallel operational semantics, justify calling $\mathcal{D}$ a declarative parallel language—in the sense discussed in Chapter 1.

Chapter 4 developed a detailed execution model, which refines the operational semantics of $\mathcal{D}$ and is geared to the architecture of distributed-memory parallel computers. This execution model identifies long latencies induced by remote accesses as a major efficiency concern and employs compiler-controlled multi-threading to hide such latencies. The model is realized by an abstract machine (HIPAR) that extends the Spineless Tagless G-machine (Peyton Jones 1992) by supporting multi-threading and the co-ordinative forms introduced with $\mathcal{D}$. In more general terms, it extends compiled graph-reduction by explicitly supporting the distribution of the graph over multiple processor elements and by providing facilities to cover the latencies that consequently arise. Abstract notions of distribution, threads, and thread synchronization were introduced at the level of the abstract-machine language KCODE. This enabled optimizations to be made with respect to distribution and thread scheduling—for example, optimizations increasing the tolerance of the code to remote-access latency. This approach to parallel graph-reduction is new, the common view being that parallelism is a low-level implementation detail, which is simply ignored in the execution model. This common view fails to expose to the compiler the concern about the efficient management of distributed resources.

We applied the principle of compiler-controlled self-scheduling, which avoids the use of an external scheduler. Instead, the program performs thread-scheduling on-the-fly, and exposes to the compiler opportunities for thread-level optimizations. This helps avoid the overhead introduced by an external scheduler and supports a natural adaptation of the scheduling process to the executed program. A scheduling hierarchy differentiates between threads, which are used to tolerate the latency of remote access, and tasks, which are created by parallel composition and which represent true parallelism. This distinction reflects the need for fast context switches between threads and the desire to ameliorate the overhead of task migration against the work load induced by a task.

6.1 Research Contributions

The present dissertation contains two main research contributions. First, it demonstrates that the typed lambda calculus can be conservatively extended to incorporate agents, making it a declarative parallel language. Second, it provides the design for an abstract machine that implements compiled graph-reduction on distributed-memory parallel computers, such that compiler-controlled multi-threading can be used to cover communication latencies.

6.1.1 A New Foundation for Declarative Parallelism

Chakravarty, Guo, Köhler & Lock (1997) introduce a logic and an operational semantics for a kernel language of GOFFIN, but this logic semantics fails to capture some important aspects of the language. In particular, because the logic semantics is based on classical higher-order logic, committed-choice indeterminism cannot be faithfully represented (see Section 3.1). For the same reason, the evaluation strategy, which uses deferred evaluation to handle unbound variables, is not captured by the logic semantics.

The formalism $\mathcal{D}$, introduced in Chapter 3, is original in that it successfully addresses all these problems, while maintaining a clear separation of co-ordination and computation concerns. Its linear logic semantics is able to capture the strategy of deferred evaluation as well as committed-choice indeterminism; it is a faithful abstract model of the parallel operational semantics of $\mathcal{D}$. Moreover, $\mathcal{D}$ is the first extension of the typed lambda calculus that includes agents for the declarative co-ordination of parallelism.

6.1.2 Compiler-controlled Multi-threading in Graph-reduction

Although many researchers have addressed the implementation of compiled graph-reduction on parallel machines, none of their efforts explicitly supports multi-threading and self-scheduling to hide long latencies. The idea of applying multi-threading as a method to cover the latency of communication operations has mainly been used by the dataflow community (Gao, Bic & Gaudiot 1995). Although they are concerned with functional programming languages, the technology of compiled graph-reduction has received little attention.

The integration of multi-threading into compiled graph-reduction, as developed in Chapter 4, is original. In particular, the ability of KCODE to expose thread-based optimizations at the source-language level is new and, as has been demonstrated, effective. The design of the concrete implementation in Chapter 5 is new in its generalization of function activations to thread activations, and in its use of active messages.

6.2 Related Work

Previous chapters have already covered related work of special relevance to the topics presented in the respective chapters. Some general remarks remain to be made on the use of declarative languages for programming parallel computers.

We can distinguish between different approaches to the design and implementation of declarative languages (that are meant to be executed in parallel) by their view of the rôle of parallelism. Parallelism is sometimes regarded as a low-level implementation problem that should not be visible at more abstract levels. This resulted in attempts to hide parallelism in the programming language, or the execution model, or both. A good example of this are the projects concerned with a parallel implementation of Haskell—for example, the GHC-based GUM (Trinder et al. 1996). But, as a sequential language, Haskell provides no facilities for identifying worthwhile parallel computations, and consequently, ad hoc constructions are used—for example, the meta-function `par` (Trinder et al. 1996) and many others (Hammond, Loidl & Partridge 1995). A similar tactic is used to deal with parallelism in the implementation. The execution model does not mention parallelism, but it creeps into the runtime system (Trinder et al. 1996). At best, the abstract machines are designed so as to support the runtime system in its task, but they do not usually acknowledge distribution (Augustsson & Johnsson 1989, Peyton Jones 1992)—partly because they are designed for shared-memory computers.

In some approaches, parallelism is not completely degraded to a mere operational detail, but still attempts are made to hide it from the programmer. For example, the dataflow community prefers languages that maximize the degree of implicit parallelism—in the hope that actual programs will happen to have enough parallelism to keep a parallel machine busy. A good example of this is the language Id (Nikhil 1990) and its further development pH (Nikhil, Arvind, Hicks, Aditya, Augustsson, Maessen & Zhou 1995). However, it would appear that a programmer needs to understand the execution model quite well in order to deliver efficient programs, and non-declarative features have been proposed to increase efficiency (Barth et al. 1991). At the implementation level, the dataflow community is definitely most receptive to the idea that the execution model must deal with all aspects of a parallel machine, and especially that it must take distribution and the resulting latencies into account—for example, see the TAM (Culler et al. 1993) and its further development (Goldstein et al. 1996).

Nevertheless, research on parallel computing has shown that parallelism is a natural component of programming (see, for example, Chandy & Taylor 1992) and a concern in all phases of software development for parallel computers (see, for example, Foster 1995). This holds for programming languages as well as for implementations, as some researchers who are concerned with declarative programming have recognized (for example, Kelly 1989, Darlington et al. 1993).

6.3 Future Work

The present dissertation is a first step towards the implementation of declarative parallel languages that use co-ordinating relations and computing functions. There is a lot of interesting research still to be done in this area. The discussion sections of the previous chapters have already given suggestions for future work; these are not repeated here, but some further topics are discussed.

6.3.1 A Compiler

It is clear that a final evaluation of the implementation technology described in the previous chapters presupposes an implementation on a massively parallel computer. I am currently writing a compiler for this purpose. In fact, the prototypical implementation of the abstract machine mentioned in Section 4.6 forms part of it. It has been used to check the soundness of the execution model on an abstract level. This enabled design errors to be spotted earlier and more easily than in the final code generation. Previous experience with a compiler for a declarative language using a new execution model (Chakravarty & Lock 1997) has made a strong case for this stepwise approach.

6.3.2 Searching Agents

Logic programming languages often come with a notion of search—for example, the backtracking search procedure of Prolog. Search is normally used to compute multiple, alternative solutions for disjunctive subgoals. From the point of view of linear logic, it is easy to add search to $\mathcal{D}$; the connective $\&$ can be used in the encoding. Unfortunately, it is much harder to find an efficient operational interpretation for disjunctions, as they complicate the treatment of unbound variables considerably. Different alternatives may maintain different bindings for the same variable (or a variable may be unbound in one alternative, and bound in another). This has been identified as one of the major problems in parallel implementations of logic languages that exploit or-parallelism—i.e., that run alternatives concurrently. Usually, variables that may have different bindings are called *conditional variables*, and many methods have been proposed to handle such variables—an overview of these methods, together with a classification, is given by Gupta (1994).

One major insight of Gupta's work is that managing conditional variables is always performance-critical, because it is nonlinear with respect to the number of alternatives, the number of conditional variables, or the number of accesses to conditional variables. In other words, for any implementation, we can construct programs that are subject to a nonconstant performance penalty, simply because the implementation has to handle conditional variables. Gupta's thesis relates mainly to shared-memory computers. The implementation of conditional variables on distributed-memory machines is even more critical; it would complicate the execution model considerably, and would induce at least the same overhead as on shared-memory computers.

However, there are many applications that require search and that can benefit from corresponding support by the programming language. Furthermore, disjunctions are important for some constraint systems. Thus, the integration of search into the operational semantics of $\mathcal{D}$ and into the execution model is an interesting research topic.

6.3.3 Garbage Collection

An essential ingredient in the runtime system of a declarative-language implementation is the garbage collector. It removes unused data structures from the heap when this is full. An abstract machine, such as the Spineless Tagless G-machine and its derivative HIPAR, requires a compacting garbage collector that is able to collect cyclic data structures. Some work has been done on this topic (Lester 1989, Lester 1992, Sankaran 1994, Kamada, Matsuoka & Yonezawa 1994). The most promising approaches support independent, local garbage collections on the

PEs, to recover unused data structures that are not globally referenced. However, the efficient treatment of global references, especially when they may be cyclic, is still a problem.

Appendix A

A SHORT INTRODUCTION TO LINEAR LOGIC

Linear logic was introduced as a refinement of classical logic by Girard (1987). An extended discussion of its syntax and semantics is available elsewhere (Girard 1995).

A.1 Contraction and Weakening

The basic idea in linear logic is to control the use of the structural rules contraction and weakening:

$$\frac{\Gamma \vdash \Delta, A, A}{\Gamma \vdash \Delta, A}\,\text{(contraction)} \qquad \frac{\Gamma \vdash \Delta}{\Gamma \vdash \Delta, A}\,\text{(weakening)}$$

Contraction intuitively means that any formula can be used multiply, and weakening allows us to ignore a formula completely. In linear logic, both contraction and weakening are not available as structural rules, which leaves us with exchange as the single structural rule:

$$\frac{\Gamma \vdash \Delta, B, A}{\Gamma \vdash \Delta, A, B}\,\text{(exchange)}$$

Exchange asserts the commutativity of the binary logical connectives.

A complete omission of contraction and weakening would restrict the expressiveness of the logic severely—consider that contraction carries the power of unbounded recursion. Linear logic introduces the modalities ! (called *of-course* or *bang*) and ? (called *why-not*) to allow a controlled use of contraction and weakening. In particular, the left rules for why-not contain

$$\frac{\Gamma \vdash \Delta, ?A, ?A}{\Gamma \vdash \Delta, ?A}\,\text{(?-contraction)} \qquad \frac{\Gamma \vdash \Delta}{\Gamma \vdash \Delta, ?A}\,\text{(?-weakening)}$$

So, ? allows to copy or discard A. Dereliction makes use of a formula prefixed by why-not:

$$\frac{\Gamma \vdash \Delta, A}{\Gamma \vdash \Delta, ?A}\,\text{(?-dereliction)}$$

A.2 Conjunction and Disjunction

As consequence of restricting the use of contraction and weakening to formulae prefixed by why-not, the logical connectives conjunction and disjunction come in two flavours, called their *multiplicative* ($\otimes$ called *times* or *tensor* and $\invamp$ called *par*) and *additive* versions ($\&$ called *with* and $\oplus$ called *plus*). The meaning of these connectives captures different aspects of the meaning of conjunction and disjunction in classical logic. The difference between the multiplicative and the additive connectives is in their treatment of the context. The operands A and B of a multiplicative connective have to share the resources provided by the context:

$$\frac{\Gamma_1 \vdash \Delta_1, A \qquad \Gamma_2 \vdash \Delta_2, B}{\Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2, A \otimes B}\,(\otimes) \qquad \frac{\Gamma \vdash \Delta, A, B}{\Gamma \vdash \Delta, A \invamp B}\,(\invamp)$$

In the case of $\invamp$, A and B compete for the resources contained in Γ and Δ, whereas $\otimes$ forces the partitioning of the context into Γ_1 and Δ_1, which may be consumed by A, and Γ_2 and Δ_2, which may be consumed by B.

In contrast, the additive connectives provide the full context to both operands:

$$\frac{\Gamma \vdash \Delta, A \qquad \Gamma \vdash \Delta, B}{\Gamma \vdash \Delta, A \,\&\, B}\,(\&) \qquad \frac{\Gamma \vdash \Delta, A}{\Gamma \vdash \Delta, A \oplus B}\,(\oplus\text{-l}) \qquad \frac{\Gamma \vdash \Delta, B}{\Gamma \vdash \Delta, A \oplus B}\,(\oplus\text{-r})$$

As expected the inference rules of the conjunctions ($\otimes$ and $\&$) have two premises, whereas those of the two disjunctions ($\invamp$ and $\oplus$) have only one premise.

A.3 Neutrals

The truth values of classical logic are also duplicated by the modification of the structural rules. The two flavours of truth are 1 and $\top$ (being the neutrals of $\otimes$ and $\&$) and the two falsities are $\bot$ and 0 (being the neutrals of $\invamp$ and $\oplus$). They comes with the following inference rules:

$$\frac{}{\vdash 1}\,(1) \qquad \frac{\Gamma \vdash \Delta}{\Gamma \vdash \Delta, \bot}\,(\bot) \qquad \frac{}{\Gamma \vdash \Delta, \top}\,(\top) \qquad \langle\text{no rule for } 0\rangle$$

The constants 1 and $\bot$ are also called multiplicatives, and $\top$ and 0 are additives.

A.4 Quantifiers

Linear logic comes with the usual quantifiers:

$$\frac{\Gamma \vdash \Delta, [t/x]A}{\Gamma \vdash \Delta, \exists x.A}\,(\exists) \qquad \frac{\Gamma \vdash \Delta, A}{\Gamma \vdash \Delta, \forall x.A}\,(\forall) \quad \text{if } x \notin \mathrm{Fr}\,\Gamma \cup \mathrm{Fr}\,\Delta$$

A.5 Duals

Linear logic is highly symmetric—i.e., negation can be defined when we assume positive (p, q, r, etc.) and negative versions ($p^\perp$, $q^\perp$, $r^\perp$, etc.) of atoms. Negation $(\cdot)^\perp$ is called *nil* and is defined as follows:

$$\begin{array}{rlrl}
1^\perp & \triangleq \perp & \perp^\perp & \triangleq 1 \\
\top^\perp & \triangleq 0 & 0^\perp & \triangleq \top \\
(p)^\perp & \triangleq p^\perp & (p^\perp)^\perp & \triangleq p \\
(A \otimes B)^\perp & \triangleq A^\perp \parr B^\perp & (A \parr B)^\perp & \triangleq A^\perp \otimes B^\perp \\
(A \,\&\, B)^\perp & \triangleq A^\perp \oplus B^\perp & (A \oplus B)^\perp & \triangleq A^\perp \,\&\, B^\perp \\
(!A)^\perp & \triangleq ?A^\perp & (?A)^\perp & \triangleq !A^\perp \\
(\forall x.A)^\perp & \triangleq \exists x.A^\perp & (\exists x.A)^\perp & \triangleq \forall x.A^\perp
\end{array}$$

We also call $A^\perp$ the *dual* of A, and it holds

$$\Gamma, A^\perp \vdash \Delta \quad \text{iff} \quad \Gamma \vdash A, \Delta \qquad (\cdot^\perp \text{ sym})$$

A.6 Implication

Linear implication is defined by

$$A \multimap B \triangleq A^\perp \parr B \qquad (\multimap \text{ def})$$

Using this definition, we can derive the inference rule for $\multimap$

$$\frac{A, \Gamma \vdash \Delta, B}{\Gamma \vdash \Delta, A \multimap B}\,(\multimap)$$

by the following proof tree

$$\dfrac{\dfrac{\dfrac{A, \Gamma \vdash \Delta, B}{\Gamma \vdash \Delta, A^\perp, B}\,(\cdot^\perp \text{ sym})}{\Gamma \vdash \Delta, A^\perp \parr B}\,(\parr)}{\Gamma \vdash \Delta, A \multimap B}\,(\multimap \text{ def})$$

A.7 Full Linear Logic

An interesting consequence of this symmetry between pairs of the linear logic constants and connectives is that the right rule of a connective is exactly symmetric to the left rule of its dual. Consequently, we can do with just the right introduction rules and omit the antecedents in the sequents. The full inference system of linear logic, which is exploiting this property, is displayed in Figure A.1.

A.8 Forum: Linear Logic with Uniform Proofs

Miller (1996) introduces Forum as a rendering of higher-order linear logic in a form that is especially suited for computational interpretations of linear logic. Forum comes with an inference system that restricts proofs to *uniform proofs*; these are cut-free proofs where the

IDENTITY AND NEGATION.

$$\frac{}{\vdash A, A^{\perp}}\ (\text{id}) \qquad\qquad \frac{\vdash \Gamma, A^{\perp} \qquad \vdash \Delta, A}{\vdash \Gamma, \Delta}\ (\text{cut})$$

STRUCTURE.

$$\frac{\vdash \Gamma, A, B}{\vdash \Gamma, B, A}\ (\text{exchange})$$

LOGIC.

$$\frac{}{\vdash 1}\ (1) \qquad\qquad \frac{\vdash \Gamma}{\vdash \Gamma, \perp}\ (\perp)$$

$$\frac{}{\vdash \Gamma, \top}\ (\top) \qquad\qquad \langle \text{no rule for } 0 \rangle$$

$$\frac{\vdash \Gamma_1, A \qquad \vdash \Gamma_2, B}{\vdash \Gamma_1, \Gamma_2, A \otimes B}\ (\otimes) \qquad\qquad \frac{\vdash \Gamma, A, B}{\vdash \Gamma, A \parr B}\ (\parr)$$

$$\frac{\vdash \Gamma, A \qquad \vdash \Gamma, B}{\vdash \Gamma, A \mathbin{\&} B}\ (\&) \qquad\qquad \frac{\vdash \Gamma, A}{\vdash \Gamma, A \oplus B}\ (\oplus\text{-l}) \qquad \frac{\vdash \Gamma, B}{\vdash \Gamma, A \oplus B}\ (\oplus\text{-r})$$

$$\frac{\vdash \Gamma}{\vdash \Gamma, ?A}\ (?\text{-weakening}) \qquad\qquad \frac{\vdash \Gamma, A}{\vdash \Gamma, ?A}\ (?\text{-dereliction})$$

$$\frac{\vdash ?\Gamma, !A, !A}{\vdash ?\Gamma, !A}\ (!) \qquad\qquad \frac{\vdash \Gamma, ?A, ?A}{\vdash \Gamma, ?A}\ (?\text{-contraction})$$

$$\frac{\vdash \Gamma, A}{\vdash \Gamma, \forall x.A}\ (\forall) \quad \text{if } x \notin \mathrm{Fr}\,\Gamma \qquad\qquad \frac{\vdash \Gamma, [t/x]A}{\vdash \Gamma, \exists x.A}\ (\exists)$$

Figure A.1 The inference rules of linear logic.

outermost logical connectives determine the applied inference rules. Miller et al. (1991) have shown that uniform proofs facilitate the interpretation of the logical connectives as operators of a logic programming language with a fixed operational meaning.

Forum uses only the constants $\top$ and $\bot$, the modality ?, the connectives $\parr$, $\&$, $\multimap$, $\Rightarrow$, and the quantifier $\forall$. Full linear logic is recovered by exploiting the duality of the linear logic connectives and the fact that $A^{\perp} = A \multimap \bot$. The connective $\Rightarrow$ represents classical implication; in linear logic, $A \Rightarrow B$ is regarded as an abbreviation for $!A \multimap B$.

Forum sequents have the form

$$\Sigma : \Psi; \Delta \longrightarrow \Gamma; \Upsilon \qquad \text{or} \qquad \Sigma : \Psi; \Delta \xrightarrow{A} \Gamma; \Upsilon$$

The set Σ is a *signature* that associates the names of the non-logical constants with types. The formulae in the antecedent and the succedent are split into a *linear part*, Δ and Γ, and a *classical context*, Ψ and Υ. In the second form, the formula A is associated with the linear part of the antecedent; its prominent position in the notation is to emphasis that it is currently proved using left introduction rules. In other words, the above two Forum sequents correspond to

$$!\Psi, \Delta \longrightarrow \Gamma, ?\Upsilon \qquad \text{and} \qquad !\Psi, \Delta, A \longrightarrow \Gamma, ?\Upsilon$$

in plain linear logic. The proof system $\mathcal{F}$ of Forum is displayed in Figure A.2 (next page), where $\mathcal{A}$, $\mathcal{A}_1$, and $\mathcal{A}_2$ are lists of atoms and $\mathcal{A}_1 \mathbin{/\!\!\!\backslash\!\!\backslash} \mathcal{A}_2$ denotes that the two lists $\mathcal{A}_1$ and $\mathcal{A}_2$ are merged.

$$\frac{}{\Sigma : \Psi; \Delta \longrightarrow \mathcal{A}, \top, \Gamma; \Upsilon}\;(\top\mathrm{R})$$

$$\frac{\Sigma : \Psi; \Delta \longrightarrow \mathcal{A}, B, \Gamma; \Upsilon \qquad \Sigma : \Psi; \Delta \longrightarrow \mathcal{A}, C, \Gamma; \Upsilon}{\Sigma : \Psi; \Delta \longrightarrow \mathcal{A}, B \,\&\, C, \Gamma; \Upsilon}\;(\&\mathrm{R})$$

$$\frac{\Sigma : \Psi; \Delta \longrightarrow \mathcal{A}, \Gamma; \Upsilon}{\Sigma : \Psi; \Delta \longrightarrow \mathcal{A}, \bot, \Gamma; \Upsilon}\;(\bot\mathrm{R}) \qquad \frac{\Sigma : \Psi; \Delta \longrightarrow \mathcal{A}, B, C, \Gamma; \Upsilon}{\Sigma : \Psi; \Delta \longrightarrow \mathcal{A}, B \parr C, \Gamma; \Upsilon}\;(\parr)$$

$$\frac{\Sigma : \Psi; B, \Delta \longrightarrow \mathcal{A}, C, \Gamma; \Upsilon}{\Sigma : \Psi; \Delta \longrightarrow B \multimap C; \Upsilon}\;(\multimap\mathrm{R}) \qquad \frac{\Sigma : B, \Psi; \Delta \longrightarrow \mathcal{A}, C, \Gamma; \Upsilon}{\Sigma : \Psi; \Delta \longrightarrow B \Rightarrow C; \Upsilon}\;(\Rightarrow\mathrm{R})$$

$$\frac{y :: \sigma, \Sigma : \Psi; \Delta \longrightarrow \mathcal{A}, [y/x]B, \Gamma; \Upsilon}{\Sigma : \Psi; \Delta \longrightarrow \mathcal{A}, \forall x^{\sigma}.B, \Gamma; \Upsilon}\;(\forall\mathrm{R}; y \notin \Sigma) \qquad \frac{\Sigma : \Psi; \Delta \longrightarrow \mathcal{A}, \Gamma; B, \Upsilon}{\Sigma : \Psi; \Delta \longrightarrow \mathcal{A}, ?B, \Gamma; \Upsilon}\;(?\mathrm{R})$$

$$\frac{\Sigma : B, \Psi; \Delta \xrightarrow{B} \mathcal{A}; \Upsilon}{\Sigma : B, \Psi; \Delta \longrightarrow \mathcal{A}; \Upsilon}\;(\mathrm{decide!}) \qquad \frac{\Sigma : \Psi; \Delta \longrightarrow \mathcal{A}, B; B, \Upsilon}{\Sigma : \Psi; \Delta \longrightarrow \mathcal{A}; B, \Upsilon}\;(\mathrm{decide?})$$

$$\frac{\Sigma : \Psi; \Delta \xrightarrow{B} \mathcal{A}; \Upsilon}{\Sigma : \Psi; B, \Delta \longrightarrow \mathcal{A}; \Upsilon}\;(\mathrm{decide})$$

$$\frac{}{\Sigma : \Psi; \cdot \xrightarrow{A} A; \Upsilon}\;(\mathrm{initial}) \qquad \frac{}{\Sigma : \Psi; \cdot \xrightarrow{A} \cdot; A, \Upsilon}\;(\mathrm{initial?})$$

$$\frac{}{\Sigma : \Psi; \cdot \xrightarrow{\bot} \cdot; \Upsilon}\;(\bot\mathrm{L}) \qquad \frac{\Sigma : \Psi; \Delta \xrightarrow{B_i} \mathcal{A}; \Upsilon}{\Sigma : \Psi; \Delta \xrightarrow{B_1 \& B_2} \mathcal{A}; \Upsilon}\;(\&\mathrm{L}_i; i \in \{1,2\}) \qquad \frac{\Sigma : \Psi; B \longrightarrow \cdot; \Upsilon}{\Sigma : \Psi; \cdot \xrightarrow{?B} \cdot; \Upsilon}\;(?\mathrm{L})$$

$$\frac{\Sigma : \Psi; \Delta_1 \xrightarrow{B} \mathcal{A}_1; \Upsilon \qquad \Sigma : \Psi; \Delta_2 \xrightarrow{C} \mathcal{A}_2; \Upsilon}{\Sigma : \Psi; \Delta_1, \Delta_2 \xrightarrow{B \parr C} \mathcal{A}_1 \mathbin{\wedge\!\!\!\wedge} \mathcal{A}_2; \Upsilon}\;(\parr\mathrm{L}) \qquad \frac{\Sigma : \Psi; \Delta \xrightarrow{[M/x]B} \mathcal{A}; \Upsilon}{\Sigma : \Psi; \Delta \xrightarrow{\forall x^{\sigma}.B} \mathcal{A}; \Upsilon}\;(\forall\mathrm{L})$$

$$\frac{\Sigma : \Psi; \Delta_1 \longrightarrow \mathcal{A}_1, B; \Upsilon \qquad \Sigma : \Psi; \Delta_2 \xrightarrow{C} \mathcal{A}_2; \Upsilon}{\Sigma : \Psi; \Delta_1, \Delta_2 \xrightarrow{B \multimap C} \mathcal{A}_1 \mathbin{\wedge\!\!\!\wedge} \mathcal{A}_2; \Upsilon}\;(\multimap\mathrm{L})$$

$$\frac{\Sigma : \cdot; \Delta_1 \longrightarrow B; \Upsilon \qquad \Sigma : \Psi; \Delta_2 \xrightarrow{C} \mathcal{A}; \Upsilon}{\Sigma : \Psi; \Delta \xrightarrow{B \Rightarrow C} \mathcal{A}; \Upsilon}\;(\Rightarrow\mathrm{L})$$

Figure A.2 The proof system $\mathcal{F}$ of Forum (Miller 1996).

Appendix B

SOME DEFINITIONS FROM HASKELL'S PRELUDE

The functions of the Haskell prelude that are used in Chapter 4 are defined in the following. More details on Haskell are given by Hudak et al. (1992).

$$\begin{aligned}
&\mathit{const} && :: a \to b \to a\\
&\mathit{const}\ k\ \_ && = k\\[4pt]
&(\circ) && :: (b \to c) \to (a \to b) \to (a \to c)\\
&(\circ)\ f\ g\ x && = f\ (g\ x)
\end{aligned}$$

$$\begin{aligned}
&\mathit{head} && :: [a] \to a\\
&\mathit{head}\ (x : \_) && = x\\[4pt]
&\mathit{map} && :: (a \to b) \to [a] \to [b]\\
&\mathit{map}\ f\ [] && = []\\
&\mathit{map}\ f\ (x : xs) && = f\ x : \mathit{map}\ f\ xs\\[4pt]
&\mathit{foldl} && :: (a \to b \to a) \to a \to [b] \to a\\
&\mathit{foldl}\ f\ z\ [] && = z\\
&\mathit{foldl}\ f\ z\ (x : xs) && = \mathit{foldl}\ f\ (f\ z\ x)\ xs\\[4pt]
&\mathit{foldr} && :: (a \to b \to b) \to b \to [a] \to b\\
&\mathit{foldr}\ f\ z\ [] && = z\\
&\mathit{foldr}\ f\ z\ (x : xs) && = f\ x\ (\mathit{foldr}\ f\ z\ xs)\\[4pt]
&(++) && :: [a] \to [a] \to [a]\\
&xs ++ ys && = \mathit{foldr}\ (:)\ ys\ xs\\[4pt]
&\mathit{concat} && :: [[a]] \to [a]\\
&\mathit{concat} && = \mathit{foldr}\ (++)\ []\\[4pt]
&\mathit{elem} && :: \mathit{Eq}\ a \Rightarrow a \to [a] \to \mathit{Bool}\\
&\mathit{elem}\ x && = (\mathit{foldr}\ (||)\ \mathit{False}) \circ (\mathit{map}\ (\overset{\cdot}{\underset{\cdot}{=}}\ x))
\end{aligned}$$

In Haskell, (||) denotes disjunction.

```
(\\)                              :: Eq a ⇒ [a] → [a] → [a]
(\\)                              = foldl del
                                    where
                                      [] ‘del‘ _                 = []
                                      (x : xs) ‘del‘ y | x ≑ y     = xs
                                                       | otherwise = x : (xs ‘del‘ y)

length                            :: [a] → Int
length                            = foldl (λn _ → n + 1) 0

partition                         :: (a → Bool) → [a] → ([a], [a])
partition p                       = foldr select ([], [])
                                    where
                                      select x (ts, fs) | p x       = (x : ts, fs)
                                                        | otherwise = (ts, x : fs)

zipWith                           :: (a → b → c) → [a] → [b] → [c]
zipWith z (a : as) (b : bs)       = z a b : zipWith z as bs
zipWith _ _ _                     = []

zipWith3                          :: (a → b → c → d) → [a] → [b] → [c] → [d]
zipWith3 z (a : as) (b : bs) (c : cs) = z a b c : zipWith z as bs cs
zipWith3 _ _ _ _                  = []

zip                               :: [a] → [b] → [(a, b)]
zip                               = zipWith (λa b → (a, b))

zip3                              :: [a] → [b] → [c] → [(a, b, c)]
zip3                              = zipWith (λa b c → (a, b, c))

unzip                             :: [(a, b, )] → ([a], [b])
unzip                             = foldr (λ(a, b) (as, bs) → (a : as, b : bs)) ([], [])
```

Glossary

$\mathcal{D}$		
A	agents	30
agents		
$\mathsf{0}$	failure	18, 30
$\mathsf{1}$	successful termination	18, 30
$E_1 \doteq E_2$	equational constraint	19, 30
$A_1 \parallel A_2$	parallel composition, "A_1 and A_2"	18, 30
$A_1 \Rightarrow A_2$	sequential composition, "A_1 and then A_2"	18, 30
$\nu v.A$	unbound variable	19, 30
B	bindings	29
binding		
$x = E$	binding that associates the expression E with the variable x	20, 29
case alternative		
$a\,\overline{v} \to E$	E may be taken if the scrutinized value matches the data constructor application $a\,\overline{v}$	21, 29
choice		
$\top$	no alternative (error)	31
$\overline{v} : \overline{P} \doteq \overline{E} \to E'$	ask expression	19, 31
$S_1 \mathbin{[]} S_2$	excluding alternatives	19, 31
constants		
$(\cdot)\downarrow$	dereferencing	28
$\lfloor\cdot\rfloor$	naming	28
fix	fixpoint combinator	28
$\langle\cdot,\ldots,\cdot\rangle$	tuple constructor	28
$E_1 \doteq E_2$	equational constraint	28
$\top$	error	28
$\mathsf{0}$	failure	28
$\mathsf{1}$	successful termination	28
$E_1 \parallel E_2$	parallel composition, "E_1 and E_2"	28
$E_1 \Rightarrow E_2$	sequential composition, "E_1 and then E_2"	28
$E_1 \mathbin{[]} E_2$	committed-choice alternatives, "either E_1 or E_2"	28
$\Sigma\lambda v.E$	new name	28
$\mathsf{ask}\ (\lambda\overline{v}.\overline{G})\ (\lambda\overline{v}.E)$	if, for some $\overline{v}$, the $\overline{G}$ hold, then take E with these values of $\overline{v}$	28

D	data constructors and destructors	28
dom $\overline{B}$	denotes the variables that are bound in the bindings $\overline{B}$	30
E	expressions	29
expressions		
let $\overline{B}$ in E	mutually recursive binding construct	20, 29
$E\downarrow$	dereferences the name that is determined by E	21, 29
case E of $\overline{X}$	deterministic case selection depending on the value of E	21, 29
kinds		
$\star$	types of terms	27
$\sharp$	types of names	27
kinds	$\mathcal{C}_{\mathrm{K}} = \{\star, \sharp\}$	27
L	reduction context	46
N	normal form	40
P	patterns	31
S	choice expressions	31
T	data type constructors	28
types		
$\lfloor\tau\rfloor$	name type for names associated with a term of type τ	22, 28
o	the type of agents	20, 28
T	data type constructors	28
X	cases of a (case · of ·) expression	29
KCode		65
f = \\π vs -> e	function binding with arguments vs, update mode π and body e	66
v = e	value binding	66
f vs	function application	66
`choose` chs	committed choice	66
v : e	new unboxed variable v in e	66
e_1 || e_2	parallel composition	66
e_1 => e_2	sequential composition	66
v_1 :=: v_2	equational constraint	66
`succ`	successful termination	66
`fail`	failure	66
`letrec` $funs$ `in` e	recursive function bindings that are visible in e	66
`letpar` $vals$ `in` e	independent value bindings that are visible in e	66
`case` v `of` $alts$	selection of an alternative from $alts$ by scrutinizing v	66
p -> e	case alternative that selects e if the pattern p fits the scrutinized value	66
$mats$ -> e	guarded expression that may be selected when the pattern-matching guards $mats$ match	66
p := e	pattern-matching guard that binds the variables in p by matching p against the value of e	66
$\mathcal{A}[\![\cdot]\!]$	mapping of agents to linear logic processes	37
active thread	thread whose name is on top of the DFT *stack*	84
agent	an entity whose main responsibility is co-ordination, not computation	18

ask	wait until a given constraint is entailed by the *constraint store*	19
$\mathcal{B}\llbracket\cdot\rrbracket$	mapping of bindings to linear logic processes	36
basic value	*lambda-form* where the body returns an unboxed integer value	71
boxed value	stored in the *heap*, in the form of a *closure*	62
calculational proofs		15
$\bullet$	introduces a precondition in a calculational proof	15
closure	a pair of code and environment; stored in the *heap*	63, 70
constraint store	the collection of *told constraints*	19, 31
constructor	short name for data constructor or a *lambda-form* where the body applies a data constructor	71
context (term)	a term containing a *hole* marked by $\langle\cdot\rangle$, where a term can be inserted	13
contraction	structural rule that allows the multiple use of a formulae in a proof	127
deferred evaluation	an evaluation strategy where reduction suspends on unbound variables until the needed variables get bound (for example, through a constraint)	22
definitions		
$\triangleq$	defining equality	9
dereferencing	$\mathcal{D}$ construct of the form $E\downarrow$, which provides the value that is associated with a name	21, 29
dereliction	structural rule that allows access to a formulae prefixed by ?	127
DFT stack	depth-first traversal stack; records the depth-first order by storing *thread* names	84
$\mathcal{E}\llbracket\cdot\rrbracket$	mapping of general expression to linear logic processes	35
environment	valuations for the free variables; represented as a finite map in HiPar	81
forwarding closure	closure that represents an indirection	76
fresh variable	a variable that for a given term M is neither contained in $\mathrm{Fr}\,M$ nor in $\mathrm{Bd}\,M$	13
Haskell		
$\lvert s\rvert$	length of a list	79
$x \in l$	check element membership in a list	79
$l_1 \setminus l_2$	list difference	79
$k \mapsto e$	entry in a finite map; associates e with the key k	79
$m_1 \triangleright m_2$	combine operator for finite lists; prefers the first argument	79
$m \uparrow k$	inquire the association of a key or a list of keys	80
Id_{kind}	identifiers that are naming objects of the given *kind*	80
Supply	type of name supplies	80
Int i	unboxed integer value	80
$Term\ c\,\overline{u}$	unboxed term value	80
$i@x$	global address value	80

$\mathit{val}\ \rho\ \sigma\ x$	look for x first in the local environment ρ and then in the global environment σ	81
Cls	type of *closures*	81
LF	type of *lambda-forms*	81
Marker	type of markers	82
Fun	marker for function closure	82
UB	marker for unbound variables closures	82
Fwd	marker for forwarding closures	82
$\langle \overline{v} \backslash\pi\ \overline{x} \rightarrow \gamma, \overline{u}, \mu \rangle$	closure	82
Thread	type of *threads*	83
$\mathit{Upd}\ g_c$	update target	83
$\mathit{Store}\ g_t\ x$	store target	83
$\mathit{Sig}\ g_t$	signal target	83
NamedThread	type of a *thread* together with its associated name	83
$\langle \tau \bullet \gamma, \mathit{as}, \mathit{ts} \rangle$	*threads*	83
Inst	type of machine instances	84
$\mathit{Remote}\ i\ \mathit{ts}$	remote targets	83
NamedInst	type of an *instance* together with its associated name	85
Config	type of machine configurations	85
$\mathit{Eval}\ e\ \rho$	code form: evaluate expression in local environment	86
$\mathit{Enter}\ g_c$	code form: enter closure	86
$\mathit{Ret}\ u\ \mathit{cls}$	code form: return value	86
$\mathit{Sync}\ n\ \rho\ \gamma$	code form: synchronize	86
$\mathit{Suspend}\ t\ g_c\ \rho$	code form: suspend on already evaluated closure or unbound variable	86
Poll	code form: check for ready-to-run threads in the waiting list	86
$\mathit{Eq}\ v_1\ v_2$	code form: compare two values for equality	86
$\mathit{Unif}\ m\ g_1\ g_2$	code form: unify two boxed values	86
$\mathit{Choose}\ a\ \rho$	code form: select an alternative	86
$\mathit{Try}\ \mathit{ch}\ \mathit{ts}\ \rho$	code form: try an alternative	86
$\mathit{Match}\ \mathit{qs}\ e\ \mathit{ts}\ \rho$	code form: execute pattern-matching guards	86
runnable	distinguishes *suspended* threads from threads that are *ready-to-run*	87
$\mathit{thread} \rightsquigarrow i$	send the given *thread* to the instance i	87
trans	transition function that proceeds a single instance by one step	87
partrans	transition function that advances all instances that can proceed	88
heap	stores data structures in the form of *closures*	62
instance	see *machine instance*	76
kinds		10
$\mathcal{C}_{\mathrm{K}}$	set of kind names	10
$\mathcal{K}$	set of kinds	10
$\kappa_1 \rightarrow \kappa_2$	kind constructor	10
lambda terms		10

PE	processor element	61
plain lambda term	a term of the polymorphic lambda calculus (Definition 2.5)	29
process		
$\top$	abort	24
$\bot$	nil	24
1	error	24
$?(t_1 \doteq t_2)$	equation	24
$P_1 \parr P_2$	parallel composition	24
$P_1 \oplus P_2$	indeterministic choice	24
$(\forall \overline{v}.x\,\overline{v} \multimapinv P)^\perp$	receive method	24
$(\forall \overline{v}.?(t_1 \doteq t_2) \multimapinv P)^\perp$	ask method	24
$\mathbf{D}$	axioms accompanying proofs over encodings of $\mathcal{D}$ expressions	39
$\mathbf{Eq}$	equality theory	26
$\mathbf{M}$	set of process methods	24
$\mathbf{P}$	set of linear logic processes	24
$\mathbf{T}$	set of process terms	24
process combinators		
$\uparrow v$	yield result v	33
$M \succcurlyeq \lambda v.N$	execute M, then N with the result of M bound to v	33
$M \succ N$	execute M, then N (the result of M is discarded)	34
$M \upuparrows N$	execute M and N in parallel, the result of both is composed into a pair	34
$M \Lsh\!\Rsh N$	execute either M or N	34
$P \upharpoonright M$	execute P with M	34
$ask\,G\,M$	execute M if the guard G is satisfied	34
err	error	34
$fail$	abort	34
$get\,x$	get the value stored under the name x	35
$name$	type of names	35
new	yield a new name	35
$proc\,\sigma$	type of a process combinators that returns a result of type σ	33
$put\,x\,v$	store v under the name x	35
$tell$	nonlogical constant defined by a set of axioms	37
push-enter model	a function application is processed by pushing all arguments on the stack and, then, tail calling the function closure	64
ready-to-run threads	*threads* in the waiting list that do not have a *Sync* code form	87
running threads	threads that are recorded in the DFT *stack*	84
$\mathcal{S}[\![\cdot]\!]$	mapping of choice expressions to linear logic processes	38
success closure	closure whose *lambda form* contains `succ`	71
suspended threads	*threads* in the waiting list that have a *Sync* code form	87

suspension	an *agent* that is waiting for some information is suspended	21
target list	component of a *thread*, which specifies where the result of the computation is needed	83
tell	impose a constraint, which is added to the *constraint store*	19
term type	type of terms in $\mathcal{D}$	22, 27
thread	generalization of activation records, which supports asynchronous function activations	82
thunk	unevaluated expression in an implementation of a lazy language; also a *lambda-form* that represents such an expression	63, 71
tuples		
$\langle M_1, \ldots, M_n \rangle$	n-ary tuple	13
$\overline{M}$	shorthand for a tuple	13
$\lvert\overline{M}\rvert$	the arity of the tuple $\overline{M}$	13
types		10
$\mathcal{C}_{\mathrm{T}}$	set of type constructors	10
$\mathcal{V}_{\mathrm{T}}$	set of type variables	10
$\sigma_1\ \sigma_2$	type construction	10
$\sigma :: \kappa$	type σ has the kind κ	10
$\mathcal{S}$	set of *simple types*	11
$\mathcal{T}$	set of *type schemes*	11
scheme	types that may contain a quantification over type variables	11
simple	types without quantification over type variables	11
unbound variable closure	closure that represents an unbound variable	76
unboxed value	raw form of a datum, not *boxed*	62
waiting list	stores all threads of an instance that are not on the DFT *stack*	84
waiting threads	threads that are recorded in the *waiting list*	84
weakening	structural rule that allows the omission of a formula in a proof	127
$\mathcal{X}[\![\cdot]\!]$	mapping of case alternatives to linear logic processes	36

BIBLIOGRAPHY

Aho, A. V., Sethi, R. & Ullman, J. D. (1986), *Compilers — Principles, Techniques, and Tools*, Addison-Wesley Publishing Company.

Aït-Kaci, H. (1991), *Warren's Abstract Machine: A Tutorial Reconstruction*, Series in Logic Programming, The MIT Press.

Aït-Kaci, H., Lincoln, P. & Nasr, R. (1987), Le fun: Logic, equations, and Functions, *in* 'Proceedings of the 4th IEEE International Symposium on Logic Programming'.

Alexiev, V. (1994), Applications of linear logic to computation: An overview, Technical Report TR93-18, Department of Computing Science, University of Alberta, Alberta, Canada. *ftp://ftp.cs.ualberta.ca/pub/TechReports/TR93-19/TR93-18.ps.gz

Andrews, P. B. (1986), *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*, Academic Press.

Ang, B. S., Arvind & Chiou, D. (1995), StarT the next generation: Integrating global caches and dataflow architecture, *in* Gao et al. (1995).

Appel, A. W. (1992), *Compiling with Continuations*, Cambridge University Press.

Arvind, Nikhil, R. & Pingali, K. (1989), 'I-structures: Data structures for parallel computing', *ACM Transactions on Programming Languages* pp. 598–632.

Augustsson, L. (1987), Compiling Lazy Functional Languages, Part II, PhD thesis, Department of Computing Science, Chalmers University of Technology, Göteborg, Sweden.

Augustsson, L. & Johnsson, T. (1989), Parallel graph reduction with the $\langle \nu, G \rangle$-machine, *in* D. MacQueen, ed., 'Functional Programming Languages and Computer Architecture Conference', pp. 202–213.

Augustsson, L., Rittri, M. & Synek, D. (1994), 'Functional pearl: On generating unique names', *Journal of Functional Programming* **4**(1), 117–123.

Barth, P. S., Nikhil, R. S. & Arvind (1991), M-structures: Extending a parallel, non-strict, functional language with state, *in* J. Hughes, ed., 'Proceedings of the International Conference on Functional Programming and Computer Architecture', number 523 *in* 'Lecture Notes in Computer Science'.

Berry, G. & Boudol, G. (1990), The chemical abstract machine, *in* 'Proceedings of the 17th Annual Symposium on Principles of Programming Languages'.

Blelloch, G. E. (1996), 'Programming parallel algorithms', *Communications of the ACM* **39**(3), 85–97.

Blelloch, G. E., Gibbons, P. B. & Matias, Y. (1995), Provably efficient scheduling for languages with fine-grained parallelism, *in* 'Proceedings of the Symposium on Parallel Algorithms and Architectures', pp. 1–12.

Blumofe, R. D., Joerg, C. F., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H. & Zhou, Y. (1995), Cilk: An efficient multithreaded runtime system, *in* 'Proccedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming', pp. 207–216.

Breitinger, S., Loogen, R. & Ortega-Mallén, Y. (1995), Concurrency in functional and logic programming, *in* 'Mt Fuji International Workshop on Functional and Logic Programming', Japan Society of Software Science and Technology, World Scientfic Publishing, Singapore.

Breitinger, S., Loogen, R. & Ortega-Mallén, Y. (1996), Towards a declarative language for parallel and concurrent programming, *in* D. N. Turner, ed., 'Functional Programming, Glasgow 1995', University of Glasgow, Springer-Verlag.

Breitinger, S., Loogen, R., Ortega-Mallén, Y. & Peña, R. (1996), The Eden coordination model for distributed memory systems, *in* M. M. T. Chakravarty, Y. Guo & T. Ida, eds, 'Multi-Paradigm Logic Programming (Proceedings of the JICSLP'96 Post-Conference Workshop)', number 96-26 *in* 'Forschungsbericht', Technische Universität Berlin, Fachbereich Informatik, Berlin, Germany, pp. 25–34.
*http://www.cs.tu-berlin.de/~chak/mplp/proc/

Burstall, R. M. & Darlington, J. (1977), 'A transformational system for developing recursive programs', *Journal of the ACM* **24**(1), 44–67.

Carriero, N. & Gelernter, D. (1992), 'Coordination languages and their significance', *Communications of the ACM* **35**(2).

Chakaravarty, M. M. T., Guo, Y., Köhler, M. & Lock, H. C. R. (1995), Co-ordinating parallel computation through equational constraints, *in* 'Workshop Logische Programmierung '95', GMD Studie, Gesellschaft für Mathematik und Datenverarbeitung.

Chakravarty, M. M. T. (1996), Integrating multi-threading into the Spineless Tagless G-machine, *in* D. N. Turner, ed., 'Functional Programming, Glasgow 1995', University of Glasgow, Springer-Verlag, London.
*http://www.springer.co.uk/eWiC/Workshops/Functional95.html#Chakravarty

Chakravarty, M. M. T. (1997), KCode—a concurrent kernel code for constraint functional languages, Technical Report 97-8, Technische Universität Berlin, Fachbereich Informatik.

Chakravarty, M. M. T., Guo, Y. & Köhler, M. (1997), Co-ordinating concurrency by logic, Forschungsbericht, Technische Universität Berlin. To appear.

Chakravarty, M. M. T., Guo, Y., Köhler, M. & Lock, H. C. R. (1994), Deklarative Programmierung paralleler Algorithmen (declarative programming of parallel algorithms), *in* F. Simon, ed., 'GI-Workshop: Declarative Programming und Spezifikation', number 9412 *in* 'Technischer Bericht', Christian-Albrecht-Universität Kiel.
*http://www.cs.tu-berlin.de/~chak/papers/pdp_lang.ps.Z

Chakravarty, M. M. T., Guo, Y., Köhler, M. & Lock, H. C. R. (1997), 'Goffin: Higher-order functions meet concurrent constraints', *Science of Computer Programming* . To appear in a special issue on the *First International Workshop on Concurrent Constraint Programming (CCP'95)*, scheduled for volume 29.
*http://www.cs.tu-berlin.de/~chak/papers/goffin.ps.gz

Chakravarty, M. M. T. & Lock, H. C. R. (1991), The implementation of lazy narrowing, *in* J. Maluszynski & M. Wirsing, eds, 'Programming Language Implementation and Logic Programming', number 528 *in* 'Lecture Notes in Computer Science', pp. 123–134.

Chakravarty, M. M. T. & Lock, H. C. R. (1997), 'Towards the uniform implementation of declarative languages', *Journal of Computer Languages* . To appear.
*http://www.cs.tu-berlin.de/~chak/papers/jump.ps.gz

Chandy, K. & Taylor, S. (1992), *An Introduction to Parallel Programming*, Jones and Bartlett Publishers.

Church, A. (1940), 'A formulation of the Simple Theory of Types', *Journal of Symbolic Logic* **5**, 56–68.

Cole, M. (1989), *Algorithmic Skeletons: Structured Management of Parallel Computation*, The MIT Press.

Crossley, J. N., Mandel, L. & Wirsing, M. (1993), Untyped constrained lambda calculus, Technical Report 9318, Institut für Informatik, Ludwig-Maximilians-Universität München, Leopoldstraße 11b, 80802 München, Germany.
*http://helios.pst.informatik.uni-muenchen.de/~mandel/techber.dvi

Culler, D. E., Goldstein, S. C., Schauser, K. E. & von Eicken, T. (1993), 'TAM—a compiler controlled threaded abstract machine', *Journal of Parallel and Distributed Computing* **18**, 347—370.

Culler, D. E., Goldstein, S. C., Schauser, K. E. & von Eicken, T. (1995), Emperical study of a dataflow language on the CM-5, *in* Gao et al. (1995).

Damas, L. & Milner, R. (1982), Principal type schemes for functional programs, *in* 'Proc. of 9th ACM Symp. on Principles of Programming Languages'.

Danelutto, M. & Pelagatti, S. (1993), Structuring parallelism in a functional framework, Technical Report TR-29/93, Dipartimento di Informatica, Universitá di Pisa.

Darlington, J., Field, A. J., Harrison, P. G., Kelly, P. H. J., Sharp, D. W. N., Wu, Q. & While, R. L. (1993), Parallel programming using skeleton functions, *in* A. Bode, M. Reeve & G. Wolf, eds, 'PARLE '93: Parallel Architectures and Languages Europe', number 694 *in* 'Lecture Notes in Computer Science', Springer-Verlag, Berlin, Germany, pp. 146–160.

Darlington, J. & Reeve, M. (1981), ALICE: A multi-processor reduction machine for the parallel evaluation of applicative languages, *in* Arvind & J. B. Dennis, eds, 'Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture', ACM Press, pp. 65–75.

Didrich, K., Fett, A., Gerke, C., Grieskamp, W. & Pepper, P. (1994), OPAL: Design and implementation of an algebraic programming language, *in* J. Gutknecht, ed., 'Programming Languages and System Architectures', Vol. 782 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 228–244.

Dijkstra, E. W. & Scholten, C. S. (1990), *Predicate Calculus and Program Semantics*, Texts and Monographs in Computer Science, Springer-Verlag, chapter 4, pp. 21–29. Also circulated as EWD999.

Dupré, L. (1995), *Bugs in Writing: A Guide to Debugging Your Prose*, Addison-Wesley.

Engler, D. R., Andrews, G. R. & Lowenthal, D. K. (1993), Filaments: Efficient support for fine-grain parallelism, TR 93-13a, University of Arizona.

Felleisen, M., Friedman, D. & Kohlbecker, E. (1987), 'A syntactic theory of sequential control', *Theoretical Computer Science* **52**, 205–237.

Foster, I. (1995), *Designing and Building Parallel Programs*, Addison-Wesley.
*http://www.mcs.anl.gov/dbpp/text/book.html

Foster, I., Kesselman, C. & Tuecke, S. (1996), 'The Nexus approach to integrating multithreading and communication', *Journal of Parallel and Distributed Computing* **37**(1), 70–82.

Franzén, T., Haridi, S. & Janson, S. (1992), An overview of the Andorra Kernel Language, *in* 'Proceedings of the 2nd Workshop on Extensions to Logic Programming', LNCS, Springer-Verlag.

Gao, G. R., Bic, L. & Gaudiot, J.-L., eds (1995), *Advanced Topics in Dataflow Computing and Multithreading*, IEEE Computer Society Press.

Girard, J.-Y. (1987), 'Linear logic', *Theoretical Computer Science* **50**.

Girard, J.-Y. (1995), Linear logic: Its syntax and semantics, *in* J.-Y. Girard, Y. Lafont & L. Regnier, eds, 'Advances in Linear Logic', number 222 *in* 'London Mathematical Society Lecture Note Series', Cambridge University Press, pp. 1–42. Also in the Proceedings of the Workshop on Linear Logic, 1993.

Goldstein, S. C., Schauser, K. E. & Culler, D. E. (1996), Enabling primitives for compiling parallel languages, *in* B. K. Szymanski & B. Sinharoy, eds, 'Languages, Compilers and Run-Time Systems for Scalable Computers', Kluwer Academic Publishers, chapter 12, pp. 153–168.

Gunter, C. A. (1992), *Semantics of Programming Languages—Structures and Techniques*, Foundations of Computer Science, The MIT Press.

Guo, Y. (1994), Definitional Constraint Programming, PhD thesis, Imperial College London, Department of Computing.

Gupta, G. (1994), *Multiprocessor Execution of Logic Programs*, Kluwer Academic Publishers.

Hall, C., Peyton Jones, S. L. & Sansom, P. M. (1994), Unboxing using specialisation, *in* 'Glasgow Functional Programming Workshop', Workshops in Computing, Springer-Verlag London.

Hammond, K. (1994), Parallel functional programming: An introduction, *in* 'PASCO'94: First International Symposium on Parallel Symbolic Computation', World Scientific Publishing, Singapore.
*ftp://ftp.dcs.glasgow.ac.uk/pub/glasgow-fp/papers/parallel-introduction.ps.Z

Hammond, K., Loidl, H. & Partridge, A. (1995), Visualising granularity in parallel programs: A graphical winnowing system for Haskell, *in* A. P. W. Bohm & J. T. Feo, eds, 'High Performance Functional Computing', pp. 208–221.
*ftp://sisal.llnl.gov/pub/hpfc/papers95/paper07.ps

Hanus, M. (1994), 'The integration of functions into logic programming: from theory to practice', *Journal of Logic Programming* **19&20**.

Harper, R. & Mitchell, J. C. (1993), 'On the type structure of Standard ML', *ACM Transactions on Programming Languages and Systems* **15**(2), 211–252.

Henglein, F. & Jørgensen, J. (1994), Formally optimal boxing, *in* 'Conference Record of POPL'94: 21ST ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages', ACM Press, New York, NY, pp. 213–226.

Hesselink, W. H. (1990), 'Axioms and models of linear logic', *Formal Aspects of Computing* **2**, 139–166.

Hindley, R. (1969), 'The principal type scheme of an object in combinatory logic', *Transactions of the American Mathematical Society* **146**, 29–60.

Hoare, C. A. R. (1990), Conjunction and concurrency, *in* 'PARBASE 90'.

Hoare, C. A. R. (1994), 'Unified theories of programming', published online. Oxford University Computing Laboratory.
*ftp://ftp.comlab.ox.ac.uk/pub/Documents/techpapers/Tony.Hoare/theory94.ps

Hoare, C. A. R., He, J. & Sampaio, A. (1995), 'Algebraic derivation of an operational semantics', published online. Oxford University Computing Laboratory.
*ftp://ftp.comlab.ox.ac.uk/pub/Documents/techpapers/Tony.Hoare/hhs94.ps

Horn, M. (1996), Improved integration of multithreading into the STGM, Report B-96-07, Freie Universität Berlin, Institut für Informatik.

Hudak, P. et al. (1992), 'Haskell special issue', *ACM SIGPLAN Notices* **27**(5).

Huet, G., ed. (1990), *Logical Foundations of Functional Programming*, University of Texas at Austin Year of Programming Series, Addison-Wesley.

Hughes, J. (1989), 'Why functional programming matters', *The Computer Journal* **32**, 98—107.

Hum, H. H. et al. (1995), A design study of the EARTH multiprocessor, *in* L. Bic, W. Böhm, P. Evripidou & J.-L. Gaudiot, eds, 'Proceedings of the IFIP WG 10.3 Working Conference on Parallel Architectures and Compilation Techniques, PACT '95', ACM Press, pp. 59–68.
*ftp://ftp-acaps.cs.mcgill.ca/pub/doc/papers/PACT95.ps.gz

Jagadeesan, R. & Pingali, K. (1992), Abstract semantics for a higher order functional language with logic variables, *in* 'Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages', ACM Press, New York, NY. Also appeared as TR91-1220, Cornell University.

Janson, S. (1994), AKL—A Multiparadigm Programming Language Based on a Concurrent Constraint Framework, PhD thesis, Computing Science Department, Uppsala University.

Johnsson, T. (1987), Compiling Lazy Functional Languages, PhD thesis, Department of Computing Science, Chalmers University of Technology, Göteborg, Sweden.

Jones, M. P. (1995), 'A system of constructor classes: Overloading and implicit higher-order polymorphism', *Journal of Functional Programming* **5**(1).

Kamada, T., Matsuoka, S. & Yonezawa, A. (1994), Efficient parallel global garbage collection on massively parallel computers, *in* 'Proceedings of the Conference on Supercomputing', IEEE Computer Society Press, Los Alamitos, CA, pp. 79–88.

Kelly, P. H. J. (1989), *Functional Programming for Loosely-Coupled Multiprocessors*, Pitman.

Kingdon, H., Lester, D. R. & Burn, G. L. (1991), 'The HDG-machine: a highly distributed graph-reducer for a transputer network', *The Computer Journal* **34**(4).

Kobayashi, N. & Yonezawa, A. (1994), Higher-order concurrent linear logic programming, *in* 'Proceedings of Theory and Practice of Parallel Programming', number 907 *in* 'LNCS', Springer-Verlag.

Krishnamurthy, A., Culler, D. E., Dusseau, A., Goldstein, S. C., Lumetta, S., von Eicken, T. & Yelick, K. (1993), Parallel programming in Split-C, *in* B. Borchers, ed., 'Proceedings of the Supercomputing '93 Conference', IEEE Computer Society Press, pp. 262–273.

Lafont, Y. (1996), 'The undecidability of second order linear logic without exponentials', *Journal of Symbolic Logic* **61**(2), 541–548.

Launchbury, J. (1993), A natural semantics for lazy evaluation, *in* 'Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages', ACM Press, New York, NY, pp. 144–154.
*http://www.cse.ogi.edu/~jl/Papers/lazySem.ps

Leroy, X. (1992), Unboxed objects and polymorphic typing, *in* 'Principles of Programming Languages', pp. 177–188.

Lester, D. R. (1989), An efficient distributed garbage collection algorithm, *in* E. Odijk, M. Rem & J.-C. Syre, eds, 'Proceedings of Parallel Architectures and Languages Europe (Volume 1)', number 365 *in* 'LNCS', Springer Verlag.

Lester, D. R. (1992), Distributed garbage collection of cyclic structures, *in* 'Proceedings of the 4th International Workshop on the Parallel Implementation of Functional Languages'. Aachener Informatik Berichte 92-19.

Lloyd, J. W. (1987), *Foundations of Logic Programming (Second Edition)*, Springer.

Lock, H. C. R. (1992), The Implementation of Functional Logic Programming Languages, PhD thesis, Technische Universität Berlin, Fachbereich Informatik. Published by R. Oldenbourg Verlag.

Lock, H. C. R., Keller, G. & Chakravarty, M. M. T. (1995), Generating fast garbage collection methods for declarative languages, *in* 'Mt Fuji International Workshop on Functional and Logic Programming', Japan Society of Software Science and Technology, World Scientific Publishing, Singapore.

Loogen, R. (1993), 'Relating the implementation techniques of functional and functional logic languages', *New Generation Computing* **11**, 179–215.

Loogen, R., Kuchen, H., Indermark, K. & Damm, W. (1989), Distributed implementation of parallel graph reduction, *in* 'Parallel Architectures and Languages Europe, LNCS 365'.

Marriott, K. & Odersky, M. (1995), A confluent calculus for concurrent constraint programming with guarded choice, *in* 'Proceedings of CP'95 Conference on Principles and Practice of Constraint Programming'.

Miller, D. (1992), The π-calculus as theory in linear logic: Preliminary results, *in* E. Lamma & P. Mello, eds, 'Workshop on Extensions to Logic Programming', Springer Verlag.

Miller, D. (1995), 'A survey of linear logic programming', *Newsletter of the Network of Excellence on Computational Logic* **3**.
*ftp://www.cis.upenn.edu/pub/papers/miller/CompuNet95/llsurvey.html

Miller, D. (1996), 'Forum: A multiple-conclusion specification logic', *Theoretical Computer Science* **165**.

Miller, D., Nadathur, G., Pfenning, F. & Scedrov, A. (1991), 'Uniform proofs as a foundation for logic programming', *Annals of Pure and Applied Logic* **51**, 125–157.

Milner, R. (1991), The polyadic π-calculus: a tutorial, *in* F. L. Bauer, W. Brauer & H. Schwichtenberg, eds, 'Logic and Algebra of Specification', Springer-Verlag, pp. 201–246.

Miniussi, A., Metzemakers, T., Sherman, D. J. & Strandh, R. I. (1994), Improving arithmetic performance using fine-grain unfolding, *in* M. Hermenegildo & J. Penjam, eds, 'International Conference on Programming Language Implementation and Logic Programming', number 844 *in* 'Lecture Notes in Computer Science', Springer-Verlag, pp. 324–339.

Mohr, E., Kranz, D. & Halstead, Jr., R. (1991), 'Lazy task creation: A technique for increasing the granularity of parallel programs', *IEEE Transactions on Parallel and Distributed Systems* **2**(3), 264–280.

Mück, A. (1992), Camel: An extension of the categorical abstract machine to compile functional/logic programs, *in* 'Proceedings of the 4th International Symposium on Programming Language Implementation and Logic Programming', number 631 *in* 'Lecture Notes in Computer Science', Springer-Verlag, Berlin, Germany, pp. 341–354.

Mück, A., Streicher, T. & Lock, H. C. R. (1994), A tiny constraint functional logic language and its continuation semantics, *in* D. Sannella, ed., 'Fifth European Symposion on Programming (ESOP '94)', number 788 *in* 'Lecture Notes in Computer Science', Springer-Verlag.

Nikhil, R. (1990), Id (version 90.0) reference manual, Technical report, MIT Laboratory for Computer Science. CSG Memo 284-1.

Nikhil, R. S., Arvind, Hicks, J. E., Aditya, S., Augustsson, L., Maessen, J.-W. & Zhou, Y. (1995), pH language reference manual, version 1.0, Technical Report CSG-Memo-369, Massachussets Institute of Technology, Laboratory for Computer Science.
*ftp://csg-ftp.lcs.mit.edu/pub/papers/csgmemo/memo-369.ps

Nikhil, R. S., Papdopoulos, G. M. & Arvind (1992), *T: A multithreaded massively parallel architecture, *in* 'Proceedings of the 19th Annual International Symposium on Computer Architecture', ACM Press, New York, NY, pp. 156–167.

Nöcker, E. G. J. M. H., Smetsers, J. E. W., van Eekelen, M. C. J. D. & Plasmeijer, M. J. (1991), Concurrent Clean, *in* 'Proceedings of PARLE '91', number 505/506 *in* 'Lecture Notes in Computer Science', Springer-Verlag, pp. 202–220.

Odersky, M. (1995), Applying π: Towards a basis for concurrent imperative programming, *in* 'Proceedings of the 2nd ACM SIGPLAN Workshop on State in Programming Languages'.

Peterson, J. et al. (1996), Haskell 1.3: a non-strict, purely functional language, Research Report YALEU/DCS/RR-1106, Yale University.
*http://haskell.cs.yale.edu/haskell-report/haskell-report.html

Peyton Jones, S. L. (1987), *The Implementation of Functional Programming Languages*, Series in Computer Science, Prentice-Hall.

Peyton Jones, S. L. (1992), 'Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine', *Journal of Functional Programming* **2**(2).

Peyton Jones, S. L. (1996), Compiling Haskell by program transformation: a report from the trenches, *in* H. R. Nielson, ed., 'Proceedings of the European Symposium on Programming', Vol. 1058 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, pp. 18–44.

Peyton Jones, S. L., Clack, C. & Salkild, J. (1989), High-performance parallel graph reduction, *in* E. Odijk, M. Rem & J.-C. Syre, eds, 'Proceedings of PARLE (Volume 1)', Springer Verlag. LNCS 365.

Peyton Jones, S. L., Hall, C. V., Hammond, K., Partain, W. D. & Wadler, P. L. (1993), The Glasgow Haskell Compiler: a technical overview, *in* 'Proc. of the UK Joint Framework for Information Technology (JFIT) Technical Conference'.

Peyton Jones, S. L. & Launchbury, J. (1991), Unboxed values as first class citizens in a non-strict functional language, *in* J. Hughes, ed., 'Proceedings of the International Conference on Functional Programming and Computer Architecture'.

Peyton Jones, S. L., Partain, W. & Santos, A. (1996), Let-floating: Moving bindings to give faster programs, *in* 'Proccedings of the International Conference on Functional Programming'.

Peyton Jones, S. L. & Salkild, J. (1988), The spineless tagless G-machine, *in* 'Workshop on Implementations of Lazy Functional Languages, Aspensas, Schweden'.

Sangiorgi, D. (1993), An investigation into functions as processes, *in* S. Brookes, M. Main, A. Melton, M. Mislove & D. Schmidt, eds, 'Proceedings of the IX Conference on the Mathematical Foundations of Programming Semantics, (MFPS '93)', Vol. 802 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 143–159.
*ftp://ftp.dcs.ed.ac.uk/pub/sad/la.ps.Z

Sankaran, N. (1994), 'A bibliography on garbage collection and related topics', *SIGPLAN Notices* **29**(9), 149–158.

Saraswat, V. A. (1993), *Concurrent Constraint Programming*, The MIT Press.

Saraswat, V. & Lincoln, P. (1992), 'Higher-order, linear, concurrent constraint programming', Manuscript.

Schauser, K. E., Culler, D. E. & Goldstein, S. C. (1995), Separation constraint partitioning - a new algorithm for partitioning non-strict programs into sequential threads, *in* 'Proceedings of the Symposium on Principles of Programming Languages'.

Sestoft, P. (1997), 'Deriving a lazy abstract machine', *Journal of Functional Programming* **7**(3). To appear.

Shapiro, E. (1989), 'The family of concurrent logic programming languages', *ACM Computing Surveys* **21**(3), 412–510.

Smolka, G. (1994), A calculus for higher-order concurrent constraint programming with deep guards, Research Report RR-94-03, DFKI, Postfach 2080, D-67608 Kaiserslautern.

Smolka, G. (1995), The Oz programming model, *in* J. van Leeuwen, ed., 'Computer Science Today', Vol. 1000 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, pp. 324–343.

Somogyi, Z., Henderson, F. J. & Conway, T. C. (1996), 'The execution algorithm of Mercury, an efficient purely declarative logic programming language', *Journal of Logic Programming* **29**(1–3), 17–64.

Sterling, L. & Shapiro, E. (1986), *The Art of Prolog: Advanced Programming Techniques*, The MIT Press.

Südholt, M. (1994), Data distribution algebras — a formal basis for programming using skeletons, *in* 'Proceedings of the 4th International Conference on Programming Methods, Concepts, and Calculi (PROCOMET '94)', IFIP, Consiglio Nazionale delle Ricerche, pp. 38–57.

Taura, K., Matsuoka, S. & Yonezawa, A. (1994), StackThreads: An abstract machine for scheduling fine-grain threads on stock CPUs, *in* 'Proceedings of the Workshop on Theory and Practice of Parallel Programming', Lecture Notes in Computer Science, Springer-Verlag, Berlin, Germany.

Trinder, P. W., Hammond, K., Mattson Jr, J. S., Partridge, A. S. & Peyton Jones, S. L. (1996), GUM: a portable parallel implementation of Haskell, *in* 'Proceedings of Programming Languages Design and Implementation'.

Van Roy, P. L. (1990), Can Logic Programming Execute as Fast as Imperative Programming, PhD thesis, University of California, Berkley.

Volpe, P. (1994), Concurrent logic programming as uniform linear proofs, *in* G. Levi & M. Rodríguez-Artalejo, eds, 'Proceedings of the Fourth International Conference on Algebraic and Logic Programming', number 850 *in* 'Lecture Notes in Computer Science', Springer-Verlag, pp. 133–149.

von Eicken, T., Culler, D. E., Goldstein, S. C. & Schauser, K. E. (1992), Active messages: a mechanism for integrated communication and computation, *in* 'Proceedings of the 19th International Symposium on Computer Architecture', ACM Press.

Wadler, P. (1992), 'Comprehending monads', *Mathematical Structures in Computer Science* **2**, 461–493.

Wadler, P. (1996), 'Lazy vs. strict', *ACM Computing Surveys* .

Waite, W. W. & Goos, G. (1985), *Compiler Construction*, second edn, Springer-Verlag.

Warren, D. H. (1983), An abstract prolog instruction set, Technical Note 309, SRI International.