DISTRIBUTING ABSTRACT MACHINES

by

Olle Fredriksson

A thesis submitted to the
University of Birmingham
for the degree of
Doctor of Philosophy

School of Computer Science
The University of Birmingham
April 2015

Abstract

Today's distributed programs are often written using either explicit message passing or Remote Procedure Calls that are not natively integrated in the language. It is difficult to establish the correctness of programs written this way compared to programs written for a single computer.

We propose a generalisation of Remote Procedure Calls that are natively integrated in a functional programming language meaning e.g. that they have support for higher-order Remote Procedure Calls across node boundaries. There are already several languages that provide this feature, but there is a lack of details on how they can be compiled correctly and efficiently, which is what this thesis focusses on.

We present four different solutions, given as readily implementable abstract machines. Two of them are based on interaction semantics — the Geometry of Interaction and game semantics — and two of them are moderate extensions of conventional abstract machines — the Krivine machine and the SECD machine. To target general distributed systems our solutions additionally support higher-order Remote Procedure Calls *without sending actual code*, since this is not generally possible when running on heterogeneous systems in a statically compiled setting.

We prove the correctness of the abstract machines with respect to their single-node execution, and show their viability for use as the basis for compilation by implementing prototype compilers based on them. In the case of the machines based on conventional machines we additionally show that they enable efficient programs and that single-node performance is not lost when they are used.

Our intention is that these abstract machines can form the foundation for future programming languages that use the idea of higher-order Remote Procedure Calls.

Acknowledgements

First of all, I would like to thank my supervisor Dan R. Ghica, who has been an ever encouraging mentor keeping me en route and my morale high throughout my years as a doctoral student. I also thank my friends and colleagues at the University of Birmingham whose warm and inclusive attitude made me feel welcome right from the start. A particularly big thank you goes to Martín Escardó and Paul Blain Levy who were in my thesis group and spent a lot of time reading paper drafts. I am grateful that you always provided constructive criticism. Thanks to my office mates in office 117: Olaf, Mark, Ben, Kat, and Chris were there from the start; Mohammed, Abdessalam, and Ahmed joined us later. You were always there just to banter but also eager to help with any real difficulties that came up — like a family. A big thanks to the theory group at the university, which is an incredibly knowledgeable, friendly, and inquisitive group of researchers. And thanks to Bertie Wheen who provided invaluable help in implementing Floskel and its benchmarks.

This work was supported by Microsoft Research through its PhD Scholarship Programme. Thanks to my co-supervisors at Microsoft Research Cambridge: Satnam Singh until he joined Google and thereafter Nick Benton. I'm especially grateful to Nick who enabled me to spend an unforgettable summer as an intern in Cambridge.

Thanks to Andrea, Antonio, Luca C, Luca R, Veljko, and Xiaoyu. I will cherish the good times we had making beer together, and I hope we can do it again some time. Thanks also to my friends outside of Birmingham, who I wish I could see more often.

All my love to Ingrid Idunn, my partner and best friend, who I am endlessly grateful to for believing in me at all times.

Last, I want to thank my family for their unconditional love and support.

Olle Fredriksson
April 2015

Contents

Conventional abstract machines

Finale

Figures

Tables

Overture

Chapter 1

Overview

Writing a computer program in a high-level programming language means that we are freed from worrying about many of the laborious details associated with low-level languages. We increase our programmer effectiveness as well as the safety and the correctness of our programs. This is done by employing features such as recursive procedures and control flow statements in place of jumps, automatic garbage collection in place of manual memory management, and static type systems in place of the language being untyped. That programs written using high-level languages sometimes run slower than their low-level counterparts has been an objection to using them from their conception [10], but proponents of high-level programming languages commonly believe that this is outweighed by the benefits of faster development times, extra safety, and correctness.

An important feature of high-level programming languages is *machine-independence*, which is an idea said to originate from the early language Fortran [11].[1] A machine-independent language is one whose programs can be recompiled to different targets without having to be rewritten, since they do not rely on machine-specific details.

This thesis presents the idea of lifting the conventional idea of machine-independent programming languages to *architecture-independence* in the context of distributed computing. Distributed systems are everywhere, but the programs that they run are often written using language features that reflect the details of the target system in the source code, meaning that they are not reusable — large parts of the programs may have to be rewritten if they are to be ported to a different system. Furthermore, the programs are often written using tools that are either not natively integrated in the language, or so low-level and error-prone that they may be compared to programming with *goto*.

[1] The idea of machine-independence should, however, probably not be attributed to Fortran's creators, because initial version included machine-dependent features [10]. It is more likely that it was Fortran's popularity at the time that led to machine-independence: it created an incentive for hardware vendors to implement versions of the language for their own hardware.

1.1 THE PROBLEM

Mental burden Imagine that we are writing a distributed program in a make-believe language with native support for message passing. In this language we write processes as procedures that may use the *pid* ! *message* operation (pronounced "bang!") to send a message asynchronously (without blocking) to the node with process identifier *pid*, and the *message* ← **receive** operation to receive messages synchronously (blocking until a message is received) from anyone.

Let's say that we want to use this language to serve two functions, $q_1, q_2 : Q \to R$, from two different nodes in a network. We may think of the two functions as performing queries in two databases. Being seasoned programmers, we immediately seize the opportunity to avoid repetition, and write the following code:

```
server f = repeat {
  (fromPid, arg) ← receive
  fromPid ! f arg
}
```

A *server* of a function *f* is a program that repeatedly receives a message containing an argument *arg*, and a process identifier, *fromPid*. Each time this happens, the server passes the argument to the function *f* and sends the result back to the network node identified by *fromPid*. Our two servers are now easily expressed by invoking our *server* procedure with either the argument q_1 or q_2:

```
server1 = server q1
server2 = server q2
```

If we want to invoke one of our servers from a remote location, it is a matter of sending the right message to it:

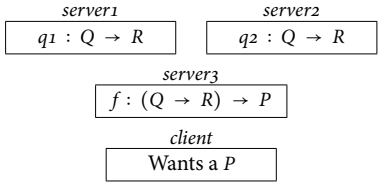
```
server1Pid ! (myPid () , myArg)
```

Here we assume that *server1* runs on the node identified by *server1Pid*, and that *myPid* () returns the process identifier of the calling process.

So far our code looks elegant. But let us now look at what happens if we want to add what may be called a *higher-order* node — one that invokes a node whose identifier is given as an argument. A simple example of such a node is a server that makes some queries to a given database and then returns a processed result. Conceptually, we can think of the type of that server being $(Q \to R) \to P$, i.e. a function that takes as its argument one of the aforementioned query functions q_1 and q_2.

Since it is a server, we rightfully expect that its functionality fits into the *server* mould that we have already written, but since it is a bit more complicated it is probably a good idea to first spell out the new server in full detail:

```
server3 = repeat {
  (fromPid, databasePid) ← receive
  databasePid ! (myPid () , query)
  result ← receive
  databasePid ! (myPid () , anotherQuery)
  anotherResult ← receive
  fromPid ! process result anotherResult
}
```



Our new server receives a message containing *fromPid*, the identifier of the caller, and *databasePid*, identifying the database to call. It then invokes the given database server twice, both times by sending a message containing its own process identifer and a query. When the two results have been received, it passes them to an assumed *process* function, and sends the result back to the caller using *fromPid*.

As expected, we can also rewrite *server3* using the *server* function, which means that we do not have to keep track of the process identifier of the caller:

```
server3' = server {λdatabasePid.
  databasePid ! (myPid () , query)
  result ← receive
  databasePid ! (myPid () , anotherQuery)
  anotherResult ← receive
  process result anotherResult
}
```

To actually use *server3*, we invoke it by sending our own and the target node's process identifiers, and then receiving the result:

```
server3Pid ! (myPid () , server1Pid)
p1 ← receive
server3Pid ! (myPid () , server2Pid)
p2 ← receive
```

Around this point, it becomes difficult for me to keep track of the process identifiers. If your working memory is more capacious than mine, try writing a server of even higher order — it helps to draw a picture.

The above example should have convinced you that it can be tricky to write distributed programs using message passing, *even if they are relatively simple*. The difficulty stems from having to keep track of multiple network locations that provide different functionalities and from passing around references to these network locations — which is reminiscent of passing around code pointers to jump to. Using an abstraction like *server* helped, but note that this is only possible if the program fits into that mould. The servers did not have to keep track of more than three process identifiers at any given time and did not preserve any internal state across invocations, which would have complicated matters even further.

Obscuration of meaning Programs written using message passing can also obscure the intended meaning of the program. As an example, consider the two programs in Figure 1.1.

The second example uses a *selective* receive expression, which synchronously receives a message and picks the branch depending on the received value. In the example, the third component of the received triple — which is one of the constructors **Request1** or **Request2** — determines what branch to pick.

Both of these programs perform the same function, which we would write simply as let $f = \lambda x.\ x * x$ in $f\ 3 + f\ 4$ if it was running on a single node, except that the deployment of the two programs differs. The first program is distributed on two nodes whereas the second is distributed on three nodes. If you did not know that the two programs performed the same function, would you be able to easily tell? Perhaps, but not from a quick glance. Message passing in this style is similar to programming with goto, because the control flow of the system as a whole jumps back and forth between processes without much structure. The structure that we have is what we can invent ourselves ad hoc, by e.g. sending a different datum to invoke different functionalities (**Request1** and **Request2** in this case).

What about Remote Procedure Calls? To mitigate the issues with explicit message passing, many systems make use of the Remote Procedure Call (RPC) [17], which is a mechanism for implementing inter-process communication. An RPC is what its name suggests: a call to a procedure located on a remote node. The idea is that an RPC looks and acts like an ordinary procedure call, and performs the necessary message passing under the hood.

> *An important problem in distributed computing is to provide a user with a nondistributed view of a distributed system.*
>
> Lamport and Lynch [91]

```
squareServer =
  (fromPid, x) ← receive
  fromPid ! x * x
main =
  squareServerPid ! (myPid () , 3)
  x ← receive
  squareServerPid ! (myPid () , 4)
  y ← receive
  x + y
```

```
squareServer =
  (fromPid, x) ← receive
  fromPid ! x * x
proxy =
  receive
    (fromPid, toPid, Request1) →
      toPid ! (myPid () , 3)
      x ← receive
      fromPid ! x
    (fromPid, toPid, Request2) →
      toPid ! (myPid () , 4)
      x ← receive
      fromPid ! x
main =
  proxyPid ! (myPid () , squareServerPid, Request1)
  x ← receive
  proxyPid ! (myPid () , squareServerPid, Request2)
  y ← receive
  x + y
```

Figure 1.1: Examples using message passing

Both of these programs perform the same function, which we would write simply as `let f = λx. x * x in f 3 + f 4` *if it was running on a single node, except that the deployment of the two programs differs. The first program is distributed on two nodes whereas the second is distributed on three nodes.*

Some might say that RPCs are the solution to the above problem in the context of programming languages. However, RPCs tend to be added to a language as an afterthought, e.g. having support only for arguments of ground type, rather than being tightly integrated into it. Even though a remote call *looks* like an ordinary function call, this means that it cannot be *treated* like one. Tanenbaum and Van Renesse [128] observed this almost 30 years ago, but the solution — more transparently integrating RPCs into the language — appears to be dismissed for efficiency reasons. It is time to revisit this issue, not least because there has been substantial progress in the efficiency and capacity of computers since then.

1.2 THIS THESIS

Our proposed solution to the problem of Lamport and Lynch [91] — to provide the user with a nondistributed view of a distributed system — is to separate the architecture-specific from the algorithmic parts of a program. In the context of distributed computing, the architecture-specific parts of a program are the details of its run time deployment and process management. To be architecture-independent, the program thus has to abstract from these features.

We propose the following way to write distributed programs to achieve this. Instead of using explicit message passing, the programmer indicates the location of definitions or subterms using location annotations, written _@_. The first example from the previous section — the "higher-order" database queries — can then be rewritten as follows:

$$
\begin{aligned}
&q1 \mathbin{@} server1 \;=\; \ldots \\
&q2 \mathbin{@} server2 \;=\; \ldots \\
&f \;\; \mathbin{@} server3 \;=\; \ldots \\
&main \mathbin{@} client \;=\; process\ (f\ q1)\ (f\ q2)
\end{aligned}
$$

The difference between this program and one using Remote Procedure Calls is that we are here free to use higher-order functions across node boundaries. It is the job of the compiler and the runtime system to realise any deployment that the programmer can conceive. This approach is also in obvious contrast to languages and libraries that use message passing such as Message Passing Interface (MPI) [65] and Erlang [8]. For full generality, we also want to do this *without sending actual code*, in contrast to e.g. Remote Evaluation [126]. The reason is that this is sometimes necessary: not all code is meaningful on all nodes, for example because of the location of a resource or because of platform differences in a heterogeneous system. It should be noted that this does not necessarily prevent us from sending code (or references to already existing code) when this is desired, e.g. for efficiency.

1.2.1 *What this thesis is actually about*

The above way of writing programs is not entirely new, but is similar to e.g. para-functional programming [75] and Caliban [85]. The focus of this thesis is on the core evaluation mechanism, in the form of abstract machines, that is used to actually run programs with location annotations, which is something that has not been investigated in full before (see Section 2.1 and Section 7.9 for relevant literature reviews). Our first requirement for this mechanism is correctness with respect to the same program without annotations, i.e. that we really are providing a nondistributed view of the system. In distributed computing, this is also called *network transparency* [27]. The second requirement is that we should *enable* the programs to be efficient. We cannot *guarantee* that all programs are efficient: it is easy indeed to come up with examples that can be expected to be slow-running (think e.g. of mapping a remotely located function over a long list). Our performance requirements are instead that we do not lose *single-node* performance when we are using our language without annotations and that we do not put an excessive burden on the network when we do.

Obviously, there are some architecture-specific remnants still in the code above, namely the locations. This is convenient when describing semantics and compilation, and it is furthermore not difficult to construct a language where the configuration of the program is separated from the algorithm and then straightforwardly translated into our language. Our expectation is that location annotations are general enough that future researchers or engineers can build programming languages based on this work.

Since the location annotations are similar to Remote Procedure Calls, this thesis may more accurately be seen as the answer to the following question:

> *From a programming language perspective, how can we do higher-order Remote Procedure Calls correctly, generally, and without sacrificing single-node performance?*

1.2.2 *What this thesis is not*

Even though distributed systems are often used for the purpose of speeding up computations by parallelising their evaluation, the goal of this dissertation is *not* to achieve speedups through parallelisation. We want to increase the expressiveness and automate some aspects of the programming of distributed systems using more conventional languages, i.e. those not tailored for distributed computing. Distributed systems are also naturally concurrent, which is something that we have to take into consideration when modelling the systems. Our proof of concept implementations do not generally include constructs for

concurrency, but it should be stressed that our work does not preclude parallel and concurrent execution — it is just not its focus.

1.3 CONTRIBUTION

Our original contributions to knowledge are in the following areas:

1.3.1 *Conventional abstract machines*

We present new extensions of two classic abstract machines — the Krivine machine (Chapter 6) and the Stack-Environment-Control-Dump (SECD) machine (Chapter 7) — giving them the ability to run in distributed systems and support for higher-order Remote Procedure Calls through the usage of location annotations. We formally prove the soundness of the extensions, and in the case of the SECD machine also the completeness, by exhibiting simulation relations between the extensions and the original machines. The full formalisations can be found in the online appendix: `http://epapers.bham.ac.uk/1985/`.

1.3.2 *Implementations*

For each of the distributing abstract machines presented in this thesis we have also made an implementation, the source code of which is also in the online appendix.

Our main implementation, Floskel (Section 7.2), is based on our extension of the SECD machine. It is a full-featured programming language implementation with support for both location annotations, i.e. code fragments that are tied to a specific location, and *ubiquitous* functions, i.e. functions that can freely be transmitted over the network. It additionally has support for algebraic datatypes and pattern matching.

1.3.3 *Interaction semantics*

We present novel applications of interaction semantics to the compilation of programming languages with higher-order Remote Procedure Calls targeting distributed systems by presenting new abstract machines for the Geometry of Interaction (Chapter 3) and game semantics (Chapter 4). These abstract machines serve as an intermediary between theory and implementation: they allow us to show the soundness of the interpretations while corresponding closely to conventional computers. They also have remarkable features like

requiring no garbage collection. We illustrate their potential by implementing prototype compilers based on them.

1.3.4 *Fault-tolerance*

We present a simple way to achieve fault-tolerance (Chapter 8) for abstract machines similar to those above, by layering a commit-and-rollback mechanism on top of them.

1.4 PREVIOUS PUBLICATIONS

This dissertation is based on the following publications, which were — as indicated — written in collaboration with several co-authors:

- Olle Fredriksson and Dan R. Ghica. "Seamless Distributed Computing from the Geometry of Interaction". In: *Trustworthy Global Computing - 7th International Symposium, TGC 2012, Newcastle upon Tyne, UK, September 7-8, 2012, Revised Selected Papers*. Ed. by Catuscia Palamidessi and Mark Dermot Ryan. Vol. 8191. Lecture Notes in Computer Science. Springer, 2012, pp. 34–48
- Olle Fredriksson and Dan R. Ghica. "Abstract Machines for Game Semantics, Revisited". In: *28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013, New Orleans, LA, USA, June 25-28, 2013*. IEEE Computer Society, 2013, pp. 560–569
- Olle Fredriksson. "Distributed call-by-value machines". In: *CoRR* abs/1401.5097 (2014)
- Olle Fredriksson and Dan R. Ghica. "Krivine nets: a semantic foundation for distributed execution". In: *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*. Ed. by Johan Jeuring and Manuel M. T. Chakravarty. ACM, 2014, pp. 349–361
- Olle Fredriksson, Dan R. Ghica, and Bertram Wheen. "Towards native higher-order remote procedure calls". In: *Proceedings of the 26th Symposium on Implementation and Application of Functional Languages, Boston, MA, USA, October 1-3, 2014*. 2014

	Interaction semantics		Conventional abstract machines		
	SIC machine	**Game nets**	**DKrivine machine**	**DCESH machine**	**Floskel**
Location	Chapter 3	Chapter 4	Chapter 6	Chapter 7	Section 7.2
Basis	GOI	Game semantics	Krivine machine	SECD machine	DCESH machine
Evaluation	CBN	CBN	CBN	CBV	CBV
Language	PCF + @	PCF + @ + local state + par	PCF + @	PCF + @	PCF + @ + algebraic datatypes + pattern matching + ubiquitous functions
Garbage collector	Not required	Not required	Distributed	Distributed	Local

Table 1.1: Thesis overview

1.5 ORGANISATION

This dissertation describes the core evaluation mechanism that is required for a language supporting seamless distribution. See Table 1.1 for an overview of the features of the abstract machines and implementations that are presented in this thesis. We give four alternative mechanisms, in the form of abstract machines with support for native higher-order Remote Procedure Calls, and a more realistic implementation called Floskel. For the machines we have also implemented proof-of-concept compilers, and shown the machines' correctness with respect to single-node execution. The source language that the machines operate on is PCF, either call-by-name or call-by-value, with the additional *t @ node* operation and some language extensions. An important consideration for implementations of distributing machines is how garbage collection is carried out. Our first two solutions, those based on interaction semantics, do not require garbage collection, whereas our extensions of conventional machines require distributed garbage collection. Our most full-fledged implementation gets by with *local* garbage collection by moving more data between nodes.

The work is exploratory in nature; new solutions spring from the uncovering of problems in earlier solutions. To make the narrative logical, the parts are presented in the order they were written. This means that the solutions that we deem to be the strongest are late in the thesis, but note that there is no mutual dependence between the solutions presented in the thesis; a reader just looking for a pre-canned solution can and might want to skip ahead.

The online appendix, available at `http://epapers.bham.ac.uk/1985/`, contains the source code for each of the solutions.

Chapter 2

Background

Synopsis This chapter introduces some of the prerequisite concepts that will be used in the rest of this dissertation.

We introduce the related work that is presentable without the context of our work (Section 2.1), we describe what an abstract machine is (Section 2.2) and the use of abstract machines as a bridge between operational semantics and implementations.

We give a short definition of distributed computing (Section 2.3).

We also introduce Agda (Section 2.4), an interactive proof assistant and programming language based on intuitionistic type theory, which is used to prove some of the theorems in this dissertation. We motivate its usage and give a short description of its syntax. Agda gives us several powerful means for abstraction. We will put this to use by factoring out the common functionality of the abstract machines that this dissertation gives. More precisely, we define a parameterised module of networks that can be instantiated with an underlying abstract machine's transition relation (Section 2.5).

2.1 RELATED WORK

Programming languages and libraries for distributed and client-server computing — which is a simpler form of distribution — are a vast area of research. Relevant to us are *functional* programming languages for distributed execution, and several surveys are available [131, 98].

The following sections are a categorisation of the existing work with comparisons to the goals of our work. Some of the more specialised related work is however presented after we have presented our own work (in Section 4.4, Section 3.5, and Section 7.9), enabling us to make more in-depth comparisons.

2.1.1 *Explicit*

Functional programming languages for distributed systems take different approaches in terms of process and communication management. Languages such as Erlang [8], which are meant for system-level development offer a low-level view of distribution in which both the process and communication are managed explicitly; Erlang is similar to the language that we used for con-

trasting effect in the introduction (Chapter 1). The Akka [5] toolkit, Cloud Haskell [37], and the more low-level MPI [65] library (which has bindings for a multitude of programming languages) use a conceptually similar programming model based on message passing. Some languages in this category use mechanisms imported from process calculi, such as Occam [102], Pict [135], and Nomadic Pict [139, 125]. Nomadic Pict extends the calculus with constructs for location dependence making it more suitable for distributed computing. Nomadic Pict and the distributed join calculus [42] both support a notion of mobility for distributed agents (that is, the process you are sending to may move), which enables expressing dynamic distributions of programs. The work on Nomadic Pict describes how a higher-level language can be based on a small set of constructs with the capability to do location-independent communication in the presence of mobile agents. To achieve this, they use a combination of a central server that keeps track of where every process is, and local caching to eliminate some requests to the central server. The caching is reminiscent of using forward pointers, where a migrating process leaves a forwarding pointer on its old node, which can then forward requests. Here, software agents are explicitly sent across the network between running processes. By contrast, in our methodology no code needs to be transmitted (though it can).

Programming languages do not need to be created from scratch to include improved language support for communication. Session types have been used to extend a variety of languages, including functional languages, with safer, typed communication primitives [136] or to provide language-independent frameworks for integrating distributed applications, such as Scribble [140]. There are some parallels to be drawn between this line of work and ours. For instance, our way of compiling a single program to multiple nodes that we will see in the coming chapters can be likened to the projection operator in multiparty session types [72].

Glasgow Distributed Haskell [118] extends the existing parallelism constructs in Haskell for use in a distributed system. These constructs are at a low level of abstraction, providing familiar constructs from parallel and concurrent programming such as locks and channels ported to the world of distributed programming.

Even though these languages have a low-level view of distribution, it does not mean that it is not possible to build abstractions in them. We saw an example of this in the introduction (Chapter 1), namely the *server* abstraction. The Open Telecom Platform (OTP) [129] provides constructs similar in spirit — but obviously more elaborate — for Erlang programs. This significantly raises the level of abstraction, at least for programs that fit into the moulds.

2.1.2 *Implicit*

Our approach is quite different compared to those based on explicit communication. Our aim is to make communication implicit, or *seamless*. In some sense this is already widely used in programming practice, especially in the context of client-server applications, in the form of RPCs [17] and related technologies such as Simple Object Access Protocol (SOAP). What we aim to do is to integrate these approaches into the programming language so that from a programmer perspective there is no distinction between a remote and local call, even at higher order. A project close to our aim is Remote Evaluation (REV) [126], which is another generalisation of RPC that enables the use of higher-order functions across node boundaries. The main differences between REV and our work is that REV relies on sending unevaluated code.

Languages that use location annotations similar to ours have also been proposed. The first one is called para-functional programming [75], where the language is a functional language with an additional *e on e'* construct for expressing distribution. This is close to our location annotations, with the difference that *e'* is an expression in the language which evaluates to the process identifier of the target node. The distribution is thus dynamic. In our work we want to make it possible to run programs in heterogeneous systems where not all code is meaningful at all nodes, which in general precludes this kind of dynamic distribution. It should be noted that our work on ubiquitous functions (presented in Section 7.2; essentially function references that are safe to transmit between nodes) could be extended in a straightforward manner to support this. Since we already have the machinery in place to invoke ubiquitous functions on arbitrary remote nodes, we are just a minor compiler update away from making the node selection dynamic.

The Caliban language [85] offers more separation between the algorithmic parts and those to do with the distribution of the language. The distribution is specified programmatically also here, but required to be statically known (through a process that can be likened to partial evaluation).

Related object oriented approaches are Emerald [83], Obliq [23], and the recent RPyC [121]. Kanor [71] is another project that similarly aims to simplify the development of distributed programs by providing a declarative language for specifying communication patterns inside an imperative host language (C++). Object orientation raises a slightly different set of challenges than functional languages, especially to give a consistent view of objects that may be mutated and migrate between nodes. A few interesting calling conventions arise naturally in this setting: call-by-reference, where a remote reference is sent to the destination node, call-by-visit, where the argument object is moved to the destination node and moved back to the source when the

method call has finished, and call-by-move, where the object is moved to the destination node when the call is performed but not moved back afterwards. As objects can move around the network and keep references to each other distributed garbage collection is an issue.

This wealth of prior work gives us strong evidence that our approach is reasonable and viable. Our main innovation is not on the language side, but in giving ways to compile languages that use similar distribution mechanisms in a sound way by using abstract machines, which is something that has not been done before.

Totoo, Deligiannis, and Loidl [130] present a comparison of the high-level parallelism features of the Haskell, F#, and Scala programming languages. Even though this work is about parallelism, it drives home a point that is also worth mentioning here. The main takeaway is that the language with the highest level of abstraction (Haskell using Glasgow Parallel Haskell [132]) actually performed the best and was easier to parallelise than the others, using so called *skeletons*, i.e. higher-order functions providing a parallel algorithm — in this case parallel map. A concluding remark in the paper is that it is important for languages to support *primitive* operations for parallel operations, as compared to library implementations of them. In Haskell's case, this is realised through operations that talk to its runtime system, for instance to implement light-weight threads, which would have been difficult to do in a library. These results should also be taken into account when designing languages for distributed computing.

2.1.3 *Distributed execution engines*

Distributed execution engines are software frameworks that provide a high-level programming model for data processing applications by protecting the programmer from many of the difficulties associated with writing such programs, e.g. task scheduling, data transmission, and fault-tolerance. As such, some execution engines fall into the implicit category above, but since they are geared towards data processing they often impose constraints on the data access and communication patterns of their allowed programs to increase performance.

A well-known but comparatively limited example is MapReduce [35], which is a programming model for large-scale distriuted systems with several implementations. It allows the programmer to specify a map function, which is first mapped over the data set, and a reduce operation, which combines the results of the map. It can thus only solve problems on a form that is easily decomposed into independent map and reduce tasks.

Other examples include CIEL [109] and Dryad [79], which are execution engines for distributed *data-flow* programs. Execution is performed by the traversal of a directed acyclic graph of tasks written in an imperative programming language, which can be dynamic and data-dependent. The distribution is at the task level. In the case of CIEL, the traversal is lazy; it starts from the result and works through the graph to determine what tasks need to be run to get the result. The evaluation is eager at the task level. The operational semantics of both systems are presented informally in text-form, which can make it hard to reason about the systems and their correctness. On the application side, they both present impressive performance results, and make use of a central server that takes care of fault-tolerance (through periodic heartbeat messages and restarting of the affected parts of a computation), task allocation, and load balancing.

2.1.4 *Tierless computing*

In the last ten years a number of *tierless* languages have appeared. Examples include Hop [124], Links [29], Ocsigen [12], ML5 [137], and Ur/Web [24, 25]. These languages have some similarities with the implicit languages above, but the emphasis here is not on network transparency, but on unified all-in-one languages for web programming, which is typically limited to client-server computing. Being unified and all-in-one means that the same language can be used for both client and server, and (using embedded domain-specific languages) for type-safe database queries and HTML generation.

Another related approach for building web applications is Swift [26], where the focus is on automation and security; the program is automatically partitioned into client (running Javascript) and server (running Java) such that security critical code only ever runs on the server.

2.2 ABSTRACT MACHINES

Programming languages are typically implemented by compilation to machine-code or other suitably low-level target languages. But when we want to describe how compilation is done it would be arduous to describe the translation all the way to machine-code, especially for the person having to read that description. Operational semantics such as small-step [116] or big-step [84] semantics can be enough to define an *interpreter* for a language, but do not necessarily give away the details required to write a *compiler* for the language, especially one that generates efficient output programs.

As an example, the small-step β-rule used in the reduction of lambda terms,

$$(\lambda x.t)\ t' \longrightarrow_\beta t[t'/x],$$

is too abstract for a compiler-writer to efficiently implement, since a naive substitution requires a traversal of the whole syntax tree of the term, whereas the reduction could be implemented in constant time by a seasoned compiler writer. Something is lost in abstraction in the presentation of the β-rule. In the Krivine machine [86], an abstract machine for call-by-name reduction of lambda terms, this operation is instead done by adding the argument (and its environment) to an environment,

$$((\lambda x.t)\ t', e, s) \longrightarrow^2 (t, e \uplus \{x \mapsto (t', e)\}, s),$$

an operation that at least gives us a hint for how an efficient implementation can be done. By using the De Bruijn index notation [20], where variables are natural numbers that stand for their index into a sequential environment, we can even implement that operation as a simple stack pushing operation:

$$((\lambda.t)\ t', e, s) \longrightarrow^2 (t, (t', e) :: e, s),$$

Like the Krivine machine, an abstract machine typically consists of a code component — sometimes represented as an intermediate bytecode, and sometimes just a source language term — and one or more data components — typically stacks and heaps [36].

The term abstract machine has been used to refer to constructions at a spectrum of different levels of abstraction, as indicated by the following quote:

> *Some abstract machines are more abstract than others.*
>
> Diehl, Hartel, and Sestoft [36][1]

In this dissertation, we will consider abstract machines as a kind of operational semantics whose purpose is to describe compilation schemes at a level of abstraction somewhere between operational semantics that use substitution, and the actual compiled program. Using an abstract machine to describe a programming language implementation thus divides the compiler's job in two parts: compiling the source code to an abstract machine and compiling the abstract machine to a concrete machine.

[1] Dean Martin, the famous American singer, would probably have wished that the authors of this quotation would send the pillow that they dream on so he could dream on it too.

Abstract machines are typically deterministic, i.e. they can make at most one transition from any state, and this is usually governed by their code component. For the compiler writer, determinism means that a given transition sequence can be implemented as a sequence of machine instructions. Since this thesis concerns languages for distributed and concurrent systems, we deviate from this guideline when modelling the system as a whole, but single-threaded execution will remain deterministic, which is enough for a compiler writer.

2.3 DISTRIBUTED COMPUTING

Definition 2.3.1 (Distributed system)**.** A *distributed system* is a computer system with a number of processing units, *nodes*, that communicate over a network by sending messages to each other. Sending messages may be the only means of communication between the nodes, meaning that they do not generally share an address space.

A *distributed program* is one running and making use of a distributed system. *Distributed computing* is computing in distributed systems.

2.4 AGDA

The definitions and proofs in this dissertation are intricate and often consist of many cases, so carrying them out manually is arduous and can be error-prone, which is something that we noticed while working on the early chapters of this thesis. Proof assistants can alleviate this burden by providing proof checking and automation. They can also be helpful tools in producing proofs, providing interactive environments in which to play with alternative definitions — even wrong ones. Our tool of choice is Agda [111], which is both an interactive proof assistant and a programming language. To eliminate another source of error, we present the Agda code as is when this is possible.

Although the work is not about Agda *per se*, this presentation should be beneficial also to you, the reader, since you can trust that the propositions do not contain mistakes. It also means that we can present technical results with a high degree of confidence while letting the focus be on the exposition rather than tedious proof details. Since Agda builds on a constructive foundation, it also means that a formalisation can act as a verified prototype implementation.

2.4.1 *Syntax and notation for code*

We assume a certain familiarity with the syntax of Agda, but since it is close to that of several popular functional programming languages it should not cause

much difficulty for the audience. If it does, there are several excellent guides and tutorials available (e.g. [18]). We will use *Set* for the type of types. We will use *implicit parameters*, written e.g. $f : \{A : Set\} \to ...$ which means that f takes, as its first argument, a type A that does not need to be explicitly spelled out when it can be inferred from the context. We will sometimes use the same name for constructors of different types, and rely on context for disambiguation. Constructors will be written in **boldface** and keywords without serifs. We make liberal use of Agda's ability to define *mixfix* operators. An example is **ifo_then_else_** which a constructor that accepts arguments in the positions of the underscores: **ifo** b **then** t **else** f. We will also explain some Agda idioms when they are used.

2.5 NETWORK MODEL

In this section we define two models for distributed communicating networks, based either on synchronous message passing (blocking send) or on asynchronous message passing (non-blocking send). The model (the Agda module called *Network*[2]) is parameterised by the underlying transition relation of the machines:

$$\_ \vdash \_ \longrightarrow \langle \_ \rangle \_ : Node \to Machine \to Tagged\ Msg \to Machine \to Set$$

The types *Node*, *Machine*, and *Msg* are additional parameters. Elements of *Node* will act as node identifiers, and we assume that these enjoy decidable equality — in MPI, a low-level library for message passing, they would correspond to the so called integer "node ranks". The type *Machine* is the type of the nodes' configurations, and *Msg* the type of messages that the machines can send. The presence of the *Node* argument means that the configuration of a node knows about and may depend on its own identifier. The type constructor *Tagged* is used to separate the different kinds of local transitions: a *Tagged Msg* can be $\boldsymbol{\tau}$ (i.e. a silent transition), **send** *msg*, or **receive** *msg* (for $msg : Msg$).[3] These tagged messages are inspired by *actions* in process calculi such as the Calculus of Communicating Systems (CCS) [104].

Both kinds of networks are modelled by two-level transition systems, similar to the Distributed Eden Abstract Machine (DREAM) [19]. A *global level* describes the transitions of the system as a whole, and a *local level* the transitions of the nodes in the system. Synchronous communication is modelled by *rendezvous*, i.e. that two nodes have to be ready to send and receive a message at a single point in time. Asynchronous communication is modelled using

[2]Online appendix: `krivine/formalisation` directory, `Network` module.
[3]Online appendix: `krivine/formalisation` directory, `Tagged` module.

$$\frac{i \vdash nodes\ i \longrightarrow \langle \boldsymbol{\tau} \rangle\ m'}{nodes \xrightarrow[Sync]{} nodes[i \mapsto m']}\ \textbf{silent-step}$$

$$\frac{\begin{array}{c} s \vdash nodes\ s \longrightarrow \langle \textbf{send}\ msg \rangle\ sender' \\ r \vdash nodes'\ r \longrightarrow \langle \textbf{receive}\ msg \rangle\ receiver' \qquad nodes' = nodes[s \mapsto sender'] \end{array}}{nodes \xrightarrow[Sync]{} nodes'[r \mapsto receiver']}\ \textbf{comm-step}$$

```
data _⟶[Sync]_ (nodes : SyncNetwork) : SyncNetwork → Set where
  silent-step : ∀ {i m'} → (i ⊢ nodes i ⟶ ⟨ τ ⟩ m') → nodes ⟶[Sync] (nodes [ i ↦ m' ])
  comm-step : ∀ {s r msg sender' receiver'} →
    let nodes' = (nodes [ s ↦ sender' ]) in
    (s ⊢ nodes s ⟶ ⟨ send msg ⟩ sender') →
    (r ⊢ nodes' r ⟶ ⟨ receive msg ⟩ receiver') →
    nodes ⟶[Sync] (nodes' [ r ↦ receiver' ])
```

Figure 2.1: Network transitions, synchronous

In a SyncNetwork *messages are passed directly between machines. Network transitions are either a* ***silent-step*** *when a node makes a* $\boldsymbol{\tau}$ *transition, or* ***comm-step*** *when two nodes exchange information by rendezvous. In* ***comm-step*** *a node* s *first takes a step sending a message, and afterwards a node* r *(which can be the same as* s*) takes a step receiving the same message. We show the rules using conventional syntax at the top and Agda syntax at the bottom.*

a "message soup", representing messages currently in transit, inspired by the Chemical Abstract Machine (CHAM) [15].[4] Formally, the two kinds of networks are:

$$\begin{aligned} SyncNetwork &= Node \to Machine \\ AsyncNetwork &= (Node \to Machine) \times List\ Msg \end{aligned}$$

This means that the asynchronous network is, in addition to a family of machines indexed by *Node* identifiers, a global multiset of messages *List Msg* — the aforementioned message soup — in which sent messages are placed, and from which received messages are retrieved.

The definitions of the networks' transition relations are given in Figure 2.1 and Figure 2.2. We show the rules using both conventional syntax and Agda syntax. In conventional mathematics we would define these relations as the subsets of $SyncNetwork \times SyncNetwork$ (in the synchronous case) generated by some given transition rules. In Agda we define the type of such a relation as $SyncNetwork \to SyncNetwork \to Set$, where *Set* is the type of types. If we have a

[4]Note that the CHAM is not an abstract machine in our usage of the term (Section 2.2).

$$\frac{i \vdash nodes\ i \longrightarrow \langle tmsg \rangle\ m' \qquad (msgin\ ,\ msgout)\ =\ detag\ tmsg}{(nodes\ ,\ msgsl\ +\!\!+\ msgin\ +\!\!+\ msgsr) \xrightarrow[Async]{} (nodes[i \mapsto m']\ ,\ msgsl\ +\!\!+\ msgout\ +\!\!+\ msgsr)}\ \textbf{step}$$

```
data _⟶[Async]_ : AsyncNetwork → AsyncNetwork → Set where
  step : ∀ {nodes} msgsl msgsr {tmsg m' i} →
    let (msgin , msgout) = detag tmsg in
    (i ⊢ nodes i ⟶ ⟨tmsg⟩ m') →
    (nodes , msgsl ++ msgin ++ msgsr) ⟶[Async] (nodes[i ↦ m'] , msgsl ++ msgout ++ msgsr)
```

Figure 2.2: Network transitions, asynchronous

An AsyncNetwork *has only one rule,* ***step****, because no synchronisation is needed. A machine on a node can take a* τ *step or a communication step, case in which a message is placed or removed from the global set of messages. Here we use the list append function,* _++_*, on the lists of messages, allowing sent and received messages to be taken from any position in the message list. We show the rules using conventional syntax at the top and Agda syntax at the bottom.*

relation R of that type, two *SyncNetwork*s a and b are taken to be R-related precisely when the type $R\ a\ b$ is inhabited. Given this representation of relations, it is convenient to define relations as inductive datatypes with a constructor per transition rule.

In the synchronous *SyncNetwork* messages are passed directly between machines. Network transitions are either a **silent-step** when a node makes a τ transition, or **comm-step** when two nodes exchange information by rendezvous. In **comm-step** a node s first takes a step sending a message, and afterwards a node r (which can be the same as s) takes a step receiving the same message.

The asynchronous *AsyncNetwork* has only one rule, **step**, because no synchronisation is needed. A machine on a node can take a τ step or a communication step, case in which a message is placed or removed from the global set of messages. Here we use the list append function, _++_, on the lists of messages, allowing sent and received messages to be taken from any position in the message list. The function *detag* is used to determine what messages a node is sending and receiving, allowing one rule for all three cases, as at most one of *msgin* and *msgout* in the rule is non-empty:

```
detag : {A : Set} → Tagged A → List A × List A
detag τ          = [] , []
detag (send x)   = [] , [ x ]
detag (receive x) = [ x ] , []
```

To explain this function in more detail, we consider what happens in the three cases: if the node takes a *silent* step, the list stays the same before and after; if the node *sends* a message, it has to be there *after*; if the node *receives* a message, the message has to be in the list *before* the transition. Note that the return type of *detag* is larger than necessary; it could have returned a pair of *Maybe A*s. However, we would then have to convert the *Maybe*s to *List*s when using the function in the transition rule.

Another helper function used in the definitions is _[_ ↦ _], which updates the state of a node in the network. It is the usual function update, commonly written as $(f \mid x \mapsto y)$, here relying on the assumption that the set of node identifiers has decidable equality ($\_\stackrel{?}{=}\_$). It is formally defined as:

```
_[_ ↦ _] : {A : Set} → (Node → A) → Node → A → Node → A
nodes[n ↦ m] n' with n' ≟ n
nodes[n ↦ m] n'       | yes _ = m
nodes[n ↦ m] n'       | no _  = nodes n'
```

In Agda, the with keyword introduces patterns additional to the arguments in a function definition.

The following theorem shows that asynchronous networks subsume synchronous networks, i.e. that we can always convert a synchronous trace to an asynchronous one.

Theorem 2.5.1. If $(a \xrightarrow[Sync]{} b)$ then $(a\,,\,[]) \xrightarrow[Async]{}^{+} (b\,,\,[])$.

Here $\_^{+}$ takes the transitive closure of a relation. We prove this in Agda by constructing a function mapping a *Sync* transition to an *Async* one by placing, then removing, the message in the global message pool (we construct elements of $\_^{+}$ with a list-like notation):

```
Sync-to-Async⁺ : ∀ {a b} → (a       --Sync-->    b) →
                           (a , []) --Async-->⁺ (b , [])
Sync-to-Async⁺ (silent-step s)     = [ step [] [] s ]
Sync-to-Async⁺ (comm-step s₁ s₂)   = step [] [] s₁ :: [ step [] [] s₂ ]
```

Going in the other direction is not possible in general, but for some specific instances of the underlying transition relation it is, as we will see later.

Variation I

INTERACTION SEMANTICS

Chapter 3

Geometry of Interaction

One of the most profound discoveries in theoretical computer science is the fact that logical and computational phenomena can be subsumed by relatively simple communication protocols. This understanding came independently from Girard's work on the Geometry of Interaction (GOI) [61] and Milner's work on process calculi [106], and influenced the subsequent development of game semantics (see [51] for a historical survey). Of the three, game semantics proved to be particularly effective at producing precise mathematical models for a large variety of programming languages, solving a long-standing open problem concerning higher-order sequential computation, namely full abstraction for Programming Computable Functions (PCF) [3, 77].

An appealing features of game semantics is that it has a dual denotational and operational character. By *denotational* we mean that it is compositionally defined on the syntax and by *operational* we mean that it can be effectively presented and can form a basis for compilation [52]. This feature was apparent from the earliest presentations of game semantics [78], although the operational aspects are less perspicuous than in interpretations based on process calculi or GOI, which quickly found applications in compiler [100] or interpreter [14] development and optimisation.

Our interest in these interaction-based semantics is slightly different. The support of higher-order Remote Procedure Calls (RPCs) in a programming language can at first seem to require sending code between nodes. But, as stated in the introduction, this is needlessly restrictive: to be as general as possible we also want to support such calls without transmitting code between nodes. Interaction-based techniques have the potential to give us what we need to do that, because they tell us exactly how a program or subprogram may interact — communicate — with its environment. By taking inspiration from such techniques, we can expect to solve the problem in a correct and elegant way.

This part of the thesis explores the idea of using interaction-based programming language semantics as the basis for compilation targeting distributed systems. Since the communication between subterms is inherent in these models, the remaining difficulty is to control the granularity of the communication networks. We present these systems in the form of new abstract machines that are readily implementable in actual compilers, evident by our prototype compiler

implementations. The first compiler (Chapter 3) works by compiling to token-passing abstract machines based on the Geometry of Interaction [61]. The second approach (Chapter 4) is based on game semantics, and has support for parallelism and local state.

Synopsis We show how the paradigmatic, higher-order, functional, recursive programming language PCF [117], extended with location annotations, can be compiled to distributed systems using a technique directly inspired by the Geometry of Interaction (GOI).

The GOI model reduces a program to a static network of elementary nodes. Although this is suitable for e.g. hardware synthesis [53], where the elementary nodes become elementary circuits and the network becomes the circuit interconnect, it is generally too fine-grained for distributed computing. We address this technical challenge by introducing a new style of abstract machine with elementary instructions for both control (jumps) and communication. These machines can (almost) arbitrarily be *combined* by replacing communication with jumps, which gives a high degree of control over the granularity of the network.

Our compiler works in several stages. First it creates the fine-grained network of elementary communicating abstract machines. Then, using node annotations (labels), it combines all machines which arise out of the compilation of terms using the same label. The final step is to compile the abstract machines down to executable code using C for local execution and Message Passing Interface (MPI) for inter-machine communication.

3.1 PCF AND ITS GOI MODEL

Girard's Geometry of Interaction [61] is a model for linear logic [62] used for the study of the dynamics of computation, seeing a proof in the logic as a proof net, executed through the passing of a token along its edges. The initial aim of GOI was to avoid the syntactic bureaucracy of proofs, but it is well-known that it can be extended to also interpret programming languages and that it is useful in compiling programs to low-level machine code [100]. Here we will use it as an interpretation for terms in our language, but we will use the notion of a proof net quite literally in that our programs will be compiled into networks of distributed communicating nodes that operate on and send a token to each other.

Our source language is call-by-name PCF with natural numbers as the only base type, extended with an annotation $t@A$, for specifying the location of a

term's evaluation:

$$t ::= x \mid \lambda x.\ t \mid tt \mid \textbf{if } t \textbf{ then } t \textbf{ else } t \mid n \mid t \oplus t \mid \mathbf{Y}\ t \mid t@A.$$

We choose PCF because it is a semantically well-understood language which lies at the foundation of practical programming languages such as Haskell. The additional annotation is to be thought of as a compiler directive: the operational semantics that we use as our specification and the typing rules ignore the location annotation and are otherwise standard [117]. The reason for ignoring the location annotations at this stage is that our goal is to achieve network transparency, i.e. that the location of a program's subterms does not affect its final result. The $\oplus$ operator stands for a primitive binary operation on natural numbers, e.g. addition or multiplication.

We give an interpretation of terms in our language following the (standard) GOI interpretations of Hoshino [73] and Mackie [100], encoding terms into linear logic proof nets. We use Girard's call-by-name embedding [62] into linear terms, where $\theta \to \theta'$ is translated to the linear type $!\theta \multimap \theta'$. The ! in the translation represents the fact that the function may use its argument an arbitrary number of times.

The term interpretation is not new — it is an adaptation of a standard technique — so we will present only what is necessary for our work, omitting some of the theoretical background on the subject. We refer the interested reader to the original work [61] for details.

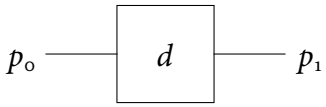
Term interpretations are built by connecting graphical components that we think of as the nodes in a network. Connected components can communicate bidirectionally using data tokens, defined by the grammar:

$$\mathit{Token} \ni e ::= \bullet \mid \mathrm{o} \mid \mathbf{S}\ e \mid \mathbf{inl}\ e \mid \mathbf{inr}\ e \mid (e, e).$$

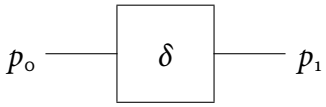
We first give a reading of these components as partial maps between data tokens, which will act as a specification for a new, lower-level abstract machine that we will give in the next section.

The standard GOI components are given in Figure 3.1: d for dereliction ($!\theta \multimap \theta$), δ for comultiplication ($!\theta \multimap !!\theta$), and c for contraction ($!\theta \multimap !\theta \otimes !\theta$). These components correspond to structural rules in linear logic. We do not give an explicit component for weakening, instead letting weakened components have their ports unconnected with the understanding that they will then act like inert sinks. Components of exponential type will expect tuples where the left component of the tuple is "routing information" that identifies the caller. The structural rules thus become bookkeeping of routing information.

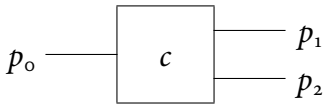
The components are bidirectional and their behaviour is given by a function mapping the values of a port at a given moment to their values at the next



$$d((\bullet, e), \bot) = (\bot, e)$$
$$d(\bot, e) = ((\bullet, e), \bot)$$



$$\delta(((e, e'), e''), \bot) = (\bot, (e, (e', e'')))$$
$$\delta(\bot, (e, (e', e''))) = (((e, e'), e''), \bot)$$



$$c((\mathbf{inl}\ e, e'), \bot, \bot) = (\bot, (e, e'), \bot)$$
$$c((\mathbf{inr}\ e, e'), \bot, \bot) = (\bot, \bot, (e, e'))$$
$$c(\bot, (e, e'), \bot) = ((\mathbf{inl}\ e, e'), \bot, \bot)$$
$$c(\bot, \bot, (e, e')) = ((\mathbf{inr}\ e, e'), \bot, \bot)$$

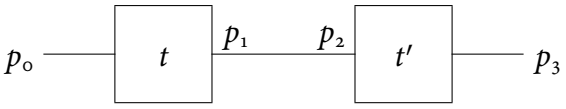
Figure 3.1: Components for structural rules

Dereliction d ($!\theta \multimap \theta$), comultiplication δ ($!\theta \multimap !!\theta$), and contraction c ($!\theta \multimap !\theta \otimes !\theta$).

moment. We denote the value on a port which sends/receives no data as $\perp$. Two well-formedness conditions of GOI nets are that at most one port is not $\perp$ (i.e. at most a single token is received at any moment) and $\overline{\perp} = (\perp, \ldots, \perp)$ is a fixed-point for any net (i.e. no spontaneous output is created). An equivalent formulation of components would be deterministic binary relations over tuples of sets of ports and data tokens.

Let π_1, π_2 be the first and second projections. Components are connected by functional composition (in both directions) on the shared port, represented graphically as:

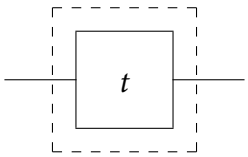
$$(t; t')(e, \perp) = t'(\pi_2 \circ t(e, \perp), \perp)$$
$$(t; t')(\perp, e) = t(\perp, \pi_1 \circ t'(\perp, e)).$$



This operation generalises to composing components with more than one port on each side in an obvious way. We will also allow feedback (e.g. *trace* operations) by letting the component be undefined for any input that results in an infinite loop.

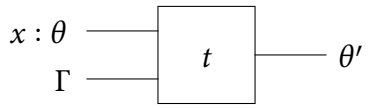
Exponentials Tokens need to carry both data and 'routing' information, which is easily performed just by using the token's pairing constructor. Basic components, however, should have no access to the routing information but act on data only. The role of the exponential functor (!) is to remove this routing information upon entering the enclosed component, pass the data to the component, then restore the routing information. Diagrammatically this is represented as a dotted box around a network, defined formally as:

$$!t((e, e'), \perp) = (\perp, (e, (\pi_2 \circ t(e', \perp))))$$
$$!t(\perp, (e, e')) = ((e, (\pi_1 \circ t(\perp, e))), \perp).$$



Types as interfaces We interpret terms into networks of components. The interface of a net is determined by the typing judgement of the term it interprets. The $\mathbb{N}$ type corresponds to one port; the function type, $\theta \to \theta'$, induces an interface which is the disjoint union of those for θ and θ'. A typing environment $\Gamma = x_1 : \theta_1, \ldots x_n : \theta_n$ induces an interface which is the disjoint union of the interfaces for each θ_i. The interface of a term with typing judgement $\Gamma, x : \theta \vdash t : \theta'$ is given by the environment on the left and its type on the right. Diagrammatically this is:

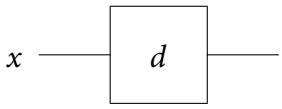
$$[\![x : \theta, \Gamma \vdash t : \theta']\!] =$$



In our graphical notation we will bundle several wires into one, for readability, and lift the standard components to work on bundles of wires (i.e. all types) by a pointwise lifting. Formally, the lifting is defined by structural recursion on the given type.

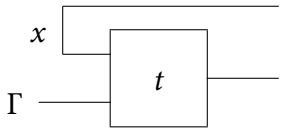
Terms as networks As the variables in the context correspond to the linear ! type, the GOI interpretation requires the use of dereliction:

$$[\![x : \theta, \Gamma \vdash x : \theta]\!] =$$



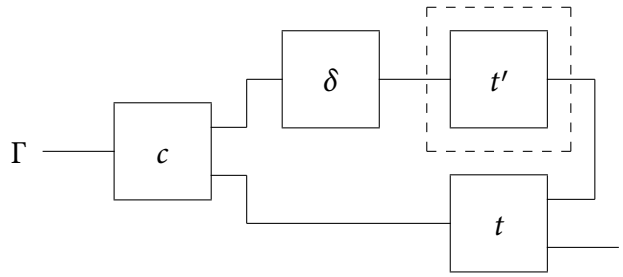
Abstraction and application When interpreting an abstraction, the variable is added to the context of the inner term, just like in the typing rule, and exposed in the interface of the final component. We only show the diagrammatic definition, since the formalisation readily follows from it:

$$[\![\Gamma \vdash \lambda x.\, t : \theta \to \theta']\!] =$$



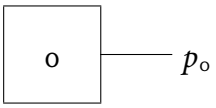
In the interpretation of application note the use of dereliction and exponentiation in the way the argument t' is connected to function t; this corresponds to the linear decomposition of call-by-name evaluation. Also note the use of contraction to explicitly share the identifiers in the context Γ.

$$[\![\Gamma \vdash t\ t' : \theta']\!] =$$



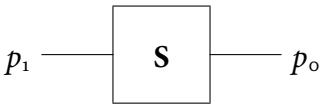
Constants The interpretation of a constant is a simple component that answers with o when requested, i.e. $\mathbf{o}(\bullet) = \mathrm{o}$. Diagrammatically,

$$[\![\Gamma \vdash \mathrm{o} : \mathbb{N}]\!] =$$



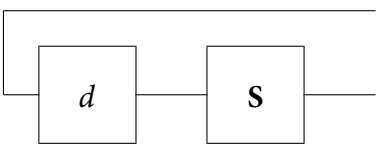
Successor A new component is needed for the interpretation of the successor function. This handles the **S** operation directly on a natural number.

$$\mathbf{S}\ (\bot, \bullet) = (\bullet, \bot)$$
$$\mathbf{S}\ (n, \bot) = (\bot, \mathbf{S}\ n)$$



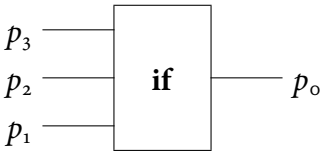
To make it possible to use this component in our interpretation it needs to be wrapped up in an abstraction and a dereliction to bring the argument to the right linear type.

$$[\![\Gamma \vdash \mathbf{S} : \mathbb{N} \to \mathbb{N}]\!] =$$



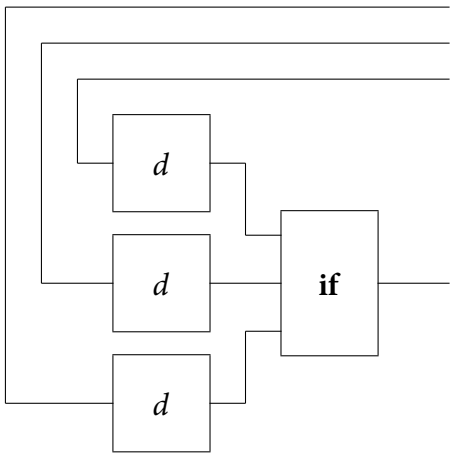
Conditionals Similarly to how addition was handled, conditionals are done by constructing a new component and then wrapping it up as a function.

$$\begin{aligned}
\mathbf{if}(\bullet, \bot, \bot, \bot) &= (\bot, \bullet, \bot, \bot) \\
\mathbf{if}(\bot, \mathrm{o}, \bot, \bot) &= (\bot, \bot, \bullet, \bot) \\
\mathbf{if}(\bot, \mathbf{S}\ n, \bot, \bot) &= (\bot, \bot, \bot, \bullet) \\
\mathbf{if}(\bot, \bot, n, \bot) &= (n, \bot, \bot, \bot) \\
\mathbf{if}(\bot, \bot, \bot, n) &= (n, \bot, \bot, \bot)
\end{aligned}$$



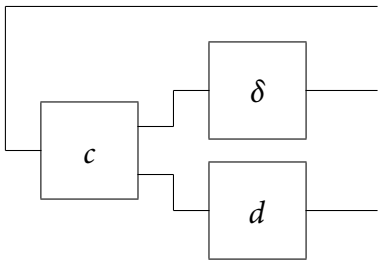
For the final interpretation of conditionals in the language, we add derelictions (since arguments are of exponential type):

$$[\![\ \Gamma \vdash \mathbf{if} \cdot \mathbf{then} \cdot \mathbf{else} \cdot : \mathbb{N} \to \mathbb{N} \to \mathbb{N} \to \mathbb{N}\ ]\!] =$$



Recursion Recursion is interpreted as a component that connects to itself, following Mackie [100].

$$[\![\Gamma \vdash \mathbf{Y} : (\theta \to \theta) \to \theta]\!] =$$



The abstract token machine interpretation given in this section is known to be sound [73, 100].

Theorem 3.1.1 (GOI soundness). Let $\vdash t : \mathbb{N}$ be a closed PCF program at ground type and $[\![t]\!]$ its GOI abstract-token machine representation. If t evaluates to n ($t \Downarrow n$) then $[\![t]\!](\bullet) = n$.

3.2 THE SIC MACHINE

To be able to describe the inner workings of the components and see how they can be compiled to executable distributed networks we construct an abstract machine, the Stack-Interaction-Control (SIC) machine, which has a small instruction set tailor-made for that purpose. The SIC machine works similarly to Mackie's [100] but with the important distinction that it also allows sending and receiving messages to and from other machines, to model networked distribution.

The machine descriptions and configurations are specified in Figure 3.2. The sets *Label* and *Port* are taken to be some countably infinite sets from which we can draw elements to use as distinct identifiers for the internal labels and external ports of the machines. We distinguish between the statics and the dynamics of SIC machines. The statics, or a machine *description*, consists of a *PortMap*, mapping external ports to internal labels, and a *LabelMap*, mapping internal labels to chunks of code. The dynamics, or a machine *configuration*, additionally has a possibly running thread, *Maybe* $(Code \times Token)$ and a *Stack* (that persists the thread). A fragment of code is a list of instructions that ends in a branch or a send instruction. Note that there is no receive instruction; the transition relation will always let an inactive machine receive messages, obviating the need for such an instruction. An initial configuration for a machine description (P, L) is given by the function $initial\ (P, L) = (nothing, [], P, L)$.

3.2.1 *SIC semantics*

The transition relation of the machines is given in Figure 3.3. The machine instructions make it possible to manipulate the data token of an active (**just**)

Label	$l \in \mathit{Label}$		
Port	$p \in \mathit{Port}$		
Instruction	$\mathit{Instr} \ni i$	$::=$ `inl` | `inr`	Tags
		| `fst` | `snd`	Projections
		| `unfst` | `unsnd`	Reverse projections
		| `swap` | `push` | `pop`	Stack operations
		| `zero` | `suc`	Natural numbers
Code	$\mathit{Code} \ni c$	$::= i;\ c$	Sequencing
		| `jump` l	Jump to label l
		| `match` $l_1 l_2$	Conditional jump
		| `if` $l_1\ l_2$	Conditional jump (nat)
		| `send` p	Send on port p
Message	Msg	$\triangleq \mathit{Port} \times \mathit{Token}$	
Stack	$S \in \mathit{Stack}$	$\triangleq \mathit{List}\ \mathit{Token}$	
Port map	$P \in \mathit{PortMap}$	$\triangleq \mathit{Port} \to \mathit{Label}$	
Label map	$L \in \mathit{LabelMap}$	$\triangleq \mathit{Label} \to \mathit{Code}$	
Descr.		$\triangleq \mathit{PortMap} \times \mathit{LabelMap}$	
Config.	$M \in \mathit{Config}$	$\triangleq \mathit{Maybe}\ (\mathit{Code} \times \mathit{Token})$	Thread
		$\times\ \mathit{Stack}$	
		$\times\ \mathit{PortMap}$	
		$\times\ \mathit{LabelMap}$	

Figure 3.2: SIC machine definition

The sets Label *and* Port *are taken to be some countably infinite sets from which we can draw elements to use as distinct identifiers for the internal labels and external ports of the machines. We distinguish between the statics and the dynamics of SIC machines. The statics, or a machine* description*, consists of a* PortMap*, mapping external ports to internal labels, and a* LabelMap*, mapping internal labels to chunks of code. The dynamics, or a machine* configuration*, additionally has a possibly running thread,* Maybe (Code × Token) *and a* Stack *(that persists the thread). A fragment of code is a list of instructions that ends in a branch or a send instruction.*

$$
\begin{array}{rcl}
(\textbf{just}\ ((\texttt{inl};\ C), e), S, P, L) & \to & (\textbf{just}\ (C, (\textbf{inl}\ e)), S, P, L) \\
(\textbf{just}\ ((\texttt{inr};\ C), e), S, P, L) & \to & (\textbf{just}\ (C, (\textbf{inr}\ e)), S, P, L) \\
(\textbf{just}\ ((\texttt{fst};\ C), (e_1, e_2)), S, P, L) & \to & (\textbf{just}\ (C, e_1), e_2 :: S, P, L) \\
(\textbf{just}\ ((\texttt{snd};\ C), (e_1, e_2)), S, P, L) & \to & (\textbf{just}\ (C, e_2), e_1 :: S, P, L) \\
(\textbf{just}\ ((\texttt{unfst};\ C), e_1), e_2 :: S, P, L) & \to & (\textbf{just}\ (C, (e_1, e_2)), S, P, L) \\
(\textbf{just}\ ((\texttt{unsnd};\ C), e_2), e_1 :: S, P, L) & \to & (\textbf{just}\ (C, (e_1, e_2)), S, P, L) \\
(\textbf{just}\ ((\texttt{swap};\ C), e), e_1 :: e_2 :: S, P, L) & \to & (\textbf{just}\ (C, e), e_2 :: e_1 :: S, P, L) \\
(\textbf{just}\ ((\texttt{push};\ C), e), S, P, L) & \to & (\textbf{just}\ (C, \bullet), e :: S, P, L) \\
(\textbf{just}\ ((\texttt{pop};\ C), e_1), e_2 :: S, P, L) & \to & (\textbf{just}\ (C, e_1), S, P, L) \\
(\textbf{just}\ ((\texttt{zero};\ C), e), S, P, L) & \to & (\textbf{just}\ (C, \mathsf{o}), S, P, L) \\
(\textbf{just}\ ((\texttt{suc};\ C), n), S, P, L) & \to & (\textbf{just}\ (C, \mathbf{S}\ n), S, P, L) \\
(\textbf{just}\ ((\texttt{jump}\ l), e), S, P, L) & \to & (\textbf{just}\ (L(l), e), S, P, L) \\
(\textbf{just}\ ((\texttt{match}\ l_1 l_2), (\textbf{inl}\ e)), S, P, L) & \to & (\textbf{just}\ (L(l_1), e), S, P, L) \\
(\textbf{just}\ ((\texttt{match}\ l_1 l_2), (\textbf{inr}\ e)), S, P, L) & \to & (\textbf{just}\ (L(l_2), e), S, P, L) \\
(\textbf{just}\ ((\texttt{if}\ l_1\ l_2), \mathsf{o}), S, P, L) & \to & (\textbf{just}\ (L(l_2), \bullet), S, P, L) \\
(\textbf{just}\ ((\texttt{if}\ l_1\ l_2), \mathbf{S}\ n), S, P, L) & \to & (\textbf{just}\ (L(l_1), \bullet), S, P, L) \\
(\textbf{just}\ ((\texttt{send}\ p), e), S, P, L) & \xrightarrow{\texttt{send}\ (p,e)} & (\textbf{nothing}, S, P, L) \\
(\textbf{nothing}, S, P, L) & \xrightarrow{\texttt{receive}\ (p,e),\ p \in dom(P)} & (\textbf{just}\ (L(P(p)), e), S, P, L)
\end{array}
$$

Figure 3.3: SIC transition relation

*The machine instructions make it possible to manipulate the data token of an active (**just**) machine, using the stack for state and intermediate results. The last two rules handle sending and receiving messages.*

machine, using the stack for state and intermediate results. The last two rules handle sending and receiving messages.

Example 3.2.1. The following code fragment can be used to construct a tuple $(1, 2)$ and send it on the port p:

$$C = \texttt{zero; suc; push; zero; suc; suc; unsnd; send } p$$

Executing the above code, we get the following trace:

$$\begin{aligned}
(\textbf{just}\ (C, e), S, P, L) &\to^3 \\
(\textbf{just}\ ((\texttt{zero; suc; suc; unsnd; send } p), \bullet), 1 :: S, P, L) &\to^3 \\
(\textbf{just}\ ((\texttt{unsnd; send } p), 2), 1 :: S, P, L) &\to \\
(\textbf{just}\ (\texttt{send } p, (1, 2)), S, P, L) &\xrightarrow{\texttt{send } (p,(1,2))} \\
(\textbf{nothing}, S, P, L)&
\end{aligned}$$

We now define a machine network by instantiating our general network formalism (Section 2.5) with the SIC transition relation. We note that its type does not quite fit since it is missing the initial node name, but this is easily remedied by constructing a new relation based on SIC that just ignores this additional component. An initial network configuration for a family of machine descriptions, $\mathit{initial}(N)$, is the pointwise lifting of the function on machine descriptions of the same name.

We will use the notation $M_1, \ldots, M_k$ for an indexed family of k machine configurations (or descriptions when dealing with statics).

To connect the port p_0 of machine M to p_1 of M' we rename them in the machine descriptions to the same port name p. A port must be connected to at most one other port; in this case the resulting network is *deterministic*, as each message can be received by at most one other machine. A port of a machine which is not connected to a port of another machine is said to be a port *of the network*. By $\mathit{inputs}(M)$ ($\mathit{outputs}(M)$) we mean the inputs (outputs) of a machine, whereas by $\mathit{inputs}(N)$ ($\mathit{outputs}(N)$) we mean the inputs (outputs) of a network. Similarly, by $\pi(M)$ ($\pi(N)$) we mean the ports of a machine (network).

Let *active* be a function on families of machines that returns the machines in its argument that are in the **just** state.

Proposition 3.2.2. For any families of machine configurations N and N' and lists of messages $\mathcal{M}$ and $\mathcal{M}'$, if $(N, \mathcal{M}) \to (N', \mathcal{M}')$, then $|\mathcal{M}'| + |\mathit{active}(N')| = |\mathcal{M}| + |\mathit{active}(N)|$.

The proofs of this proposition and other theorems from this chapter are given in Appendix A so as to not break the flow of reading. In particular, this proposition means that if we start out with one message and no active machines, there can be at most one active machine at any point in the network's execution — the execution is *single-token*.

3.2.2 *Components as SIC code*

In each instance of the components we initially take all ports p_x and labels l_x to be distinct. To emphasise the input/output role of a port we sometimes write them as p_x^i when serving as input and p_x^o when serving as output. The machine descriptions for the different components are described by giving their port mappings P and label mappings L as a tuple.

The following three machines are stateless. They use the stack internally for intermediate results, but ultimately return the stack to the initial empty state.

Dereliction, d, removes the first component of the token tuple when going from left to right, and adds it when going in the other direction:

$$\text{dereliction} = \left\langle \begin{array}{l} p_{\text{o}}^i \mapsto l_{\text{o}} \\ p_1^i \mapsto l_1 \end{array} \quad , \quad \begin{array}{l} l_{\text{o}} \mapsto \texttt{snd; pop; send } p_1^o \\ l_1 \mapsto \texttt{push; unfst; send } p_{\text{o}}^o \end{array} \right\rangle$$

Comultiplication, δ, reassociates the data token to go from $((e, e'), e'')$ to $(e, (e', e''))$ and back:

$$\text{comult} = \left\langle \begin{array}{l} p_{\text{o}}^i \mapsto l_{\text{o}} \\ p_1^i \mapsto l_1 \end{array} \quad , \quad \begin{array}{l} l_{\text{o}} \mapsto \texttt{fst; snd; swap; unfst; unsnd; send } p_1^o \\ l_1 \mapsto \texttt{snd; fst; swap; unsnd; unfst; send } p_{\text{o}}^o \end{array} \right\rangle$$

Contraction, c, uses matching on the first component of the token to choose the port to send on when going from left to right.

$$\text{contraction} = \left\langle \begin{array}{l} p_{\text{o}}^i \mapsto l_{\text{o}} \\ p_1^i \mapsto l_1 \\ p_2^i \mapsto l_2 \end{array} \quad , \quad \begin{array}{l} l_{\text{o}} \mapsto \texttt{fst; match } l_3 l_4 \\ l_1 \mapsto \texttt{fst; inl; unfst; send } p_{\text{o}}^o \\ l_2 \mapsto \texttt{fst; inr; unfst; send } p_{\text{o}}^o \\ l_3 \mapsto \texttt{unfst; send } p_1^o \\ l_4 \mapsto \texttt{unfst; send } p_2^o \end{array} \right\rangle$$

Dotted boxes with k inputs are defined as follows: Let $P_{\text{in}}^i = \{p_{\text{o}}^i, \ldots, p_{k-1}^i\}$, $P_{\text{out}}^i = \{p_k^i, \ldots, p_{2k-1}^i\}$, Let $P_{\text{in}}^o = \{p_{\text{o}}^o, \ldots, p_{k-1}^o\}$, $P_{\text{out}}^o = \{p_k^o, \ldots, p_{2k-1}^o\}$, and $L_{\text{out}} = \{l_k, \ldots, l_{2k-1}\}$. The box for these sets of ports and labels is then the following (note that boxes use the stack for storing their state):

$$\text{box} = \left\langle p_i^i \mapsto l_i \mid p_i^i \in P_{\text{in}}^i \cup P_{\text{out}}^i \quad , \quad \begin{array}{l} l_i \mapsto \texttt{snd; send } p_{i+k}^o \mid l_i \in L_{\text{in}} \\ l_i \mapsto \texttt{unsnd; send } p_{i-k}^o \mid l_i \in L_{\text{out}} \end{array} \right\rangle$$

For the constant o component, the abstract machine is defined as follows:

$$\text{constant} = \left\langle \; p^i_{\text{o}} \mapsto l_{\text{o}} \quad , \quad l_{\text{o}} \mapsto \texttt{zero; send } p^o_{\text{o}} \; \right\rangle$$

The successor machine first asks for its argument, then runs the successor instruction on that.

$$\text{suc} = \left\langle \begin{array}{ll} p^i_{\text{o}} \mapsto l_{\text{o}} & \\ p^i_1 \mapsto l_1 & , \quad l_{\text{o}} \mapsto \texttt{send } p^o_1 \\ & \quad l_1 \mapsto \texttt{suc; send } p^o_{\text{o}} \end{array} \right\rangle$$

The conditional uses the matching instruction `if` to choose the branch depending on the natural number of the token.

$$\text{if} = \left\langle \begin{array}{ll} p^i_{\text{o}} \mapsto l_{\text{o}} & \\ p^i_1 \mapsto l_1 & \quad l_{\text{o}} \mapsto \texttt{send } p^o_1 \\ p^i_2 \mapsto l_2 & , \quad l_1 \mapsto \texttt{if } l_{\text{true}}\ l_{\text{false}} \\ p^i_3 \mapsto l_2 & \quad l_{\text{true}} \mapsto \texttt{send } p^o_2 \\ & \quad l_{\text{false}} \mapsto \texttt{send } p^o_3 \\ & \quad l_2 \mapsto \texttt{send } p^o_{\text{o}} \end{array} \right\rangle$$

Theorem 3.2.3 (Soundness). Let $\vdash t : \mathbb{N}$ be a closed PCF program of ground type, $[\![t]\!]$ its GOI abstract-token machine representation and N its SIC-net implementation. If t evaluates to n ($t \Downarrow n$) then $[\![t]\!](\bullet) = n$ and $(N, \{(p^i, \bullet)\}) \to^* (N, \{(p^o, n)\})$.

Proof outline The soundness of the SIC implementation of each of the components above, i.e. that they give the same result as the GOI interpretation wherever it is defined, can be verified by a straightforward symbolic execution for each clause of the GOI component definitions. We can similarly show that SIC network composition is sound with respect to GOI component composition defined above. We can then prove soundness by induction on the term since the compilation is compositional.

3.3 COMBINING MACHINES

When writing distributed applications, the location at which a computation is performed is vital. Traditional approaches usually make this clear, for instance by their usage of explicit message passing. Using our current term interpretation, thinking of each abstract machine as running on a different node in a network, we get distributed programs in which the communication is handled automatically, but we will have one abstract machine for each (very small) component — the interpretation produces extremely fine-grained networks where each node does very little work before passing the token along to another node.

It is expected that the communication is one of the most performance critical parts in a distributed network, which is why it would be better if bigger chunks of computations happened on the same node before the token was passed along.

To make this possible, we devise a way to combine the descriptions of two abstract machines in a deterministic network to get a larger abstract machine with the same behaviour as the two original machines. Informally, the way to combine two machines is to remove ports that are used internally between the two machines (if any) and replace sends on those ports with jumps. The algorithm for combining components $M_1 = (P_1, L_1)$ and $M_2 = (P_2, L_2)$ is described more formally below.

We use Δ for the symmetric difference of two sets. If $f : A \to B$ is a function we write as $f \restriction A'$ the restriction of f to the domain $A' \subseteq A$ and we extend it in the obvious way to relations. We use the standard notation $C[s/s']$ to denote the replacing of all occurrences of a string s by s' in C. We write $C[s(x)/s'(x) \mid x \in A]$ to denote the substitution of all strings of shape $s(x)$ by strings of shape $s'(x)$ with x in a list A, defined inductively as

$$\begin{aligned} C[s(x)/s'(x) \mid x \in [\,]] &= C \\ C[s(x)/s'(x) \mid x \in a :: A] &= \big(C[s(a)/s'(a)]\big)[s(x)/s'(x) \mid x \in A] \end{aligned}$$

The combination of two machines is defined by keeping the ports which are not shared and by replacing in the code the send operations to shared ports by jumps to labels given by the port mappings.

$$\begin{aligned} \mathit{combine}(M_1, M_2) = \langle &(P_1 \cup P_2) \restriction (\pi(M_1) \,\Delta\, \pi(M_2)), \\ &(L_1 \cup L_2)[\texttt{send}\ p/\texttt{jump}\ P(p) \mid p \in \pi(M_1) \cap \pi(M_2)]\rangle. \end{aligned}$$

There are two abuses of notation above. First, the union $P_1 \cup P_2$ above is on functions taken as sets of pairs and it may not result in a proper function. However, the restriction to $\pi(M_1) \,\Delta\, \pi(M_2)$ always produces a proper function. Second, $\pi(M_1) \cap \pi(M_2)$ is a set and not a list. However, the result of this substitution is independent of the order in which the elements of this set are taken from any of its possible list representations.

We also lift the combination operations to families of machines, in the obvious way:

$$\mathit{combine}(M_1, \ldots, M_k) = \mathit{combine}(M_1, \mathit{combine}(\ldots, M_k))$$

A family of machines is said to be *combinable* if combining any of its components does not change the overall network behaviour:

Definition 3.3.1. A deterministic family of machines $N = M_1, \ldots, M_k$ is *combinable* if whenever

$$(\mathit{initial}(N), [(p, e)]) \to^* (\mathit{initial}(N), [(p', e')])$$

for some p in $\mathit{inputs}(N)$, p' in $\mathit{outputs}(N)$, then for any

$$N_{\text{combined}} = \mathit{combine}(N_1), \ldots, \mathit{combine}(N_{k'})$$

obtained from a partition $N = N_1, \ldots, N_{k'}$ we have that

$$(\mathit{initial}(N_{\text{combined}}), [(p, e)]) \to^* (\mathit{initial}(N_{\text{combined}}), [(p', e')])$$

Note that this only says something about runs that do not leave anything in the stacks (as evident by the transitions' ending up in the initial state), which is enough for our purposes.

The set of combinable machines is hard to define exactly, so we would just like to find a sound characterisation of such machines which covers all the basic components we use.

Definition 3.3.2. A machine description $M = (P, L)$ is *stack-neutral* (or stateless) if for all stacks S and S', p in $\mathit{inputs}(M)$, p' in $\mathit{outputs}(M)$, if

$$((\textbf{nothing}, S, P, L), [(p, e)]) \to^* ((\textbf{nothing}, S', P, L), [(p', e')])$$

then $S = S'$.

Definition 3.3.3. A machine network N of k machines described by port mappings P_i and label mappings L_i is *stack-neutral*, if for all stacks S_i and S'_i, p in $\mathit{inputs}(N)$, p' in $\mathit{outputs}(N)$, if

$$\begin{aligned}(\{(\textbf{nothing}, S_i, P_i, L_i) \mid i \in \{1, \ldots, k\}\}, [(p, e)]) \to^* \\ (\{(\textbf{nothing}, S'_i, P_i, L_i) \mid i \in \{1, \ldots, k\}\}, [(p', e')])\end{aligned}$$

then all $S_i = S'_i$.

Note that this definition is more general than having a list of stack-neutral machines, as a stack-neutral network's machines may use the stack for state after they have been exited as long as the stack is cleared before an output on a network port.

Proposition 3.3.4. If two machine networks N_1 and N_2 (of initially passive machines) are stack-neutral, combinable and the composition N_1, N_2 is deterministic, then N_1, N_2 is stack-neutral and combinable.

Recall the update notation from Section 2.5:

$$(M_1, \ldots, M_k)[i \mapsto M] \triangleq M_1, \ldots, M_{i-1}, M, M_{i+1}, M_k$$

Lemma 3.3.5. Let

$$N = (T_1, S_1, P_1, L_1), \ldots, (T_k, S_k, P_k, L_k)$$

$$N' = (T'_1, S'_1, P'_1, L'_1), \ldots, (T'_k, S'_k, P'_k, L'_k).$$

If

$$(N, \mathcal{M}) \to^* (N, \mathcal{M}')$$

then for any S

$$(N[i \mapsto (T_i, S_i :: S, P_i, L_i)], \mathcal{M}) \to^* (N'[i \mapsto (T'_i, S'_i :: S, P'_i, L'_i)], \mathcal{M}').$$

Proof. This is a simple fact about SIC machines — there are no instructions that branch depending on how many elements there are on the stack. □

For any SIC net N let $box(N)$ be N with an additional box machine M with input ports $outputs(N)$ and output ports $inputs(N)$, defined as in Section 3.2.2.

Proposition 3.3.6. If a machine network N is stack-neutral and combinable then $box(N)$ is stack-neutral and combinable.

From the above two results it follows by induction on the structure of the generated nets that

Theorem 3.3.7. If $\Gamma \vdash t : \theta$ is a PCF term, $[\![t]\!]$ its GOI abstract-token machine representation and N the implementation of $[\![t]\!]$ as a SIC net then N is combinable.

With the ability to combine components, we can now exploit the $t@A$ annotations in the language. They make it possible to specify where a piece of code should be located (A is a node identifier). When this construct is encountered in compilation, the components generated in compiling t are tagged with A (without overwriting tags stemming from inside t).

Next, the components with the same tag are combined using the algorithm above and their combined machine placed on the node identified by the tag. This allows the programmer to arbitrarily choose where the compiled representation of a part of a term is placed. Soundness (Theorem 3.2.3) along with the freedom to combine nets (Theorem 3.3.7) ensures that the resulting network is a correct implementation of any (terminating) PCF program.

3.4 COMPILING PCF

We developed an experimental compiler[1] that compiles to C, using MPI for communication, using SIC abstract machines as an intermediate formalism. Each machine description in a network is mapped to a C source file, using a function for each machine instruction and global variables for the data token and the stack. An example of a predefined instruction is that for the `swap` instruction:

```
inline void swap() {
  Data d1 = pop_stack();
  Data d2 = pop_stack();
  push_stack(d1);
  push_stack(d2);
}
```

An abstract machine's label *l* corresponds to a C function `void l()` whose definition is a list of calls to the predefined machine instruction functions. In this representation, jumps are function calls. All predefined functions are small and not used recursively so they can be efficiently inlined.

Each process in MPI has a unique identifier called its *rank*, and messages can also be assigned a *tag*. A port in a SIC machine is uniquely determined by its tag, but also has to be assigned a rank so that the message can be sent to the correct node. This is resolved at compile time. The main loop for a machine listening on ports corresponding to tags 0 and 1 looks like this:

```
while(1) {
  int port = receive();
  switch (port) {
    case 0: l0(); break;
    case 1: l1(); break;
    default: break;
  }
}
```

Here `l0` and `l1` are functions corresponding to the labels associated with the ports. The predefined function `receive` calls `MPI_Recv`, which is an MPI function that blocks until a message is received. A process in this state thus corresponds to a machine in **nothing** state. Upon receiving a message, the `receive` function deserialises the message and assigns it to the global data token

[1] Online appendix: `goi` directory.

variable before returning the message's tag. The predefined function for the `send` instruction now has to take two parameters: the destination node's rank and the port's tag:

```
inline void send(int node, int port);
```

The function takes care of serialising the data token and sending it to the correct node using `MPI_Send`.

When all machines have been compiled to C, these can in turn be compiled to executables and run on different machines in a network where they use message passing for communication.

3.5 RELATED WORK

Here we outline the work that is immediately related to the GOI interpretation; work related in more general ways, such as using interaction semantics in general or aiming to solve a problem similar to ours can be found in Section 2.1, Section 4.4, and Section 7.9.

There are two contrasting views of the execution taking place in GOI networks, although independent of the original formalism [61]. The first one is to reduce the networks themselves, i.e. to perform graph reduction in interaction nets [88]. This line of work has for example provided insight into optimal lambda reduction [63] where there are parallels to the work of Lamping [90] on the same topic. The second one, that we also use, is to push a token through a static network, which was pioneered by Mackie [100] who provides essential inspiration for our work. While the graph reducing line of work is a fruitful area of research it seems to be at odds against our method of compilation which relies on the modification of a relatively static network at compile-time to control granularity. Mackie [100] also shows that GOI is useful not only for the study of semantics, but also as a methodology for compiling programs. Another similar line of work and inspiration for our work is the Geometry of Synthesis, that uses a GOI-like compilation of a call-by-name programming language with recursion to target reconfigurable digital circuits [53, 58, 59, 60].

Hoshino [73], that we also borrow our graphical notation from, presents another use of GOI for the semantics of programming languages, namely a linear functional programming language.

Danos, Herbelin, and Regnier [31] give an abstract machine for GOI, called the Interaction Abstract Machine (IAM). Insight into the call-return symmetry of the legal paths in GOI shows that this machine performs redundant work, which they optimise by introducing the Jumping Abstract Machine (JAM), an environment machine, that essentially forgoes work in the return direction

by jumping [32]. The JAM has interesting connections with the Krivine machine [86] — they are isomorphic under certain embeddings of the linear calculus into the lambda calculus — but the usage of environments seems to prohibit using a distributed interpretation of nets (see Chapter 6 for how different the formulation becomes for an environment machine). An additional optimisation was found by Fernández and Mackie [39]: whereas the JAM shows that it is possible to avoid reverse computation, this work shows that it is possible to avoid recomputing repeated subpaths, yielding a call-by-value interpretation.

Our main innovation compared to this previous work is the application of GOI to distributed systems and the abstract machine with granularity control via combinations.

Berry and Boudol [15] introduce the Chemical Abstract Machine (CHAM) which provides an inspiration for the communication part of the SIC abstract machine (Section 2.5). The paper does not provide any insight into the implementation of a system based on a CHAM, although that is outside of the scope of their work which deals with *reasoning* about systems.

Banâtre, Coutant, and Métayer [13] does provide some insight on the implementation of such a machine but relies either on broadcasting the multiset on which it is performing its transformations to all nodes or on using shared memory, which suggests that it is not suitable for distributed systems where such an operation can be costly. This is why we use messages tagged with ports in our machines, meaning that the messaging is point-to-point and can be implemented efficiently using message passing.

Schöpp [123] shows that there is a relation between interaction-based compilation methods and more conventional methods such as translations to continuation-passing style and defunctionalisation.

The *Hume box calculus* [66] has a semantics-preserving horizontal combination operation that is conceptually similar to our combination operation, but in the context of boxes in the Hume language [67], a language targeting resource-bound systems.

3.6 CONCLUSION

We have shown a programming language and compilation model for higher-order RPCs, that provides freedom in choosing the location at which a computation takes place with implicitly handled communication. This was achieved by basing the model on the Geometry of Interaction and constructing a way to produce nodes that are more coarse-grained than the standard elementary nodes, and showing that this is still correct.

We defer benchmarking the compiler to Section 7.8, so that we can also compare our implementation to the abstract machines that will be defined in subsequent chapters. We will however make some remarks about the performance of the compiler here as a motivation for the next chapter. For single-node computation of programs doing simple arithmetic, the compiler is between 30 and 200 times slower than a naive implementation of the Krivine machine [86]. The source of this inefficiency is in part due to the embedding into linear logic, which means that programs do work to share (using contraction) the variables in the environment. For multi-node programs, the programs compiled using GOI use messages that sometimes grow big. This is because the tokens contain what amounts to (part of) the computational context of the terms.

The GOI model that we use also seems to be difficult to parallelise, because of the boxes that use global state. Although we stated in the introduction, Chapter 1, that the focus of this work is not parallelism or concurrency, it would still be preferable if we at least did not prohibit such features in the compilation model.

The next chapter aims to overcome some of these problems by using a conceptually similar compilation model based instead on game semantics.

Chapter 4

Game semantics

Synopsis We define new abstract machines for game semantics which correspond to networks of conventional computers, and can be used as an intermediate representation for compilation targeting distributed systems. This is achieved in two steps. First we introduce the Heap and Register Abstract Machine (HRAM), an abstraction of a conventional multi-threaded computer, which can be structured into HRAM nets, an abstract point-to-point network model. Game Abstract Machines (GAMs), are HRAMs with additional structure at the interface level, but no special operational capabilities. We show that GAMs cannot be naively composed, but composition must be mediated using appropriate HRAM combinators. We start from a formulation of game semantics in the nominal model [48], which has two benefits. First, pointer manipulation requires no encoding or decoding, as in integer-based representations, but exploits the HRAM ability to create locally fresh *names*. Second, token size is constant as only names are passed around; the computational history of a token is stored by the HRAM rather than passing it around (cf. IAM [31] and the GOI compiler (Chapter 3)). HRAMs are also flexible enough to allow the representation of game models for languages with state (*non-innocent* games) or concurrency (*non-alternating* games). We illustrate the potential of this technique by implementing a distributing compiler for Idealised Concurrent Algol (ICA), a higher-order programming language with shared state concurrency [55], thus significantly extending our previous distributing PCF compiler based on GOI (Chapter 3). We show that compilation is sound and memory safe, i.e. no (distributed or local) garbage collection is necessary.

4.1 SIMPLE NETS

In this section we introduce a class of basic abstract machines for manipulating heap structures, which also have primitives for communication and control. They represent a natural intermediate stage for compilation to machine language, and will be used as such in Section 4.3. The machines can naturally be organised into communication networks which give an abstract representation of distributed systems. We find it formally convenient to work in a nominal model in order to avoid the difficulties caused by concrete encoding of game structures, especially *justification pointers*, as integers. We assume from the

reader a certain familiarity with basic nominal concepts. The interested reader is referred to the literature ([49] is a starting point).

4.1.1 *Heap and Register Abstract Machines (HRAMs)*

We fix a set of *port names* ($\mathbb{A}$) and a set of *pointer names* ($\mathbb{P}$) as disjoint sets of atoms. Let $L \triangleq \{\mathbf{O}, \mathbf{P}\}$ be the set of polarities of a port. To maintain an analogy with game semantics from the beginning, port names correspond to game semantic *moves* and input/output polarities correspond to opponent/proponent. A *port structure* is a tuple $(l, a) \in \mathit{Port} \triangleq L \times \mathbb{A}$. An *interface* $A \in \mathcal{P}_{\mathit{fin}}(\mathit{Port})$ is a set of port structures such that all port names are unique, i.e. $\forall p = (l, a), p' = (l', a') \in A$, if $a = a'$ then $p = p'$. Let the support of an interface be $\mathit{sup}(A) \triangleq \{a \mid (l, a) \in A\}$, its set of port names.

The *tensor* of two interfaces is defined as

$$A \otimes B \triangleq A \cup B, \text{ where } \mathit{sup}(A) \cap \mathit{sup}(B) = \varnothing.$$

The duals of interfaces, port structures, and polarities are defined as

$$\begin{aligned} A^* &\triangleq \{p^* \mid p \in A\} \\ (l, a)^* &\triangleq (l^*, a) \\ \mathbf{O}^* &\triangleq \mathbf{P} \\ \mathbf{P}^* &\triangleq \mathbf{O}. \end{aligned}$$

An arrow interface is defined in terms of tensor and dual,

$$A \Rightarrow B \triangleq A^* \otimes B.$$

We introduce notation for opponent ports of an interface $A^{(\mathbf{O})} \triangleq \{(\mathbf{O}, a) \in A\}$. The player ports of an interface $A^{(\mathbf{P})}$ is defined analogously. The set of all interfaces is denoted by $\mathcal{I}$. We say that two interfaces *have the same shape* if they are equivariant, i.e. there is a permutation $\pi : \mathbb{A} \to \mathbb{A}$ such that

$$\{\pi \cdot p \mid p \in A_1\} = A_2,$$

and we write $\pi \vdash A_1 =_{\mathbb{A}} A_2$, where $\pi \cdot (l, a) \triangleq (l, \pi(a))$ is the permutation action of π. We may write only $A_1 =_{\mathbb{A}} A_2$ if π is obvious or unimportant.

Let the set of data $\mathcal{D}$ be $\varnothing \in \mathbb{1}$, pointer names $a \in \mathbb{P}$ or integers $n \in \mathbb{Z}$. Let the set of instructions *Instr* be as below, where $i, j, k \in \mathbb{N} + \mathbb{1}$ (which permits ignoring results and allocating "null" data).

- $i \leftarrow$ `new` j, k allocates a new pointer in the heap and populates it with the values stored in registers j and k, storing the pointer in register i.
- $i, j \leftarrow$ `get` k reads the tuple pointed at by the name in the register k and stores it in registers i and j.
- `update` i, j writes the value stored in register j to the second component of the value pointed to by the name in register i.
- `free` i releases the memory pointed to by the name in the register i and resets the register.
- `swap` i, j swaps the values of registers i and j.
- $i \leftarrow$ `set` j sets register i to value j.

Let code fragments $\mathcal{C}$ be $\mathcal{C} ::= \mathit{Instr}; \mathcal{C} \mid \texttt{ifzero}\ \mathbb{N}\ \mathcal{C}\ \mathcal{C} \mid \texttt{spark}\ a \mid \texttt{end}$. The port names occurring in the code fragment are $\mathit{sup} \in \mathcal{C} \to \mathcal{P}_{\mathit{fin}}(\mathbb{A})$, defined in the obvious way (only the `spark` a instruction can contribute names). An `ifzero` i instruction will branch according to the value stored in register i. A `spark` a will either jump to a or send a message to a, depending on whether a is a local port or not.

An *engine* is an interface together with a port map, $E = (A, P) \in \mathcal{I} \times (\mathit{sup}(A^{(\mathbf{O})}) \to \mathcal{C})$ such that for each code fragment $c \in \mathit{cod}\ P$ and each port name $a \in \mathit{sup}(c)$, $(\mathbf{P}, a) \in A$, meaning that ports that are "sparked" must be output ports of the interface A. The set of all engines is $\mathcal{E}$.

Engines have threads and shared heap. All threads have a fixed number of registers r, which is a global constant. For the language ICA we will need four registers, but languages with more kinds of pointers in the game model, e.g. control pointers [89], may need and use more registers.

A *thread* is a tuple $t = (c, \overline{d}) \in T = \mathcal{C} \times \mathcal{D}^r$: a code fragment and an r-tuple of data register values.

An *engine configuration* is a tuple $k = (\overline{t}, h) \in \mathcal{K} = \mathcal{P}_{\mathit{fin}}(T) \times (\mathbb{P} \rightharpoonup \mathbb{P} \times \mathcal{D})$: a set of threads and a heap that maps pointer names to pairs of pointer names and data items.

A pair consisting of an engine configuration and an engine will be written using the notation $k : E \in \mathcal{K} \times \mathcal{E}$. Define the function $\mathit{initial} \in \mathcal{E} \to \mathcal{K} \times \mathcal{E}$ as $\mathit{initial}(E) \triangleq (\varnothing, \varnothing) : E$ for an engine E. This function pairs the engine up with an engine configuration consisting of no threads and an empty heap.

HRAMs communicate using *messages*, each consisting of a port name and a vector of data items of size r_m: $m = (x, \overline{d}) \in \mathcal{M} = \mathbb{A} \times \mathcal{D}^{r_m}$. The constant r_m specifies the size of the messages in the network, and has to fulfil $r_m \leqslant r$. For

a set $X \subseteq \mathbb{A}$, define $\mathcal{M}_X = X \times \mathcal{D}^{r_m}$, the subset of $\mathcal{M}$ whose port names are limited to those of X.

We specify the operational semantics of an engine $E = (A, P)$ as a transition relation $- \xrightarrow[E,\chi]{} - \subseteq \mathcal{K} \times (\{\bullet\} \cup (L \times \mathcal{M})) \times \mathcal{K}$. Like the *Tagged* representation in Section 2.5, the relation is either labelled with $\bullet$ — a silent transition — or a polarised message — an observable transition. The messages will be constructed simply from the first r_m registers of a thread, meaning that on certain actions part of the register contents become observable in the transition relation.

To aid readability, we use the following shorthands:

- $n \xrightarrow[E,\chi]{} n'$ means $n \xrightarrow[E,\chi]{\bullet} n'$ (silent transitions).
- $n \xrightarrow[E,\chi]{(a,\overline{d})} n'$ means $n \xrightarrow[E,\chi]{(\mathbf{P},(a,\overline{d}))} n'$ (output transitions).
- $n \xrightarrow[E,\chi]{(a,\overline{d})^\bullet} n'$ means $n \xrightarrow[E,\chi]{(\mathbf{O},(a,\overline{d}))} n'$ (input transitions).

We use the notation $\overline{d}$ for n-tuples of registers and then d_i for the (zero-based) ith component of $\overline{d}$, and $d_\varnothing \triangleq \varnothing$. For updating a register, we use $\overline{d}[i := d] \triangleq (d_0, \cdots, d_{i-1}, d, d_{i+1}, \cdots, d_{n-1})$ and $\overline{d}[\varnothing := d] \triangleq \overline{d}$.

To construct messages from the register contents of a thread, we use the functions $msg \in \mathcal{D}^r \to \mathcal{D}^{r_m}$, which takes the first r_m components of its input, and $regs \in \mathcal{D}^{r_m} \to \mathcal{D}^r$, which pads its input with $\varnothing$ at the end (i.e. $regs(\overline{d}) \triangleq (d_0, \ldots, d_{r_m-1}, \varnothing, \ldots)$).

The network connectivity is specified by the function χ, which will be described in more detail in the next subsection. For a port name a, $\chi(a)$ can be read as "the port that a is connected to". The full operational rules for HRAMs are given in Figure 4.1. The interesting rule is that for `spark` because it depends on whether the port where the next computation is "sparked" is local or not. If the port is local then `spark` makes a jump, and if the port is non-local then it produces an output token and the current thread of execution is terminated, similar to the IAM. Compared to the SIC machine, which had separate instructions for jumping and sending, this means that we do not have to modify the code of the machines when combining them.

4.1.2 *HRAM nets*

A well-formed *HRAM net* $S \in \mathcal{S}$ is a set of engines, a function over port names specifying what ports are connected, and an external interface, $S = (\overline{E}, \chi, A)$, where $E \in \mathcal{E}$, $A \in \mathcal{I}$, and χ is a bijection between the net's output and input port

$$
\begin{array}{rcl}
((i \leftarrow \texttt{new}\ j,k;\ C,\overline{d}) \cup \overline{t}, h) & \xrightarrow[E,\chi]{} & ((C,\overline{d}[i := p]) \cup \overline{t}, h \cup \{p \mapsto (d_j,d_k)\}) \text{ if } p \notin sup(h) \\
((i,j \leftarrow \texttt{get}\ k;\ C,\overline{d}) \cup \overline{t}, h \cup \{d_k \mapsto (d,d')\}) & \xrightarrow[E,\chi]{} & ((C,\overline{d}[i := d][j := d']) \cup \overline{t}, h \cup \{d_k \mapsto (d,d')\}) \\
((\texttt{update}\ i,j;\ C,\overline{d}) \cup \overline{t}, h \cup \{d_i \mapsto (d,d')\}) & \xrightarrow[E,\chi]{} & ((C,\overline{d}[i := d][j := d']) \cup \overline{t}, h \cup \{d_i \mapsto (d,d_j)\}) \\
((\texttt{free}\ i;\ C,\overline{d}) \cup \overline{t}, h \cup \{d_i \mapsto (d,d')\}) & \xrightarrow[E,\chi]{} & ((C,\overline{d}[i := \varnothing]) \cup \overline{t}, h) \\
((\texttt{swap}\ i,j;\ C,\overline{d}) \cup \overline{t}, h) & \xrightarrow[E,\chi]{} & ((C,\overline{d}[i := d_j][j := d_i]) \cup \overline{t}, h) \\
((i \leftarrow \texttt{set}\ j;\ C,\overline{d}) \cup \overline{t}, h) & \xrightarrow[E,\chi]{} & ((C,\overline{d}[i := j]) \cup \overline{t}, h) \\
((\texttt{ifzero}\ i\ c_1\ c_2;\ C,\overline{d}[i := \mathsf{o}]) \cup \overline{t}, h) & \xrightarrow[E,\chi]{} & ((c_1,\overline{d}[i := \varnothing]) \cup \overline{t}, h) \\
((\texttt{ifzero}\ i\ c_1\ c_2;\ C,\overline{d}[i := n+1]) \cup \overline{t}, h) & \xrightarrow[E,\chi]{} & ((c_2,\overline{d}[i := \varnothing]) \cup \overline{t}, h) \\
((\texttt{spark}\ a,\overline{d}) \cup \overline{t}, h) & \xrightarrow[E,\chi]{(\chi(a),msg(\overline{d}))} & (\overline{t}, h) \text{ if } (\mathbf{O},\chi(a)) \notin A \\
((\texttt{spark}\ a,\overline{d}) \cup \overline{t}, h) & \xrightarrow[E,\chi]{} & ((P(\chi(a)),regs(msg(\overline{d}))) \cup \overline{t}, h) \text{ if } (\mathbf{O},\chi(a)) \in A \\
(\overline{t}, h) & \xrightarrow[E,\chi]{(a,\overline{d})^\bullet} & ((P(a),regs(\overline{d})) \cup \overline{t}, h) \text{ if } (\mathbf{O},a) \in A \\
((\texttt{end},\overline{d}) \cup \overline{t}, h) & \xrightarrow[E,\chi]{} & (\overline{t}, h)
\end{array}
$$

Figure 4.1: Operational semantics of HRAMs

The `spark` *instruction depends on whether the port where the next computation is "sparked" is local or not. If the port is local then* `spark` *makes a jump, and if the port is non-local then it produces an output token and the current thread of execution is terminated.*

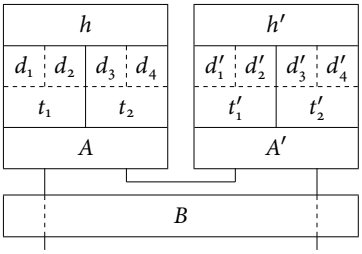


Figure 4.2: Example HRAM net

An HRAM net with two HRAMs (interfaces A, A', two ports each), each with two running threads (t_i, t'_i) with local registers (d_i, d'_i) and shared heaps (h, h'). Two of the HRAM ports are connected and two are part of the global interface B.

names. Specifically, χ has to be in $sup(A^{(\mathbf{O})} \otimes A_{\overline{E}}^{(\mathbf{P})}) \to sup(A^{(\mathbf{P})} \otimes A_{\overline{E}}^{(\mathbf{O})})$, where $A_{\overline{E}} = \otimes\{A \mid (A, P) \in \overline{E}\}$.

Note that HRAM nets are not exactly the same as the networks defined in Section 2.5, even though they are similar, which is why we cannot use the same formalism here. The HRAM nets put a greater emphasis on composability by explicitly including an interface (i.e. the type that governs what nets can be composed), through which the net can communicate with an external environment. This makes it possible to reason about the semantics of open terms compiled to nets.

Figure 4.2 shows a diagram of an HRAM net with two HRAMs (interfaces A, A', two ports each), each with two running threads (t_i, t'_i) with local registers (d_i, d'_i) and shared heaps (h, h'). Two of the HRAM ports are connected and two are part of the global interface B.

The function χ gives the net connectivity. It being in $sup(A^{(\mathbf{O})} \otimes A_{\overline{E}}^{(\mathbf{P})}) \to sup(A^{(\mathbf{P})} \otimes A_{\overline{E}}^{(\mathbf{O})})$ means that it maps each input port name of the net's interface and output port name of the net's engines to either an output port name of the net's interface or an input port name of one of its engines. Since it is a bijection, each port name (and thus port) is connected to exactly one other port name, so the abstract network model we are using is point-to-point.

For an engine $e = (A, P)$, we define a *singleton* net with e as its sole engine as $singleton(e) = (\{e\}, \chi, A')$, where A' is an interface such that $\chi \vdash A =_{\mathbb{A}} A'$ and χ is given by:

$$\chi(a) \triangleq \pi(a) \text{ if } a \in sup(A^{(\mathbf{P})})$$
$$\chi(a) \triangleq \pi^{-1}(a) \text{ if } a \in sup(A'^{(\mathbf{O})})$$

A *net configuration* is a set of tuples of engine configurations and engines and a multiset of pending messages: $n = (\overline{e : E}, \overline{m}) \in \mathcal{N} = \mathcal{P}_{fin}(\mathcal{K} \times \mathcal{E}) \times \mathbf{Mset}_{fin}(\mathcal{M})$. Define the function $initial \in \mathcal{S} \to \mathcal{N}$ as

$$initial(\overline{E}, \chi, A) \triangleq (\{initial(E) \mid E \in \overline{E}\}, \varnothing),$$

a net configuration with only initial engines and no pending messages.

The operational semantics of a net $S = (\overline{E}, \chi, A)$ is specified as a transition relation $- \bar{\to} - \subseteq \mathcal{N} \times (\{\bullet\} \cup (L \times \mathcal{M}_{sup(A)})) \times \mathcal{N}$, the middle component for communication with the environment. The semantics is given in the style of the CHAM [15], where HRAMs are "molecules" and the pending messages of the HRAM net is the "solution". HRAM inputs (outputs) are to (from) the set of pending messages. Silent transitions of any HRAM are silent transitions of the net. The rules are given in Figure 4.3. The first three rules are the same as those for the asynchronous networks given in Section 2.5, allowing HRAMs to communicate within the net. The last two rules are new and let the net as a whole make observable transitions. This means that we can reason about the semantics of a net in terms of how it might communicate with an external environment.

4.1.3 *Semantics of HRAM nets*

We define *List A* for a set A to be finite sequences of elements from A, and use $s{::}s'$ for concatenation. A *trace* for a net $(\overline{E}, \chi, A)$ is a *finite sequence* of messages with polarity: $s \in List\ (L \times \mathcal{M}_{sup(A)})$. Write $\alpha \in L \times \mathcal{M}_{sup(A)}$ for single polarised messages. We use the same notational convention as before to identify inputs $(-^\bullet)$.

For a trace $s = \alpha_1{::}\alpha_2{::}\cdots{::}\alpha_n$, define $\xrightarrow{s}$ to be the following composition of relations on net configurations: $\xrightarrow{\alpha_1}\to^*\xrightarrow{\alpha_2}\to^*\cdots\xrightarrow{\alpha_n}$, where $\to^*$ is the reflexive transitive closure of $\to$, i.e. any number of silent steps are allowed in between those that are observable.

Write $traces_A$ for the set $List\ (L \times \mathcal{M}_{sup(A)})$. The denotation $[\![S]\!] \subseteq traces_A$ of a net $S = (\overline{E}, \chi, A)$ is the set of traces of observable transitions reachable from the initial net configuration $initial(S)$ using the transition relation:

$$[\![S]\!] \triangleq \{s \in traces_A \mid \exists n.initial(S) \xrightarrow{s} n\}$$

The denotation of a net includes the empty trace and is prefix-closed by construction.

As with interfaces, we are not interested in the actual port names occurring in a trace, so we define *equivariance* for sets of traces. Let $S_1 \subseteq traces_{A_1}$ and

$$\frac{e \xrightarrow[E,\chi]{} e'}{(e : E \cup \overline{e : E}, \overline{m}) \to (e' : E \cup \overline{e : E}, \overline{m})}$$

$$\frac{e \xrightarrow[E,\chi]{m} e'}{(e : E \cup \overline{e : E}, \overline{m}) \to (e' : E \cup \overline{e : E}, \{m\} \uplus \overline{m})}$$

$$\frac{e \xrightarrow[E,\chi]{m^\bullet} e'}{(e : E \cup \overline{e : E}, \{m\} \uplus \overline{m}) \to (e' : E \cup \overline{e : E}, \overline{m})}$$

$$\frac{(\mathbf{P}, a) \in A}{(\overline{e : E}, \{(a, \overline{d})\} \uplus \overline{m}) \xrightarrow{(a,\overline{d})} (\overline{e : E}, \overline{m})}$$

$$\frac{(\mathbf{O}, a) \in A}{(\overline{e : E}, \overline{m}) \xrightarrow{(a,\overline{d})^\bullet} (\overline{e : E}, \{(\chi(a), \overline{d})\} \uplus \overline{m})}$$

Figure 4.3: Operational semantics of HRAM nets
The first three rules allow HRAMs to communicate within the net. The last two rules let the net as a whole make observable transitions.

$S_2 \subseteq traces_{A_2}$ for $A_1, A_2 \in \mathcal{I}$. $S_1 =_{\mathbb{A}} S_2$ if and only if there is a permutation $\pi \in \mathbb{A} \to \mathbb{A}$ such that $\{\pi \cdot s \mid s \in S_1\} = S_2$, where $\pi \cdot \epsilon \triangleq \epsilon$ and $\pi \cdot (s{::}(l, (a, \overline{d}))) \triangleq (\pi \cdot s){::}(l, (\pi(x), \overline{d}))$.

Define the *deletion* operation $s{-}A$ which removes from a trace all elements $(l, (x, \overline{d}))$ if $x \in sup(A)$ and define the interleaving of sets of traces $S_1 \subseteq traces_A$ and $S_2 \subseteq traces_B$ as $S_1 \otimes S_2 \triangleq \{s \mid s \in traces_{A \otimes B} \wedge s{-}B \in S_1 \wedge s{-}A \in S_2\}$.

Define the composition of the sets of traces

$$S_1 \subseteq traces_{A \Rightarrow B} \text{ and}$$
$$S_2 \subseteq traces_{B' \Rightarrow C} \text{ where } \pi \vdash B =_{\mathbb{A}} B'$$

as the usual *synchronisation and hiding* in trace semantics:

$$S_1; S_2 \triangleq \{s{-}B \mid s \in traces_{A \otimes B \otimes C} \wedge s{-}C \in S_1 \wedge \pi \cdot s^{*B}{-}A \in S_2\}$$

(where s^{*B} is s where the messages from B have reversed polarity.)

Two nets, $f = (\overline{E}_f, \chi_f, I_f)$ and $g = (\overline{E}_g, \chi_g, I_g)$ are said to be *structurally equivalent* if they are graph-isomorphic, i.e. $\pi \cdot \overline{E}_f = \overline{E}_g$, $\pi \vdash I_f =_{\mathbb{A}} I_g$ and $\chi_g \circ \pi = \pi \circ \chi_f$.

Theorem 4.1.1. If S_1 and S_2 are structurally equivalent nets, then $[\![S_1]\!] =_{\mathbb{A}} [\![S_2]\!]$.

Proof. A straightforward induction on the trace length, in both directions. □

4.1.4 *HRAM nets as a category*

In this subsection we will show that HRAM nets form a symmetric compact-closed category. This establishes that our definitions are sensible and that HRAM nets are equal up to topological isomorphisms. This result also shows that the structure of HRAM nets is very loose.

The category, called **HRAMnet**, is defined as follows:

- Objects are interfaces $A \in \mathcal{P}_{fin}(Port)$ identified up to equivariance.
- A morphism $f : A \to B$ is a well-formed net on the form $(\overline{E}, \chi, A \Rightarrow B)$, for some $\overline{E}$ and χ. We will identify morphisms that have the same denotation, i.e. if $[\![f]\!] =_{\mathbb{A}} [\![g]\!]$ then $f = g$ (in the category).
- The identity morphism for an object A is

$$id_A \triangleq (\varnothing, \chi, A \Rightarrow A')$$

for an A' such that $\pi \vdash A =_{\mathbb{A}} A'$ and

$$\chi(a) \triangleq \pi(a) \text{ if } a \in sup(A^{*(\mathbf{O})})$$
$$\chi(a) \triangleq \pi^{-1}(a) \text{ if } a \in sup(A'^{(\mathbf{O})}).$$

Note that $A \Rightarrow A' = A^* \cup A'$. This means that the identity is *pure connectivity*.

- Composition of two morphisms $f = (\overline{E}_f, \chi_f, A \Rightarrow B) : A \to B$ and $g = (\overline{E}_g, \chi_g, B' \Rightarrow C) : B' \to C$, such that $\pi \vdash B =_{\mathbb{A}} B'$, is

$$f; g = (\overline{E}_f \cup \overline{E}_g, \chi_{f;g}, A \Rightarrow C) : A \to C$$

where

$$\chi_{f;g}(a) \triangleq \chi_f(a) \text{ if } a \in sup(A^{*(\mathbf{O})} \otimes I_f^{(\mathbf{P})}) \wedge \chi_f(a) \notin sup(B)$$
$$\chi_{f;g}(a) \triangleq \chi_g(a) \text{ if } a \in sup(C^{(\mathbf{O})} \otimes I_g^{(\mathbf{P})}) \wedge \chi_g(a) \notin sup(B')$$
$$\chi_{f;g}(a) \triangleq \chi_g(\pi(\chi_f(a))) \text{ if } a \in sup(A^{*(\mathbf{O})} \otimes I_f^{(\mathbf{P})}) \wedge \chi_f(a) \in sup(B)$$
$$\chi_{f;g}(a) \triangleq \chi_f(\pi^{-1}(\chi_g(a))) \text{ if } a \in sup(C^{(\mathbf{O})} \otimes I_g^{(\mathbf{P})}) \wedge \chi_g(a) \in sup(B')$$

and

$$I_f \triangleq \otimes\{A \mid (A, P) \in \overline{E}_f\}$$
$$I_g \triangleq \otimes\{A \mid (A, P) \in \overline{E}_g\}.$$

Note We identify HRAMs with interfaces of the same shape in the category, which means that our objects and morphisms are in reality unions of equivariant sets, i.e. sets of interfaces whose elements are the same but with different port name permutations. In defining the operations of our category we use *representatives* for these sets, and require that the representatives are chosen such that their sets of port names are disjoint (but same-shaped when the operation calls for it). The composition operation may appear to be partial because of this requirement, but we can always find equivariant representatives that fulfil it.

It is possible to find other representations of interfaces that do not rely on equivariance. For instance, an interface could simply be two natural numbers — the number of input and output ports. Another possibility would be to make the tensor the *disjoint* union operator. Both of these would, however, lead to a lot of bureaucracy relating to injection functions to make sure that port connections are routed correctly. Our formulation, while seemingly complex, leads to very little bureaucracy, and is easy to implement.

Proposition 4.1.2. **HRAMnet** is a category.

The proofs of this proposition and other theorems from this chapter are of a technical nature and are given in Appendix B so as to not break the flow of reading.

We will now show that **HRAMnet** is a symmetric monoidal category:

- The tensor product of two objects A, B, $A \otimes B$ has already been defined. We define the tensor of two morphisms

$$f = (\overline{E}_f, \chi_f, A \Rightarrow B) \text{ and}$$
$$g = (\overline{E}_g, \chi_g, C \Rightarrow D)$$

  as

$$f \otimes g = (\overline{E}_f \cup \overline{E}_g, \chi_f \otimes \chi_g, A \otimes C \Rightarrow B \otimes D).$$

- The unit object is the empty interface, $\varnothing$.
- Since $A \otimes (B \otimes C) = A \cup B \cup C = (A \otimes B) \otimes C$ we define the associator $\alpha_{A,B,C} \triangleq id_{A \otimes B \otimes C}$ with the obvious inverse.
- Similarly, since $\varnothing \otimes A = \varnothing \cup A = A = A \cup \varnothing = A \otimes \varnothing$, we define the left unitor $\lambda_A \triangleq id_A$ and the right unitor $\rho_A \triangleq id_A$.
- Since $A \otimes B = A \cup B = B \cup A = B \otimes A$ we define the braiding $\gamma_{A,B} \triangleq id_{A \otimes B}$.

Proposition 4.1.3. **HRAMnet** is a symmetric monoidal category.

Next we show that **HRAMnet** is a compact-closed category:

- We have already defined the dual A^* of an object A.
- Since $\varnothing \Rightarrow (A^* \otimes A') = \varnothing^* \cup (A^* \cup A') = A \Rightarrow A'$ we can define the unit $\eta_A \triangleq id_A$ and since $A \otimes A'^* \Rightarrow \varnothing = (A \cup A'^*)^* \cup \varnothing = A^* \cup A' = A \Rightarrow A'$ we can define the counit $\varepsilon_A \triangleq id_A$.

This leads us directly to the following result — what we set out to show:

Proposition 4.1.4. **HRAMnet** is a symmetric compact-closed category.

The following two theorems can be proved by induction on the trace length, and provide a connection between the **HRAMnet** tensor and composition and trace interleaving and composition.

Theorem 4.1.5. If $f : A \to B$ and $g : C \to D$ are morphisms of **HRAMnet** then $[\![f \otimes g]\!] = [\![f]\!] \otimes [\![g]\!]$.

Theorem 4.1.6. If $f : A \to B$ and $g : B' \to C$ are morphisms of **HRAMnet** such that $\pi \vdash B =_{\mathbb{A}} B'$ then $[\![f; g]\!] = [\![f]\!]; [\![g]\!]$.

The following result explicates how communicating HRAMs can be combined into a single machine, where the intercommunication is done with jumping rather than message passing, in a sound way:

Theorem 4.1.7. If $E_1 = (A_1, P_1)$ and $E_2 = (A_2, P_2)$ are engines such that $S = (\{E_1, E_2\}, \chi, A)$ is a net, then $E_{12} = (A_1 \otimes A_2, P_1 \cup P_2)$ is an engine, $S' = (\{E_{12}\}, \chi, A)$ is a net, and $[\![S]\!] \subseteq [\![S']\!]$.

This corresponds to the combination construction of Chapter 3. Note that it is simpler to define and prove the soundness of the operation in this setting. There are two reasons for this. The first is that we do not have to modify the code of the machines since the `spark` instruction has different transition rules depending on whether a port is local or remote. The second is that HRAMs do not have stacks, for which the order of computation matters, which complicates the proof. Heaps, on the other hand, are of course not as order sensitive.

We define a family of projection HRAM nets $\Pi_{i,A_1 \otimes \cdots \otimes A_n} : A_1 \otimes \cdots \otimes A_n \to A_i$ by first constructing a family of "sinks" $!_A : A \to I \stackrel{\Delta}{=} \mathit{singleton}((A \Rightarrow I, P))$ where $I = \varnothing$ and $P(a) = \mathtt{end}$ for each a in its domain and then defining e.g. $\Pi_{1,A \otimes B} : A \otimes B \to A \stackrel{\Delta}{=} id_A \otimes !_B$.

4.2 GAME NETS FOR ICA

The structure of a **HRAMnet** token is determined by the number of registers r and the message size r_m, which are globally fixed. To implement machines for (our variation of) game semantics we require four message components: a port name, two pointer names, and a data fragment, meaning that $r_m = 3$. We choose $r = 4$, to get an additional register for temporary thread values to work with. From this point on, messages in nets and traces will be restricted to this form.

The message structure is intended to capture the structure of a move when game semantics is expressed in the nominal model. The port name is the move, the first name is the "point" whereas the second name is the "butt" of a justification arrow, and the data is the value of the move. This direct and abstract encoding of the justification pointer as names is quite different to that used in the Pointer Abstract Machine (PAM) and in other GOI-based token machines. In

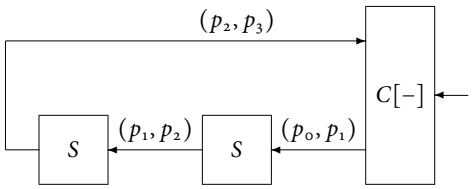


Figure 4.4: Non-locality of names in HRAM composition

The token is received by S and propagated to the other S HRAM, this time with tokens (p_1, p_2)*. This trace of events* $(p_0, p_1)::(p_1, p_2)$ *corresponds to the existence of a justification pointer from the second action to the first in the game model. The essential correctness invariant for a well-formed trace representing a game semantic play is that each token consists of a* known *name and a* fresh *name (if locally created, or* unknown *if externally created). However, the second S machine will respond with* (p_2, p_3) *to* (p_1, p_2)*, leading to a situation where* $C[-]$ *receives a token formed from two unknown tokens.*

PAM the pointer is represented by a sequence of integers encoding the hereditary justification of the move, which is a snapshot of the computational causal history of the move, just like in GOI-based machines. Such encodings have an immediate negative consequence, as tokens can become impractically large in complex computations, especially involving recursion. Large tokens entail not only significant communication overheads but also the computational overheads of decoding their structure. A subtler negative consequence of such an encoding is that it makes supporting the semantic structures required to interpret state and concurrency needlessly complicated and inefficient. The nominal representation is simple and compact, and efficiently exploits local machine memory (heap) in a way that previous abstract machines, of a "functional" nature, do not.

The price that we pay is a failure of compositionality, which we will illustrate shortly. The rest of the section will show how compositionality can be restored without substantially changing the HRAM framework. If in HRAM nets compositionality is "plug-and-play", as apparent from its compact-closed structure, GAM composition must be mediated by a family of operators which are themselves HRAMs.

In this simple motivating example it is assumed that the reader is familiar with game semantics, and several of the notions to be introduced formally in the next subsections are anticipated. We trust that this will not be confusing.

Let S be a HRAM representing the game semantic model for the *successor* operation $S : nat \to nat$. The HRAM net in Figure 4.4 represents a (failed) attempt to construct an interpretation for the term $x : nat \vdash S(S(x)) : nat$ in a context $C[-_{nat}] : nat$. This is the standard way of composing GOI-like machines.

The labels along the edges of the HRAM net trace a token (a, p_o, p_1, d) sent by the context $C[-]$ in order to evaluate the term. We elide a and d, which are irrelevant, to keep the diagram uncluttered. The token is received by S and propagated to the other S HRAM, this time with tokens (p_1, p_2). This trace of events $(p_o, p_1)::(p_1, p_2)$ corresponds to the existence of a justification pointer from the second action to the first in the game model. The essential correctness invariant for a well-formed trace representing a game semantic play is that each token consists of a *known* name and a *fresh* name (if locally created, or *unknown* if externally created). However, the second S machine will respond with (p_2, p_3) to (p_1, p_2), leading to a situation where $C[-]$ receives a token formed from two unknown tokens.

In game semantics, the composition of $(p_o, p_1)::(p_1, p_2)$ and $(p_1, p_2)::(p_2, p_3)$ should lead to $(p_o, p_1)::(p_1, p_3)$, as justification pointers are "extended" so that they never point into a move hidden through composition. This is precisely what the composition operator, a specialised HRAM, will be designed to achieve.

4.2.1 *Game Abstract Machines and nets*

Definition 4.2.1. We define a *game interface* (cf. *arena*) as a tuple

$$\mathfrak{A} = (A, qst_{\mathfrak{A}}, ini_{\mathfrak{A}}, \vdash_{\mathfrak{A}})$$

where

- $A \in \mathcal{I}$ is an interface. For game interfaces $\mathfrak{A}, \mathfrak{B}, \mathfrak{C}$ we will write A, B, C and so on for their underlying interfaces.
- The set of ports is partitioned into a subset of question port names $qst_{\mathfrak{A}}$ and answer port names, $ans_{\mathfrak{A}}$, $qst_{\mathfrak{A}} \uplus ans_{\mathfrak{A}} = sup(A)$.
- The set of initial port names $ini_{\mathfrak{A}}$ is a subset of the **O**-labelled question ports.
- The enabling relation $\vdash_{\mathfrak{A}}$ relates question port names to non-initial port names such that if $a \vdash_{\mathfrak{A}} a'$ for port names $a \in qst_{\mathfrak{A}}$ with $(l, a) \in A$ and $a' \in sup(A) \setminus ini_{\mathfrak{A}}$ with $(l', a') \in A$, then $l \neq l'$.

For notational consistency, write $opp_{\mathfrak{A}} \triangleq sup(A^{(\mathbf{O})})$ and $prop_{\mathfrak{A}} \triangleq sup(A^{(\mathbf{P})})$. Call the set of all game interfaces $\mathcal{I}_{\mathfrak{G}}$. Game interfaces are equivariant, $\pi \vdash \mathfrak{A} =_{\mathbb{A}} \mathfrak{B}$, if and only if $\pi \vdash A =_{\mathbb{A}} B$, $\{\pi(a) \mid a \in qst_{\mathfrak{A}}\} = qst_{\mathfrak{B}}$, $\{\pi(a) \mid a \in ini_{\mathfrak{A}}\} = ini_{\mathfrak{B}}$ and $\{(\pi(a), \pi(a')) \mid a \vdash_{\mathfrak{A}} a'\} = \vdash_{\mathfrak{B}}$.

Definition 4.2.2. For game interfaces (with disjoint sets of port names) $\mathfrak{A}$ and $\mathfrak{B}$, we define:

$$\mathfrak{A} \otimes \mathfrak{B} \triangleq (A \otimes B, qst_{\mathfrak{A}} \cup qst_{\mathfrak{B}}, ini_{\mathfrak{A}} \cup ini_{\mathfrak{B}}, \vdash_{\mathfrak{A}} \cup \vdash_{\mathfrak{B}})$$
$$\mathfrak{A} \Rightarrow \mathfrak{B} \triangleq (A \Rightarrow B, qst_{\mathfrak{A}} \cup qst_{\mathfrak{B}}, ini_{\mathfrak{B}}, \vdash_{\mathfrak{A}} \cup \vdash_{\mathfrak{B}} \cup (ini_{\mathfrak{B}} \times ini_{\mathfrak{A}})).$$

A *GAM net* is a tuple $G = (S, \mathfrak{A}) \in \mathcal{S} \times \mathcal{I}_{\mathfrak{G}}$ consisting of a net and a game interface such that $S = (\overline{E}, \chi, A)$, i.e. the interface of the underlying net is the same as that of the game interface. The denotational semantics of a GAM net $G = (S, \mathfrak{A})$ is just that of the underlying HRAM net: $[\![G]\!] \triangleq [\![S]\!]$.

4.2.2 *Game traces*

To be able to use game semantics as the specification for game nets we define the usual legality conditions on traces, following the formulation of nominal games [48].

Definition 4.2.3. The bound and free pointers bp and $fp \in traces \to \mathcal{P}(\mathbb{P})$ are:

$$bp(\epsilon) \triangleq \varnothing$$
$$bp(s{::}(l, (a, p, p', d))) \triangleq bp(s) \cup \{p'\}$$
$$fp(\epsilon) \triangleq \varnothing$$
$$fp(s{::}(l, (a, p, p', d)) \triangleq fp(s) \cup (\{p\} \setminus bp(s))$$

The pointers of a trace are defined as $ptrs(s) = bp(s) \cup fp(s)$.

Definition 4.2.4. Define $enabled_{\mathfrak{A}} \in traces_A \to \mathcal{P}(sup(A) \times \mathbb{P})$ inductively as follows:

$$enabled_{\mathfrak{A}}(\epsilon) \triangleq \varnothing$$
$$enabled_{\mathfrak{A}}(s{::}(l, (a, p, p', d))) \triangleq enabled_{\mathfrak{A}}(s) \cup \{(a', p') \mid a \vdash_{\mathfrak{A}} a'\}$$

Definition 4.2.5. We define the following relations over traces:

- Write $s' \leqslant s$ if and only if there is a trace s_1 such that $s'{::}s_1 = s$, i.e. s' is a *prefix* of s.
- Write $s' \leqslant s$ if and only if there are traces s_1, s_2 such that $s_1{::}s'{::}s_2 = s$, i.e. s' is a *segment* of s.

Definition 4.2.6. For an arena $\mathfrak{A}$ and a trace $s \in traces_A$, we define the following legality conditions:

- s has *unique pointers* when $s' :: (l, (a, p, p', d)) \leqslant s$ implies $p' \notin ptrs(s')$.
- s is *correctly labelled* when $(l, (a, p, p', d)) \subseteq s$ implies $a \in sup(A^{(l)})$.
- s is *justified* when $s' :: (l, (a, p, p', d)) \leqslant s$ and $a \notin ini_{\mathfrak{A}}$ implies $(a, p) \in enabled_{\mathfrak{A}}(s')$.
- s is *well-opened* when $s' :: (l, (a, p, p', d)) \leqslant s$ and $a \in ini_{\mathfrak{A}}$ implies $s' = \epsilon$.
- s is *strictly scoped* when $(l, (a, p, p', d)) :: s' \subseteq s$ with $a \in ans_{\mathfrak{A}}$ implies $p \notin fp(s')$.
- s is *strictly nested* when $(l_1, (a_1, p, p', d_1)) :: s' :: (l_2, (a_2, p', p'', d_2)) :: s'' :: (l_3, (a_3, p', p''', d_3)) \subseteq s$ implies $(l_4, (a_4, p'', -, d_4)) \subseteq s''$ for port names $a_1, a_2 \in qst_{\mathfrak{A}}$ and $a_3, a_4 \in ans_{\mathfrak{A}}$.
- s is *alternating* when $(l_1, m_1) :: (l_2, m_2) \subseteq s$ implies $l_1 \neq l_2$.

Definition 4.2.7. We say that a question message $\alpha = (l, (a, p, p', d))$ $(a \in qst_{\mathfrak{A}})$ is *pending* in a trace $s = s_1 :: \alpha :: s_2$ if and only if there is no answer $\alpha' = (l', (a', p', p'', d')) \subseteq s_2$ $(a' \in ans_{\mathfrak{A}})$, i.e. the question has not been answered.

Write $P_{\mathfrak{A}}$ for the subset of $traces_A$ consisting of the traces that have unique pointers, are correctly labelled, justified, strictly scoped and strictly nested.

For a set of traces P, write P^{alt} for the subset consisting of only alternating traces, and P^{st} (for single-threaded) for the subset consisting of only well-opened traces.

Definition 4.2.8. If $s \in traces$ and $X \subseteq \mathbb{P}$, define the *hereditarily justified* trace $s \upharpoonright X$, where inductively $(s', X') = s \upharpoonright X$:

$$
\begin{aligned}
\epsilon \upharpoonright X &\triangleq (\epsilon, X) \\
s :: (l, (a, p, p', d)) \upharpoonright X &\triangleq (s' :: (l, (a, p, p', d)), B \cup \{p'\}) && \text{if } p \in X' \\
s :: (l, (a, p, p', d)) \upharpoonright X &\triangleq (s', B) && \text{if } p \notin X'
\end{aligned}
$$

Write $s \upharpoonright X$ for s' when $s \upharpoonright X = (s', X')$ when it is convenient.

4.2.3 *Copycat*

The quintessential game semantic behaviour is that of the *copycat strategy*, as it appears in various guises in the representation of all structural morphisms of any category of strategies. A copycat not only replicates the behaviour of its opponent in terms of moves, but also in terms of justification structures.

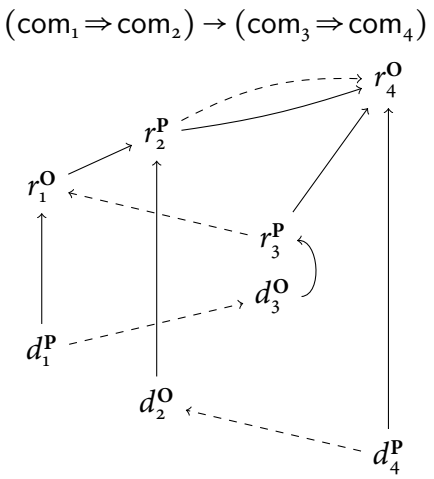


Figure 4.5: A typical play for copycat

The full lines represent justification pointers, and the dashed lines "copycat links", which we use to preserve the justification structure.

Because of this, the copycat strategy needs to be either history-sensitive (stateful) or the justification information needs to be carried along with the token. We take the former approach, in contrast to IAM and other GOI-inspired machines. To use a metaphor, the GOI approach is to update a map containing the route from the starting location as you go, and our approach is to leave a trail of bread crumbs behind us.[1]

Consider the identity (or copycat) strategy on $\mathsf{com} \Rightarrow \mathsf{com}$, where com is a two-move arena (one question, one answer). A typical play may look as in Figure 4.5. The full lines represent justification pointers, and the trace (play) is represented nominally as

$$(r_4, p_0, p_1)::(r_2, p_1, p_2)::(r_1, p_2, p_3)::(r_3, p_1, p_4)::(d_3, p_4)\cdots$$

To preserve the justification structure, a copycat engine only needs to store "copycat links", which are shown as dashed lines in the diagram between question moves. In this instance, for an input on r_4, a heap value mapping a freshly created p_2 (the pointer to r_2) to p_1 (the pointer from r_4) is added.

The reason for mapping p_2 to p_1 becomes clear when the engine later gets an input on r_1 with pointers p_2 and p_3. It can then replicate the move to r_3, but using p_1 as a justifier. By following the p_2 pointer in the heap it gets p_1 so it can produce (r_3, p_1, p_4), where p_4 is a fresh heap value mapping to p_3. When receiving an answer, i.e. a d move, the copycat link can be dereferenced and then *discarded* from the heap.

[1] But unlike the fairy tale, there are no birds in our formalism.

The following HRAM macro-instructions are useful in defining copycat machines to, respectively, handle the pointers in an initial question, a non-initial question and an answer:

$$\texttt{cci} \triangleq \texttt{swap}\ 0,1;\ 1 \leftarrow \texttt{new}\ 0,3$$
$$\texttt{ccq} \triangleq 1 \leftarrow \texttt{new}\ 1,3;\ 0,3 \leftarrow \texttt{get}\ 0$$
$$\texttt{cca} \triangleq \texttt{swap}\ 0,1;\ 0,3 \leftarrow \texttt{get1};\ \texttt{free}\ 1$$

For game interfaces $\mathfrak{A}$ and $\mathfrak{A}'$ such that $\pi \vdash \mathfrak{A} =_{\mathbb{A}} \mathfrak{A}'$, we define a generalised copycat engine as $\mathbb{CC}_{C,\pi,\mathfrak{A}} = (A \Rightarrow A', P)$, where:

$$\begin{aligned} P \triangleq\ & \{q_2 \mapsto C;\ \texttt{spark}\ q_1 \mid q_2 \in ini_{\mathfrak{A}'} \wedge q_1 = \pi^{-1}(q_2)\} \\ & \cup \{q_2 \mapsto \texttt{ccq};\ \texttt{spark}\ q_1 \mid q_2 \in (opp_{\mathfrak{A}'} \cap qst_{\mathfrak{A}'}) \setminus ini_{\mathfrak{A}'} \wedge q_1 = \pi^{-1}(q_2)\} \\ & \cup \{a_2 \mapsto \texttt{cca};\ \texttt{spark}\ a_1 \mid a_2 \in opp_{\mathfrak{A}'} \cap ans_{\mathfrak{A}'} \wedge a_1 = \pi^{-1}(a_2)\} \\ & \cup \{q_1 \mapsto \texttt{ccq};\ \texttt{spark}\ q_2 \mid q_1 \in opp_{\mathfrak{A}} \cap qst_{\mathfrak{A}} \wedge q_2 = \pi(q_1)\} \\ & \cup \{a_1 \mapsto \texttt{cca};\ \texttt{spark}\ a_2 \mid a_1 \in opp_{\mathfrak{A}} \cap ans_{\mathfrak{A}} \wedge a_2 = \pi(a_1)\} \end{aligned}$$

This copycat engine is parameterised by an initial instruction C, which is run when receiving an initial question. The engine for an ordinary copycat, i.e. the identity of games, is $\mathbb{CC}_{\texttt{cci},\pi,\mathfrak{A}}$. By slight abuse of notation, write $\mathbb{CC}_{\mathfrak{A}}$ for the singleton copycat game net $(singleton(\mathbb{CC}_{\texttt{cci},\pi,\mathfrak{A}}), \mathfrak{A} \Rightarrow \pi \cdot \mathfrak{A})$.

Following [48], we define a partial order $\leqslant$ over polarities, L, as $\mathbf{O} \leqslant \mathbf{O}, \mathbf{O} \leqslant \mathbf{P}, \mathbf{P} \leqslant \mathbf{P}$ and a preorder $\preccurlyeq$ over traces from $P_{\mathfrak{A}}$ to be the least reflexive and transitive relation such that if $l_1 \leqslant l_2$ then

$$\begin{aligned} s_1{::}(l_1,(a_1,p_1,p_1',d_1)){::}(l_2,(a_2,p_2,p_2',d_2)){::}s_2 \\ \preccurlyeq s_1{::}(l_2,(a_2,p_2,p_2',d_2)){::}(l_1,(a_1,p_1,p_1',d_1)){::}s_2, \end{aligned}$$

where $p_1' \neq p_2$. A set of traces $S \subseteq P_{\mathfrak{A}}$ is *saturated* if and only if, for $s, s' \in P_{\mathfrak{A}}$, $s' \preccurlyeq s$ and $s \in S$ implies $s' \in S$. If $S \subseteq P_{\mathfrak{A}}$ is a set of traces, let $sat(S)$ be the smallest saturated set of traces that contains S.

The usual definition of the copycat strategy (in the alternating and single-threaded setting) as a set of traces is

$$cc^{st,alt}_{\mathfrak{A},\mathfrak{A}'} \triangleq \{s \in P^{st,alt}_{\mathfrak{A}\Rightarrow\mathfrak{A}'} \mid \forall s' \leqslant_{\text{even}} s.\ s'^{*} \upharpoonright A =_{\mathbb{AIP}} s' \upharpoonright A'\}$$

Definition 4.2.9. A set of traces S_1 is **P**-*closed* with respect to a set of traces S_2 if and only if $s' \in S_1 \cap S_2$ and $s = s'{::}(\mathbf{P},(a,p,p',d)) \in S_1$ implies $s \in S_2$.

The intuition of **P**-closure is that if the trace s' is "legal" according to S_2, then any outputs that can occur after s' in S_1 are also legal.

Definition 4.2.10. We say that a GAM net f *implements* a set of traces S if and only if $S \subseteq [\![f]\!]$ and $[\![f]\!]$ is **P**-closed with respect to S.

This is the form of the statements of correctness for game nets that we want; it certifies that the net f can accommodate all traces in S and, furthermore, that it only produces legal outputs when given valid inputs.

The main result of this section establishes the correctness of the GAM for copycat.

Theorem 4.2.11. $\mathbb{C}_{\pi,\mathfrak{A}}$ implements $cc_{\mathfrak{A},\pi\cdot\mathfrak{A}}$.

The first part of this theorem, but for single-threaded and alternating traces, is proved in Lemma 4.2.17. The second part, but for single-threaded traces, is point 2 of Lemma 4.2.22. Lemma 4.2.16 essentially lets us lift proofs on single-threaded traces to multi-threaded traces, and Lemma 4.2.13 similarly lets us lift the proofs to non-alternating traces.

Lemma 4.2.12. If $n_1 = (\overline{e : E}, \overline{m})$ and $n_1' = (\overline{e' : E}, \overline{m}')$ are net configurations of a net $f = (\overline{E}, \chi, A)$, and $n_1 \xrightarrow{(x)} n_1'$ $((x) \in \{\bullet\} \cup (L \times \mathcal{M}_{sup(A)})$ then $n_2 \xrightarrow{(x)} n_2'$ where $n_2 = (\overline{e : E}, \overline{m} \uplus \{m\})$ and $n_2' = (\overline{e' : E}, \overline{m}' \uplus \{m\})$.

Lemma 4.2.13. If f is a net and s a trace, then

1. $s = s_1{::}(l, m_1){::}(\mathbf{O}, m){::}s_2 \in [\![f]\!]$ with witness $initial(f) \xrightarrow{s} n$ implies $s' = s_1{::}(\mathbf{O}, m){::}(l, m_1){::}s_2 \in [\![f]\!]$ with $initial(f) \xrightarrow{s'} n$ and

2. $s = s_1{::}(\mathbf{P}, m){::}(l, m_1){::}s_2 \in [\![f]\!]$ with witness $initial(f) \xrightarrow{s} n$ implies $s' = s_1{::}(l, m_1){::}(\mathbf{P}, m){::}s_2 \in [\![f]\!]$ with $initial(f) \xrightarrow{s'} n$.

A special case of this lemma is that if $G = (f, \mathfrak{A})$ and, for a set of traces $S \subseteq P_{\mathfrak{A}}$, $S \subseteq [\![G]\!]$ holds, then $sat(S) \subseteq [\![G]\!]$.

Lemma 4.2.14. If $s, s' \in P_{\mathfrak{A}}$ and $s' \preccurlyeq s$, then

1. $enabled(s) = enabled(s')$,

2. $bp(s) = bp(s')$, and

3. $fp(s) = fp(s')$.

Lemma 4.2.15. Let $S \subseteq P_{\mathfrak{A}}$ be a saturated set of traces. If $s, s' \in S$ are traces such that $s' \preccurlyeq s$ and $s{::}\alpha \in S$, then $s'{::}\alpha \in S$.

Lemma 4.2.16. For any game net $f = (S, \mathfrak{A})$ and trace $s \in P_{\mathfrak{A}}$, $s \in [\![f]\!]$ if and only if $\forall p \in fp(s).s \upharpoonright \{p\} \in [\![f]\!]$.

Lemma 4.2.17. $cc^{st,alt}_{\mathfrak{A},\pi\cdot\mathfrak{A}} \subseteq [\![\mathbb{C}_{\pi,\mathfrak{A}}]\!]$.

Lemma 4.2.18. If $s = s_1{::}o{::}s_2 \in cc_{\mathfrak{A},\mathfrak{A}'}$ and $p \notin s_2$, then $s{::}p \in cc_{\mathfrak{A},\mathfrak{A}'}$, where $o = (\mathbf{O}, (a, p, p', d))$ and $p = (\mathbf{P}, (\tilde{\pi}_{\mathbb{A}}(a), \tilde{\pi}_{\mathbb{P}}(p), \tilde{\pi}_{\mathbb{P}}(p'), d))$ (i.e. the "copy" of o).

Definition 4.2.19. Define the multiset of messages that a net configuration n is ready to immediately send as $ready(n) \triangleq \{(\mathbf{P}, m) \mid \exists n'.\ n \to^* \xrightarrow{(\mathbf{P},m)} n'\}$.

Definition 4.2.20. If s is a trace, h is a heap, $\mathfrak{A}$ is a game interface, and $\pi_{\mathbb{P}}$ is a permutation over $\mathbb{P}$, we say that h is a *copycat heap* for s over $\mathfrak{A}$ if and only if:

For every pending $\mathbf{P}$-question from $\mathfrak{A}$ in s, i.e. $(\mathbf{P}, (a, p, p', d)) \subseteq s$ ($a \in qst_{\mathfrak{A}}$), $h(p') = (\tilde{\pi}_{\mathbb{P}}(p'), \varnothing)$.

Lemma 4.2.21. If $s \in cc$ is a trace such that $initial(\mathbb{C}) \xrightarrow{s} n$, then the following holds:

1. If $n \to^* n'$ then $ready(n) = ready(n')$.

2. If $n \to^* \xrightarrow{(\mathbf{P},m)} n'$, then $ready(n) = ready(n') \cup \{(\mathbf{P}, m)\}$.

As we are only interested in what is observable, the trace s is thus equivalent to one where silent steps are only taken in one go by one thread right before outputs.

Lemma 4.2.22. If $s \in cc^{st}$ is a trace such that $initial(\mathbb{C}) \xrightarrow{s} n$ for an $n = (\{(\overline{t}, h) \ :\ E\}, \overline{m})$, then there exists a permutation $\pi_{\mathbb{P}}$ over $\mathbb{P}$ such that the following holds:

1. The heap h is a copycat heap for s over $\mathfrak{A} \Rightarrow \mathfrak{A}'$.

2. The set of messages that n can immediately send, $ready(n)$, is exactly the set of messages p such that $s = s_1{::}o{::}s_2$ and $p \notin s_2$ where the form of o and p is $o = (\mathbf{O}, (a, p, p', d))$ and $p = (\mathbf{P}, (\tilde{\pi}_{\mathbb{A}}(a), \tilde{\pi}_{\mathbb{P}}(p), \tilde{\pi}_{\mathbb{P}}(p'), d))$ (i.e. the "copy" of o).

4.2.4 *Composition*

The definition of composition in Hyland-Ong games [77] is nearly indistinguishable from our definition of trace composition, so we might expect HRAM net composition to correspond to it. That is, however, only superficially true: the nominal setting that we are using [48] brings to light what happens to the justification pointers in composition.

If A is an interface, $s \in \mathit{traces}_A$ and $X \subseteq \mathit{sup}(A)$, we define the *reindexing deletion* operator $s \upharpoonright X$ as follows, where $(s', \rho) = s \upharpoonright X$ inductively:

$$\begin{aligned}
\epsilon \upharpoonright X &\triangleq (\epsilon, id) \\
s{::}(l,(a,p,p',d)) \upharpoonright X &\triangleq (s'{::}(l,(a,\rho(p),p',d)),\rho) && \text{if } a \notin X \\
s{::}(l,(a,p,p',d)) \upharpoonright X &\triangleq (s',\rho \cup \{p' \mapsto \rho(p)\}) && \text{if } a \in X
\end{aligned}$$

We write $s \upharpoonright X$ for s' when $s \upharpoonright X = (s', \rho)$ in the following definition:

Definition 4.2.23. The *game composition* of the sets of traces $S_1 \subseteq \mathit{traces}_{A \Rightarrow B}$ and $S_2 \subseteq \mathit{traces}_{B' \Rightarrow C}$ with $\pi \vdash B =_{\mathbb{A}} B'$ is

$$S_1 ;_{\mathfrak{G}} S_2 \triangleq \{ s \upharpoonright B \mid s \in \mathit{traces}_{A \otimes B \otimes C} \wedge s \upharpoonright C \in S_1 \wedge \pi \cdot s^{*B} \upharpoonright A \in S_2 \}$$

Clearly we have $S_1 ; S_2 \neq S_1 ;_{\mathfrak{G}} S_2$ for some sets of traces S_1 and S_2, which reinforces the practical problem in the beginning of this section.

Composition is constructed from three copycat-like behaviours, as sketched in Figure 4.6 for a typical play at some types A, B and C. As a trace in the nominal model, this is:

$$\begin{aligned}
&(q_6, p_0, p_1){::}(q_4, p_1, p_2){::}(q_3, p_2, p_3){::} \\
&\qquad (q_2, p_1, p_4){::}(q_1, p_4, p_5){::}(q_5, p_1, p_6){::}(a_5, p_6){::} \\
&\qquad\qquad (a_1, p_5){::}(a_2, p_4){::}(a_3, p_3){::}(a_4, p_2){::}(a_6, p_1)
\end{aligned}$$

We see that this *almost* corresponds to three interleaved copycats as described above; between A, B, C and A', B', C'. But there is a small difference: the move q_1, if it were to blindly follow the recipe of a copycat, would dereference the pointer p_4, yielding p_3, and so incorrectly make the move q_5 justified by q_3, whereas it really should be justified by q_6 as in the diagram. This is precisely the problem explained in the beginning of this section.

To make a pointer *extension*, when the B-initial move q_3 is performed, it should map p_4 not only to p_3, but also to the pointer that p_2 points to, which is p_1 (the dotted line in the diagram). When the A-initial move q_1 is performed, it has access to both of these pointers that p_4 maps to, and can correctly make the q_5 move by associating it with pointers p_1 and a fresh p_6.

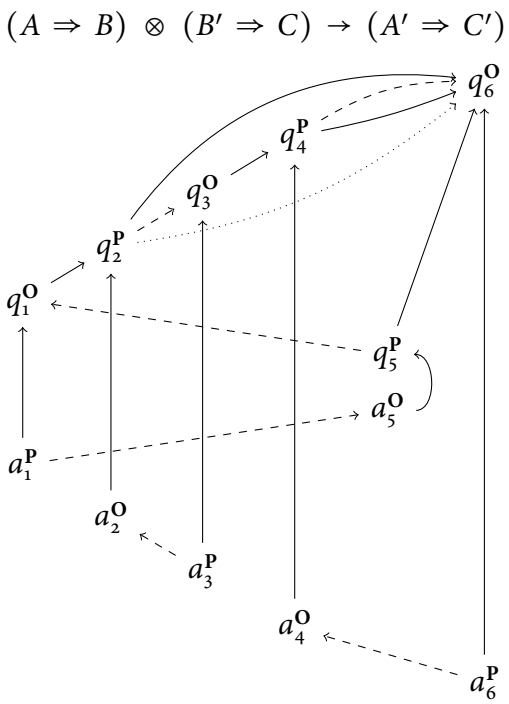


Figure 4.6: Composition from copycat

Composition is constructed from three copycat-like behaviours for a typical play at some types A, B and C. It almost *corresponds to three interleaved copycats; between A, B, C and A', B', C'. But there is a small difference: the move q_1, if it were to blindly follow the recipe of a copycat, would dereference the pointer p_4, yielding p_3, and so incorrectly make the move q_5 justified by q_3, whereas it really should be justified by q_6. The dotted lines represent pointer* extensions*, which are used to alleviate this issue.*

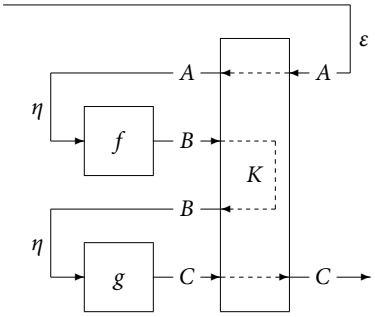


Figure 4.7: Composing GAMs using the K HRAM

Composition is mediated by the operator K, which preserves the locality of freshly generated names, exchanging non-local pointer names with local pointer names and storing the mapping between the two as copycat links, indicated diagrammatically by dashed lines.

Let $\mathfrak{A}'$, $\mathfrak{B}'$, and $\mathfrak{C}'$ be game interfaces such that $\pi_{\mathfrak{A}} \vdash \mathfrak{A} =_{\mathbb{A}} \mathfrak{A}'$, $\pi_{\mathfrak{B}} \vdash \mathfrak{B} =_{\mathbb{A}} \mathfrak{B}'$, $\pi_{\mathfrak{C}} \vdash \mathfrak{C} =_{\mathbb{A}} \mathfrak{C}'$, and

$$
\begin{aligned}
(A' \Rightarrow A, P_A) &= \mathbb{C}_{\mathtt{exq},\pi_{\mathfrak{A}}^{-1},\mathfrak{A}'} \\
(B \Rightarrow B', P_B) &= \mathbb{C}_{\mathtt{exi},\pi_{\mathfrak{B}},\mathfrak{B}} \\
(C \Rightarrow C', P_C) &= \mathbb{C}_{\mathtt{cci},\pi_{\mathfrak{C}},\mathfrak{C}}, \text{ where} \\
&\qquad \mathtt{exi} \triangleq \mathtt{o}, 3 \leftarrow \mathtt{get}\ \mathtt{o};\ 1 \leftarrow \mathtt{new}\ 1, \mathtt{o} \\
&\qquad \mathtt{exq} \triangleq \varnothing, \mathtt{o} \leftarrow \mathtt{get}\ \mathtt{o};\ 1 \leftarrow \mathtt{new}\ 1, 3
\end{aligned}
$$

Then the game composition operator $K_{\mathfrak{A},\mathfrak{B},\mathfrak{C}}$ is:

$$K_{\mathfrak{A},\mathfrak{B},\mathfrak{C}} \triangleq ((A \Rightarrow B) \otimes (B' \Rightarrow C) \Rightarrow (A' \Rightarrow C'), P_A \cup P_B \cup P_C).$$

Using the game composition operator K we can define GAM net composition using **HRAMnet** compact closed combinators. Let $f : \mathfrak{A} \Rightarrow \mathfrak{B}, g : \mathfrak{B} \Rightarrow \mathfrak{C}$ be GAM nets. Then their composition is defined as

$$
\begin{aligned}
&f ;_{GAM} g \triangleq \Lambda_A^{-1}(\Lambda_A(f) \otimes \Lambda_B(g)); K_{\mathfrak{A},\mathfrak{B},\mathfrak{C}})), \text{where} \\
&\Lambda_A(f : A \to B) \triangleq (\eta_A; (id_{A^*} \otimes f)) : I \to A^* \otimes B \\
&\Lambda_A^{-1}(f : I \to A^* \otimes B) \triangleq ((id_A \otimes f); (\varepsilon_A \otimes id_B)) : A \to B.
\end{aligned}
$$

Composition is represented diagrammatically as in Figure 4.7. Note the comparison with the naive composition from Figure 4.4. HRAMs f and g are

not plugged in directly, although the interfaces match. Composition is mediated by the operator K, which preserves the locality of freshly generated names, exchanging non-local pointer names with local pointer names and storing the mapping between the two as copycat links, indicated diagrammatically by dashed lines in K.

Theorem 4.2.24. If $f : \mathfrak{A} \to \mathfrak{B}$ and $g : \mathfrak{B}' \to \mathfrak{C}$ are game nets such that $\pi_{\mathfrak{B}} \vdash \mathfrak{B} =_{\mathbb{A}} \mathfrak{B}'$, f implements $S_f \subseteq P_{\mathfrak{A} \Rightarrow \mathfrak{B}}$, and g implements $S_g \subseteq P_{\mathfrak{B}' \Rightarrow \mathfrak{C}}$, then $f ;_{GAM} g$ implements $(S_f ;_{\mathfrak{G}} S_g)$.

This follows from Lemma 4.2.26 and Lemma 4.2.27.

Definition 4.2.25. If s is a trace, h is a heap, $\mathfrak{A}$ is a game interface, and $\pi_{\mathbb{P}}$ is a permutation over $\mathbb{P}$, we say that h is an *extended copycat heap* for s over $\mathfrak{A}$ if and only if:

1. For every pending $\mathbf{P}$-question non-initial in $\mathfrak{A}$ in s, i.e. $(\mathbf{P}, (a, p, p', d)) \subseteq s$ $(a \in qst_{\mathfrak{A}} \setminus ini_{\mathfrak{A}})$, $h(p') = (\tilde{\pi}_{\mathbb{P}}(p'), \varnothing)$.

2. For every pending $\mathbf{P}$-question initial in $\mathfrak{A}$ in s and its justifying move, i.e. $(\mathbf{O}, (a_1, p_1, p, d_1)) :: s' :: (\mathbf{P}, (a_2, p, p_2, d_2)) \subseteq s$ $(a_2 \in ini_{\mathfrak{A}})$, $h(p_2) = (\tilde{\pi}_{\mathbb{P}}(p_2), \tilde{\pi}_{\mathbb{P}}(p_1))$.

Lemma 4.2.26. If $f : \mathfrak{A} \to \mathfrak{B}$ and $g : \mathfrak{B}' \to \mathfrak{C}$ are game nets such that $\pi_{\mathfrak{B}} \vdash \mathfrak{B} =_{\mathbb{A}} \mathfrak{B}'$, f implements $S_f \subseteq P_{\mathfrak{A} \Rightarrow \mathfrak{B}}$, and g implements $S_g \subseteq P_{\mathfrak{B}' \Rightarrow \mathfrak{C}}$, then $(S_f ;_{\mathfrak{G}} S_g)^{st,alt} \subseteq_{\mathbb{AP}} [\![f ;_{GAM} g]\!] = [\![\Lambda_A^{-1}(\Lambda_A(f) \otimes \Lambda_{B'}(g); K_{\mathfrak{A},\mathfrak{B},\mathfrak{C}})]\!]$.

Lemma 4.2.27. If $f : \mathfrak{A} \to \mathfrak{B}$ and $g : \mathfrak{B}' \to \mathfrak{C}$ are game nets such that $\pi_{\mathfrak{B}} \vdash \mathfrak{B} =_{\mathbb{A}} \mathfrak{B}'$, f implements $S_f \subseteq P_{\mathfrak{A} \Rightarrow \mathfrak{B}}$, and g implements $S_g \subseteq P_{\mathfrak{B}' \Rightarrow \mathfrak{C}}$, then $[\![(f ;_{GAM} g)]\!]$ is $\mathbf{P}$-closed with respect to $(S_f ;_{\mathfrak{G}} S_g)$.

4.2.5 *Diagonal*

For game interfaces $\mathfrak{A}_1, \mathfrak{A}_2, \mathfrak{A}_3$ and permutations π_{ij} such that $\pi_{ij} \vdash \mathfrak{A}_i =_{\mathbb{A}} \mathfrak{A}_j$ for $i \neq j \in \{1, 2, 3\}$, we define the family of diagonal engines as:

$$\delta_{\pi_{12}, \pi_{13}, \mathfrak{A}} = (A_1 \Rightarrow A_2 \otimes A_3, P_1 \otimes P_2 \otimes P_3)$$

where, for $i \in \{2,3\}$,

$$
\begin{aligned}
P_1 \triangleq\ & \{q_1 \mapsto \texttt{ccq};\ \texttt{ifzero}\ 3\ (\texttt{spark}\ q_2)\ (\texttt{spark}\ q_3) \\
& \quad \mid q_1 \in opp_{\mathfrak{A}_1} \cap qst_{\mathfrak{A}_1} \wedge q_2 = \pi_{12}(q_1) \wedge q_3 = \pi_{13}(q_1)\} \\
& \cup \{a_1 \mapsto \texttt{cca};\ \texttt{ifzero}\ 3\ (\texttt{spark}\ a_2)\ (\texttt{spark}\ a_3) \\
& \quad \mid a_1 \in opp_{\mathfrak{A}_1} \cap ans_{\mathfrak{A}_1} \wedge a_2 = \pi_{12}(a_1) \wedge a_3 = \pi_{13}(a_1)\} \\
P_i \triangleq\ & \{q_i \mapsto 3 \leftarrow \texttt{set}\ (i-2);\ \texttt{cci};\ \texttt{spark}\ q_1 \mid q_i \in ini_{\mathfrak{A}_i} \wedge q_1 = \pi_{1i}^{-1}(q_i)\} \\
& \cup \{q_i \mapsto \texttt{ccq};\ \texttt{spark}\ q_1 \mid q_i \in (opp_{\mathfrak{A}_i} \cap qst_{\mathfrak{A}_i}) \setminus ini_{\mathfrak{A}_i} \wedge q_1 = \pi_{1i}^{-1}(q_i)\} \\
& \cup \{a_i \mapsto \texttt{cca};\ \texttt{spark}\ a_1 \mid a_i \in opp_{\mathfrak{A}_i} \cap ans_{\mathfrak{A}_i} \wedge a_1 = \pi_{1i}^{-1}(a_i)\}.
\end{aligned}
$$

The diagonal is almost identical to the copycat, except that an integer value of 0 or 1 is associated, in the heap, with the name of each message arriving on the A_2 and A_3 interfaces (hence the `set` instructions, to be used for routing back messages arriving on A_1 using `ifzero` instructions). By abuse of notation, we also write δ for the net $singleton(\delta)$.

Lemma 4.2.28. The δ net is the diagonal net, i.e. $[\![\delta_{\pi_{12},\pi_{23},\mathfrak{A}}; \Pi_i]\!] = [\![\mathbb{C}_{\pi_i,\mathfrak{A}}]\!]$.

4.2.6 *Fixpoint*

We define a family of GAMs $Fix_{\mathfrak{A}}$ with interfaces $(\mathfrak{A}_1 \Rightarrow \mathfrak{A}_2) \Rightarrow \mathfrak{A}_3$ where there exist permutations $\pi_{i,j}$ such that $\pi_{i,j} \vdash \mathfrak{A}_i =_{\mathbb{A}} \mathfrak{A}_j$ for $i \neq j \in \{1,2,3\}$. The fixpoint engine is defined as $Fix_{\pi_{12},\pi_{13},\mathfrak{A}} = \Lambda_A^{-1}(\delta_{\pi_{12},\pi_{13},\mathfrak{A}})$.

Let $fix_{\pi_{12},\pi_{13},\mathfrak{A}} : (\mathfrak{A} \Rightarrow \pi_{12} \cdot \mathfrak{A}) \Rightarrow \pi_{13} \cdot \mathfrak{A}$ be the game semantic strategy for fixpoint in Hyland-Ong games [77, p. 364].

Theorem 4.2.29. $Fix_{\pi_{12},\pi_{13},\mathfrak{A}}$ implements $fix_{\pi_{12},\pi_{13},\mathfrak{A}}$.

The proof of this is immediate considering the three cases of moves from the definition of the game semantic strategy. It is interesting to note here that we "*force*" a HRAM with interface $A_1 \Rightarrow A_2 \otimes A_3$ into a GAM with game interface $(\mathfrak{A}_3 \Rightarrow \mathfrak{A}_1) \Rightarrow \mathfrak{A}_2$, which has underlying interface $(A_3 \Rightarrow A_1) \Rightarrow A_2$. In the **HRAMnet** category, which is symmetric compact-closed, the two interfaces are isomorphic (with $A_1^* \otimes A_2 \otimes A_3$), but as game interfaces they are not. It is rather surprising that we can reuse our diagonal GAMs in such brutal fashion: in the game interface for fixpoint there is a reversed enabling relation between A_3 and A_1. The reason why this still leads to legal plays only is because the onus of producing the justification pointers in the initial move for A_3 lies with the opponent, which cannot exploit the fact that the diagonal is "wired illegally". It only sees the fixpoint interface and must play accordingly. It is fair to say that that fixpoint interface is more restrictive to the opponent than the

diagonal interface, because the diagonal interface allows extra behaviours, e.g. sending initial messages in A_3, which are no longer legal.

4.2.7 *Other ICA constants*

A GAM net for an integer literal n can be defined using the following engine (whose interface corresponds to the ICA exp type).

$$lit_n \triangleq (\{(\mathbf{O}, q), (\mathbf{P}, a)\}, P), \text{ where}$$
$$P \triangleq \{q \mapsto \texttt{swap}\ 0, 1;\ 1 \leftarrow \texttt{set}\ \varnothing;\ 2 \leftarrow \texttt{set}\ n;\ \texttt{spark}\ a\}$$

We see that upon getting an input question on port q, this engine will respond with a legal answer containing n as its value (register 2).

The conditional at type exp can be defined using the following engine, with the convention that $\{(\mathbf{O}, q_i), (\mathbf{P}, a_i)\} = \mathsf{exp}_i$.

$$\mathit{if} \triangleq (\mathsf{exp}_1 \Rightarrow \mathsf{exp}_2 \Rightarrow \mathsf{exp}_3 \Rightarrow \mathsf{exp}_4, P), \text{ where}$$
$$\begin{aligned} P \triangleq \{&q_4 \mapsto \texttt{cci};\ \texttt{spark}\ q_1,\\ &a_1 \mapsto \texttt{cca};\ \texttt{swap}\ 0, 1;\ \texttt{cci};\ \texttt{ifzero}\ 2\ (\texttt{spark}\ q_3)\ (\texttt{spark}\ q_2),\\ &a_2 \mapsto \texttt{cca};\ \texttt{spark}\ a_4,\\ &a_3 \mapsto \texttt{cca};\ \texttt{spark}\ a_4\} \end{aligned}$$

We can also define primitive operations, e.g. $+ : \mathsf{exp} \Rightarrow \mathsf{exp} \Rightarrow \mathsf{exp}$, in a similar manner. An interesting engine is that for *newvar*:

$$\mathit{newvar} \triangleq ((\mathsf{exp}_1 \otimes (\mathsf{exp}_2 \Rightarrow \mathsf{com}_3) \Rightarrow \mathsf{exp}_4) \Rightarrow \mathsf{exp}_5, P)$$
$$\begin{aligned} P \triangleq \{&q_5 \mapsto 3 \leftarrow \texttt{set}\ 0;\ \texttt{cci};\ \texttt{spark}\ q_4,\\ &q_1 \mapsto \varnothing, 2 \leftarrow \texttt{get}\ 0;\ \texttt{swap}\ 0, 1;\ 1 \leftarrow \texttt{set}\ \varnothing;\ \texttt{spark}\ a_1,\\ &q_3 \mapsto \texttt{swap}\ 0, 1;\ 1 \leftarrow \texttt{new}\ 0, 1;\ \texttt{spark}\ q_2,\\ &a_2 \mapsto \varnothing, 3 \leftarrow \texttt{get}\ 0;\ \texttt{update}\ 3\ 2;\ \texttt{cca};\ \texttt{spark}\ a_3,\\ &a_4 \mapsto \texttt{cca};\ \texttt{spark}\ a_5\} \end{aligned}$$

We see that we store the variable in the second component of the justification pointer that justifies q_4, so that it can be accessed in subsequent requests. A slight problem is that moves in exp_2 will actually not be justified by this pointer which we remedy in the q_3 case, by storing a pointer to the pointer with the variable in the second component of the justifier of q_2, which means that we can access and update the variable in a_2.

We can easily extend the HRAMs with new instructions to interpret parallel execution and semaphores, but we omit them from the current presentation, since parallelism is not our focus.

4.3 SEAMLESS DISTRIBUTED COMPILATION FOR ICA

4.3.1 *The language ICA*

ICA is PCF extended with constants to facilitate local effects. Its ground types are expressions and commands (exp, com), with the type of assignable variables desugared as $\mathsf{var} \triangleq \mathsf{exp} \times (\mathsf{exp} \to \mathsf{com})$. Dereferencing and assignment are desugared as the first and second projections from the type of assignable variables. The local variable binder is $\mathsf{new} : (\mathsf{var} \to \mathsf{com}) \to \mathsf{com}$. ICA also has a type of split binary semaphores $\mathsf{sem} \triangleq \mathsf{com} \times \mathsf{com}$, with the first and second projections corresponding to set, get, respectively (see [55] for the full definition, including the game semantic model).

In this section we give a compilation method for ICA into GAM nets. The compilation is compositional on the syntax and uses the constructs of the previous section. ICA types are compiled into GAM interfaces which correspond to their game semantic arenas in the obvious way. We will use $A, B, \ldots$ to refer to an ICA type and to the GAM interface. Section 4.2 has already developed all the infrastructure needed to interpret the constants of ICA (Section 4.2.7), including fixpoint (Section 4.2.6). Given an ICA type judgment $\Gamma \vdash M : A$ with Γ a list of variable-type assignments $x_i : A_i$, M a term and A a type, a GAM G_M implementing it is defined compositionally on the syntax as follows:

$$
\begin{aligned}
G_{\Gamma \vdash MM' : A} &= \delta_{\pi_1, \pi_2, \Gamma} ;_{GAM} \left(G_{\Gamma \vdash M : A \to B} \otimes G_{\Gamma \vdash M' : B} \right) ;_{GAM} eval_{A,B} \\
G_{\Gamma \vdash \lambda x : A . M : A \to B} &= \Lambda_A \left(G_{\Gamma, x : A \vdash M : B} \right) \\
G_{x : A, \Gamma \vdash x : A} &= \Pi_{\mathfrak{G}A} ; \mathbb{C}_{A,\pi},
\end{aligned}
$$

Where $eval_{A,B} \triangleq \Lambda_B^{-1}(\mathbb{C}_{A \Rightarrow B, \pi})$ for a suitably chosen port renaming π and where $\Pi_{\mathfrak{G}A}$, $\Pi_{\mathfrak{G}1}$, and $\Pi_{\mathfrak{G}2}$ are HRAMs with signatures $\Pi_{\mathfrak{G}i} = (A_1 \otimes A_2 \Rightarrow A_3, P_i)$ such that they copycat between A_3 and A_i and ignore $A_{j \neq i}$. The interpretation of function application, which is the most complex, is shown diagrammatically in Figure 4.8. The copycat connections are shown using dashed lines.

Theorem 4.3.1. If M is an ICA term, G_M is the GAM implementing it and σ_M its game semantic strategy then G_M implements σ_M.

The correctness of compilation follows directly from the correctness of the individual GAM nets and the correctness of GAM composition $;_{GAM}$.

4.3.2 *Prototype implementation*

Following the recipe in the previous section we can produce an implementation of any ICA term as a GAM net. GAMs are just special purpose HRAMs,

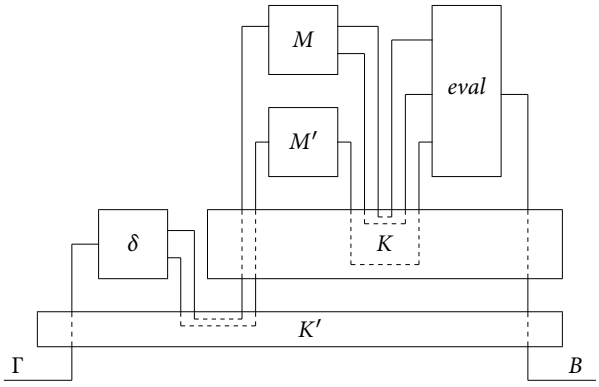


Figure 4.8: GAM net for application

This net uses two composition GAMs, one eval, and one diagonal.

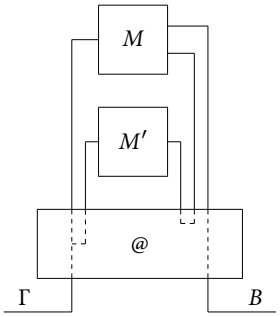


Figure 4.9: Optimised GAM net for application

The functionality of the two compositions, the diagonal, and the eval GAMs from the naive implementation of application have been combined and optimised into a single GAM, requiring only one pointer renaming before reaching M.

with no special operations. HRAMs, in turn, can easily be implemented on any conventional computer with the usual store, control and communication facilities. A GAM net is also just a special purpose HRAM net, which is a powerful abstraction of communication processes, as it subsumes through the `spark` instruction communication between processes (threads) on the same physical machine or located on distinct physical machines and communicating via a point-to-point network. We have built a prototype compiler[2] based on GAMs by implementing them in C, managing processes using standard UNIX threads and physical network distribution using MPI [65].

The actual distribution is achieved using light pragma-like code annotations. In order to execute a program at node A but delegate one computation to node B and another computation to node C we simply annotate an ICA program with node names, e.g.:

$$\{\mathsf{new}\ x.x := \{f(x)\}@B + \{g(x)\}@C; !x\}@A$$

Note that this gives node B, via function f, read-write access to memory location x which is located at node A. Accessing non-local resources is possible, albeit possibly expensive.

Several facts make the compilation process quite remarkable:

- It is *seamless* (in the sense of [46]), allowing distributed compilation where communication is never explicit but always realised through function calls.
- It is *flexible*, allowing any syntactic subterm to be located at any designated physical location, with no impact on the semantics of the program. The access of *non-local* resources is always possible, albeit possibly at a cost (latency, bandwidth, etc.).
- It does not require any form of *garbage collection*, even on local nodes, although the language combines (ground) state, higher-order functions and concurrency. This is because a pointer associated with a pointer is not needed if and only if the question is answered; then it can be safely deallocated.

The current implementation performs few optimisations, and the resulting code is inefficient. Looking at the implementation of application in Figure 4.8 it is quite clear that a message entering the GAM net via port A needs to undergo four pointer renamings before reaching the GAM for M. This is the cost we pay for compositionality. However, the particular configuration

[2]Online appendix: `games` directory.

for application can be significantly simplified using standard peephole optimisation, and we can reach the much simpler, still correct implementation in Figure 4.9. Here the functionality of the two compositions, the diagonal, and the *eval* GAMs have been combined and optimised into a single GAM, requiring only one pointer renaming before reaching M. Other optimisations can be introduced to simplify GAM nets, in particular to obviate the need for the use of composition GAMs K, for example by the observation that composition of closed first-order terms (such as those used for most constants) can be done directly.

4.4 RELATED WORK

Section 3.5 outlined some pieces of work relevant to the GOI interpretation of terms, which is quite closely related also to the games interpretation. In this section we outline work related to the interpretation of programming languages using interaction semantics (other than GOI). For a presentation of work that aims to solve problems similar to ours, see Section 2.1.

4.4.1 *Games*

Our work is based on game semantics [77], which is a denotational semantics for programming languages well-known for achieving full abstraction for PCF. Varying the conditions on the allowed strategies — the interpretation of a program — in the models allows achieving definability — the ability to construct terms from elements in the model — results for many different language constructs. It is thus possible to pick a model that fits the language that you are trying to model [30].

From the perspective of distributed computing, what is striking is the encoding of function application (much like GOI), which becomes an interaction between the two involved parties (i.e. function and argument), which suggested a way to automatically infer the needed communication of a program running in such a system from its encoding as a strategy.

To be able to sufficiently distinguish certain programs (of order higher than two; it has been shown that they are not needed for second order programs, whose semantics are regular [54]) a notion of justification pointers was introduced, providing information about why a move is justified. The initial presentation used sequences of numbers to indicate the pointer structure [77], which led to a certain amount of bureaucracy in formally defining the different operations on move sequences that are required in any presentation of games. Our work is based on nominal games [48], which ameliorates the bureaucracy by using *names* to represent the pointing structures of the sequences.

4.4.2 *Process calculi*

The idea of reducing computation to interaction also appeared in the context of process calculi [106, 107]. This was a development largely independent of game semantics and GOI which happened around the same time. The π-calculus, introduced by Milner, Parrow, and Walker [107], described in depth by e.g. Sangiorgi and Walker [122], is a calculus for describing concurrent systems and their behaviour.

Pict [114] is a programming language for concurrent systems that aims to be to π-calculus what Haskell or ML is to λ-calculus, meaning that it adds some new features and syntactic sugar to make it more programmer-friendly. Note that the methodology is different to our approach where we aim to make communication implicit — here the communication is like in approaches using message passing. In his doctoral thesis, Turner [135, chapters 7-9] gives an abstract machine for Pict and a proof that the reduction it does is a valid reduction. Furthermore, its compilation into to the C programming language is described. While this is an impressive feat that is even fairly performant (within an order of magnitude of the equivalent ML program according to the source), it is only for running the programs *locally on one machine*, and it does not perform the reduction in parallel — only concurrently (or interleaved). It is therefore inadequate for distributed systems in its current form.

Gardner, Laneve, and Wischik [50] makes the problem of using π-calculus in a distributed setting clearer. Consider a term like $P = x(y).y(z).Q$, a process that receives a channel y along x and then listens on y. The problem this poses in a distributed system is that when another process wants to send on y, it needs to know where y is. It seems impossible to do this without either using broadcasting or a central server keeping track of where every channel is. One "solution" would be to disallow terms like this, but in this paper the problem is solved by using *linear forwarders*, which act as proxies, forwarding messages to the right location.

4.4.3 *Hardware synthesis*

Sutherland [127] describes the architecture of hardware circuits that run asynchronously instead of the more typical clock-based, synchronous designs. This is one of the ideas behind the Geometry of Synthesis series [53, 58, 59, 60]. The set of problems that one might face in hardware are not necessarily the same as the ones in distributed computing, but some of the ideas of event-driven circuits carry over to distributed computing if we take the view that a circuit's component is a node in the distributed system and an event on a wire is a message in the network. As a comparison, hardware components generally have

limited functionality whereas in distributed computing there is often few restrictions to what each node can do. In the Geometry of Synthesis it is shown how to compile a general-purpose programming language into hardware circuits. The main idea is inspired by game semantics in that function application is reduced to interaction between circuits (when the result of a function is requested, the function can request its argument) and that the moves that a circuit can play are known in advance (based on the type). Evaluation is based on local reduction rules, which is suitable for compilation into circuits. These papers shows that it is possible to compile general-purpose programming languages to targets that are traditionally programmed using *domain-specific* languages.

4.4.4 *Algol-like languages*

The Algol family are well-studied languages [120] with both functional and imperative features, but with only *local*, ground-type state, which circumvents implementation issues such as the *funargs* problem [138] and thus does not always require garbage collection. This compromise, sacrificing some expressivity for not having to do distributed garbage collection (for which, as concluded by Abdullahi and Ringwood [2], there is no solution that is satisfactory in all regards), is therefore suitable for distributed computing.

However, there are subtleties in mixing imperative, stateful constructs and functional constructs in a call-by-name (or -reference) language. Reynolds [119] (later improved by O'Hearn et al. [112]) calls this interference, which can for instance be aliasing of procedure arguments: if f is a binary procedure that mutates its arguments, calling it with the same argument, as in $f(x, x)$ can yield (perhaps) unexpected results. It might also be procedures running in parallel with interfering side-effects. The solution presented is called *Syntactic Control of Interference*, and as the name suggests, the potential for interference is something that can be disallowed syntactically, by a type checker. Disallowing interference goes in two ways: two terms are non-interfering if they do not share any active identifiers ($f(x, x)$ is not permitted, as in affine linear logic). Also, if two terms are passive (or pure), then they are non-interfering.

We use Idealised Concurrent Algol (ICA) [55] as our source language. Ghica, Murawski, and Ong [56] show a way to achieve decidable may-equivalence in ICA, through a type system that keeps track of and puts bounds on concurrency. Ghica and Smith [58] later use it in their Geometry of Synthesis series for compiling concurrent programs into static hardware and also show how it can be translated to Syntactic Control of Interference, as already mentioned.

These developments, which deal with disallowing unsafe programs and adding resource constraints to the programs through the type system could

also be applied to distributed computing. Interference is as much of a problem there as in single-computer programs, and concurrency bounds could be useful, if dealing with a system with constrained resources like a GPU (or even interfacing with a hardware circuit on a Field-Programmable Gate Array).

4.5 CONCLUSION

In this chapter we have seen how game semantics can be expressed operationally using abstract machines very similar to networked conventional computers. We believe that many of the programming languages with a semantic model expressed as Hyland-Ong-style pointer games [77] can be represented using GAMs and then compiled to a variety of platforms such as MPI. However, if the language includes sum types — which seems to require answers that justify questions [103] — we might not be able to get by without a garbage collector as we have done here.

Like the GOI compiler, this compilation model provides freedom in choosing the location at which a computation takes place. It additionally has support for language features like local state and mutable references.

Benchmarks are given later, in Section 7.8, but we make some remarks about the performance here. Even with the optimised implementation of application shown in Figure 4.9, single-node programs compiled using the GAM formalism are roughly 4 to 8 times slower than those compiled with a naive implementation of the Krivine machine [86]. This is significantly better than the performance of the GOI compiler, but still unsatisfactory. The performance problems stem from the excessive heap pointer manipulation required in almost all HRAMs, and from having to use contraction for variables not used linearly — just like in the GOI compiler. The heap pointer manipulation, which perhaps does not look that bad on paper, may lead to CPU cache thrashing. The good news is that the distributed programs communicate using messages of fixed size — a significant improvement over our previous compiler.

4.6 DISCUSSION

In the introduction of this thesis (Chapter 1) we argued that distributed computing would benefit from the existence of architecture-independent, seamless compilation methods for conventional programming languages which can allow the programmer to focus on solving algorithmic problems without being overwhelmed by the minutiae of driving complex computational systems. More concretely, our proposition was to generalise RPCs to also handle higher-order functions across node boundaries.

This part of the dissertation has given two compilation schemes for higher-order RPCs based on interaction semantics. They both achieve seamless distribution but have certain apparent unavoidable inefficiencies. The compiler based on the Geometry of Interaction (Chapter 3) has a possibly insurmountable communication overhead, whereas the compiler based on game semantics (Chapter 4) communicates efficiently but requires a very high computational overhead on each node. These inefficiencies stem from their being inspired by denotational semantics where efficiency of execution is not the top priority. Another problem that basing them on denotational semantics gives rise to is that they are exotic: it would be difficult to use them as the basis for adding higher-order RPCs to an existing compiler since most of the compiler would have to be changed. Exoticness can also mean that it is hard to adapt certain conventional compiler optimisation techniques, which further exacerbates the efficiency problem.

Variation II

CONVENTIONAL ABSTRACT MACHINES

Chapter 5

Source language

The following two chapters will use the same source language, which is presented below.[1]

We will be compiling the untyped applied lambda calculus, i.e. an untyped Programming Computable Functions (PCF), in two different ways. We will technically compile two different languages since one will implement call-by-name and one call-by-value, but they will share the same syntax.

For the sake of a concrete yet simple presentation we assume that the only data is natural numbers, and the constants are numeric literals, arithmetic operators and if-then-else. Informally, the grammar of the language is

$$\begin{aligned} M ::= {} & x \mid \lambda x.M \mid M\ M \mid \mathsf{if}\ M\ \mathsf{then}\ M\ \mathsf{else}\ M \mid n \\ & \mid M \oplus M \mid M@A. \end{aligned}$$

Formally, we define the data type of *terms* with the following constructors:

```
data Term : Set where
  λ_   : Term → Term
  _$_  : (t t' : Term) → Term
  var  : ℕ → Term
  lit  : ℕ → Term
  op   : (f : ℕ → ℕ → ℕ) (t t' : Term) → Term
  ifo_then_else_ : (b t f : Term) → Term
  _@_  : Term → Node → Term
```

Above, *Set* is the previously mentioned type of types — signifying that we are defining a new type. The constructor for function application has an explicit name (`_$_`), for clarity. We use the De Bruijn index notation [20] to represent variables, so abstraction (`λ_`) is a unary operator and each variable (`var`) is a natural number. The value of the index denotes the number of binders between the variable and its binder. Note that we do not make use of the by now standard technique of representing lambda terms as an inductive family of data types indexed by context (or number of free variables) [6] which can be used to ensure that it is only possible to construct well-typed (or closed) terms. This is partly because our source language is untyped, and partly because it

[1] Online appendix: `krivine/formalisation` directory, `Lambda` module.

would needlessly complicate the presentation without making our life easier — we would have to add additional indices to every function on terms. Instead, we will permit the abstract machines to get stuck if something goes wrong (e.g. when the types do not match at run time or when encountering a free variable), noting that this would not happen in a typical implementation [105] since its frontend would include a type-checker.

Example 5.0.1. The term $(\lambda x.\lambda y.y + x)\,3\,4$ is represented as

```
termExample : Term
termExample = λ (λ (var 0 +' var 1)) $ lit 3 $ lit 4
  where _+'_ = op _+_
```

Numeric literals (**lit**) and conditionals (**ifo_then_else_**) are obvious, noting that the constructor for the latter is a *mixfix operator*. Binary arithmetic operators (**op**) take three arguments: the function giving the operation and two terms.

We also introduce syntactic support in the language (_@_, another infix operator) for specifying the location for *closed* subterms. This is the same as the location annotations that we saw Chapter 3 and Chapter 4 with the difference that it is here limited to closed subterms. Node assignment is a "compiler pragma" and will have no bearing on observational properties of the programming language. The requirement that node assignment is specified for closed terms keeps the presentation as simple as possible. This apparent restriction can easily be overcome using lambda lifting, i.e. by transforming every open subterm t @ A to $(\lambda\, fv\ t.\ t)$ @ A $(fv\ t)$ at compile time.

5.1 SEMANTICS

We do not give an operational semantics for the language here; it is standard and can be found elsewhere — see [116] for small-step and [84] for big-step semantics. We rely instead on conventional abstract machines with already established correctness results as our specification. For the correctness of the Krivine machine, we refer the reader to Krivine [86] who gives such a proof for the lambda calculus fragment of our language. For the correctness of the extensions required in the applied language, Leroy [97] treats strict evaluation and Hannan and Miller [68] treat conditionals and primitive operations by giving a derivation of the machine from a call-by-name big-step semantics. Danvy and Millikin [33] present a similar derivation and correctness proof for the Stack-Environment-Control-Dump (SECD) machine and several variations thereof.

Chapter 6

The Krivine machine

In this part of the thesis we present a solution to problems with the approaches based on interaction semantics that we outlined in Section 4.6. Instead of building wholly new abstract machines we take *conventional* abstract machines and make conservative extensions to them to support distributed execution. This new approach combines the best of the two previous compilers: it communicates efficiently by keeping the size of the messages within a small fixed bound and it executes efficiently on each node. In fact, the compilation scheme degenerates to that of the conventional abstract machines if the whole program is deployed on a single node. An additional advantage that basing the work on conventional machines offers is that, unlike the exotic Geometry of Interaction (GOI) and games-based approaches, it is a standard approach to compiler construction so that common optimisation techniques more readily apply to it and so that it can interface trivially with legacy code which was compiled to the abstract machine in question. We perform this act of making conservative extensions to the quintessential abstract machines for call-by-name and call-by-value: the Krivine machine [86] and the SECD machine [93]. By using the SECD machine we thus additionally extend our previous work also by exploring the usage of the call-by-value calling convention. Note that, while the call-by-value evaluation order subsumes call-by-name in the sense that we can simulate call-by-name in a call-by-value language by using thunking, a direct implementation has the potential to be more efficient and instructive than such a simulation. Our extensions are called the DKrivine machine (presented in this chapter) and the DCESH machine (Chapter 7). Finally, we model a general-purpose fault-tolerant environment for machines similar to the DKrivine and the DCESH machine (Chapter 8) by adding a layer consisting of a transactional machine that provides a simple commit-and-rollback mechanism for underlying abstract machines that may unexpectedly fail.

Synopsis In this chapter we define a new approach to compilation to distributed architectures based on networks of abstract machines. Using it we can implement a generalised and fully transparent form of Remote Procedure Call that supports calling higher-order functions across node boundaries, without sending actual code. Our starting point is the classic Krivine machine, which implements reduction for untyped call-by-name PCF. We successively add the

features that we need for distributed execution and show the correctness of each addition. Our final system, the *Krivine net*, is shown to be a correct distributed implementation of the Krivine machine, preserving both termination and non-termination properties. We also implement a prototype compiler which we later compare (Section 7.8) with our previous distributing compilers based on Girard's GOI (Chapter 3) and on game semantics (Chapter 4).

6.1 THE MACHINE

The Krivine machine [86] is the standard abstract machine for call-by-name.[1] It has three components: code, environment, and stack. The stack and the environment contain *thunks*, which are closures representing unevaluated function arguments. The evaluations are delayed until the values are needed. For the pure lambda calculus, the Krivine machine uses three instructions:

POPARG pop an argument from the stack and add it to the environment.

PUSHARG push a thunk for some code given as argument.

VAR look up the argument in the environment and start evaluation.

For the applied lambda calculus the machine becomes more complex; arithmetic operations are strict, so additional mechanisms are required to force the evaluation of arguments.

In Agda, we define closures and environments by mutual induction:

```
mutual
  Closure = Term × Env
  data EnvEl : Set where
    clos : Closure → EnvEl
  Env = List EnvEl
```

The constructor **clos** that takes a *Closure* into an environment element *EnvEl* is needed for formal reasons, to prevent the Agda type-checker from reporting a circular definition.

Stacks and configurations are:

[1] Online appendix: `krivine/formalisation` directory, `Krivine` module.

```
data StackElem : Set where
  arg : Closure                        → StackElem
  ifo : Closure          → Closure →   StackElem
  op₂ : (ℕ → ℕ → ℕ) → Closure →       StackElem
  op₁ : (ℕ → ℕ)                        → StackElem
Stack  = List StackElem
Config = Term × Env × Stack
```

Stack elements represent the evaluation context, i.e. the current continuation. The context for application, commonly written $-\ t$, is constructed using **arg**, whereas **ifo**, $\mathbf{op_2}$, $\mathbf{op_1}$ are used by the constants.

The signature of the Krivine machine is given as a data type, defining a *Rel*ation on *Config*urations of the Krivine machine:

```
data _⟶ₖ_ : Rel Config Config where
```

We define the relation type *Rel A B* to be $A \to B \to Set$, so two elements a and b are R-related exactly when $R\ a\ b$ is inhabited, given $R : Rel\ A\ B$. Each rule, i.e. each instruction of the machine, will thus correspond to a constructor. We explain the definition of each rule.

```
POPARG : {t : Term} {e : Env} {c : Closure} {s : Stack} →
  (ƛ t , e , arg c :: s)  ⟶ₖ  (t , clos c :: e , s)
```

POPARG handles abstractions $\boldsymbol{\lambda}\ t$ by moving the top of the stack **arg** c into the first position of the environment e. The constructors **arg**, **clos** are needed for type-checking and would be omitted in an informal presentation. The constructor arguments (t, e, c, s) are implicit, indicated syntactically in Agda by curly braces.

```
PUSHARG : {t t' : Term} {e : Env} {s : Stack} →
  ((t $ t') , e , s)  ⟶ₖ  (t , e , arg (t' , e) :: s)
```

PUSHARG handles application $t\ \$\ t'$ by creating a new closure **arg** (t', e) and pushing it onto the stack, then carrying on with the execution of the function body t.

```
VAR : {n : ℕ} {e e' : Env} {t : Term} {s : Stack} →
  lookup n e ≡ just (clos (t , e')) →
  (var n , e , s)  ⟶ₖ  (t , e' , s)
```

The **VAR** rule looks up the variable n in the current environment e and, if successful, retrieves the closure at that position (t, e') and proceeds to execute from

it, with the current stack. In Agda the ≡ operator denotes *propositional* equality, which necessitates a proof, whereas = is used to introduce new definitions.

Because this is an applied lambda calculus we need additional operations for conditionals and operators. Here we omit the types of the implicit arguments since they can be inferred:

$$
\begin{aligned}
&\mathbf{COND} : \forall \{b\ t\ f\ e\ s\} \rightarrow \\
&\quad (\mathbf{ifo}\ b\ \mathbf{then}\ t\ \mathbf{else}\ f\ ,\ e\ ,\ s) \longrightarrow_{\mathcal{K}} (b\ ,\ e\ ,\ \mathbf{ifo}\ (t\ ,\ e)\ (f\ ,\ e) :: s) \\
&\mathbf{COND\text{-}o} : \forall \{e\ t\ e'\ f\ s\} \rightarrow \\
&\quad (\mathbf{lit}\ o\ ,\ e\ ,\ \mathbf{ifo}\ (t\ ,\ e')\ f :: s) \longrightarrow_{\mathcal{K}} (t\ ,\ e'\ ,\ s) \\
&\mathbf{COND\text{-}suc} : \forall \{n\ e\ t\ f\ e'\ s\} \rightarrow \\
&\quad (\mathbf{lit}\ (1 +\ n)\ ,\ e\ ,\ \mathbf{ifo}\ t\ (f\ ,\ e') :: s) \longrightarrow_{\mathcal{K}} (f\ ,\ e'\ ,\ s) \\
&\mathbf{OP} : \forall \{f\ t\ t'\ e\ s\} \rightarrow \\
&\quad (\mathbf{op}\ f\ t\ t'\ ,\ e\ ,\ s) \longrightarrow_{\mathcal{K}} (t\ ,\ e\ ,\ \mathbf{op_2}\ f\ (t'\ ,\ e) :: s) \\
&\mathbf{OP_2} : \forall \{n\ e\ f\ t\ e'\ s\} \rightarrow \\
&\quad (\mathbf{lit}\ n\ ,\ e\ ,\ \mathbf{op_2}\ f\ (t\ ,\ e') :: s) \longrightarrow_{\mathcal{K}} (t\ ,\ e'\ ,\ \mathbf{op_1}\ (f\ n) :: s) \\
&\mathbf{OP_1} : \forall \{n\ e\ f\ s\} \rightarrow \\
&\quad (\mathbf{lit}\ n\ ,\ e\ ,\ \mathbf{op_1}\ f :: s) \longrightarrow_{\mathcal{K}} (\mathbf{lit}\ (f\ n)\ ,\ []\ ,\ s)
\end{aligned}
$$

Example 6.1.1. We can see the Krivine machine at work in Figure 6.1.[2] It shows the execution trace of the term in Example 5.0.1. For brevity it is written informally, omitting the constructors **op**, **var**, **arg**, etc.

Finally, we include a (degenerate) instruction for remote execution:

$$\mathbf{REMOTE} : \forall \{t\ i\ e\ s\} \rightarrow (t\ @\ i\ ,\ e\ ,\ s) \longrightarrow_{\mathcal{K}} (t\ ,\ []\ ,\ s)$$

This instruction is included strictly so that the _@_ construct for node assignment does not make the machine get stuck or trigger a run time error, but it is effectively a *no-op*: it simply erases the environment e, since node assignment is meant to be applied only to closed terms. In the following section we will define the distributed Krivine machine, where the **REMOTE** instruction is meaningful.

[2] Online appendix: `krivine/formalisation` directory, `Trace` module.

$((\boldsymbol{\lambda}\ (\boldsymbol{\lambda}\ \_+\_\ 0\ 1)\ \$\ 3\ \$\ 4)\ ,\ []\ ,\ [])$
$\quad\longrightarrow\langle\ \textbf{PUSHARG}\ \rangle$
$((\boldsymbol{\lambda}\ (\boldsymbol{\lambda}\ \_+\_\ 0\ 1)\ \$\ 3)\ ,\ []\ ,\ [\ (4\ ,\ [])\ ])$
$\quad\longrightarrow\langle\ \textbf{PUSHARG}\ \rangle$
$(\boldsymbol{\lambda}\ (\boldsymbol{\lambda}\ \_+\_\ 0\ 1)\ ,\ []\ ,\ [\ (3\ ,\ [])\ ,\ (4\ ,\ [])\ ])$
$\quad\longrightarrow\langle\ \textbf{POPARG}\ \rangle$
$(\boldsymbol{\lambda}\ \_+\_\ 0\ 1\ ,\ [\ (3\ ,[])\ ],\ [\ (4\ ,[])\ ])$
$\quad\longrightarrow\langle\ \textbf{POPARG}\ \rangle$
$(\_+\_\ 0\ 1\ ,\ [\ (4\ ,\ [])\ ,\ (3\ ,\ [])\ ]\ ,\ [])$
$\quad\longrightarrow\langle\ \textbf{OP}\ \rangle$
$(0\ ,\ [\ (4\ ,\ [])\ ,\ (3\ ,\ [])\ ]\ ,\ [op_2\ \_+\_\ (1\ ,\ [\ (4\ ,\ [])\ ,\ (3\ ,\ [])\ ])\ ])$
$\quad\longrightarrow\langle\ \textbf{VAR refl}\ \rangle$
$(4\ ,\ []\ ,\ [op_2\ \_+\_\ (1\ ,\ [\ (4\ ,\ [])\ ,\ (3\ ,\ [])\ ])\ ])$
$\quad\longrightarrow\langle\ \textbf{OP}_\textbf{2}\ \rangle$
$(1\ ,\ [\ (4\ ,\ [])\ ,\ (3\ ,\ [])\ ]\ ,\ [op_1\ (\_+\_\ 4)\ ])$
$\quad\longrightarrow\langle\ \textbf{VAR refl}\ \rangle$
$(3\ ,\ []\ ,\ [op_1\ (\_+\_\ 4)\ ])$
$\quad\longrightarrow\langle\ \textbf{OP}_\textbf{1}\ \rangle$
$(7\ ,\ []\ ,\ [])$

Figure 6.1: Example Krivine machine execution trace
The execution trace of the term $(\lambda x.\lambda y.y + x)\,3\,4$.

6.2 KRIVINE NETS

We now extend the Krivine machine so that it supports an arbitrary pattern of distribution by letting several instances of the extended machine run in a network. We call these machines *DKrivine* machines and they form *Krivine nets*.[3] The DKrivine machines extend the Krivine machines conservatively by adding new features. Each such machine is identified as a *node* in the network and has a dedicated heap. A pointer into a heap may be tagged with a node identifier, case in which it is a *remote pointer*, which can now be stored in the environment along with local closures. The stack may now have as a bottom element a remote pointer indicating the existence of a *remote stack extension*, i.e. the fact that the information which logically belongs to this stack is physically located on a different node. Finally, the configuration of the Krivine machine is now called a *thread* indicating that its execution can be dynamically started and halted. Internally, the heap structure is used for storing persistent data that needs to outlive the run time of a thread. The new definitions are as follows:

$$
\begin{aligned}
&\mathit{RPtr} \;=\; \mathit{Ptr} \times \mathit{Node}\\
&\mathit{ContPtr} \;=\; \mathit{RPtr}\\
&\mathsf{data}\ \mathit{EnvElem} \;:\; \mathit{Set}\ \mathsf{where}\\
&\quad \mathbf{local} \;:\; \mathit{Closure} \to \quad \mathit{EnvElem}\\
&\quad \mathbf{remote} \;:\; \mathit{ContPtr} \to \mathbb{N} \to \mathit{EnvElem}\\
&\mathit{Stack} \;=\; \mathit{List}\ \mathit{StackElem} \times \mathit{Maybe}\ (\mathit{ContPtr} \times \mathbb{N} \times \mathbb{N})\\
&\mathit{ContHeap} \;=\; \mathit{Heap}\ \mathit{Stack}\\
&\mathit{Thread} \;=\; \mathit{Term} \times \mathit{Env} \times \mathit{Stack}\\
&\mathit{Machine} \;=\; \mathit{Maybe}\ \mathit{Thread} \times \mathit{ContHeap}
\end{aligned}
$$

The definitions are straightforward, except for the **remote** environment element and the definition of stacks which require explanation. A remote *ContPtr* is a pointer to a continuation stack, and the constructor **remote** takes an additional natural number argument indicating the offset in that continuation stack where the referred closure is stored. As stated, the stack now possibly includes a remote stack extension. This extension is to be thought of as being located at the bottom of the local stack, and consists of a *ContPtr* pointing into the heap of a remote node holding the stack, and two natural numbers that form the current node's *view* of that stack. The second number is the offset into the remote stack that the view starts from, and the first number stores how many consecutive arguments there are on on it.

Because DKrivine machines are networked they exchange messages, which fall into three categories, formalised as constructors for the *Msg* data type:

[3] Online appendix: `krivine/formalisation` directory, `DKrivine` module.

REMOTE messages initiate remote evaluation, and are defined as:

$$\textbf{REMOTE} : \mathit{Term} \to \mathit{Node} \to \mathit{ContPtr} \to \mathbb{N} \to \mathit{Msg}$$

The message consists of a *Term*, a destination *Node* identifier, a *ContPtr* to the sender's current continuation stack and a natural number indicating how many arguments are on that stack.

The design decision to make a *Term* part of the message structure is for simplicity of formalisation only. In the actual implementation only a *code pointer* needs to be sent to the node, which already has the required code available. The mechanism through which compiled code arrives at each node is handled by a *distributed program loader* which is part of the runtime system and, as such, beyond the scope of this work. It should be obvious that distributed program loading is possible in principle here when all code is static and available at compile time.

RETURN messages are sent when computation has terminated and reached a literal, and the value must be returned to the node that has initiated the computation. The definition is:

$$\textbf{RETURN} : \mathit{ContPtr} \to \mathbb{N} \to \mathbb{N} \to \mathit{Msg}$$

The message contains a *ContPtr* to the remote stack of the machine that is receiving the message, the natural number calculated and another number indicating to the receiving machine how many arguments can now be discarded from the stack, corresponding to the offset in the sending node's view of the stack.

VAR is a message used to access remotely located variables. It consists of a remote *ContPtr*, an offset into the remote continuation stack, a local continuation stack and the number of arguments on it.

$$\textbf{VAR} : \mathit{ContPtr} \to \mathbb{N} \to \mathit{ContPtr} \to \mathbb{N} \to \mathit{Msg}$$

We need to send the continuation stack pointer of the calling node (like in the **REMOTE** rule) because the remote variable may refer to a function, in which case the arguments are supplied by the calling node, or it may be part of an operation on the calling node, in which case the resulting number needs to be returned there once it has been calculated.

Deliberate in the design of the Krivine nets is the need to minimise message exchange. To achieve this, machines do not send remote "pop" messages for manipulating remote stack extensions, but perform this operation locally. When

a node sends a pointer to a new continuation stack it also sends the number of arguments that are on that stack, so that the receiving node can pop arguments from its local view of that stack.

We can now start describing the transitions of the DKrivine machine. The signature of the transition relation is:

```
data _ ⊢ _⟶ 𝒟𝒦⟨_⟩_ (i : Node) :
  Machine → Tagged Msg → Machine → Set
```

DKrivine transitions are parameterised by the current node identifier and map a *Machine* state and a *Tagged Msg* into a new *Machine* state. The tag (see also Section 2.5) applied to the message indicates whether the message is sent, received or absent (i.e. a $\boldsymbol{\tau}$ transition):

```
data Tagged (Msg : Set) : Set where
  τ       : Tagged Msg
  send    : Msg → Tagged Msg
  receive : Msg → Tagged Msg
```

All the old rules are present, but now expressed in the presence of the continuation heap.

```
POPARG : ∀ {t e c s r ch} →
  i ⊢ (just (λ t , e , arg c :: s , r) , ch)  ⟶_𝒟𝒦 ⟨ τ ⟩
      (just (t , local c :: e , s , r) , ch)
```

Compared to the **POPARG** rule of the original machine, the only differences are the tag on the configuration (**just** ...), which expresses the fact that the DKrivine thread is running, and the continuation heap *ch* which remains constant during the application of this rule. The environment element constructor **local** now emphasises that the variable is local. Because the transition involves only one node it is $\boldsymbol{\tau}$, i.e. no messages are exchanged.

The other old transition rules are embedded into the DKrivine machine in a similar way. They are all silent and the continuation heap *ch* stays unchanged:

```
PUSHARG : ∀ {t t' e s r ch} →
  i ⊢ (just ((t $ t') , e , s , r) , ch) ⟶_DK ⟨ τ ⟩
      (just (t , e , arg (t' , e) :: s , r) , ch)
VAR : ∀ {n e s r ch t e'} →
  lookup n e ≡ just (local (t , e')) →
  i ⊢ (just (var n , e , s , r) , ch) ⟶_DK ⟨ τ ⟩
      (just (t , e' , s , r) , ch)
COND : ∀ {b t f e s r ch} →
  i ⊢ (just (ifo b then t else f , e , s , r) , ch) ⟶_DK ⟨ τ ⟩
      (just (b , e , ifo (t , e) (f , e) :: s , r) , ch)
COND-o : ∀ {e t e' f s r ch} →
  i ⊢ (just (lit o , e , ifo (t , e') f :: s , r) , ch) ⟶_DK ⟨ τ ⟩
      (just (t , e' , s , r) , ch)
COND-suc : ∀ {n e t e' f s r ch} →
  i ⊢ (just (lit (1 + n) , e , ifo t (f , e') :: s , r) , ch) ⟶_DK ⟨ τ ⟩
      (just (f , e' , s , r) , ch)
OP : ∀ {f t t' e s r ch} →
  i ⊢ (just (op f t t' , e , s , r) , ch) ⟶_DK ⟨ τ ⟩
      (just (t , e , op₂ f (t' , e) :: s , r) , ch)
OP₂ : ∀ {n e f t e' s r ch} →
  i ⊢ (just (lit n , e , op₂ f (t , e') :: s , r) , ch) ⟶_DK ⟨ τ ⟩
      (just (t , e' , op₁ (f n) :: s , r) , ch)
OP₁ : ∀ {n e f s r ch} →
  i ⊢ (just (lit n , e , op₁ f :: s , r) , ch) ⟶_DK ⟨ τ ⟩
      (just (lit (f n) , [] , s , r) , ch)
```

The REMOTE execution rule is now meaningful, and it has a **send** and a **receive** version:

```
REMOTE-send : ∀ {t i' e s ch} →
  let (ch' , kp) = i ⊢ ch ▷ s in
  i ⊢ (just (t @ i' , e , s) , ch)
      ⟶_DK ⟨ send (REMOTE t i' kp (num-args s)) ⟩
      (nothing , ch')
```

The operation $i \vdash ch \triangleright s$ signifies allocating at node i in heap ch a new pointer pointing at stack s, and it returns a pair of the updated heap ch' and the newly allocated remote pointer kp. The remote execution directive t @ i' is carried out by sending a REMOTE message to i' consisting of the (pointer to) code t, the destination i', the local continuation-stack pointer kp and the number of arguments on it. After sending the remote execution message the thread halts, i.e. its state is **nothing**.

The function that calculates the number of arguments on the stack is quite subtle and we give its expression below:

$$
\begin{array}{lll}
\mathit{num\text{-}args} : \mathit{Stack} & \to \mathbb{N} \\
\mathit{num\text{-}args}\ ([] \quad , \mathbf{nothing}) & = o \\
\mathit{num\text{-}args}\ ([] \quad , \mathbf{just}\ (\_ , n , \_)) & = n \\
\mathit{num\text{-}args}\ (\mathbf{arg}\ \_ :: s \quad , r) & = 1 + \mathit{num\text{-}args}\ (s , r) \\
\mathit{num\text{-}args}\ (\mathbf{ifo}\ \_\ \_ :: \_ \ , \_) & = o \\
\mathit{num\text{-}args}\ (\mathbf{op_2}\ \_\ \_ :: \_ , \_) & = o \\
\mathit{num\text{-}args}\ (\mathbf{op_1}\ \_ :: \_ \quad , \_) & = o
\end{array}
$$

The function returns the number of arguments at the top of the stack, but it takes into account the possibility that some arguments are local and some arguments are remote. Recall that the remote pointer that we store at the bottom of the stack, pointing to the remote stack extension, also has a natural number *numargs* expressing how many arguments are stored remotely. This is an important optimisation because it makes it possible for this function to be evaluated *locally*, without querying the remote machine where the stack extension is physically located.

The counterpart **REMOTE-receive** rule is:

$$
\begin{array}{l}
\mathbf{REMOTE\text{-}receive} : \forall\ \{ch\ t\ kp\ numargs\} \to \\
\quad i \vdash (\mathbf{nothing} , ch) \\
\qquad \longrightarrow_{\mathcal{DK}} \langle\, \mathbf{receive}\ (\mathbf{REMOTE}\ t\ i\ kp\ numargs) \,\rangle \\
\qquad (\mathbf{just}\ (t , [] , [] , \mathbf{just}\ (kp , numargs , o)) , ch)
\end{array}
$$

The thread on node i is halted when it receives the **REMOTE** execution message, with the same contents as above. The code t becomes the currently executed code in an empty environment — t is, as we explained before, closed — and empty stack remotely extended by kp to the originating machine stack.

Additionally, some of the original rules now have *send* and *receive* counterparts to handle the situation when remote variables or continuations need to be processed. Remarkably, it is possible to avoid sending messages when popping a remote argument, and we can get by with the following new instruction:

$$
\begin{array}{l}
\mathbf{POPARG\text{-}remote} : \forall\ \{t\ e\ kp\ args\ m\ ch\} \to \\
\quad i \vdash (\mathbf{just}\ (\boldsymbol{\lambda}\ t , e , [] , \mathbf{just}\ (kp , 1 + \ args , m)) , ch) \\
\qquad \longrightarrow_{\mathcal{DK}} \langle\, \boldsymbol{\tau} \,\rangle \\
\qquad (\mathbf{just}\ (t , \mathbf{remote}\ kp\ m :: e , [] , \mathbf{just}\ (kp , args , 1 + \ m)) , ch)
\end{array}
$$

Note that this is a silent ($\boldsymbol{\tau}$) transition. A machine does not really "pop" the arguments of a remote stack extension but changes its view of this remote stack.

This avoids instituting a whole class of messages for stack management and it also gives a more robust stack management framework in which stacks, along with heaps and any other data structures involved, are only changed *locally*.

This rule is triggered when a **POPARG** action encounters a local empty stack, which means that the remote stack extension needs to be used. Just like in the case of a local **POPARG**, the environment is updated, but this time with the remote pointer *kp* which has its offset set at *m*. The offset in the view of the remote stack extension is updated (to $1 + m$) to reflect the fact that another argument has been "popped".

It is not difficult to imagine a different, perhaps more obvious, way to perform this instruction's functionality: simply sending a message to the remote node holding the stack that we have a pointer to, instructing it to pop an argument. This would however mean that there would be a larger distribution overhead for invoking a remote function *even if it does not use its argument*. Since we here store how many arguments there are on the remote stack we can safely create a remote closure with an index into the stack without having to tell the remote node about it.

The rules that need genuine remote counterparts are **VAR**, for accessing remote variables, and **RETURN**, for returning a literal from a remote computation.

```
VAR-send : ∀ {n e s rkp index ch} →
  lookup n e ≡ just (remote rkp index) →
  let (ch' , kp) = i ⊢ ch ▷ s in
  i ⊢ (just (var n , e , s) , ch)
      ⟶_DK ⟨ send (VAR rkp index kp (num-args s)) ⟩
      (nothing , ch')
```

The **VAR-send** rule is triggered when the machine detects a **remote** pointer in its environment *e*. Just like in the case of the **REMOTE** instruction, the current continuation stack is saved in the continuation heap of the machine *i*, at address *kp*. The machine then sends a **VAR**-tagged message onto the network, with the structure discussed before, and halts, i.e. its thread is **nothing**. Note that the left-hand-side of the transition triggered by the **VAR-send** rule is almost the same as that of the local **VAR** rule.

Upon receiving a **VAR** message, a (halted) machine executes the **VAR-receive** instruction:

```
VAR-receive : ∀ {ch kp s n rkp m el} →
  ch ! kp ≡ just s →
  stack-index s n ≡ just el →
  i ⊢ (nothing , ch)
      ⟶DK ⟨ receive (VAR (kp , i) n rkp m) ⟩
      (just (var o , el :: [] , [] , just (rkp , m , o)) , ch)
```

The right-hand-side of the **VAR-receive** rule introduces a new variable **var** *o*, perhaps surprisingly. In order to avoid having special cases where the retrieved variable index is itself either local or remote, we create the dummy variable **var** *o* referring to the variable pointed-to by the received **VAR** message. This is what the *stack-index* : *Stack* → ℕ → *Maybe EnvElem* function, invoked on the stack that *kp* points to, achieves. If the stack element at index *n* in the stack is a local argument, then it returns that closure as a **local** environment element. If the element at index *n* refers to an argument on the remote stack extension, it returns a corresponding **remote** environment element. Afterwards we can use the existing local **VAR** or **VAR-send** rules depending on whether the variable is local or remote also to this node.

```
RETURN-send : ∀ {n e kp m ch} →
  i ⊢ (just (lit n , e , [] , just (kp , o , m)) , ch)
      ⟶DK ⟨ send (RETURN kp n m) ⟩
      (nothing , ch)
RETURN-receive : ∀ {ch kp s s' n m} →
  ch ! kp ≡ just s → drop-stack s m ≡ just s' →
  i ⊢ (nothing , ch)
      ⟶DK ⟨ receive (RETURN (kp , i) n m) ⟩
      (just (lit n , [] , s') , ch)
```

Finally, the **RETURN-send** and **RETURN-receive** rules are triggered when a machine has reached a literal and has a remote stack extension without any arguments, implying that the remote stack is either empty (i.e. it is located at the root node of the whole execution) or it has a continuation requiring a natural number literal. In both cases we want to send the literal back to the node where the stack is located. The one thing to notice is that the message includes the number *m* to be used by the receiver to drop the correct number of elements from the top of the stack. This is handled by the *drop-stack* function, defined as follows:

```
drop-stack : Stack → ℕ → Maybe Stack
drop-stack (s , r)                    o          = just (s , r)
drop-stack ([] , just (_ , o , _))    (1 + _)    = nothing
drop-stack ([] , just (kp , 1 + n , m)) (1 + i) =
  drop-stack ([] , just (kp , n , 1 + m)) i
drop-stack ([] , nothing)             (1 + _)    = nothing
drop-stack (arg _ :: s , r)           (1 + i)    = drop-stack (s , r) i
drop-stack (_ :: _ , _)               (1 + _)    = nothing
```

As in the case of *num-args* the function may change the local view of a remote stack extension, without requiring further message exchanges between nodes. If not enough arguments are on the stack the function returns **nothing**, which should not happen during a normal execution since we take care to keep the stack views consistent.

The definition of network transitions (Section 2.5) is parameterised by a machine transition relation $\longrightarrow_{\mathcal{M}}$, which is subsequently instantiated to $\longrightarrow_{\mathcal{DK}}$, and initialised by starting from a designated node i with code t and all other constituents empty.

```
open import Network Node _≟_ _ ⊢ _⟶ 𝒟𝒦⟨_⟩_ public
initial-network_Sync : Term → Node → SyncNetwork
initial-network_Sync t i =
  let inactives = λ i → (nothing , ∅)
      active    = (just (t , [] , [] , nothing) , ∅)
  in inactives[i ↦ active]
initial-network_Async : Term → Node → AsyncNetwork
initial-network_Async c i = initial-network_Sync c i , []
```

As mentioned in Section 2.5, it is immediate to show that a *SyncNetwork* can be represented by the more expressive *AsyncNetwork*. The other direction is not as trivial, and is formalised by the following lemma, stating that whenever some DKrivine machines can make an *Async* transition with the global pool of messages remaining the same (empty, for simplicity), the same transition could be made in a *SyncNetwork*:

Lemma 6.2.1. If there is exactly one active node in a family of nodes *nodes* and $((nodes\,,\,[])\ \xrightarrow[Async]{}{}^{+}\ (nodes',\,[]))$, then then $nodes\ \xrightarrow[Sync]{}{}^{+}\ nodes'$.[4]

The lemma is stated as follows in Agda:

[4]Online appendix: `krivine/formalisation` directory, `DKrivine.Properties` module.

$$
\begin{aligned}
\mathit{Async^+\text{-}to\text{-}Sync^+} \;:\; & \forall\, \{\mathit{nodes}\ \mathit{nodes'}\}\ i \rightarrow \\
& \mathit{all\ nodes\ except\ i\ are\ inactive} \rightarrow \\
& ((\mathit{nodes}\,,\,[]) \xrightarrow[\mathit{Async}]{}{}^{+}\ (\mathit{nodes'}\,,\,[])) \rightarrow \\
& \mathit{nodes} \xrightarrow[\mathit{Sync}]{}{}^{+}\ \mathit{nodes'} \\
\mathit{Async^+\text{-}to\text{-}Sync^+} \;=\; & \mathit{Async^+\text{-}to\text{-}Sync^+\text{-}lemma}\ \mathbf{refl}\ \mathbf{refl}
\end{aligned}
$$

The proof is an immediate application of a more complex lemma which can be found in the online appendix. In contrast to the *Sync-to-Async*$^+$ embedding,[5] this embedding is specific to DKrivine machines. More precisely, two properties of these machines make this possible. The first one is that the DKrivine machines halt after each message **send** and **receive** only from halting states. The second one is that they are deterministic. Intuitively, it is fairly clear that the two styles of communication are equivalent under these circumstances.

These two results about Krivine nets are interesting because they show that we do not need to commit to a synchronous or asynchronous network of DKrivine machines since they are equivalent. We may therefore use whichever is more convenient for correctness proofs in the knowledge that the properties we prove transfer immediately to the other one.

6.2.1 *Example*

Let us briefly compare the execution of a rather simple term,

$$((\lambda f.\lambda x.f\ x)@B)(\lambda y.y)\ \mathrm{o}$$

on a single machine and on a distributed machine.[6] The program is located on (the default) node *A*, except for $\lambda f.\lambda x.f\ x$ which is on node *B*. This program is similar to our introductory example in that it does a remote function call, and additionally shows that higher-order remote function calls are also possible.

As we discussed earlier, the Krivine machine ignores the @ construct (the **REMOTE** rule is a no-op), producing the execution trace **PUSHARG**; **PUSHARG**; **REMOTE**; **POPARG**; **POPARG**; **PUSHARG**; **VAR**; **POPARG**; **VAR**; **VAR**, which leaves the machine in state (**lit** *o* , [] , []).

The Krivine net of two nodes produces the following trace (informally, indicating machine state only when interesting). Node *A* starts with **PUSHARG**; **PUSHARG**; **REMOTE-send**, which produces the message

$$\mathbf{REMOTE}\ (\boldsymbol{\lambda}\ (\boldsymbol{\lambda}\ (\mathbf{var}\ 1\ \$\ \mathbf{var}\ o)))\ B\ (\mathit{ptr}_1\,,\,A)\ 2$$

where ptr_1 points to the stack ([($\boldsymbol{\lambda}$ **var** *o* , []) , (*o* , [])] , **nothing**).

[5] Online appendix: `krivine/formalisation` directory, `Network` module.

[6] Online appendix: `krivine/formalisation` directory, `Trace` module.

Node B receives the message and executes **REMOTE-receive**; **POPARG-remote**; **POPARG-remote**; **PUSHARG**; **VAR-send**, which produces the message

$$\textbf{VAR}\ (ptr_1\ ,\ A)\ o\ (ptr_2\ ,\ B)\ 1$$

where ptr_2 points to the stack

$$[\ (\textbf{var}\ o\ ,\ [remote\ (ptr_1\ ,\ A)\ 1\ ,\ \textbf{remote}\ (ptr_1\ ,\ A)\ o\ ])\ ]\ ,\ \textbf{just}\ ((ptr_1\ ,\ A)\ ,\ o\ ,\ 2)$$

Note that the two traces are essentially the same, except for the **REMOTE** rule becoming meaningful. As we explained before, the **POPARG-remote** rule only changes the local view of the remote stack extension and generates no communication overhead. Also note that the stack at ptr_2 extends remotely to the stack at ptr_1 and uses it in its own stored closures.

The rest of the dialogue is as follows:

Node A : **VAR-receive**; **VAR**; **POPARG-remote**; **VAR-send**
Node B : **VAR-receive**; **VAR**; **VAR-send**
Node A : **VAR-receive**; **VAR**; **RETURN-send**
Node B : **RETURN-receive**; **RETURN-send**
Node A : **RETURN-receive**; **RETURN-send**
Node B : **RETURN-receive**; **RETURN-send**
Node A : **RETURN-receive**

Compared to the Krivine trace, the **VAR** instructions is here broken into a **send** and **receive** version if the requested variable is remote. There is also the additional **VAR** rule needed to avoid a case statement on whether a variable is local or remote. The **RETURN** instructions are new, required to forward computed values to the caller.

After the execution, the heaps of the two nodes are:

$$\begin{aligned}
A\ :\ \{&ptr_1 \mapsto ([arg\ (\boldsymbol{\lambda}\ \textbf{var}\ o\ ,\ [])\,;\textbf{arg}\ (\textbf{lit}\ o\ ,\ [])\ ]\ ,\ \textbf{nothing})\ ,\\
&ptr_3 \mapsto ([]\ ,\ \textbf{just}\ ((ptr_2\ ,\ B)\ ,\ o\ ,\ 1))\}\\
B\ :\ \{&ptr_2 \mapsto ([arg\ (\textbf{var}\ o\ ,\ [remote\ (ptr_1\ ,\ A)\ 1\,;\textbf{remote}\ (ptr_1\ ,\ A)\ o\ ])\ ]\ ,\\
&\qquad \textbf{just}\ ((ptr_1\ ,\ A)\ ,\ o\ ,\ 2))\ ,\\
&ptr_4 \mapsto ([]\ ,\ \textbf{just}\ ((ptr_3\ ,\ A)\ ,\ o\ ,\ o))\}
\end{aligned}$$

A graphical representation of the final heap is in Figure 6.2, with stack extension pointers in black and remote variables in grey.

Unlike the Krivine machine, the Krivine nets will result in non-empty heaps (*garbage*) in the individual DKrivine machines. We will discuss how to deal with this in Section 9.3.

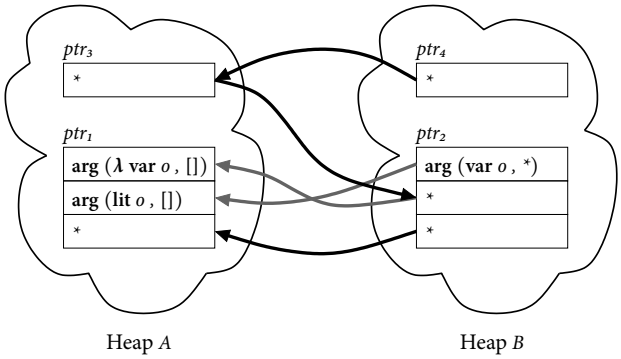


Figure 6.2: Final heap

A graphical representation of the final heap of the example, with stack extension pointers in black and remote variables in grey.

6.3 CORRECTNESS

We prove the correctness of the DKrivine machine by exhibiting a simulation between the conventional Krivine machine and a Krivine net.[7] The simulation is then used to prove the following *Soundness theorems*:[8]

Theorem 6.3.1. If $cfg \downarrow_{\mathcal{K}} \mathbf{lit}\ n$ and $R_{Sync}\ cfg\ nodes$, then $nodes \downarrow_{Sync} \mathbf{lit}\ n$.

Theorem 6.3.2. If $cfg \uparrow_{\mathcal{K}}$ and $R_{Sync}\ cfg\ nodes$, then $nodes \uparrow_{Sync}$.

These theorems correspond to the following Agda definitions:

$$\begin{aligned} \mathit{termination\text{-}agrees}_{Sync} &: \forall\ cfg\ nodes\ n \to R_{Sync}\ cfg\ nodes \to \\ &\quad cfg \downarrow_{\mathcal{K}} \mathbf{lit}\ n \to nodes \downarrow_{Sync} \mathbf{lit}\ n \\ \mathit{divergence\text{-}agrees}_{Sync} &: \forall\ cfg\ nodes \to R_{Sync}\ cfg\ nodes \to \\ &\quad cfg \uparrow_{\mathcal{K}} \to nodes \uparrow_{Sync} \end{aligned}$$

The termination theorem states that for any Krivine machine configuration *cfg* and any Krivine net configuration *nodes*, if we have a *simulation relation* R_{Sync} between them then for any literal *n*, if the Krivine machine starting from *cfg* produces the literal *n*, then the Krivine net starting from configuration *nodes* produces the same. Note that we are using *Sync* nets, because they are more convenient and because *Async* nets can be reduced to *Sync* nets in the case of Krivine nets, as discussed previously. The divergence theorem makes a similar point about non-termination: from related states, if the Krivine machine diverges then the Krivine net diverges.

[7] Online appendix: `krivine/formalisation` directory, `DKrivine.Simulation` module.

[8] Online appendix: `krivine/formalisation` directory, `DKrivine.Soundness` module.

6.3.1 *The simulation relation*

The most important ingredient of the correctness proof is defining and exhibiting the appropriate simulation relation. At the top level, the relation between the Krivine machine and Krivine net configurations is defined as follows:

$$
\begin{aligned}
&R_{Sync} \;:\; \mathit{Rel\ Config\ SyncNetwork} \\
&R_{Sync}\ \mathit{cfg\ nodes} \;=\; \exists\ \lambda\ i \rightarrow \\
&\quad \mathit{all\ nodes\ except\ i\ are\ inactive} \times \\
&\quad R_{Machine}\ (\mathit{proj}_2 \circ \mathit{nodes})\ \mathit{cfg}\ (\mathit{proj}_1\ (\mathit{nodes}\ i))
\end{aligned}
$$

In Agda notation the existential statement $\exists i.P(i)$ is written $\exists\ \lambda\ i \rightarrow P\ i$. The predicate *all_except_are_* is defined as

$$
\mathit{all\ f\ except\ x\ are\ P} \;=\; \forall\ x' \rightarrow x' \neq x \rightarrow P\ (f\ x')
$$

and *inactive node* holds exactly when the thread of *node* is **nothing**. A simulation between machine and net configurations exists only when precisely one node i is active in the net. The machine at that node ($\mathit{proj}_1\ (\mathit{nodes}\ i)$) must be related to the configuration of the Krivine machine through the following machine-simulation relation:

$$
\begin{aligned}
&R_{Machine} \;:\; \mathit{Heaps} \rightarrow \mathit{Rel\ Config}\ (\mathit{Maybe\ Thread}) \\
&R_{Machine}\ \mathit{hs}\ (t_1\ ,\ e_1\ ,\ s_1)\ (\mathbf{just}\ (t_2\ ,\ e_2\ ,\ s_2)) \;= \\
&\quad R_{Term}\ t_1\ t_2 \times R_{Env}\ \mathit{hs}\ e_1\ e_2 \times (\exists\ \lambda\ \mathit{rank} \rightarrow R_{Stack}\ \mathit{rank}\ \mathit{hs}\ s_1\ s_2) \\
&R_{Machine}\ \mathit{hs}\ (t_1\ ,\ e_1\ ,\ s_1)\ \mathbf{nothing} \;=\; \bot
\end{aligned}
$$

The relation is indexed by the distributed heap of the Krivine net $\mathit{hs} \;:\; \mathit{Heaps}$, which is the *Node*-indexed family of all the individual heaps. This relation $R_{Machine}$ simply distributes the relation further to terms using R_{Term}, environments using R_{Env} and stacks using R_{Stack}. In order for this to be possible it is required that the DKrivine machine is not halted (**nothing** : *Maybe Thread*).

On terms, the relation R_{Term} is just propositional equality, while R_{Env} and R_{Stack} are more subtle and require a non-trivial proof technique. R_{Stack} is similar to a *step-indexed* relation [7] on stacks. It is defined by induction on a natural number *rank* in order to ensure that the cascading remote stack extensions do not have any cycles. Unlike a step-indexed relation, *rank* means that we do exactly *rank* remote-pointer dereferencings in the process of relating two stacks, and R_{Stack} requires that this number is known. $R_{EnvElem}$, used by R_{Env} to relate environment elements, is defined by induction on a *rank* for the same reason.

6.3.2 *Relating environments*

On environments, the definition of the relation is:

$$
\begin{aligned}
&R_{Env} \;:\; \mathit{Heaps} \to \mathit{Rel}\ \mathit{Krivine.Env}\ \mathit{DKrivine.Env} \\
&R_{Env}\ \mathit{hs}\ []\ \ \ \ \ \ []\ \ \ \ \ \ = \top \\
&R_{Env}\ \mathit{hs}\ []\ \ \ \ \ \ (x_2 :: e_2)\ = \bot \\
&R_{Env}\ \mathit{hs}\ (x_1 :: e_1)\ []\ \ \ \ \ \ = \bot \\
&R_{Env}\ \mathit{hs}\ (x_1 :: e_1)\ (x_2 :: e_2)\ = \\
&\quad (\exists\ \lambda\ \mathit{rank} \to R_{EnvElem}\ \mathit{rank}\ \mathit{hs}\ x_1\ x_2) \times R_{Env}\ \mathit{hs}\ e_1\ e_2
\end{aligned}
$$

Empty environments are trivially related, but environments of different shapes cannot be related. If both environments are non-empty then the definition is inductive on the structure of the environment. Environment elements are related by requiring that there exists a *rank* such that they are related by $R_{EnvElem}$:

$$
\begin{aligned}
&R_{EnvElem} \;:\; \mathbb{N} \to \mathit{Heaps} \to \mathit{Rel}\ \mathit{Krivine.EnvElem}\ \mathit{DKrivine.EnvElem} \\
&R_{EnvElem}\ o\ \ \ \ \ \ \ \ \ \ \mathit{hs}\ (\mathbf{clos}\ c_1)\ (\mathbf{local}\ c_2)\ = R_{Closure}\ \mathit{hs}\ c_1\ c_2 \\
&R_{EnvElem}\ (1 +\ \mathit{rank})\ \mathit{hs}\ (\mathbf{clos}\ c_1)\ (\mathbf{local}\ c_2)\ = \bot \\
&R_{EnvElem}\ o\ \ \ \ \ \ \ \ \ \ \mathit{hs}\ (\mathbf{clos}\ c_1)\ (\mathbf{remote}\ \mathit{contptr}\ \mathit{index})\ = \bot \\
&R_{EnvElem}\ (1 +\ \mathit{rank})\ \mathit{hs}\ (\mathbf{clos}\ c_1)\ (\mathbf{remote}\ \mathit{contptr}\ \mathit{index})\ = \\
&\quad \mathit{stack\text{-}ext\text{-}pred}\ \mathit{hs}\ \mathit{contptr} \\
&\qquad (\lambda\ s_2 \to \exists\ \lambda\ ee_2 \to \mathit{stack\text{-}index}\ s_2\ \mathit{index} \equiv \mathbf{just}\ ee_2 \\
&\qquad\qquad \times R_{EnvElem}\ \mathit{rank}\ \mathit{hs}\ (\mathbf{clos}\ c_1)\ ee_2)
\end{aligned}
$$

Local closures of the DKrivine machine relate to closures of the Krivine machine ($R_{Closure}$) if their terms are equal and their environments are related through R_{Env}. Relating remote closures (**remote** *contptr index*) of the DKrivine machine to the closures of the Krivine machine (**clos** c_1) is perhaps the most subtle part of the definition. It uses the following helper function which ensures that, given a distributed heap $\mathit{hs} : \mathit{Heaps}$, a remote pointer $(\mathit{ptr}, \mathit{loc}) : \mathit{ContPtr}$ and a predicate on distributed stacks $\mathit{DKrivine.Stack} \to \mathit{Set}$, the pointer points to a stack in the heap of node *loc* such that the predicate holds:

$$
\begin{aligned}
&\mathit{stack\text{-}ext\text{-}pred} \;:\; \mathit{Heaps} \to \mathit{ContPtr} \to (\mathit{DKrivine.Stack} \to \mathit{Set}) \to \mathit{Set} \\
&\mathit{stack\text{-}ext\text{-}pred}\ \mathit{hs}\ (\mathit{ptr}\ ,\ \mathit{loc})\ P\ = \exists\ \lambda\ s \to (\mathit{hs}\ \mathit{loc}\ !\ \mathit{ptr} \equiv \mathbf{just}\ s) \times P\ s
\end{aligned}
$$

The pointer dereferencing operation is $\mathit{hs}\ \mathit{loc}\ !\ \mathit{ptr}$. The predicate which we use in the definition of $R_{EnvElem}$ is that there exists an element ee_2 in the environment of the DKrivine machine such that it $R_{EnvElem}$-relates to the Krivine closure **clos** c_1 in one less step.

Note that the *rank* has to be o to relate local elements, and it has to be $1 +$ *rank* to relate a remote element. The recursive call is done with the predecessor *rank*. This makes sure that there are *exactly rank* pointers to follow to reach a local closure if we have an element of $R_{EnvElem}\ \mathit{rank}\ \mathit{hs}\ x_1\ x_2$. This means, in particular, that there can be no circular sequences of pointers between the nodes of the

system. When there are no circular sequences of pointers between the nodes of the system it is enough to use distributed reference counting [16] — full-blown distributed garbage collection is not a necessity.

6.3.3 *Relating stacks*

Relating stacks is somewhat similar.

$$
\begin{array}{l}
R_{Stack} : \mathbb{N} \to Heaps \to Rel\ Krivine.Stack\ DKrivine.Stack \\
R_{Stack}\ rank \quad hs\ (x_1 :: s_1)\ ([]\ ,\ \mathbf{nothing}) = \bot \\
R_{Stack}\ rank \quad hs\ []\ (x_2 :: s_2\ ,\ r) = \bot \\
R_{Stack}\ o \quad hs\ []\ ([]\ ,\ \mathbf{nothing}) = \top \\
R_{Stack}\ (1 + rank)\ hs\ []\ ([]\ ,\ \mathbf{nothing}) = \bot \\
R_{Stack}\ rank \quad hs\ (x_1 :: s_1)\ (x_2 :: s_2\ ,\ r) = \\
\quad R_{StackElem}\ hs\ x_1\ x_2 \times R_{Stack}\ rank\ hs\ s_1\ (s_2\ ,\ r) \\
R_{Stack}\ o \quad hs\ s_1\ ([]\ ,\ \mathbf{just}\ (contptr\ ,\ args\ ,\ drop)) = \bot \\
R_{Stack}\ (1 + rank)\ hs\ s_1\ ([]\ ,\ \mathbf{just}\ (contptr\ ,\ args\ ,\ drop)) = \\
\quad stack\text{-}ext\text{-}pred\ hs\ contptr\ (\lambda\ s_2 \to \\
\qquad \exists\ \lambda\ ds_2 \to drop\text{-}stack\ s_2\ drop \equiv \mathbf{just}\ ds_2 \\
\qquad\qquad \times\ num\text{-}args\ ds_2 \equiv args \times R_{Stack}\ rank\ hs\ s_1\ ds_2)
\end{array}
$$

Empty stacks, with no remote extensions, are related if the *rank* is 0, whereas empty and non-empty are not related. Two non-empty stacks are related if the elements on top are related by $R_{EnvElem}$ and the remaining stacks are related. The relation is interesting when remote pointer extensions are involved. If there is a remote stack extension but the step-index is 0 then it cannot be related to a Krivine stack. If there is a non-zero step-index then, using the same helper function *stack-ext-pred*, we require that the substack ds_2 of s_2 obtained by dropping the *drop* arguments required by the remote stack extension pointer **just** (*contptr* , *args* , *drop*) is related to the Krivine stack s_1 using a smaller (by one) index.

Finally, stack elements are related if they have the same head constructor, and the constituents are related:

$$
\begin{array}{lll}
R_{StackElem} : Heaps \to Rel\ Krivine.StackElem\ DKrivine.StackElem & & \\
R_{StackElem}\ hs\ (\mathbf{arg}\ c_1) & (\mathbf{arg}\ c_2) & = R_{Closure}\ hs\ c_1\ c_2 \\
R_{StackElem}\ hs\ (\mathbf{ifo}\ c_1\ c_1\text{'}) & (\mathbf{ifo}\ c_2\ c_2\text{'}) & = R_{Closure}\ hs\ c_1\ c_2 \times \\
 & & \quad R_{Closure}\ hs\ c_1\text{'}\ c_2\text{'} \\
R_{StackElem}\ hs\ (\mathbf{op_2}\ f\ c_1) & (\mathbf{op_2}\ g\ c_2) & = f \equiv g \times \\
 & & \quad R_{Closure}\ hs\ c_1\ c_2 \\
R_{StackElem}\ hs\ (\mathbf{op_1}\ f) & (\mathbf{op_1}\ g) & = f \equiv g \\
R_{StackElem}\ hs\ \_ & \_ & = \bot
\end{array}
$$

6.3.4 *Proof outline*

In order to prove the main property we need to first establish the monotonicity of all the heap-indexed relations relative to heap inclusion: if two machine configurations, environments, environment elements, or stacks are related in a family of heaps *hs* they are also related in any larger family of heaps *hs* $\subseteq_s$ *hs'*. The _$\subseteq s$_ relation is a pointwise lifting of heap inclusion $h \subseteq h'$, which states that any element in *h* is also in *h'*. The main property is the following:

Lemma 6.3.3. If *hs* $\subseteq_s$ *hs'* then $R_{Machine}$ *hs cfg m* implies $R_{Machine}$ *hs' cfg m*.[9]

The properties are proved in a local Agda module parameterised by the heap inclusion property, and therefore it does not need to be included in each statement — it is a background assumption:

```
module HeapUpdate (hs hs' : Heaps) (inc : hs ⊆s hs') where
  envelem   : ∀ rank el el' → R_EnvElem rank hs el el'
                            → R_EnvElem rank hs' el el'
  env       : ∀ e e' → R_Env hs e e' → R_Env hs' e e'
  stackelem : ∀ el el' → R_StackElem hs el el'
                       → R_StackElem hs' el el'
  stack     : ∀ rank s s' → R_Stack rank hs s s'
                          → R_Stack rank hs' s s'
  machine   : ∀ cfg m → R_Machine hs cfg m
                      → R_Machine hs' cfg m
```

The proofs are largely straightforward, inductive on the structure of the data structure the lemma is concerned with. The key auxiliary property that makes monotonicity of the relations true is the fact that any predicate which relies on heap dereferencing is preserved:

```
s-ext-pred : ∀ contptr {P Q} → (∀ s → P s → Q s) →
  stack-ext-pred hs contptr P → stack-ext-pred hs' contptr Q
```

For example, for environments, environment elements, and closures the proofs are mutually recursive, inductive on their structures:

[9] Online appendix: `krivine/formalisation` directory, `DKrivine.Simulation` module.

```
closure : ∀ c c' → R_Closure hs c c' → R_Closure hs' c c'
envelem : ∀ rank el el' → R_EnvElem rank hs el el'
                        → R_EnvElem rank hs' el el'
envelem o          (clos c) (local c') Rcc' = closure c c' Rcc'
envelem (1 + rank) (clos c) (local c') Rcc' = Rcc'
envelem o          (clos c) (remote contptr index) Relel' = Relel'
envelem (1 + rank) (clos c) (remote contptr index) Relel' =
  s-ext-pred contptr f Relel'
  where
    f : ∀ s →
    (∃ λ ee' → stack-index s index ≡ just ee'
       × R_EnvElem rank hs (clos c) ee') →
       ∃ λ ee' → stack-index s index ≡ just ee'
       × R_EnvElem rank hs' (clos c) ee'
    f s (ee' , si , Rcee') = ee' , si , envelem rank (clos c) ee' Rcee'


env : ∀ e e' → R_Env hs e e' → R_Env hs' e e'
env []       []       Ree' = Ree'
env []       (x :: e') Ree' = Ree'
env (x :: e) []       Ree' = Ree'
env (x :: e) (x' :: e') ((rank , Rxx') , Ree') =
  (rank , envelem rank x x' Rxx') , env e e' Ree'
closure (t , e) (t' , e') (Rtt' , Ree') = Rtt' , env e e' Ree'
```

The soundness theorem $termination\text{-}agrees_{Sync}$ stated at the beginning of this section follows directly from two important lemmas, called $simulation_{Sync}$ and *termination-return*. The former is the main technical result of this work on distributing the Krivine machine (soundness is merely a corollary of it) and the latter is used to handle the remaining non-trivial case of the soundness proof, that of cascading **RETURN** statements at the end of an execution.

Theorem 6.3.4. The relation R_{Sync} is a *Simulation* relation between the $\longrightarrow_{\mathcal{K}}$ and $\xrightarrow[Sync]{}^{+}$ transition relations.[10]

This theorem is proved by the following Agda definition:

```
simulation_Sync : Simulation _⟶_K_ _⟶_Sync^+_ R_Sync
```

A simulation relation is defined in the standard way, where $\longrightarrow$ and $\longrightarrow'$ are transition relations that parameters of the enclosing module:[11]

[10] Online appendix: `krivine/formalisation` directory, `DKrivine.Simulation` module.
[11] Online appendix: `krivine/formalisation` directory, `Relation` module.

$$
\begin{aligned}
&\mathit{Simulation} : (\_R\_ : \mathit{Rel}\ A\ B) \to \mathit{Set} \\
&\mathit{Simulation}\ \_R\_ = \forall\ a\ a'\ b \to (a \longrightarrow a') \to a\ R\ b \to \\
&\qquad\qquad \exists\ \lambda\ b' \to (b \longrightarrow' b') \times a'\ R\ b'
\end{aligned}
$$

The proof of $\mathit{simulation}_{\mathit{Sync}}$ is lengthy but largely routine. The non-trivial cases are:

- **RETURN** actions of the DKrivine machines, which are handled by the lemma *simulation-return*:

$$
\begin{aligned}
&\mathit{simulation\text{-}return} : \forall\ n\ e\ s\ \mathit{cfg'}\ e'\ s'\ i\ \mathit{nodes}\ \mathit{srank}\ \mathit{conth} \to \\
&\quad \mathsf{let}\ \ \mathit{cfg} = (\mathbf{lit}\ n\ ,\ e\ ,\ s) \\
&\qquad\ \ \mathit{hs} = \mathit{proj}_2 \circ \mathit{nodes} \\
&\qquad \mathsf{in}\ \mathit{cfg} \longrightarrow_{\mathcal{K}} \mathit{cfg'} \to \\
&\qquad\quad \mathit{all\ nodes\ except\ i\ are\ inactive} \to \\
&\qquad\quad \mathit{nodes}\ i \equiv \mathbf{just}\ (\mathbf{lit}\ n\ ,\ e'\ ,\ s')\ ,\ \mathit{conth} \to \\
&\qquad\quad R_{\mathit{Stack}}\ \mathit{srank}\ \mathit{hs}\ s\ s' \to \exists\ \lambda\ \mathit{nodes'} \to \\
&\qquad\qquad \mathit{nodes} \xrightarrow[\mathit{Sync}]{}{}^{+}\ \mathit{nodes'} \times R_{\mathit{Sync}}\ \mathit{cfg'}\ \mathit{nodes'}
\end{aligned}
$$

- **VAR** remote actions of the DKrivine machine, which are handled by the lemma *simulation-var*:

$$
\begin{aligned}
&\mathit{simulation\text{-}var} : \forall\ t\ e\ s\ n\ e'\ s'\ \mathit{nodes}\ i\ \mathit{conth}\ \mathit{el} \to \\
&\quad \mathsf{let}\ \mathit{hs} = \mathit{proj}_2 \circ \mathit{nodes}\ \mathsf{in} \\
&\qquad (\exists\ \lambda\ \mathit{rank} \to R_{\mathit{EnvElem}}\ \mathit{rank}\ \mathit{hs}\ (\mathbf{clos}\ (t\ ,\ e))\ \mathit{el}) \to \\
&\qquad (\exists\ \lambda\ \mathit{rank} \to R_{\mathit{Stack}}\ \mathit{rank}\ \mathit{hs}\ s\ s') \to \\
&\qquad \mathit{all\ nodes\ except\ i\ are\ inactive} \to \\
&\qquad \mathit{nodes}\ i \equiv \mathbf{just}\ (\mathbf{var}\ n\ ,\ e'\ ,\ s')\ ,\ \mathit{conth} \to \\
&\qquad \mathit{lookup}\ n\ e' \equiv \mathbf{just}\ \mathit{el} \to \\
&\qquad \exists\ \lambda\ \mathit{nodes'} \to (\mathit{nodes} \xrightarrow[\mathit{Sync}]{}{}^{+}\ \mathit{nodes'}) \times R_{\mathit{Sync}}\ (t\ ,\ e\ ,\ s)\ \mathit{nodes'}
\end{aligned}
$$

What is interesting about these two lemmas, which establish the conditions under which the simulation relation is preserved by transitions related to the integer operations and **VAR** rules, is that it requires a different proof technique, induction on the *rank*. This is because the distributed machine may need to perform a cascade of returns (or variable accesses) between different nodes before it reaches a configuration related to that of the Krivine machine, as we saw in the example in Section 6.2.1.

The *termination-return* lemma mentioned earlier uses a similar proof technique (induction on the *rank*); its full statement is:

$$
\begin{aligned}
&\mathit{termination\text{-}return} : \forall\ n\ e'\ s'\ i\ \mathit{nodes}\ \mathit{srank}\ \mathit{conth} \to \\
&\quad \mathsf{let}\ \mathit{hs} = \mathit{proj}_2 \circ \mathit{nodes} \\
&\qquad \mathsf{in}\ \mathit{all\ nodes\ except\ i\ are\ inactive} \to \\
&\qquad\quad \mathit{nodes}\ i \equiv \mathbf{just}\ (\mathbf{lit}\ n\ ,\ e'\ ,\ s')\ ,\ \mathit{conth} \to \\
&\qquad\quad R_{\mathit{Stack}}\ \mathit{srank}\ \mathit{hs}\ []\ s' \to \mathit{nodes} \downarrow_{\mathit{Sync}} \mathbf{lit}\ n
\end{aligned}
$$

The second part of the soundness proof is the agreement on divergence between the Krivine machine and the Krivine net. This proof relies essentially on the fact that a Krivine net transition is deterministic whenever only one node is active and that the Krivine machine transition's codomain is decidable in the following sense:[12]

$$
\begin{aligned}
&\_\mathit{is\text{-}deterministic\text{-}at}\_ : \{A\ B : \mathit{Set}\}\ (R : \mathit{Rel}\ A\ B)\ (x : A) \to \mathit{Set} \\
&\_R\_\ \mathit{is\text{-}deterministic\text{-}at}\ a = \forall\ \{b\ b'\} \to a\ R\ b \to a\ R\ b' \to b \equiv b' \\
&\_\mathit{is\text{-}decidable} : \{A\ B : \mathit{Set}\}\ (\_R\_ : \mathit{Rel}\ A\ B) \to \mathit{Set} \\
&\_R\_\ \mathit{is\text{-}decidable} = \forall\ a \to \mathit{Dec}\ (\exists\ \lambda\ b \to a\ R\ b)
\end{aligned}
$$

Theorem 6.3.5. If all nodes except one are *inactive* in a *SyncNetwork nodes*, then $\_\xrightarrow[\mathit{Sync}]{}\_$ *is-deterministic-at nodes*, i.e. the next transition is deterministic.

Theorem 6.3.6. It is decidable whether a *SyncNetwork* can make a transition or not.

In Agda, these theorems are stated as:

$$
\begin{aligned}
&\mathit{determinism}_{\mathit{Sync}} : \forall\ \mathit{nodes}\ i \to \mathit{all\ nodes\ except\ i\ are\ inactive} \to \\
&\quad \_\xrightarrow[\mathit{Sync}]{}\_\ \mathit{is\text{-}deterministic\text{-}at\ nodes} \\
&\mathit{decidable}_{\mathcal{K}} : \_\longrightarrow_{\mathcal{K}}\_\ \mathit{is\text{-}decidable}
\end{aligned}
$$

To conclude this section, we need to show that initial configurations are related so that we have a starting point for the simulation. This is easy to do since the environments and stacks are empty:

$$
\mathit{initial\text{-}related}_{\mathit{Sync}} : \forall\ t\ \mathit{root} \to R_{\mathit{Sync}}\ (t\ ,\ []\ ,\ []\ )\ (\mathit{initial\text{-}network}_{\mathit{Sync}}\ t\ \mathit{root})
$$

[12] Online appendix: `krivine/formalisation` directory, `DKrivine.Properties` module.

6.4 PROOF OF CONCEPT IMPLEMENTATION

We have implemented a prototype compiler for Krivine nets.[13] Except for the _@_ directive, compilation to Krivine nets is implemented by using the same standard compilation scheme used to compile Krivine machines. The aim is not efficiency as much as simplicity. Since the machine is deterministic, we compile each constructor of the source language to a given code sequence. Each argument to a function in the source language becomes a separate C function, such that its address can be taken. As an example, pushing an argument is translated into a bytecode instruction **PUSHARG** (*f*) where *f* is the (statically known) address of the function obtained from compiling the argument. Each bytecode instruction of the Krivine machine is in turn translated into separate C functions, and message passing is implemented using Message Passing Interface (MPI).

The runtime system of the DKrivine machine takes into account whether pointers are local or remote and behaves accordingly. A remote pointer is represented as the following C struct:

```
struct RemotePtr {
  void* ptr;
  int location;
}
```

The environment uses tags to distinguish between local and remote pointers just like the Agda definition.

The _@_ directive is translated directly into a predefined **REMOTE** bytecode instruction, which constructs and sends a **REMOTE** message at run time. As mentioned, we avoid sending code by grouping fragments of output code that correspond to the same node, and compiling each group as a separate binary. The fragment of code that corresponds to *t* inside a subterm $t \mathbin{@} A$ is assigned, at compile time, a global identifier that an invoking node can use to activate *t* on node *A*, meaning that no actual code has to be sent at run time.

The main loop of each node is set up to receive messages and act depending on their tag, conceptually like the following code skeleton:

```
while(1) {
  Msg message = receive();
  switch(message.tag) {
      case REMOTE_MSG_TAG: ...
      case RETURN_MSG_TAG: ...
```

[13] Online appendix: `krivine/implementation` directory.

```
        case VAR_MSG_TAG: ...
        default: break;
    }
}
```

The compiler is not certified or extracted from the proofs, so we choose an implementation that is, as much as reasonably possible, "clearly correct."

We defer the benchmarks of the compiler to Section 7.8, so that we can also compare our implementation to the implementation of our last abstract machine, which will be presented next (Chapter 7).

Chapter 7

The SECD machine

We have seen how to construct a moderate extension of the Krivine machine (Chapter 6) to allow the execution of distributed programs. A natural question to ask at this point is whether this is also possible to do for other abstract machines. In particular, it might be interesting to investigate machines that implement the call-by-value evaluation strategy. That is what we will do in this chapter.

Synopsis We present another abstract machine, called DCESH, which models the execution of higher-order programs running in distributed architectures. The machine is conceptually similar to the DKrivine machine, with the difference that it uses call-by-value and is based on a modernised version of the SECD machine. It enriches this version of the SECD machine with the specialised communication features required for implementing the Remote Procedure Call (RPC) mechanism. The key correctness result is that the termination behaviour of the RPC is indistinguishable (bisimilar) to that of a local call. The correctness proofs and the requisite definitions for DCESH and other related abstract machines are formalised using Agda. The most technically challenging part of the formalisation requires the use of the *step-indexed relations* technique [7].

We use the DCESH as a target architecture for compiling a conventional call-by-value functional language ("Floskel") which can be annotated with node information. Benchmarks show that the single-node performance of Floskel is comparable to that of OCaml, a semantically similar language, and that distribution overheads are not excessive.

7.1 TECHNICAL OUTLINE

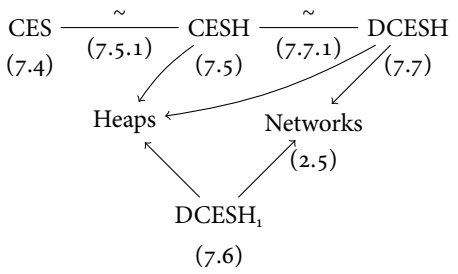
The presentation is divided into the following two main parts:

Compiler and runtime We describe the syntax and implementation of Floskel (Section 7.2), a general-purpose functional language with native RPCs. Our basis is a conventional compiler for such a language, and we show how it is modified to support RPCs and, additionally, *ubiquitous* functions, i.e. functions available on all nodes. Our benchmarks suggest that Floskel's perform-

ance is comparable to the state of the art OCaml compiler for single-node execution.

Abstract machines The semantics of a core of Floskel has been formalised in Agda (Section 7.3) in the form of an abstract machine that can be used to guide an implementation. To achieve this we make gradual refinements to a machine, based on Landin's SECD machine [93], that we call the *CES machine*. First we add heaps for dynamically allocating closures, forming the *CESH machine*; we show that the execution of CES and CESH are bisimilar. We then add communication primitives (synchronous and asynchronous) by instantiating our previously defined general form of networks (Section 2.5) with two different underlying abstract machines. We first illustrate the idea of subsuming function calls by communication protocols by constructing a degenerate distributed machine, DCESH_1, that decomposes some machine instructions into message passing, but only runs on one node. Execution on the fully distributed CESH machine, called DCESH, is shown to be bisimilar to the CESH machine (and thus the CES machine) — our main theoretical result.

The formalisation is organised as follows, where the arrows denote dependence, the lines with ~ symbols bisimulations, and the parenthesised numerals section numbers:



7.2 FLOSKEL: A LOCATION-AWARE LANGUAGE

This section describes the Floskel programming language and its compiler, runtime, and performance.[1]

7.2.1 *Syntax*

At the core of the Floskel language is a call-by-value functional language with user-definable algebraic data types and pattern matching. Floskel is semantically similar to languages in the ML [64] family, and syntactically similar to lan-

[1] Online appendix: `floskel` directory.

guages such as Miranda [134] and Haskell [74]. The main thing that sets Floskel's syntax apart is that pattern matching clauses are given without the leading function name,[2] to avoid repetition, and that type annotations are given after a single colon, as in the following example:

```
map : (a → b) → [a] → [b]
  f []     = []
  f (x::xs) = f x :: map f xs
```

Node annotations An ordinary function definition, like *map*, is a *ubiquitous* function by default. This means that it is made available on all nodes in the system, and a call to such a function is always done locally — a plain old function call.

On the other hand, a function or subterm defined with a *node annotation*, such as

```
query@Database : Query → Response
  x = ...,
```

is *located* and compiled only to the specified node (here *Database*). In the rest of the program *query* can be used like any other function, but the compiler and runtime system treat it differently. A call to *query* from a node other than *Database* is a *remote* call.

Since the programmer can use located functions like any other functions, and this is a functional language, it means that the language has, by necessity, support for higher-order functions across node boundaries. For instance the function

```
f@A : (Query → Response) → X
  q = ... use q ...
```

can be applied to *query* yielding *f query* : *X*.

Node annotations can also be applied to subexpressions, as in the following example:

```
sum []     = 0
    (x::xs) = x + sum xs
xs@A = ...
ys@B = ...
result@C = (sum xs) @A + (sum ys) @B
```

[2] After having received feedback on how confusing this syntax is both from anonymous reviewers and those reading a draft of this dissertation, I am inclined to think that deviating from the "standard" syntax was a mistake, especially in a presentation that has little to do with the syntax of pattern matching clauses. It does, however, give further proof that Wadler's law is true.

Here we want to calculate the sum, on node *C*, of the elements of two lists located on nodes *A* and *B*. If the lists are lengthy, it is better to calculate the sums on *A* and *B*, and to then send the final sum to *C*, since this saves us having to send the full lists over the network.

7.2.2 *Compilation*

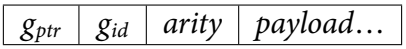
The Floskel compiler[3] currently targets C using the MPI library [65] for communication, though other targets are possible since we do not make use of any features that are unique to our target. Any compatible combination of low-level language and message passing library would work. Most of the compiler's pipeline is standard for a functional language implementation. It works by applying a series of standard transformations to the source program until reaching a level low enough to be straightforwardly translated to C. Since the source language has pattern matching, it first compiles the pattern matching to simple case trees [9]. Local definitions are then lifted to the top-level using lambda lifting [81], and lastly the program is closure converted [108] to support partially applied functions.

Up until the lambda lifting, a node annotation is a constructor in the abstract syntax tree of the language's expressions. The lambda lifter lifts such subexpressions to the top-level such that annotations are afterwards associated with definitions (and not expressions). This is for simplicity: it means that there are fewer cases to consider when we make the annotations work.

The main work specific to Floskel is done in the closure conversion and the runtime system that the compiled programs make use of.

Closures For applications, the closure converter distinguishes between known functions — those that are on the top-level and have a known arity, and unknown functions — those that are provided e.g. as function arguments.

A known function f that is either ubiquitous or available on the same node as the definition that is being compiled is compiled to an ordinary function call if there are enough arguments. If there are not, and the function is ubiquitous we have to construct a *partial application* closure, which contains a pointer to the function and the arguments so far. The compiler maintains the invariant that unknown functions are always in the form of a closure, whose general layout is:



[3] Online appendix: `floskel` directory.

Since the function may require access to the payload of the closure, g_{ptr} is a function of arity *arity* + 1: when applying a closure *cl* as above to arguments $x_1 , \ldots , x_{arity}$, the call becomes $g_{ptr}\ (cl , x_1 , \ldots , x_{arity})$ meaning that the function has access to the payload through *cl*. To construct the initial closure for a partial application of a function f of arity *arity* with *nargs* arguments, we have to conform to this rule, so we construct the closure $(f'_{ptr} , f'_{id} , n , y_1 , \ldots , y_{nargs})$ where $n = arity - nargs$ and f' is a new ubiquitous top-level function defined as follows:

$$
\begin{array}{l}
f'\ cl\ x_1 \ldots x_n\ =\ \mathsf{case}\ cl\ \mathsf{of} \\
\quad (\_\ \_\ \_\ y_1 , \ldots , y_{nargs})\ \rightarrow\ f\ (y_1 , \ldots , y_{nargs} , x_1 , \ldots , x_n)
\end{array}
$$

A family of $apply_i$ functions handle, in a standard way, applications (of i arguments) of unknown functions by inspecting the arity stored in the closure to decide whether to construct a new partial application closure with the additional arguments or to apply the function.

The field f_{id} is an integer identifier assigned to every function at compile time and used as a system-wide identifier if the function is ubiquitous, or a node-specific identifier if not. If there are k ubiquitous functions they are assigned the first k identifiers, and the nodes of the system may use identifiers greater than k for their respective located functions. Determining if a function is ubiquitous is thus a simple comparison: $f_{id} < k$. Additionally, every node has a table of functions that maps ubiquitous or local located function identifiers to local function pointers, which is used by the deserialiser.

If we have a saturated call to a known remote function, we make a call to the function $rApply_{arity}$, defined in the runtime system (to be described). If we have a non-saturated call to a known remote or located function, we construct the closure $(f'_{ptr} , f'_{id} , arity , y_1 , \ldots , y_{nargs})$ where f' is a new ubiquitous top-level function defined as follows:

$$
\begin{array}{l}
f'\ cl\ x_1 \ldots x_n\ =\ \mathsf{case}\ cl\ \mathsf{of} \\
\quad (\_\ \_\ \_\ y_1 , \ldots , y_{nargs})\ \rightarrow \\
\qquad \mathsf{if}\ myNode\ \equiv\ f_{node}\ \mathsf{then} \\
\qquad\quad lookup\ (f_{id})\ (y_1 , \ldots , y_{nargs} , x_1 , \ldots , x_n) \\
\qquad \mathsf{else} \\
\qquad\quad rApply_{arity}\ (f_{node} , f_{id} , y_1 , \ldots , y_{nargs} , x_1 , \ldots , x_n)
\end{array}
$$

Here *myNode* is the identifier of the node the code is currently being run at. If it is the same node as the node of f, we can make an ordinary function call by looking up the function corresponding to f_{id} in the function table. Otherwise we call the runtime system function $rApply_{arity}$.

In this way, we construct a closure for located functions that looks just like the closure of an ubiquitous function, meaning that fewer special cases

are needed in the runtime system — the normal $apply_i$ function works for all closures.

7.2.3 *Runtime*

The runtime system defines a family of ubiquitous functions $rApply_{arity}$, that, as we saw above, are used for remote procedure calls and to construct closures for located functions. The function takes a function identifier, a node identifier, and *arity* arguments. It serialises the arguments and sends them together with the function identifier to the given node:

$$
\begin{array}{l}
rApply\ f_{node}\ f_{id}\ x_1 \ldots x_{arity}\ = \\
\quad \mathsf{send}\ (f_{id}\ ,\ serialise\ (x_1)\ ,\ \ldots,\ serialise\ (x_{arity}))\ \mathsf{to}\ f_{node}; \\
\quad \mathsf{receive}\ answer\ \mathsf{from}\ f_{node}\ \rightarrow \\
\quad answer
\end{array}
$$

When the node f_{node} receives this message, it looks the function up in its function table, calls it with the deserialised arguments, and sends back the result:

$$
\begin{array}{l}
\mathsf{receive}\ (f_{id}\ ,\ y_1\ ,\ \ldots\ ,\ y_{arity})\ \mathsf{from}\ remoteNode\ \rightarrow \\
\quad \mathsf{let}\ result\ =\ lookup\ (f_{id})\ (deserialise\ (y_1)\ ,\ \ldots,\ deserialise\ (y_{arity})) \\
\quad \mathsf{in\ send}\ result\ \mathsf{to}\ remoteNode
\end{array}
$$

Serialisation In a remote function call the arguments may be values from arbitrary algebraic data types (like lists and trees), in addition to primitive types and functions.

The serialisation of a primitive type is the identity function, while algebraic data types require a traversal and flattening of the heap structure. We use tags in the lower bits of a value's field to differentiate between pointers and non-pointers, which makes this flattening straightforward. The interesting part of serialisation is how to handle closures, both in the case of ubiquitous and located functions.

For closures around ubiquitous functions, we serialise the closure almost as is, but use the function identifiers to resolve the function pointer on the receiving node, as the pointer is not guaranteed to be the same on each node.

To handle located functions, the most straightforward implementation is to use "mobilised" closures that work by exchanging the located function with a ubiquitous function that calls *rApply* to perform the remote procedure call — a sort of lambda-lifting for locations. This is what our implementation currently does. Our formalisation will describe an optimised variant of this scheme,

which instead saves the closure on the sending node and sends a pointer to that. The optimised scheme means that we do not unnecessarily send closures containing (potentially large) arguments that are going to end up on the node they originated from anyway. The cost of this optimisation, however, is that it requires us to keep track of heap-allocated pointers across node boundaries using distributed garbage collection. The serialisation currently implemented does not require such garbage collection — it only requires local collections — but may be slow when dealing with large data. Our compiler uses, for simplicity of implementation, the Boehm-Demers-Weiser conservative garbage collector [1] for local garbage collection.

In detail, to serialise a closure

f_{ptr}	f_{id}	*arity*	*payload…*

we put a placeholder, `CL`, in the place of f_{ptr}:

`CL`	f_{id}	*arity*	*payload'* …

where *payload'* represents the serialised payload and `CL` is a tag that can be used to identify that this is a closure. To deserialise this on the receiving end, we look up the function pointer associated with f_{id} in the ubiquitous function table and substitute that for `CL`.

7.2.4 *Performance benchmarks*

Single-node Before we measure the performance of the implementation of the native RPC, we analyse how the single-node performance is affected by the distribution overhead even if it is not used — is it feasible for a general-purpose language to be based on the DCESH?

Table 7.1 shows absolute and relative timings of a number of small benchmarks using integers, lists, trees, recursion, and a small amount of output for printing results. We compare the performance of Floskel programs compiled with our compiler, and equivalent OCaml programs compiled using `ocaml-opt`, a high-performance native-code compiler. Since our compiler targets C, we further compile the generated files to native code using `gcc -O2`. We can see that the running time of programs compiled with our compiler is between two and six times greater than that of those compiled with `ocamlopt`. These results should be viewed in the light of the fact that our compiler only does a minimal amount of optimisation, whereas a considerable amount of time and effort has been put into `ocamlopt`.

Moreover, our compiler only produces C code rather than assembly, which is another potential source of inefficiencies.

	trees	nqueens	qsort	primes	tak	fib
Floskel	91.2s	12.2s	9.45s	19.3s	16.5s	10.0s
ocamlopt	43.0s	3.10s	3.21s	6.67s	2.85s	1.68s
relative	2.12	3.94	2.94	2.9	5.77	5.95

Table 7.1: Floskel single-node performance

The running time of programs compiled with our compiler is between two and six times greater than that of those compiled with `ocamlopt`*.*

	trees	nqueens	qsort	primes	tak	fib
μs/remote call	618	382	4.77	13.4	6.94	6.87
B/remote call	1490	25.8	28.1	27.0	32.0	24.0

Table 7.2: Floskel distribution overheads

The first row, μs*/remote call, is obtained by running the same benchmark with and without node annotations, taking the delta-time of those two, and then dividing by the number of remote invocations in the distributed program. The second row measures the amount of data transmitted per remote invocation, in bytes. We can see that we can do between* 1600 *and* 210000 *remote invocations per second on this set of benchmarks running on our machine.*

Distribution overhead We measure the overhead of our implementation of native remote procedure calls by running the same programs as for the single-node benchmarks, but distributed to between two and nine nodes. The distribution is done by adding node annotations in ways that generate large amounts of communication. We run the benchmarks on a single physical computer with local virtual nodes, which means that the contributions of network latencies are factored out. These measurements give the overhead of the other factors related to remote calls, like serialisation and deserialisation. The results are shown in Table 7.2. The first row, μs/remote call, is obtained by running the same benchmark with and without node annotations, taking the delta-time of those two, and then dividing by the number of remote invocations in the distributed program. The second row measures the amount of data transmitted per remote invocation, in bytes.

It is expected that this benchmark depends largely on the kinds of invocations that are done, since it is more costly to serialise and send a long list or a big closure than an integer. The benchmark hints at this; the program with the biggest messages is the slowest.

An outlier is the `nqueens` benchmark, which does not do remote invocations with large arguments, but still has a high overhead per call. This is prob-

ably because it intentionally uses many localised functions, meaning that its distribution is extremely fine-grained.

The full distributed Floskel programs are given in the online appendix.[4] The single-node versions are the same, only without the location specifiers, and the OCaml versions are literal translations thereof.

7.3 ABSTRACT MACHINE FORMALISATION

Having introduced the programming language, its compiler, and its runtime system, we now present the theoretical foundation for the correctness of the compiler.[5] We start with the standard abstract machine model of call-by-value computation, which we refine, in several steps, into increasingly expressive abstract machines with heap and networking capabilities, while showing that correctness is preserved along the way, via bisimulation results. All definitions and theorems are formalised using the proof assistant Agda, the syntax of which we will follow. Note that we shall not formalise the whole of Floskel but only a core language which coincides with Plotkin's (untyped) call-by-value PCF [117].

7.4 THE CES MACHINE

The starting point is a variation of Landin's SECD machine [93] called Modern SECD [96]. Since this variation does not use a dump, we will call our implementation the CES machine.

The Modern SECD machine can be traced back to the CEK machine of Felleisen [38] and to the SECD variation of Henderson [69]. Just like the CEK, the Modern SECD machine places the continuations that originally resided in the dump on the stack, which simplifies the machine configurations by obviating the need for a dump. But Modern SECD goes further; like Henderson's machine, it uses a bytecode for the control component. Although this means that a compilation step is required before running a term, which might seem like a complication, it means that we do not require as many different kinds of continuations as the CEK machine. For example, the continuations related to the evaluation of a function and its argument can now be encoded directly in the control component just by juxtaposition of code.

A CES[6] configuration (*Config*) is a tuple consisting of a fragment of code (*Code*), an environment (*Env*), and a stack (*Stack*). Evaluation begins with an

[4] Online appendix: `floskel/benchmarks` directory.

[5] Online appendix: `secd/formalisation` directory.

[6] Online appendix: `secd/formalisation` directory, `CES` module.

empty stack and environment, and then follows a *stack discipline*. Subterms push their result on the stack so that their superterms can consume them. When (and if) the evaluation terminates, the program's result is the sole stack element.

The machine operates on bytecode and does not directly interpret the source terms, so the terms need to be compiled before they can be executed.[7] The main work of compilation is done by the function *compile'*, which takes a term t and a fragment of code c used as a postlude. The postlude parameter lets us compile terms without using a costly bytecode append function; *compile'* uses a *difference list* [76] representation where append is a constant-time operation. The bold upper-case names (**CLOS**, **VAR**, and so on) are the bytecode instructions, which are sequenced using _;_. Instructions can be seen to correspond to the constructs of the source language, sequentialised.

```
compile' : Term → Code → Code
compile' (ƛ t)       c = CLOS (compile' t RET) ; c
compile' (t $ t')    c = compile' t (compile' t' (APPL ; c))
compile' (var x)     c = VAR x ; c
compile' (lit n)     c = LIT n ; c
compile' (op f t t') c = compile' t' (compile' t (OP f ; c))
compile' (ifo b then t else f) c =
  compile' b (COND (compile' t c) (compile' f c))
```

Example 7.4.1. To compile a term t we supply **END** as a postlude: *compile* t = *compile'* t **END**. The term $t = (\lambda x.\ x)\ (\lambda x\ y.\ x)$ is compiled as follows:

```
compile ((ƛ var 0) $ (ƛ (ƛ var 1))) = CLOS (VAR 0 ; RET) ;
  CLOS (CLOS (VAR 1 ; RET) ; RET) ; APPL ; END
```

Environments (*Env*) are lists of values (*List Value*), which are either natural numbers (**nat** n) or closures (**clos** *cl*). A closure (*Closure*) is a fragment of code paired with an environment (*Code* × *Env*). Stacks (*Stack*) are lists of stack elements (*List StackElem*), which are either values (**val** v) or continuations (**cont** *cl*), represented by closures.

Figure 7.1 shows the definition of the transition relation for configurations of the CES machine. A note on the Agda syntax is that the instruction constructor names are overloaded as constructors for the relation; their usage is disambiguated by context. Arguments in curly braces are *implicit* and can be automatically inferred. Propositional equality is written _≡_.

[7] Online appendix: `secd/formalisation` directory, `MachineCode` module.

```
data _⟶CES_ : Rel Config Config where
  VAR      : ∀ {n c e s v} → lookup n e ≡ just v →          (VAR n ; c , e , s) ⟶CES (c , e , val v :: s)
  CLOS     : ∀ {c' c e s} →                                 (CLOS c' ; c , e , s) ⟶CES (c , e , val (clos (c' , e)) :: s)
  APPL     : ∀ {c e v c' e' s} →  (APPL ; c , e , val v :: val (clos (c' , e')) :: s) ⟶CES (c' , v :: e' , cont (c , e) :: s)
  RET      : ∀ {e v c e' s} →                  (RET , e , val v :: cont (c , e') :: s) ⟶CES (c , e' , val v :: s)
  LIT      : ∀ {n c e s} →                                  (LIT n ; c , e , s) ⟶CES (c , e , val (nat n) :: s)
  OP       : ∀ {f c e n₁ n₂ s} → (OP f ; c , e , val (nat n₁) :: val (nat n₂) :: s) ⟶CES (c , e , val (nat (f n₁ n₂)) :: s)
  COND-0   : ∀ {c c' e s} →                          (COND c c' , e , val (nat 0) :: s) ⟶CES (c , e , s)
  COND-1+n : ∀ {c c' e n s} →                  (COND c c' , e , val (nat (1 + n)) :: s) ⟶CES (c' , e , s)
```

Figure 7.1: The transition relation of the CES machine

The stack discipline is clear in this definition. When e.g. VAR is executed, the machine looks up the value of the variable in the environment and pushes it on the stack. A somewhat subtle part of the relation is the interplay between the APPL instruction and the RET instruction. When performing an application, two values are required on the stack, one of which has to be a closure. The machine enters the closure, adding the value to the environment, and pushes a return continuation on the stack. The code inside a closure is terminated by a RET instruction, so once the machine has finished executing the closure (and thus produced a value on the stack), that value is returned to the continuation.

The stack discipline is clear in the definition of the transition relation. When e.g. **VAR** is executed, the CES machine looks up the value of the variable in the environment and pushes it on the stack. A somewhat subtle part of the relation is the interplay between the **APPL** instruction and the **RET** instruction. When performing an application, two values are required on the stack, one of which has to be a closure. The machine enters the closure, adding the value to the environment, and pushes a return continuation on the stack. The code inside a closure is terminated by a **RET** instruction, so once the machine has finished executing the closure (and thus produced a value on the stack), that value is returned to the continuation. It is straightforward to prove, by cases on the transitions, that the CES machine is deterministic, i.e. that there is at most one transition from any given state.

Example 7.4.2. We trace the execution of Example 7.4.1 defined above, which exemplifies how returning from an application works.[8] Here we write $a \xrightarrow[CES]{} \langle\, x \,\rangle\, b$ meaning that the machine uses rule x to transition from a to b.

$$
\begin{aligned}
&\mathsf{let}\ c_1 \;=\; \mathbf{VAR}\ o\ \mathbf{;}\ \mathbf{RET} \\
&\quad\ c_2 \;=\; \mathbf{CLOS}\ (\mathbf{VAR}\ 1\ \mathbf{;}\ \mathbf{RET})\ \mathbf{;}\ \mathbf{RET} \\
&\quad\ cl_1 \;=\; \mathbf{val}\ (\mathbf{clos}\ (c_1\,,\ []));\, cl_2 \;=\; \mathbf{val}\ (\mathbf{clos}\ (c_2\,,\ [])) \\
&\mathsf{in}\ (\mathbf{CLOS}\ c_1\ \mathbf{;}\ \mathbf{CLOS}\ c_2\ \mathbf{;}\ \mathbf{APPL}\ \mathbf{;}\ \mathbf{END}\,,\ []\,,\ []) \\
&\xrightarrow[CES]{} \langle\ \mathbf{CLOS}\ \rangle \quad (\mathbf{CLOS}\ c_2\ \mathbf{;}\ \mathbf{APPL}\ \mathbf{;}\ \mathbf{END}\,,\ []\,,\ [\ cl_1\ ]) \\
&\xrightarrow[CES]{} \langle\ \mathbf{CLOS}\ \rangle \quad (\mathbf{APPL}\ \mathbf{;}\ \mathbf{END}\,,\ []\,,\ [\ cl_2\,,\ cl_1\ ]) \\
&\xrightarrow[CES]{} \langle\ \mathbf{APPL}\ \rangle \quad (\mathbf{VAR}\ o\ \mathbf{;}\ \mathbf{RET}\,,\ [\ cl_2\ ]\,,\ [\ \mathbf{cont}\ (\mathbf{END}\,,\ [])\ ]) \\
&\xrightarrow[CES]{} \langle\ \mathbf{VAR\ refl}\ \rangle\ (\mathbf{RET}\,,\ [\ cl_2\ ]\,,\ [\ cl_2\,,\ \mathbf{cont}\ (\mathbf{END}\,,\ [])\ ]) \\
&\xrightarrow[CES]{} \langle\ \mathbf{RET}\ \rangle \quad (\mathbf{END}\,,\ []\,,\ [\ cl_2\ ])
\end{aligned}
$$

The final result is therefore the second closure, cl_2.

The CES machine *terminates with a value* v, written $cfg \downarrow_{CES} v$ if it, through the reflexive transitive closure of $\xrightarrow[CES]{}$, reaches the end of its code fragment with an empty environment, and v as its sole stack element. It *terminates*, written $cfg \downarrow_{CES}$ if there exists a value v such that it terminates with the value v. It *diverges*, written $cfg \uparrow_{CES}$ if it is possible to take another step from any configuration reachable from the reflexive transitive closure of $\xrightarrow[CES]{}$.[9]

We do not prove that the compilation of call-by-value PCF to the CES machine is correct here, as it is — as mentioned in Chapter 5 — a standard result [41, 33].

[8] Online appendix: `secd/formalisation` directory, `Trace` module.

[9] Online appendix: `secd/formalisation` directory, `CES.Properties` module.

```
data _⟶_ : Rel Config Config where
  ...
  CLOS : ∀ {c' c e s h} → let (h' , ptr_cl) = h ▷ (c' , e) in
    (CLOS c' ; c , e , s , h) ⟶ (c , e , val (clos ptr_cl) :: s , h')
  APPL : ∀ {c e v ptr_cl c' e' s h} → h ! ptr_cl ≡ just (c' , e') →
    (APPL ; c , e , val v :: val (clos ptr_cl) :: s , h) ⟶ (c' , v :: e' , cont (c , e) :: s , h)
```

Figure 7.2: The transition relation of the CESH machine (excerpt)

To build a closure, the machine allocates it in the heap, using the _▷_ function, which, given a heap and an element, gives back an updated heap and a pointer to the element. When performing an application, the machine has a pointer *to a closure, so it looks it up in the heap using the _!_ function, which, given a heap and a pointer, gives back the element that the pointer points to (if it exists).*

7.5 CESH: A HEAP MACHINE

In a compiler implementation of the CES machine targeting a low-level language, closures have to be dynamically allocated in a heap. However, the CES machine does not make this dynamic allocation explicit. We will now make it explicit by defining a new machine, called the CESH, which is a CES machine with an extra heap component in its configuration.[10] While heaps are not strictly necessary for a *presentation* of the CES machine, they are of great importance to us. The distributed machine that we will later define needs heaps for persistent storage of data, and the CESH machine forms an intermediate step between that and the CES machine. A CESH configuration is defined as

$$Config = Code \times Env \times Stack \times Heap\ Closure$$

where *Heap* is a type constructor for heaps parameterised by the type of its content.[11] The only difference in the definition of the configuration constituents, compared to the CES machine, is that a closure value (the **clos** constructor of the *Value* type) does not contain an actual closure, but just a pointer (*Ptr*). The stack is as in the CES machine.

Figure 7.2 shows those rules of the CESH machine that are significantly different from the CES: **CLOS** and **APPL**. To build a closure, the CESH allocates it in the heap, using the _▷_ function, which, given a heap and an element, gives back an updated heap and a pointer to the element. When performing an application, the machine has a *pointer* to a closure, so it looks it up in the

[10] Online appendix: `secd/formalisation` directory, `CESH` module.

[11] Online appendix: `secd/formalisation` directory, `Heap` module.

heap using the _!_ function, which, given a heap and a pointer, gives back the element that the pointer points to (if it exists).

A CESH configuration *cfg* can *terminate with a value* v, written as $cfg \downarrow_{CESH} v$, *terminate* ($cfg \downarrow_{CESH}$), or *diverge* ($cfg \uparrow_{CESH}$).[12] These are analogous to the definitions for the CES machine, except that the CESH machine is allowed to terminate with *any* heap:

$$cfg \downarrow_{CESH} v = \exists\, \lambda\, h \to cfg \xrightarrow[CESH]{}^{*} (\mathbf{END}\,,\, []\,,\, [\ \mathbf{val}\ v\ ]\,,\, h)$$

7.5.1 *Correctness*

To show that our definition of the machine is correct, we construct a bisimulation between the CES and CESH, which given the similarity between the two machines, is almost equality. The difference is dealing with closure values, since the CESH stores pointers rather than closures. The relation for closure values must be parameterised by the heap of the CESH configuration, where the (dereferenced) value of the closure pointer is related to the CES closure.

In Agda, the relation is constructed separately for the different components of the machine configurations.[13] Since they run the same bytecode, the relation for code is equality.

```
R_Code : Rel Code Code
R_Code c₁ c₂ = c₁ ≡ c₂
```

For closures it is defined component-wise. Since we have used the same names for some of the components of the CES and CESH machines, we qualify them, using Agda's qualified imports, by prepending *CES.* and *CESH.* to their names. These components may contain values, so we have to parameterise the relations by a closure heap (here *ClosHeap = Heap CESH.Closure*).

```
R_Env : ClosHeap → Rel CES.Env CESH.Env
R_Clos : ClosHeap → Rel CES.Closure CESH.Closure
R_Clos h (c₁ , e₁) (c₂ , e₂) = R_Code c₁ c₂ × R_Env h e₁ e₂
```

Values are related only if they have the same head constructor and related constituents: if the two values are number literals, they are related if they are equal; a CES closure and a pointer are related only if the pointer leads to a CESH closure that is in turn related to the CES closure.

[12] Online appendix: `secd/formalisation` directory, `CESH.Properties` module.
[13] Online appendix: `secd/formalisation` directory, `CESH.Simulation` module.

$$
\begin{aligned}
&R_{Val} \;:\; \mathit{ClosHeap} \to \mathit{Rel}\ \mathit{CES.Value}\ \mathit{CESH.Value} \\
&R_{Val}\ h\ (\mathbf{nat}\ n_1)\ (\mathbf{nat}\ n_2) \;=\; n_1 \equiv n_2 \\
&R_{Val}\ h\ (\mathbf{nat}\ \_)\ (\mathbf{clos}\ \_) \;=\; \bot \\
&R_{Val}\ h\ (\mathbf{clos}\ \_)\ (\mathbf{nat}\ \_) \;=\; \bot \\
&R_{Val}\ h\ (\mathbf{clos}\ c_1)\ (\mathbf{clos}\ ptr) \;=\; \exists\ \lambda\ c_2 \to \\
&\quad h\ !\ ptr \equiv \mathbf{just}\ c_2 \times R_{Clos}\ h\ c_1\ c_2
\end{aligned}
$$

Environments are related if they have the same list spine and their values are pointwise related.

$$
\begin{aligned}
&R_{Env}\ h\ []\ \ \ \ \ \ \ \ []\ \ \ \ \ \ \ \ \;=\; \top \\
&R_{Env}\ h\ []\ \ \ \ \ \ \ \ (x_2 :: e_2) \;=\; \bot \\
&R_{Env}\ h\ (x_1 :: e_1)\ [] \ \ \ \ \ \ \ \ \;=\; \bot \\
&R_{Env}\ h\ (x_1 :: e_1)\ (x_2 :: e_2) \;=\; R_{Val}\ h\ x_1\ x_2 \times R_{Env}\ h\ e_1\ e_2
\end{aligned}
$$

Note that we use $\top$ and $\bot$ to represent true and false, represented in Agda by the unit type and the uninhabited type. The relation on stacks is defined similarly, using the relation on values and continuations. Finally, two configurations are R_{Cfg}-related if their components are related. Here we pass the heap of the CESH configuration as an argument to the environment and stack relations.

$$
\begin{aligned}
&R_{Cfg} \;:\; \mathit{Rel}\ \mathit{CES.Config}\ \mathit{CESH.Config} \\
&R_{Cfg}\ (c_1\ ,\ e_1\ ,\ s_1)\ (c_2\ ,\ e_2\ ,\ s_2\ ,\ h_2) \;= \\
&\quad R_{Code}\ c_1\ c_2 \times R_{Env}\ h_2\ e_1\ e_2 \times R_{Stack}\ h_2\ s_1\ s_2
\end{aligned}
$$

In the formalisation we define heaps and their properties *abstractly*, rather than using a specific heap implementation.[14] The first key property we require is that dereferencing a pointer in a heap where that pointer was just allocated with a value gives back the same value:

$$
\forall\ h\ x \to \mathsf{let}\ (h'\ ,\ ptr) \;=\; h \triangleright x\ \mathsf{in}\ h'\ !\ ptr \equiv \mathbf{just}\ x
$$

Following the proof structure used for Krivine nets (Chapter 6), we will require a preorder $\subseteq$ for *subheaps*. The intuitive reading for $h \subseteq h'$ is that h' can be used where h can, i.e. that h' contains at least the allocations of h. The formal definition is:

$$
\begin{aligned}
h \subseteq h' \;=\; \forall\ ptr\ \{x\} \to h\ \ !\ ptr \equiv \mathbf{just}\ x \to \\
\qquad\qquad\qquad\qquad\qquad\qquad h'\ !\ ptr \equiv \mathbf{just}\ x
\end{aligned}
$$

[14] Online appendix: `secd/formalisation` directory, `Heap` module.

The second key property that we require of a heap implementation is that allocation does not overwrite any previously allocated memory cells ($proj_1$ means first projection):

$$\forall\ h\ x \to h \subseteq proj_1\ (h \triangleright x)$$

Also like in Chapter 6, we prove the monotonicity of R_{Cfg} with respect to heap inclusion, i.e.

Theorem 7.5.1. For any two heaps h and h' such that $h \subseteq h'$, if $R_{Cfg}\ cfg\ (c,e,s,h)$, then $R_{Cfg}\ cfg\ (c,e,s,h')$.

Our first correctness result is the following:

Theorem 7.5.2. R_{Cfg} is a *Simulation* relation.[15]

The proof is by cases on the *CES* transition, and, in each case, the *CESH* machine can make analogous transitions. The property mentioned above is then used to show that R_{Cfg} is preserved.

It is helpful to introduce the notion of a *presimulation* relation — a generalisation of a simulation relation that does not require the target states of the transitions to be related — defined as:[16]

$$\begin{aligned} &Presimulation\ \_\longrightarrow\_\ \_\longrightarrow'\_\ \_R\_\ = \\ &\quad \forall\ a\ a'\ b \to (a \longrightarrow a') \to a\ R\ b \to \exists\ \lambda\ b' \to (b \longrightarrow' b') \end{aligned}$$

Theorem 7.5.3. The inverse of R_{Cfg} is a *Presimulation*.[17]

In general, the following holds:

Theorem 7.5.4. If R is a *Simulation* between relations $\longrightarrow$ and $\longrightarrow'$, R^{-1} is a *Presimulation*, and $\longrightarrow'$ is deterministic at states b related to some a, then R^{-1} is a *Simulation*.[18]

In Agda this is:

[15] Online appendix: `secd/formalisation` directory, `CESH.Simulation` module.
[16] Online appendix: `secd/formalisation` directory, `Relation` module.
[17] Online appendix: `secd/formalisation` directory, `CESH.Presimulation` module.
[18] Online appendix: `secd/formalisation` directory, `Relation` module.

```
presimulation-to-simulation
  : (_R_ : Rel A B)
  → Simulation ⟶ ⟶' _R_
  → Presimulation ⟶' ⟶ (_R_ ⁻¹)
  → (∀ a b → a R b → ⟶' is-deterministic-at b)
  → Simulation ⟶' ⟶ (_R_ ⁻¹)
presimulation-to-simulation R sim presim det = sim⁻¹
  where
    sim⁻¹ : Simulation ⟶' ⟶ (R ⁻¹)
    sim⁻¹ b b' a bstep aRb
      = let (a' , astep)          = presim b b' a bstep aRb
            (b'' , bstep' , a'Rb'') = sim a a' b astep aRb
        in a' , astep , subst (λ b'' → R a' b'')
                              (sym (det a b aRb bstep bstep'))
                              a'Rb''
```

Here $\longrightarrow$: *Rel A A* and $\longrightarrow'$: *Rel B B* are additional parameters, *subst* a function that substitutes equal for equal in a term's type using a propositional equality (in this case obtained from the determinism *det*), and *sym* is the symmetry property of propositional equality.

Theorem 7.5.5. R_{Cfg} is a *Bisimulation*.[19]

This follows from *presimulation-to-simulation*, because we have already established that the CESH is deterministic. The idea that the backward simulation can be obtained cheaply in a deterministic setting is also used by Leroy [94], who notes that the forward simulation is often easier to prove directly than the backward simulation. Our experience confirms this.

A corollary of the above theorem is the following:

Corollary 7.5.6. If $R_{Cfg}\ cfg_1\ cfg_2$ then $cfg_1\ \downarrow_{CES}\ \mathbf{nat}\ n \leftrightarrow cfg_2\ \downarrow_{CESH}\ \mathbf{nat}\ n$ and $cfg_1\ \uparrow_{CES}\ \leftrightarrow cfg_2\ \uparrow_{CESH}$.

To finalise the proof we note that there are configurations in R_{Cfg}. One such example is the initial configuration for a fragment of code: For any c, we have $R_{Cfg}\ (c\ ,\ []\ ,\ []\ )\ (c\ ,\ []\ ,\ []\ ,\ \varnothing)$ (where $\varnothing$ is the empty heap).

7.6 DCESH$_1$: A TRIVIALLY DISTRIBUTED MACHINE

In higher-order distributed programs containing location specifiers, we will sometimes encounter situations where a function is not available locally. For

[19] Online appendix: `secd/formalisation` directory, `CESH.Bisimulation` module.

example, when evaluating the function f in the term $(f\ @\ A)\ (g\ @\ B)$, we may need to apply the remotely available function g. Our general idea is to do this by decomposing some instructions into communication. In the example, the function f may send a message requesting the evaluation of g, meaning that the **APPL** instruction is split into a pair of instructions: **APPL-send** and **APPL-receive**.

This section outlines an abstract machine, called DCESH_1, which decomposes all application and return instructions into communication.[20] The machine is trivially distributed, because it runs as the sole node in a network, sending messages only to itself. Although it is not used as an intermediate step for the proofs, it is included because it illustrates this decomposition.

A configuration of the DCESH_1 machine (*Machine*) is a tuple consisting of a possibly running thread (*Maybe Thread*), a closure heap (*Heap Closure*), and a "continuation heap" (*Heap* (*Closure* × *Stack*)). Since the language is sequential we have at most one thread running at once. The thread resembles a CES configuration, $\mathit{Thread} = \mathit{Code} \times \mathit{Env} \times \mathit{Stack}$, but stacks are defined differently. A stack is now a list of values paired with an optional pointer (into the continuation heap), $\mathit{Stack} = \mathit{List\ Val} \times \mathit{Maybe\ ContPtr}$ (*ContPtr* is a synonym for *Ptr*). When performing an application, when CES would push a continuation on the stack, the DCESH_1 machine is going to stop the current thread and send a message, which means that it has to save the continuation and the remainder of the stack in the heap for them to persist the thread's lifetime.

The optional pointer in *Stack* is an element at the *bottom* of the list of values. Comparing it to the definition of the CES machine, where stacks are lists of either values or continuations (which are closures), we can picture their relation: Whereas the CES machine stores the values and continuations in a single, contiguous stack, the DCESH_1 machine stores first a contiguous block of values until reaching a continuation, at which point it stores a pointer to the continuation closure and the rest of the stack.

The definition of closures, values, and environments are otherwise just like in the CESH machine. The machine communicates with itself using two kinds of messages, **APPL** and **RET**, corresponding to the instructions that we are replacing with communication.

Figure 7.3 defines the transition relation for the DCESH_1 machine, written $m \xrightarrow{tmsg} m'$ for a tagged message *tmsg* and machine configurations m and m'. Most transitions are the same as in the CESH machine, framed with the additional heaps and the **just** meaning that the thread is running. We elide them for brevity.

[20] Online appendix: `secd/formalisation` directory, `DCESH1` module.

The interesting rules are the decomposed rules for application and return. When an application is performed, an **APPL** message containing a pointer to the closure to apply, the argument value and a pointer to a return continuation (which is first allocated) is sent, and the thread is stopped (**nothing**). We call such a machine *inactive*. The machine can receive an application message if the thread is not running. When that happens, the closure pointer is dereferenced and entered, adding the received argument to the environment. The stack is left empty apart from the continuation pointer of the received message. When returning from a function application, the machine sends a return message containing the continuation pointer and the value to return.

On the receiving end of that communication, it dereferences the continuation pointer and enters it, putting the result value on top of the stack.

Example 7.6.1. We trace the execution of Example 7.4.1 in a synchronous network of nodes indexed by the unit type. Heaps with pointer mappings are written $\{ptr \mapsto element\}$. The last list shown in each step is the message list of the asynchronous network.

$$
\begin{aligned}
&\mathsf{let}\ h_{cl} \quad = \{ptr_1 \mapsto (c_1\ ,\ [])\} \\
&\quad\ h'_{cl} \quad = \{ptr_1 \mapsto (c_1\ ,\ [])\ ,\ ptr_2 \mapsto (c_2\ ,\ [])\} \\
&\quad\ h_{cnt} \quad = \{ptr_{cnt} \mapsto ((\mathit{END},\ [])\ ,\ [\ ]\ ,\ \mathbf{nothing})\} \\
&\mathsf{in}\ (\mathbf{just}\ (\mathbf{CLOS}\ c_1\ ;\ \mathbf{CLOS}\ c_2\ ;\ \mathbf{APPL}\ ;\ \mathbf{END}\ ,\ [\ ]\ ,\ [\ ]\ ,\ \mathbf{nothing})\ ,\ \varnothing\ ,\ \varnothing)\ ,\ [\ ] \\
&\longrightarrow\langle\ \mathbf{step\ CLOS}\ \rangle \\
&(\mathbf{just}\ (\mathbf{CLOS}\ c_2\ ;\ \mathbf{APPL}\ ;\ \mathbf{END}\ ,\ [\ ]\ ,\ [\ \mathbf{clos}\ ptr_1\ ]\ ,\ \mathbf{nothing})\ ,\ h_{cl}\ ,\ \varnothing)\ ,\ [\ ] \\
&\longrightarrow\langle\ \mathbf{step\ CLOS}\ \rangle \\
&(\mathbf{just}\ (\mathbf{APPL}\ ;\ \mathbf{END}\ ,\ [\ ]\ ,\ [\ \mathbf{clos}\ ptr_2\ ,\ \mathbf{clos}\ ptr_1\ ]\ ,\ \mathbf{nothing})\ ,\ h'_{cl}\ ,\ \varnothing)\ ,\ [\ ] \\
&\longrightarrow\langle\ \mathbf{step\ APPL\text{-}send}\ \rangle \\
&(\mathbf{nothing}\ ,\ h'_{cl}\ ,\ h_{cnt})\ ,\ [\ \mathbf{APPL}\ ptr_1\ (\mathbf{clos}\ ptr_2)\ ptr_{cnt}\ ] \\
&\longrightarrow\langle\ \mathbf{step\ APPL\text{-}receive}\ \rangle \\
&(\mathbf{just}\ (\mathbf{VAR}\ o\ ;\ \mathbf{RET}\ ,\ [\ \mathbf{clos}\ ptr_2\ ]\ ,\ [\ ]\ ,\ \mathbf{just}\ ptr_{cnt})\ ,\ h'_{cl}\ ,\ h_{cnt})\ ,\ [\ ] \\
&\longrightarrow\langle\ \mathbf{step}\ (\mathbf{VAR\ refl})\ \rangle \\
&(\mathbf{just}\ (\mathbf{RET}\ ,\ [\ \mathbf{clos}\ ptr_2\ ]\ ,\ [\ \mathbf{clos}\ ptr_2\ ]\ ,\ \mathbf{just}\ ptr_{cnt})\ ,\ h'_{cl}\ ,\ h_{cnt})\ ,\ [\ ] \\
&\longrightarrow\langle\ \mathbf{step\ RET\text{-}send}\ \rangle \\
&(\mathbf{nothing}\ ,\ h'_{cl}\ ,\ h_{cnt})\ ,\ [\ \mathbf{RET}\ ptr_{cnt}\ (\mathbf{clos}\ ptr_2)\ ] \\
&\longrightarrow\langle\ \mathbf{step\ RET\text{-}receive}\ \rangle \\
&(\mathbf{just}\ (\mathbf{END}\ ,\ [\ ]\ ,\ [\ \mathbf{clos}\ ptr_2\ ]\ ,\ \mathbf{nothing})\ ,\ h'_{cl}\ ,\ h_{cnt})\ ,\ [\ ]
\end{aligned}
$$

Comparing this to Example 7.4.2 we can see that an **APPL-send** followed by an **APPL-receive** amounts to the same thing as the **APPL** rule in the CES machine, and similarly for the **RET** instruction.

$$\textsf{data}\ \_\overset{-}{\rightarrow}\_ : \mathit{Machine} \rightarrow \mathit{Tagged\ Msg} \rightarrow \mathit{Machine} \rightarrow \mathit{Set}\ \textsf{where}$$

...

$$\textbf{APPL-send} : \forall \{c\ e\ v\ ptr_{cl}\ s\ r\ h_{cl}\ h_{cnt}\} \rightarrow \textsf{let}\ (h'_{cnt}\ ,\ ptr_{cnt}) = h_{cnt} \triangleright ((c\ ,\ e)\ ,\ s\ ,\ r)\ \textsf{in}$$

$$(\textbf{just}\ (\textbf{APPL}\ ;\ c\ ,\ e\ ,\ v :: \textbf{clos}\ ptr_{cl} :: s\ ,\ r)\ ,\ h_{cl}\ ,\ h_{cnt}) \xrightarrow{\textbf{send}\ (\textbf{APPL}\ ptr_{cl}\ v\ ptr_{cnt})} (\textbf{nothing}\ ,\ h_{cl}\ ,\ h'_{cnt})$$

$$\textbf{APPL-receive} : \forall \{h_{cl}\ h_{cnt}\ ptr_{cl}\ v\ ptr_{cnt}\ c\ e\} \rightarrow h_{cl}\ !\ ptr_{cl} \equiv \textbf{just}\ (c\ ,\ e) \rightarrow$$

$$(\textbf{nothing}\ ,\ h_{cl}\ ,\ h_{cnt}) \xrightarrow{\textbf{receive}\ (\textbf{APPL}\ ptr_{cl}\ v\ ptr_{cnt})} (\textbf{just}\ (c\ ,\ v :: e\ ,\ []\ ,\ \textbf{just}\ ptr_{cnt})\ ,\ h_{cl}\ ,\ h_{cnt})$$

$$\textbf{RET-send} : \forall \{e\ v\ ptr_{cnt}\ h_{cl}\ h_{cnt}\} \rightarrow$$

$$(\textbf{just}\ (\textbf{RET}\ ,\ e\ ,\ v :: []\ ,\ \textbf{just}\ ptr_{cnt})\ ,\ h_{cl}\ ,\ h_{cnt}) \xrightarrow{\textbf{send}\ (\textbf{RET}\ ptr_{cnt}\ v)} (\textbf{nothing}\ ,\ h_{cl}\ ,\ h_{cnt})$$

$$\textbf{RET-receive} : \forall \{h_{cl}\ h_{cnt}\ ptr_{cnt}\ v\ c\ e\ s\ r\} \rightarrow h_{cnt}\ !\ ptr_{cnt} \equiv \textbf{just}\ ((c\ ,\ e)\ ,\ s\ ,\ r) \rightarrow$$

$$(\textbf{nothing}\ ,\ h_{cl}\ ,\ h_{cnt}) \xrightarrow{\textbf{receive}\ (\textbf{RET}\ ptr_{cnt}\ v)} (\textbf{just}\ (c\ ,\ e\ ,\ v :: s\ ,\ r)\ ,\ h_{cl}\ ,\ h_{cnt})$$

Figure 7.3: The transition relation of the DCESH_1 machine (excerpt)

When an application is performed, an **APPL** *message containing a pointer to the closure to apply, the argument value and a pointer to a return continuation (which is first allocated) is sent, and the thread is stopped (**nothing**). The machine can receive an application message if the thread is not running. When that happens, the closure pointer is dereferenced and entered, adding the received argument to the environment. The stack is left empty apart from the continuation pointer of the received message. When returning from a function application, the machine sends a return message containing the continuation pointer and the value to return. On the receiving end of that communication, it dereferences the continuation pointer and enters it, putting the result value on top of the stack.*

7.7 DCESH: THE DISTRIBUTED CESH MACHINE

We have so far seen two refinements of the CES machine. We have seen CESH, that adds heaps, and DCESH_1, that decomposes instructions into communication in a degenerate network of only one node. Our final refinement is a distributed machine, DCESH, that supports multiple nodes.[21] The main problem that we now face is that there is no centralised heap, but each node has its own local heap. This means that, for supporting higher-order functions across node boundaries, we have to somehow keep references to closures in the heaps of *other* nodes. Another problem is efficiency; we would like a system where we do not pay the higher price of communication for locally running code. The main idea for solving these two problems is to use *remote pointers*, $\mathit{RPtr} = \mathit{Ptr} \times \mathit{Node}$, pointers paired with node identifiers signifying on what node's heap the pointer is located. This solves the heap problem because we always know where a pointer comes from. It can also be used to solve the efficiency problem since we can choose what instructions to run based on whether a pointer is local or remote. The correctness proof of the DCESH_1 will show that whenever a node holds a remote pointer, that pointer is valid on the remote node, meaning that consistency is maintained. If it is local, we run the rules of the CESH machine. If it is remote, we run the decomposed rules of the DCESH_1 machine.

The final extension to the bytecode will add support for location specifiers. We add the instruction **REMOTE** *c i* for the compilation of the term construct *t* **@** *i*. The location specifiers, *t* **@** *i*, are taken to mean that the term *t* should be evaluated on node *i*. As previously mentioned, we require that the terms *t* in all location specification subterms *t* **@** *i* are *closed*. The **REMOTE** *c i* instruction will be used to start running a code fragment *c* on node *i* in the network. We also extend the *compile*' function to handle the new term construct:

$$\mathit{compile}'\ (t\ @\ i)\ c\ =\ \textbf{REMOTE}\ (\mathit{compile}'\ t\ \textbf{RET})\ i\ ;\ c$$

Note that we reuse the **RET** instruction to return from a remote computation.

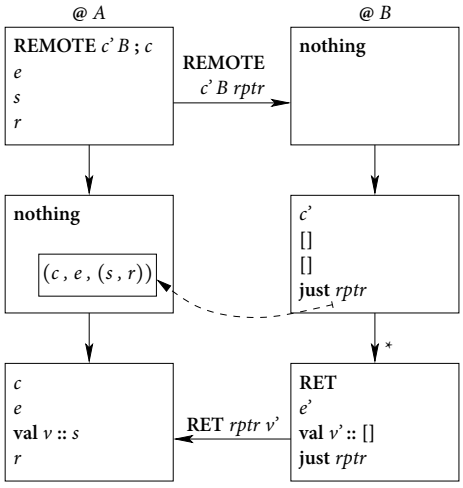
The definition of closures, values, environments and closure heaps are the same as in the CESH machine, but using *RPtr* instead of *Ptr* for closure pointers.

The stack combines the functionality of the CES machine, permitting local continuations, with that of the DCESH_1 machine, making it possible for a stack to end with a continuation on another node. A stack element is a value or a (local) continuation signified by the **val** and **cont** constructors. A stack (*Stack*) is a list of stack elements, possibly ending with a (remote) pointer to a continuation, $\mathit{List}\ \mathit{StackElem} \times \mathit{Maybe}\ \mathit{ContPtr}$ (where $\mathit{ContPtr} = \mathit{RPtr}$). Threads

[21] Online appendix: `secd/formalisation` directory, `DCESH` module.

and machines are defined like in the DCESH_1 machine. The messages that DCESH can send are those of the DCESH_1 machine but using remote pointers instead of plain pointers, plus a message for starting a remote computation, **REMOTE** *c i* $rptr_{cnt}$. Note that sending a **REMOTE** message amounts to sending code in our formalisation, which is something that we would not like to do. However, because no code is generated at run time, every machine can be preloaded with all the bytecode it needs, and the message only needs to contain a *reference* to a fragment of code.

Figure 7.4 defines the transition relation of the DCESH machine, written $i \vdash m \xrightarrow{tmsg} m'$ for a node identifier i, a tagged message *tmsg* and machine configurations m and m'. The parameter i is taken to be the identifier of the node on which the transition is taking place. For local computations, we have rules analogous to those of the CESH machine, so we omit them and show only those for remote computations. The rules use the function $i \vdash h \triangleright x$ for allocating a pointer to x in a heap h and then constructing a remote pointer tagged with node identifier i from it. When starting a remote computation, the machine allocates a continuation in the heap and sends a message containing the code and continuation pointer to the remote node in question. Afterwards the current thread is stopped.



On the receiving end of such a communication, a new thread is started, placing the continuation pointer at the bottom of the stack for the later return to the caller node. To run the apply instruction when the function closure is remote, i.e. its location is *not* equal to the current node, the machine sends a message containing the closure pointer, argument value, and continuation, like in the DCESH_1 machine. On the other end of such a communication, the machine dereferences the pointer and enters the closure with the received value.

$$
\begin{aligned}
&\textsf{data}\ \longrightarrow\!\mathit{Machine}\ (i : \mathit{Node}) : \mathit{Machine} \to \mathit{Tagged\ Msg} \to \mathit{Machine} \to \mathit{Set}\ \textsf{where}\\
&\quad \ldots\\
&\quad \textbf{REMOTE-send} \quad : \forall \{c'\ i'\ c\ e\ s\ r\ h_{cl}\ h_{cnt}\} \to \textsf{let}\ (h'_{cnt}\ ,\ rptr) = i \vdash h_{cnt} \triangleright ((c\ ,\ e)\ ,\ s\ ,\ r)\ \textsf{in}\\
&\qquad i \vdash (\textbf{just}\ (\textbf{REMOTE}\ c'\ i'\ ;\ c\ ,\ e\ ,\ s\ ,\ r)\ ,\ h_{cl}\ ,\ h_{cnt}) \xrightarrow{\textbf{send}\ (\textbf{REMOTE}\ c'\ i'\ rptr)}\\
&\qquad\quad (\textbf{nothing}\ ,\ h_{cl}\ ,\ h'_{cnt})\\
&\quad \textbf{REMOTE-receive} : \forall \{h_{cl}\ h_{cnt}\ c\ rptr_{cnt}\} \to\\
&\qquad i \vdash (\textbf{nothing}\ ,\ h_{cl}\ ,\ h_{cnt}) \xrightarrow{\textbf{receive}\ (\textbf{REMOTE}\ c\ i\ rptr_{cnt})}\\
&\qquad\quad (\textbf{just}\ (c\ ,\ []\ ,\ []\ ,\ \textbf{just}\ rptr_{cnt})\ ,\ h_{cl}\ ,\ h_{cnt})\\
&\quad \textbf{APPL-send} \quad : \forall \{c\ e\ v\ ptr_{cl}\ j\ s\ r\ h_{cl}\ h_{cnt}\} \to i \neq j \to \textsf{let}\ (h'_{cnt}\ ,\ rptr_{cnt}) = i \vdash h_{cnt} \triangleright ((c\ ,\ e)\ ,\ s\ ,\ r)\ \textsf{in}\\
&\qquad i \vdash (\textbf{just}\ (\textbf{APPL}\ ;\ c\ ,\ e\ ,\ \textbf{val}\ v :: \textbf{val}\ (\textbf{clos}\ (ptr_{cl}\ ,\ j)) :: s\ ,\ r)\ ,\ h_{cl}\ ,\ h_{cnt}) \xrightarrow{\textbf{send}\ (\textbf{APPL}\ (ptr_{cl}\ ,\ j)\ v\ rptr_{cnt})}\\
&\qquad\quad (\textbf{nothing}\ ,\ h_{cl}\ ,\ h'_{cnt})\\
&\quad \textbf{APPL-receive} \quad : \forall \{h_{cl}\ h_{cnt}\ ptr_{cl}\ v\ rptr_{cnt}\ c\ e\} \to h_{cl}\ !\ ptr_{cl} \equiv \textbf{just}\ (c\ ,\ e) \to\\
&\qquad i \vdash (\textbf{nothing}\ ,\ h_{cl}\ ,\ h_{cnt}) \xrightarrow{\textbf{receive}\ (\textbf{APPL}\ (ptr_{cl}\ ,\ i)\ v\ rptr_{cnt})}\\
&\qquad\quad (\textbf{just}\ (c\ ,\ v :: e\ ,\ []\ ,\ \textbf{just}\ rptr_{cnt})\ ,\ h_{cl}\ ,\ h_{cnt})\\
&\quad \textbf{RET-send} \quad : \forall \{e\ v\ rptr_{cnt}\ h_{cl}\ h_{cnt}\} \to\\
&\qquad i \vdash (\textbf{just}\ (\textbf{RET}\ ,\ e\ ,\ \textbf{val}\ v :: []\ ,\ \textbf{just}\ rptr_{cnt})\ ,\ h_{cl}\ ,\ h_{cnt}) \xrightarrow{\textbf{send}\ (\textbf{RET}\ rptr_{cnt}\ v)}\\
&\qquad\quad (\textbf{nothing}\ ,\ h_{cl}\ ,\ h_{cnt})\\
&\quad \textbf{RET-receive} \quad : \forall \{h_{cl}\ h_{cnt}\ ptr_{cnt}\ v\ c\ e\ s\ r\} \to h_{cnt}\ !\ ptr_{cnt} \equiv \textbf{just}\ ((c\ ,\ e)\ ,\ s\ ,\ r) \to\\
&\qquad i \vdash (\textbf{nothing}\ ,\ h_{cl}\ ,\ h_{cnt}) \xrightarrow{\textbf{receive}\ (\textbf{RET}\ (ptr_{cnt}\ ,\ i)\ v)}\\
&\qquad\quad (\textbf{just}\ (c\ ,\ e\ ,\ \textbf{val}\ v :: s\ ,\ r)\ ,\ h_{cl}\ ,\ h_{cnt})
\end{aligned}
$$

Figure 7.4: The transition relation of the DCESH machine (excerpt)

When starting a remote computation, the machine allocates a continuation in the heap and sends a message to the remote node in question. On the receiving end of such a communication, a new thread is started, placing the continuation pointer at the bottom of the stack for the later return to the caller node. To run the apply instruction when the function closure is remote, i.e. its location is not *equal to the current node, the machine sends a message containing the closure pointer, argument value, and continuation. After either a remote invocation or a remote application, the machine can return if it has produced a value on the stack and has a remote continuation at the bottom of the stack.*

The bottom remote continuation pointer is set to the received continuation pointer. After either a remote invocation or a remote application, the machine can return if it has produced a value on the stack and has a remote continuation at the bottom of the stack. To do this, a message containing the continuation pointer and the return value is sent to the location of the continuation pointer. When receiving a return message, the continuation pointer is dereferenced and entered with the received value.

A network of abstract machines is obtained by instantiating the *Network* module with the $\_\vdash\_\overset{\_}{\rightarrow}\_$ relation. From here on *SyncNetwork* and *AsyncNetwork* and their transition relations refer to the instantiated versions.

An initial network configuration, given a code fragment c and a node identifier i, is a network where only node i is active, ready to run the code fragment:

$$\begin{aligned}&\mathit{initial\text{-}network}_{Sync} : \mathit{Code} \to \mathit{Node} \to \mathit{SyncNetwork}\\&\mathit{initial\text{-}network}_{Sync}\ c\ i\ =\\&\quad (\lambda\ i' \to (\mathbf{nothing} , \varnothing , \varnothing))\ [\ i \mapsto\ (\mathbf{just}\ (c , [] , [] , \mathbf{nothing}) , \varnothing , \varnothing)\ ]\end{aligned}$$

An initial asynchronous network configuration is one where there are no messages in the message list: $\mathit{initial\text{-}network}_{Async}\ c\ i\ =\ \mathit{initial\text{-}network}_{Sync}\ c\ i , []$.

Unsurprisingly, if all nodes in a synchronous network except one are inactive, then the next step is deterministic. Another key ancillary property of DCESH networks is that synchronous or asynchronous networks for single threaded computations behave essentially the same:[22]

Theorem 7.7.1. In a family of nodes *nodes* where all except one are *inactive*, $(\mathit{nodes} , [])\ \underset{Async}{\longrightarrow}^{+}\ (\mathit{nodes'} , [])$ implies $\mathit{nodes}\ \underset{Sync}{\longrightarrow}^{+}\ \mathit{nodes'}$.

In Agda, this is:

$$\begin{aligned}&\underset{Async}{\longrightarrow}^{+}\text{-}\mathit{to}\text{-}\underset{Sync}{\longrightarrow}^{+} : \forall\ \{\mathit{nodes}\ \mathit{nodes'}\}\ i \to \mathit{all\ nodes\ except\ i\ are\ inactive} \to\\&\quad ((\mathit{nodes} , [])\ \underset{Async}{\longrightarrow}^{+}\ (\mathit{nodes'} , [])) \to (\mathit{nodes}\ \underset{Sync}{\longrightarrow}^{+}\ \mathit{nodes'})\end{aligned}$$

This means that it is enough to deal with the simpler synchronous networks.

DCESH network *nodes* can *terminate with a value* v ($\mathit{nodes}\ \downarrow_{Sync}\ v$), *terminate* ($\mathit{nodes}\ \downarrow_{Sync}$), or *diverge* ($\mathit{nodes}\ \uparrow_{Sync}$).[23] A network terminates with a value v if it can step to a network where only one node is active, and that node has reached the **END** instruction with the value v on top of its stack:

$$\begin{aligned}&\mathit{nodes}\ \downarrow_{Sync}\ v\ =\ \exists\ \lambda\ \mathit{nodes'} \to \mathit{nodes}\ \underset{Sync}{\longrightarrow}^{*}\ \mathit{nodes'} \times\\&\quad \exists\ \lambda\ i \to \mathit{all\ nodes'\ except\ i\ are\ inactive} \times \exists\ \lambda\ \mathit{heaps} \to\\&\quad \mathit{nodes'}\ i \equiv (\mathbf{just}\ (\mathbf{END} , [] , \mathbf{val}\ v :: [] , \mathbf{nothing}) , \mathit{heaps})\end{aligned}$$

The other definitions are analogous to those of the CES and CESH machines.

[22] Online appendix: `secd/formalisation` directory, `DCESH.Properties` module.

[23] Online appendix: `secd/formalisation` directory, `DCESH.Properties` module.

7.7.1 *Correctness*

To prove the correctness of the machine, we will now establish a bisimulation between the CESH and the DCESH machines.

To simplify this development, we extend the CESH machine with a dummy rule for the **REMOTE** *c i* instruction so that both machines run the same bytecode. This rule is almost a no-op, but since we are assuming that the code we run remotely is closed, the environment is emptied, and since the compiled code *c* will end in a **RET** instruction a return continuation is pushed on the stack.

$$(\textbf{REMOTE}\ c'\ i\ ;\ c\ ,\ e\ ,\ s\ ,\ h) \xrightarrow[CESH]{} (c'\ ,\ [\ ]\ ,\ \textbf{cont}\ (c\ ,\ e)\ ::\ s\ ,\ h)$$

The relation that we are about to define[24] is, as before, *almost* equality. But since values may be pointers to closures, it must be parameterised by heaps. A technical problem is that *both* machines use pointers, and the DCESH machine also uses *remote* pointers and has two heaps for each node. The relation must therefore be parameterised by all the heaps in the system. The extra parameter is a synonym for an indexed family of the closure and continuation heaps of the whole network, $Heaps = Node \to DCESH.ClosHeap \times DCESH.ContHeap$. The complexity of this relation justifies our use of mechanised reasoning.

The correctness proof itself is not routine. Simply following the recipe that we used before does not work. In the old proof, there can be no circularity, since that bisimulation was constructed inductively on the structure of the CES configuration. But now both systems, CESH and DCESH, have heaps where there is a potential for circular references (e.g. a closure, residing in a heap, whose environment contains a pointer to itself), preventing a direct proof via structural induction. This is perhaps the most mathematically (and formally) challenging point of the work on the DCESH. The solution lies in using the technique of *step-indexed relations*, adapted to the context of bisimulation relations [7]. We add an additional *rank* parameter that records how many times pointers are allowed to be dereferenced.

The rank is used in defining the relation for closure pointers $R_{rptr_{cl}}$. If the rank is zero, the relation is trivially fulfilled. If the rank is non-zero then three conditions must hold. First, the CESH pointer must point to a closure in the CESH heap; second, the remote pointer of the DCESH network must point to a closure in the heap of the location that the pointer refers to; third, the two closures must be related.

[24] Online appendix: `secd/formalisation` directory, `DCESH.Simulation-CESH` module.

$$
\begin{aligned}
&R_{rptr_{cl}} : \mathbb{N} \to \mathit{CESH.ClosHeap} \to \mathit{Heaps} \to \\
&\quad \mathit{Rel\ CESH.ClosPtr\ DCESH.ClosPtr} \\
&R_{rptr_{cl}}\ o\ \_\ \_\ \_\ \_ = \top \\
&R_{rptr_{cl}}\ (1 + \mathit{rank})\ h\ hs\ ptr_1\ (ptr_2\ ,\ loc) = \\
&\exists_2\ \lambda\ cl_1\ cl_2 \to h\ !\ ptr_1 \equiv \mathbf{just}\ cl_1 \times \\
&\qquad\qquad \mathit{proj}_1\ (hs\ loc)\ !\ ptr_2 \equiv \mathbf{just}\ cl_2 \times \\
&\qquad\qquad R_{Clos}\ \mathit{rank}\ h\ hs\ cl_1\ cl_2
\end{aligned}
$$

The relation for values is also as before, but with the extra parameters. The relation for stack elements $R_{StackElem}$ is almost as before, but now requires that the relation is true for *any* natural number *rank*, i.e. for any finite number of pointer dereferencings.

$$
\begin{aligned}
&R_{StackElem} : \mathit{CESH.ClosHeap} \to \mathit{Heaps} \to \\
&\quad \mathit{Rel\ CESH.StackElem\ DCESH.StackElem} \\
&R_{StackElem}\ h\ hs\ (\mathbf{val}\ v_1)\ \ (\mathbf{val}\ v_2)\ = \\
&\quad \forall\ \mathit{rank} \to R_{Val}\ \mathit{rank}\ h\ hs\ v_1\ v_2 \\
&R_{StackElem}\ h\ hs\ (\mathbf{val}\ \_)\ \ (\mathbf{cont}\ \_)\ = \bot \\
&R_{StackElem}\ h\ hs\ (\mathbf{cont}\ \_)\ \ (\mathbf{val}\ \_)\ = \bot \\
&R_{StackElem}\ h\ hs\ (\mathbf{cont}\ cl_1)\ (\mathbf{cont}\ cl_2) = \\
&\quad \forall\ \mathit{rank} \to R_{Clos}\ \mathit{rank}\ h\ hs\ cl_1\ cl_2
\end{aligned}
$$

The relation for stacks R_{Stack} now takes into account that the DCESH stacks may end in a pointer representing a remote continuation, requiring that the pointer points to something in the continuation heap of the location of the pointer, which is related to the CESH stack element.

$$
\begin{aligned}
&R_{Stack} : \mathit{CESH.ClosHeap} \to \mathit{Heaps} \to \\
&\qquad \mathit{Rel\ CESH.Stack\ DCESH.Stack} \\
&\ldots \\
&R_{Stack}\ h\ hs\ (cont_1 :: s_1)\ ([]\ ,\ \mathbf{just}\ (ptr\ ,\ loc)) = \\
&\quad \exists_2\ \lambda\ cont_2\ s_2 \to \mathit{proj}_2\ (hs\ loc)\ !\ ptr \equiv \mathbf{just}\ (cont_2\ ,\ s_2) \times \\
&\qquad R_{StackElem}\ h\ hs\ cont_1\ (\mathbf{cont}\ cont_2) \times \\
&\qquad R_{Stack}\ h\ hs\ s_1\ s_2
\end{aligned}
$$

Finally, a CESH configuration and a DCESH thread are R_{Thread}-related if the thread is running and the constituents are pointwise related.

$$
\begin{aligned}
&R_{Thread} : \mathit{Heaps} \to \mathit{Rel\ Config\ (Maybe\ Thread)} \\
&R_{Thread}\ hs\ \_\qquad\qquad\qquad \mathbf{nothing}\qquad\qquad\quad = \bot \\
&R_{Thread}\ hs\ (c_1\ ,\ e_1\ ,\ s_1\ ,\ h_1)\ (\mathbf{just}\ (c_2\ ,\ e_2\ ,\ s_2)) = \\
&\quad R_{Code}\ c_1\ c_2 \times (\forall\ \mathit{rank} \to R_{Env}\ \mathit{rank}\ h_1\ hs\ e_1\ e_2) \times \\
&\quad R_{Stack}\ h_1\ hs\ s_1\ s_2
\end{aligned}
$$

Then a CESH configuration is related to a synchronous network R_{Sync} if the network has exactly one running machine that is related to the configuration.

$$
\begin{aligned}
&R_{Sync} \;:\; \mathit{Rel\ Config\ SyncNetwork} \\
&R_{Sync}\ \mathit{cfg\ nodes} \;=\; \exists\, \lambda\, i \to \mathit{all\ nodes\ except\ i\ are\ inactive} \times \\
&\quad R_{Thread}\ (\mathit{proj}_2 \circ \mathit{nodes})\ \mathit{cfg}\ (\mathit{proj}_1\ (\mathit{nodes}\ i))
\end{aligned}
$$

DCESH network heaps are ordered pointwise (called $\subseteq s$ since it is the "plural" of $\subseteq$).

$$
\begin{aligned}
\mathit{hs} \subseteq s\ \mathit{hs'} \;=\; \forall\ i \to \mathsf{let}\ (h_{cl}\ ,\ h_{cnt}) \;&=\; \mathit{hs}\ i \\
(h'_{cl}\ ,\ h'_{cnt}) \;&=\; \mathit{hs'}\ i \\
\mathsf{in}\ h_{cl} \subseteq h'_{cl} \times h_{cnt} &\subseteq h'_{cnt}
\end{aligned}
$$

For any CESH closure heaps h and h' such that $h \subseteq h'$ and families of DCESH heaps hs and hs' such that $hs \subseteq s\ hs'$ the following statements hold:

Lemma 7.7.2. If $R_{Env}\ n\ h\ hs\ e_1\ e_2$ then $R_{Env}\ n\ h'\ hs'\ e_1\ e_2$.

Lemma 7.7.3. If $R_{Stack}\ h\ hs\ s_1\ s_2$ then $R_{Stack}\ h'\ hs'\ s_1\ s_2$.

Theorem 7.7.4. R_{Sync} is a *Simulation* relation.[25]

The proof proceeds by cases on the CESH transition. In each case, the DCESH network can make analogous transitions. The lemmas above are then used to show that R_{Sync} is preserved.

Theorem 7.7.5. The inverse of R_{Sync} is a *Presimulation*.[26]

This leads to, using the *presimulatation-to-simulation* theorem, the main result:

Theorem 7.7.6. R_{Sync} is a *Bisimulation*.[27]

As immediate corollaries under the assumption that $R_{Sync}\ \mathit{cfg\ nodes}$, we have:

Corollary 7.7.7. $\mathit{cfg}\ \downarrow_{CESH}\ \mathbf{nat}\ n$ if and only if $\mathit{nodes}\ \downarrow_{Sync}\ \mathbf{nat}\ n$.

Corollary 7.7.8. $\mathit{cfg}\ \uparrow_{CESH}$ if and only if $\mathit{nodes}\ \uparrow_{Sync}$.

We also have that initial configurations are in R_{Sync}:

$$
\begin{aligned}
\mathit{initial\text{-}related}_{Sync}\ :\ \forall\ c\ i \to R_{Sync}\ (c\ ,\ [\ ]\ ,\ [\ ]\ ,\ \varnothing) \\
(\mathit{initial\text{-}network}_{Sync}\ c\ i)
\end{aligned}
$$

These final results complete the picture for the DCESH machine. We have established that we get the same final result regardless of whether we choose to run a fragment of code using the CES, the CESH, or the DCESH machine.

[25] Online appendix: `secd/formalisation` directory, `DCESH.Simulation-CESH` module.
[26] Online appendix: `secd/formalisation` directory, `DCESH.Presimulation-CESH` module.
[27] Online appendix: `secd/formalisation` directory, `DCESH.Bisimulation-CESH` module.

7.8 COMPARISON

We have already seen some benchmarks of our Floskel (Section 7.2.4) implementation, but it also useful to compare our work on extending conventional abstract machines to the work using Geometry of Interaction (Chapter 3) or game semantics (Chapter 4).

A principled performance comparison between these four compilers, which all implement seamless compilation, is difficult because it cannot be a like-for-like comparison, but we will attempt to do so anyway. We summarise the differences between the four implementations below.

The GOI compiler, DCESH, and DKrivine implement PCF, but not in the same way. DKrivine and DCESH implement the type-free language and recursion is dealt with using combinators in the source language, which is potentially less efficient. On the other hand, the interaction-based compilers use a specialised fixpoint constant but also require specialised machinery to handle variable contraction. So there are several sources of inefficiencies in these compilers.

The GAMC compiler implements a larger language: a typed applied call-by-name lambda calculus with mutable references and concurrency. It is also tidier than the approaches based on conventional abstract machines, in that it explicitly deallocates memory when it is done with it and so does not require garbage collection. These features do however require a significant amount of overhead, some of which is already present in the DKrivine and DCESH infrastructure but some of which will need to be added.

There are some features that make the comparison of the compilers somewhat meaningful. The first is that there is a small intersection of source programs that they can all compile. The second is that three of the compilers use call-by-name. All four compilers are also written as straightforward representations of the semantic model of the language, with the same level of disregard for optimisations and a similar level of concern for "obvious" correctness. All four compilers target C and MPI, meaning that benchmarks can be run on the same computer.

With these caveats in mind we will attempt a rough performance comparison of the compilers in several ways. Since the intersection of supported programs is small, our benchmark cannot be very comprehensive, and is simply three small programs operating on integers:

`arith`: Computing the sum of applying a complicated integer function to the numbers in the sequence $0, \ldots, 299$.

`fib`: Computing the 10th Fibonacci number (using the exponential algorithm) 100 times and taking the sum.

	arith			fib			root		
	time	avg. size	max. size	time	avg. size	max. size	time	avg. size	max size
GOI	114%	107	172	4,017%	302	444	19,422%	717	1,312
GAMC	193%	20	24	1,481%	20	24	22,872%	20	24
DKrivine	140%	32	40	238%	32	40	890%	32	40
DCESH	133%	32	40	103%	32	40	100%	32	40

Table 7.3: Benchmarks for distribution overheads

The DKrivine and DCESH compilers are not only faster for local execution, but also have a comparatively small communication overhead. Each time entry in the table is relative to the same compiler's local execution time, which means that DKrivine and DCESH are well ahead of the others in terms of absolute *execution time. It should be noted that DCESH sends much fewer messages than the others because of its call-by-value evaluation strategy, which means that it gets low overheads also for these benchmarks.*

`root`: Compute the (integer) root of a polynomial using 20 iterations of the bisection method.

Krivine baseline. We take a naive implementation of the classic Krivine machine as a reference point and run the compilers in single-node mode. This gives a rough measure of the overall overhead of the compiler before communication costs even come into play. The benchmark programs are written without caching intermediate results, which means that they perform many needless re-computations when run in the call-by-name compilers. It is thus to be expected that the call-by-value compiler, DCESH, does comparatively better.

	arith	fib	root
GOI	3,042%	2,832%	20,222%
GAMC	765%	395%	356%
DKrivine	131%	141%	233%
DCESH	2.6%	65%	100%

In the case of the interaction-based compilers the overheads are mainly due to the implementation of contraction, and in the case of GAMC they are also due to the large amount of heap allocation and deallocation.

Single node baseline. We measure each compiler using its own single-node performance as a reference point and split the program in two nodes such that a large communication overhead is introduced. We measure it both in terms of relative execution time and in terms of average and maximum size of the messages, in bytes. The overheads are only due to the processing required by the node to send and receive the nodes and not due to network latencies —

in order to factor those out we run all the (virtual) MPI nodes on the same physical computer.

The data is shown in Table 7.3 and we can see that the DKrivine and DCESH compilers are not only faster for local execution, but also have a comparatively small communication overhead. Each time entry in the table is relative to the same compiler's local execution time, which means that DKrivine and DCESH are well ahead of the others in terms of *absolute* execution time. It should be noted that DCESH sends much fewer messages than the others because of its call-by-value evaluation strategy, which means that it gets low overheads also for these benchmarks.

All compilers except GOI use messages of a bound size, whereas GOI's messages grow, sometimes significantly, during execution. The high overhead across the call-by-name compilers for the `root` benchmark is because that benchmark does a relatively small amount of local computations before it needs to communicate. We suspect that the high overhead for GOI and GAMC in many benchmarks is also due to the large amount of "bookkeeping" C code that is required, even for simple terms. The way the C compiler optimiser works plays an important role in the performance gap between single-node and distributed execution. When all the code is on the same node the functions are aggressively inlined because they belong to the same binary output. When the code is distributed this is no longer possible.

Although the more exotic interaction-based approaches can be effective at creating correct and transparent distribution, it seems to be the case that their single-node execution model is bound to be less efficient than that of conventional abstract machines. We should also make the point that conventional techniques have the advantage that existing compilers can be extended to accommodate higher-order RPCs without extremely intrusive changes, and that there already exists a breadth of research on for example optimisation and inclusion of foreign function interfaces, which is not the case for the interaction-based techniques.

7.9 RELATED WORK

In this section we will focus on work that is closely related to distributing abstract machines like ours. See Section 2.1 for a more comprehensive survey of related work.

The execution mechanism that the tierless language Links builds on, the client/server calculus [28], is specialised to systems with two nodes, namely client and server. The two nodes are not equal peers: the server is designed to be *stateless* to be able to handle a large number of clients. The work on the

client/server calculus also spawned work on a more general parallel abstract machine, LSAM, that handles an arbitrary number of nodes [110]. A predecessor to LSAM, called dML, uses a similar operational semantics but for a richer language [113]. The main difference between these machines and ours is that they are based on higher-level semantics for call-by-value lambda calculi, that use explicit substitutions and are therefore less straightforward to use as a basis for compilation. In contrast to our work, they also assume synchronous communication models.

There are also extensions of the non-strict Spineless Tagless G-machine (STG) [82] for distributed execution. One is GUM [133], which is an implementation-oriented project to extend the support for parallel execution in Haskell to distributed architectures. The focus is on providing a large amount of automation and the work provides insight into how to mix local garbage collection with distributed weighted reference counting, but has no formal accounts of the execution mechanism.

The Eden project [99], an implementation of parallel Haskell for distributed systems, keeps most communication implicit and is thus closer to our aims. A similarity to our work is that the specification of the language is tiered: an operational semantics at the level of the language and an abstract machine semantics for execution environment, the Distributed Eden Abstract Machine (DREAM) [19]. Eden is not perfectly seamless: a small set of syntactic constructs are used to manage processes explicitly and communication is always performed using head-strict lazy lists. There are significant technical differences between DREAM and the DCESH and Krivine nets since the DREAM is a mechanism of distribution for the STG machine. In terms of emphasis, Eden is an implementation-focussed project whereas we want to create a firm theoretical foundation on which compilation to distributed platforms can be carried out. Whereas (as far as we know) no soundness results exist for the DREAM, we provide a fully formalised proof.

The DREAM, dML and the LSAM are, as far as we are aware, the only abstract machines for general distributed systems which, like the DKrivine and the DCESH machines, combine conventional execution mechanisms with communication primitives. Abstract machines have been proposed for communication only [70], taking inspiration from the Chemical Abstract Machine (CHAM) (which we also take inspiration from, to model the communication network), but they only deal with half the problem when it comes to compiling conventional languages.

Chapter 8

Fault-tolerance via transactions

We formalise a generic transaction-based method for transparently handling failure in abstract machines like DCESH and DKrivine, showing that fault-tolerance can — at least to an extent — be automated in seamlessly distributing languages like ours.[1] Node state is "backed up" (*commit*) at certain points in the execution, and if an exceptional condition arises, the backup is restored (*roll-back*).

This development is independent of the underlying transition relation, but the proofs rely on sequentiality. We assume that we have two arbitrary types *Machine* and *Msg*, as well as a transition relation over them:

$$\_\xrightarrow[Machine]{\_}\_ \;:\; Machine \to Tagged\ Msg \to Machine \to Set$$

We have no knowledge of exceptional states in *Machine*, since it is a parameter, so we define another relation, $\_\xrightarrow[Crash]{\_}\_$, as a thin layer on top of $\_\xrightarrow[Machine]{\_}\_$. The new definition is shown in Figure 8.1 and adds the exceptional state **nothing** by extending the set of states of the relation to *Maybe Machine*. The fallible machine can make a **normal-step** transition from and to **just** ordinary *Machine* states, or it can **crash** which leaves it in the exceptional state. This means that we tolerate fail-stop faults as opposed to e.g. the more general Byzantine failures [92].

The additional assumptions for sequentiality are that we have a decidable predicate, $active : Machine \to Set$ on machines, and the following functions:

$$\begin{aligned}
&inactive\text{-}receive\text{-}active \;:\; \forall\ \{m\ m'\ msg\} \to \\
&\quad (m \xrightarrow[Machine]{\mathbf{receive}\ msg} m') \to \neg\ (active\ m) \times active\ m' \\
&active\text{-}silent\text{-}active \quad\;\; :\; \forall\ \{m\ m'\} \to \\
&\quad (m \xrightarrow[Machine]{\tau} m') \to active\ m \times active\ m' \\
&active\text{-}send\text{-}inactive \quad :\; \forall\ \{m\ m'\ msg\} \to \\
&\quad (m \xrightarrow[Machine]{\mathbf{send}\ msg} m') \to active\ m \times \neg\ (active\ m')
\end{aligned}$$

These functions express the property that if a machine is invoked, i.e. it receives a message, then it must go from an inactive to an active state. If the machine then takes a silent step, it must remain active, and when it sends a message it must go back to being inactive. This gives us sequentiality; a machine cannot fork new threads, and cannot be invoked several times in parallel.

[1] Online appendix: `secd/formalisation` directory, `Backup` module.

```
data _⟶[Crash]_ : Maybe Machine → Tagged Msg →
                  Maybe Machine → Set where
  normal-step : ∀ {tmsg m m'} →
    (m ⟶[Machine]^tmsg m') → (just m ⟶[Crash]^tmsg just m')
  crash : ∀ {m} →
    (just m ⟶[Crash]^τ nothing)
```

Figure 8.1: The transition relation of a machine that may crash
The fallible machine can make a ***normal-step*** *transition from and to* ***just*** *ordinary* Machine *states, or it can* ***crash*** *which leaves it in the exceptional state.*

As the focus here is on obvious correctness and simplicity, we abstract from the method of actually detecting faults in nodes, and assume that it can be done (using e.g. a heartbeat network [4]). Similarly, we assume that we have a means of creating and restoring a backup of a node in the system; how this is done depends largely on the underlying system. We so define a machine with a backup as $Backup = Machine \times Machine$, where the second *Machine* denotes the backup. Backups are therefore done by replicating the machine state in our model. Using this definition, we define a backup strategy, given in Figure 8.2. This strategy makes a backup just after sending and receiving messages. In the case of the underlying machine crashing, it restores the backup. Note that this is only one of many possible backup strategies. This one is particularly nice from a correctness point-of-view, because it makes a backup after every observable event, although it may not be the most performant.

We define binary relations for making transitions with *some* tagged message, as follows:

$$
\begin{aligned}
&\_\xrightarrow[Machine]{}\_ : Machine \to Machine \to Set \\
&m_1 \xrightarrow[Machine]{} m_2 = \exists\, \lambda\, tmsg \to (m_1 \xrightarrow[Machine]{tmsg} m_2) \\
&\_\xrightarrow[Backup]{}\_ : Backup \to Backup \to Set \\
&b_1 \xrightarrow[Backup]{} b_2 = \exists\, \lambda\, tmsg \to (b_1 \xrightarrow[Backup]{tmsg} b_2)
\end{aligned}
$$

Using these relations we can define the observable trace of a run of a *Machine* (*Backup*), i.e. an element of the reflexive transitive closure of the above relations. First we define *IO*, the subset of tagged messages that we can observe, namely **send** and **receive**:

```
data IO (A : Set) : Set where
  send receive : A → IO A
```

$$
\begin{aligned}
&\textsf{data}\ \_\xrightarrow[Backup]{\_}\_ : \mathit{Backup} \to \mathit{Tagged\ Msg} \to \mathit{Backup} \to \mathit{Set}\ \textsf{where}\\
&\quad \textbf{silent-step} : \forall\ \{m\ n\ m'\} \to\\
&\qquad (\textbf{just}\ m \xrightarrow[Crash]{\tau} \textbf{just}\ m') \qquad \to ((m\ ,\ n) \xrightarrow[Backup]{\tau} (m'\ ,\ n))\\
&\quad \textbf{receive-step} : \forall\ \{m\ n\ m'\ msg\} \to\\
&\qquad (\textbf{just}\ m \xrightarrow[Crash]{\textbf{receive}\ msg} \textbf{just}\ m') \to ((m\ ,\ n) \xrightarrow[Backup]{\textbf{receive}\ msg} (m'\ ,\ m'))\\
&\quad \textbf{send-step} : \forall\ \{m\ n\ m'\ msg\} \to\\
&\qquad (\textbf{just}\ m \xrightarrow[Crash]{\textbf{send}\ msg} \textbf{just}\ m') \quad \to ((m\ ,\ n) \xrightarrow[Backup]{\textbf{send}\ msg} (m'\ ,\ m'))\\
&\quad \textbf{recover} : \forall\ \{m\ n\} \to\\
&\qquad (\textbf{just}\ m \xrightarrow[Crash]{\tau} \textbf{nothing}) \quad \to ((m\ ,\ n) \xrightarrow[Backup]{\tau} (n\ ,\ n))
\end{aligned}
$$

Figure 8.2: The transition relation of a crashing machine with backup

This strategy makes a backup just after sending and receiving messages. In the case of the underlying machine crashing, it restores the backup. Backups are done by replicating the machine state.

The following function now gives us the observable trace, given an element of $\xrightarrow[Machine]{}^*$ (which is defined using list-like notation) by ignoring any silent steps.

$$
\begin{aligned}
&[\![\_]\!]_M : \forall\ \{m_1\ m_2\} \to m_1 \xrightarrow[Machine]{}^* m_2 \to \mathit{List}\ (\mathit{IO\ Msg})\\
&[\![\ []\ ]\!]_M = []\\
&[\![\ ((\boldsymbol{\tau} \qquad\qquad , \_) :: \mathit{steps})\ ]\!]_M = [\![\ \mathit{steps}\ ]\!]_M\\
&[\![\ ((\textbf{send}\quad \mathit{msg}\ , \_) :: \mathit{steps})\ ]\!]_M = \textbf{send}\quad \mathit{msg} :: [\![\ \mathit{steps}\ ]\!]_M\\
&[\![\ ((\textbf{receive}\ \mathit{msg}\ , \_) :: \mathit{steps})\ ]\!]_M = \textbf{receive}\ \mathit{msg} :: [\![\ \mathit{steps}\ ]\!]_M
\end{aligned}
$$

$[\![\_]\!]_B$ is defined analogously. Given this definition, we can trivially prove the following soundness result:

Theorem 8.0.1. If we have a run $m_1 \xrightarrow[Machine]{}^* m_2$ then there exists a run of the *Backup* machine that starts and ends in the same state and has the same observational behaviour.

Formally, this is expressed as:

$$
\begin{aligned}
&\mathit{soundness} : \forall\ \{m_1\ m_2\ b_1\}\\
&\quad (\mathit{mtrace} : m_1 \xrightarrow[Machine]{}^* m_2) \to\\
&\quad \exists_2\ \lambda\ b_2\ (\mathit{btrace} : (m_1\ ,\ b_1) \xrightarrow[Backup]{}^* (m_2\ ,\ b_2)) \to\\
&\qquad [\![\ \mathit{btrace}\ ]\!]_B \equiv [\![\ \mathit{mtrace}\ ]\!]_M
\end{aligned}
$$

This is proved by constructing a crash-free *Backup* run given the *Machine* run. Obviously, the interesting question is whether we can take any *crashing* run and get a corresponding *Machine* run.

The result that we want is the following:

Theorem 8.0.2. $bs : (b_1 , b_1) \xrightarrow[Backup]{}{}^{*} (m_2 , b_2)$

there is a run

$$ms : b_1 \xrightarrow[Machine]{}{}^{*} m_2$$

with the same observational behaviour as *bs*.

The key to proving that is the following lemma:

$$\begin{aligned}
&\mathit{fast\text{-}forward\text{-}to\text{-}crash} : \forall \{m_1\ m_2\ b_1\ b_2\ n\} \rightarrow \\
&\quad (s : (m_1 , b_1) \xrightarrow[Backup]{}{}^{*} (m_2 , b_2)) \rightarrow \\
&\quad \mathit{thread\text{-}crashes}\ s \rightarrow \mathit{length}\ s \leq n \rightarrow \\
&\quad \exists\, \lambda\, (s' : ((b_1 , b_1) \xrightarrow[Backup]{}{}^{*} (m_2 , b_2))) \rightarrow \\
&\qquad (\neg\ \mathit{thread\text{-}crashes}\ s') \times ([\![\ s\ ]\!]_B \equiv [\![\ s'\ ]\!]_B) \times (\mathit{length}\ s' \leq n)
\end{aligned}$$

Here *thread-crashes* is a decidable property on backup runs, that ensures that, if m_1 is active, then it crashes and does a recovery step at some point before it performs an observable action. The proof of *fast-forward-to-crash* is done by induction on the natural number *n*. Our key result above, expressed formally as follows, can now be proved:

$$\begin{aligned}
&\mathit{completeness} : \forall \{b_1\ m_2\ b_2\} \\
&\quad (bs : (b_1 , b_1) \xrightarrow[Backup]{}{}^{*} (m_2 , b_2)) \rightarrow \\
&\quad \exists\, \lambda\, (ms : b_1 \xrightarrow[Machine]{}{}^{*} m_2) \rightarrow \\
&\qquad [\![\ bs\ ]\!]_B \equiv [\![\ ms\ ]\!]_M
\end{aligned}$$

The above result can be enhanced further by observing that if the probability of a machine crash is not 1 then the probability of the machine *eventually* having a successful execution is 1. This means that the probability for the number *n* above to exist is also 1. This argument has not been formalised in Agda.[2]

[2]The amount of work that would be required to create libraries for real numbers and probability distributions is too high compared to the importance of this observation.

Finale

Chapter 9

Conclusion

9.1 SUMMARY OF CONTRIBUTIONS

Our most important practical contribution is a compiler for a full-fledged functional language called Floskel (Section 7.2), which supports both located and ubiquitous functions. This compiler is based on our extension of the SECD machine.

On the theoretical side we have presented four novel abstract machines for the execution of programs with support for higher-order Remote Procedure Calls and made prototype implementations for them. The main feature of these abstract machines is that function calls behave, from the point of view of the programmer, in the same way whether they are local or remote.

The first two (Chapter 3 and Chapter 4) used a new application of interaction semantics to the compilation of programming languages, and additionally showed how we can construct abstract machines that are readily implementable and close to conventional machines for these semantics. These abstract machines support a novel *combination* operation where the functionality of components of the interpretation of a program can arbitrarily be combined into one node, which gives programmers the freedom to control the granularity of the programs to their liking. We have proved the correctness of these machines by pen-and-paper soundness proofs, which show that they are a correct implementation of the interaction semantics.

The other two abstract machines (Chapter 6 and Chapter 7) are moderate extensions of abstract machines that are conventionally used for compilation, namely the Krivine machine and the SECD machine. The behaviour of these machines is deliberately exactly that of the conventional machine in the case when the programs run on a single node. They give us a principled compilation model of the applied lambda calculus to abstract distributed architectures for both call-by-name and call-by-value. Our main results here are rigorous, fully formalised proofs of correctness of the new abstract machines done by comparing them to the conventional counterparts, and proof-of-concept compilers which allow us to compare these compilation schemes with our previous implementations.

The full source code for the implementations and the Agda formalisations can be found in the online appendix: `http://epapers.bham.ac.uk/1985/`.

We have additionally showed a simple way of achieving fault-tolerance (Chapter 8) that can be applied to any of the presented abstract machines, by an additional layer on top of a machine that may fail.

9.1.1 *Thesis evaluation*

In Section 1.2.1 we stated that the focus of this thesis is on the core evaluation mechanism, in the form of abstract machines, that is used to run programs with location annotations, since this is something that has not been investigated in full before (see Section 2.1 and Section 7.9 for relevant literature reviews).

Our main requirements for this mechanism were regarding correctness and runtime efficiency. First, we wanted correctness with respect to the same program without annotations, i.e. that we are providing a nondistributed view of the system. Second, we wanted to enable (but not necessarily guarantee) the programs to be efficient. Our performance requirements to achieve this were that we should not lose *single-node* performance when using our language without annotations and that we do not put an excessive burden on the network when we do.

The first requirement, correctness, is fulfilled by all four of the presented abstract machines. For each of them we have proved, in Agda or with pen-and-paper, that a program with location annotations always yields the same result as one without annotations. This means that we really do get a nondistributed view of the system.

The abstract machines have different performance characteristics (see Section 7.8 for a comparison). The first machine, based on GOI (Chapter 3), essentially stores the computational context in the messages, which means that it has potential to put a big burden on the network. The GAMC compiler (Chapter 4) ameliorates this issue by storing the context locally on the nodes and keeping references to these local pieces of context in the messages. Both GOI and GAMC are remarkable because they do not require garbage collection, but they can still not compete with conventional compilation techniques in single-node performance. This discovery was what led us into making distributing extensions of two of the abstract machines that are conventionally used for compilation — the Krivine machine (Chapter 6) and the SECD machine (Chapter 7). These extensions are constructed such that they degenerate into the original machines when they are run in single-node mode, meaning that the single-node performance requirement is fulfilled by construction. Our benchmarks (Section 7.8) confirm this. Like the GAMC, the conventionally based machines use messages of constant size.

Our most mature implementation, called Floskel (Section 7.2), trades not requiring distributed garbage collection for potentially larger messages. Its single-node performance is fast enough to come close to a state of the art compiler.

To summarise, our conventionally based machines are correct, have good single-node performance, and use constant-size messages. They thus fulfil our initial requirements. It is our hope that future compiler writers will make use of our ideas to integrate Remote Procedure Calls (RPCs) that both act native and perform well in future programming languages.

9.2 LIMITATIONS

This dissertation shows how to implement seamlessly distributing programming languages with location annotations in several different ways. It dives deep into this topic in the sense that multiple solutions *to achieve exactly that*, with different strengths and weaknesses, are presented and compared. However, there are many other problems in distributed computing that we do not solve.

Our work does not make parallel or concurrent programming easier, but it does not make it more difficult either — that is just not its focus. While distributed systems are often used for the purpose of speeding up computations by parallelising them, distribution and parallelism are orthogonal issues. Distribution deals with executing a program in a system with multiple nodes where message passing may be the only form of communication available between them (i.e. they do not generally share an address space), while parallelism means running parts of the program in parallel, which is something that can also happen in non-distributed systems. Our work does however not preclude parallel and concurrent execution, evident by one of our implementations (Chapter 4) having a *par* construct.

The possible need for distributed garbage collection (which is also be discussed in Section 9.3) may prove to be a serious limitation for implementations based on our work. If full-blown distributed garbage collection is not an option, there are several strategies that can be used. The first is to use an interaction-based compilation technique (e.g. GOI (Chapter 3) or GAMC (Chapter 4)) that does not require garbage collection. The second is to use a technique that is guaranteed not to produce cyclic garbage (e.g. DKrivine (Chapter 6)) such that distributed reference counting can be used. The third is simply to not keep remote references to data; to serialise and send whole heap structures when they are needed on remote nodes (e.g. Floskel (Section 7.2)). The last option has larger communication overheads, but in cases where those

are an issue, data access can be indirected. It is also possible to provide syntactic sugar for such indirections in the language.

Our formalisations only cover a core of the implementations; a certain amount of extrapolation took place when we implemented the functionality that the formalisations describe. The compilers are thus not extracted directly from the proofs, but are written by hand following them. While such extrapolations are not uncommon practice in research, they are a serious limitation. But we should keep in mind that having formalised part of a system is better than having formalised none of it.

Higher-order RPCs might not be flexible enough to use in all distributed scenarios; more low-level or specialised language features may be necessary for certain types of applications. The popularity of low-level libraries like MPI [65] and specialised programming models like MapReduce [34] could be an indication that this is the case. However, RPC is *also* a widely used programming model. Since our work can be seen as improved, natively integrated RPCs it should be possible to use them wherever RPCs are used today.

A more philosophical discussion of our work is given in Section 9.4.

9.3 FURTHER WORK

The main challenge of this research is, as described in the introductory chapter, to create the underlying evaluation mechanism for programming languages that are seamlessly distributed by the means of Remote Procedure Calls that are transparently, correctly, and efficiently incorporated into the programming language. To do this, there are a few main areas of research that are relevant, each of which comes with different sets of issues that need to be addressed. This section outlines the areas and possibilities for future research.

9.3.1 *Parallelism and concurrency*

Our abstract machines have the internal machinery required for parallel execution, but we restrict ourselves to sequential execution. In moving towards parallelism there are several design (how to add parallelism?) and theoretical (is compilation still correct?) challenges. Design-wise the threading mechanism of our abstract machines is flexible enough, considering that the GAMC compiler uses essentially the same mechanism as the others. An ingredient that is lacking is a synchronisation primitive, but that is not a serious difficulty. A theoretical challenge stems from the failure of the equivalence of synchronous and asynchronous networks in the presence of multiple pending messages. Futhermore, conventional abstract machines typically do *not* support parallelism, meaning that we cannot continue to use them as our specification.

9.3.2 *Language*

A language with static location annotations is too simple-minded for most realistic distributed programs. We need something more expressive.

In the examples we have seen that we still have to reflect some architecture-specific details in the source code of the programs. Taking inspiration from aspect-oriented programming [80] and orchestration languages [22] we propose that a configuration language, *separate* from the algorithmic language, should be constructed. This configuration language would be used to specify how the program should deal with issues of the distributed system not directly related to the logic of the program, such as failure response and recovery, dynamic load-balancing, etc. The node annotations could, instead of being the direct specification of location, be used as *pointcuts* that the configuration language can tie into. A configuration language would increase modularity, as there would no longer be a need to change the logic of the program to re-target an application to a different system. An additional benefit is that the correctness of the program logic only needs to be verified for a *local* instance of the program, since changing the configuration preserves its overall semantics.

Another question is how to do code and data migration in the language. Whether code or data can or should be migrated to different nodes is a question that can be answered from a safety or from an efficiency point of view. The safety angle is very well covered by type systems such as ML5's [137], which prevent the unwanted export of local resources. Another possible use of such a type system is the use of its location information to automatically infer and decide suitable locations for parts of the program, or to warn when a program's employment is potentially not what the programmer wants. The efficiency point of view can also be dealt with in a type theoretic way, as witnessed by recent work in resource-sensitive type systems [21, 57]. The flexibility of Floskel and the DCESH in terms of localising or remoting the calls (statically or even dynamically) together with a resource oriented system can pave the way towards a highly convenient automatic orchestration system in which a program is automatically distributed among several nodes to achieve certain performance objectives.

9.3.3 *Fault-tolerance*

The fault-tolerance model (Chapter 8) that we have shown makes faults invisible, which will not work for all applications. The programmer needs to have the choice to manually handle faults, because the handler can be application-specific. As an example, a MapReduce run should always have the same result in the face of errors, so our model of fault-tolerance would work. But in a

distributed transactional database, the way to handle inconsistencies will vary depending on the application.

9.3.4 *Implementation*

In our current implementations we have largely ignored the finer issues of efficiency. Our aim was to support *in-principle* efficient single-node compilation, which happens when the machines execute trivially on a single node as a conventional machine, and to reduce the communication overhead by sending only small (bounded-size) messages which are necessary. For example, our use of *views* of remote stack extensions in the DKrivine machine avoids the need to send *pop* messages. In the future we would like to examine the possibility of *efficient* compilation. In order to do this several immediate efficiency issues must be addressed.

Remote pointers In the RPC literature [17] it is argued that emulating a shared or virtual address space is infeasible since it requires each pointer to also contain location information, and that it is questionable whether acceptable efficiency can be achieved. These arguments certainly apply to our work, where we do just this. However, if we use a tagged pointer representation [101] for closure pointers it means that we can use pointer tags to distinguish between local and remote pointers without even having to dereference them. With such tags we would pay a very low, if any, performance penalty for the local pointers.

Garbage collection The execution of some of our machines creates garbage in their heaps. Distributed garbage collection can be a serious problem, but we have strong reasons to believe that it can often be avoided here, because the heap structures that get created are quite simple. For example, there are never circular linked structures in a Krivine net, otherwise the relations would not be well founded. This means that a simpler method, distributed reference counting [16], can be used. We also know that efficient memory management is possible when compiling call-by-name functional programming languages to distributed architectures. The Geometry of Interaction (GOI) compiler is purely stack-based, while the Game Abstract Machine Compiler (GAMC) compiler uses heaps but does explicit deallocations of locations that are no longer needed. The Floskel implementation does not need distributed garbage collection. The values which are the result of call-by-value evaluation are always sent, along with any required closures, to the node where the function using them as arguments is executed. With this approach local garbage collection suffices. Note that this is similar to the approach that

Links takes. If a large data structure needs to be held on a particular node the programmer needs to be aware of this requirement and indirect the access to it using located functions. However, if we wanted to automate this process as well, and prevent some data from migrating when it is too large, the current approach could not cope, and distributed garbage collection would be required. Mutable references or lazy evaluation would likely also require it. Whether this can be done efficiently is a separate topic of research (see e.g. [115, 2, 40]). An interesting question is if we can syntactically infer when it is safe for a node to only use local garbage collection, e.g. when the external interface of a node is of first order. This would restrict the set of nodes that need to participate in distributed collections.

Shortcut forwarding One of the most unpleasant features of the current Krivine net approach is the excessive forwarding of data, especially on remote returns. A way to alleviate this issue is to not create indirections when a node has a stack consisting only of a stack extension at the time of a remote invocation, meaning that the remote node could return directly to the current node's invoker.

Fortunately, the DCESH machine does not suffer from the same problem, as evident by the stronger bisimulation result.

Runtime system The compiled programs need a *runtime system* with support for automatically handling and managing the problems specific to a distributed computing like failure and restart. To do this in a general enough way and tie it into the configuration language described above is a great challenge on its own. The runtime system may also need to perform distributed garbage collection, as previously mentioned.

9.3.5 *Formalisation*

We have mentioned that Agda formalisations are given only for the abstract machines and their properties, which are the new theoretical contributions of this work. However, a full formalisation of the compiler stacks, remains a long-term ambition. Our dream is the eventual development of an end-to-end seamless distributing compiler for a higher-order imperative and parallel functional programming language, along the lines of the CakeML [87] and CompCert projects [95]. The formalisation of the correctness of the Krivine net and the DCESH, relative to the conventional machines, is the first step in this direction.

9.4 DISCUSSION

A question worth asking is whether this transparent and integrated approach to distributed computing is *practical*. There are two main possible objections:

Performance Some might say that higher-level languages have poorer performance than system-oriented programming language, which makes them impractical. This debate has been carrying on fruitlessly ever since high-level languages were introduced. We believe that the full spectrum of languages, from machine code to the most abstract, are worth investigating seriously. Seamless computing with higher-order Remote Procedure Calls focusses on the latter, somewhat in the extreme, in the belief that the principled study of heterogeneous (not just distributed, but also for instance reconfigurable) compilation techniques will broaden and deepen our understanding of programming languages in general. And, if we are lucky and diligent, it may even yield a practical and very useful programming paradigm.

Control Distributed computing raises certain specific obstacles in the way of using higher-level languages seamlessly, and this leads to more cogent arguments against their use. A distributed architecture is more volatile than a single node because individual nodes may fail, communication links may break and messages may get lost. Because of this, a remote call may fail in ways that a local call may not. Is it reasonable to present them to the programmer as if they are the same thing? We argue that there is a significant class of applications where the answer is *yes*. If the programmer's objective is to develop algorithms rather than systems, it does not seem right to burden them with the often onerous task of failure management in a distributed system. Another argument against higher-level languages is that they may hide the details of the program's dataflow and not provide enough control to eliminate bottlenecks. To us it seems that the right way to manage both failure and dataflow issues in distributed *algorithmic* programming requires a separation of concerns. Suitable runtime systems must present a more robust programming interface; MapReduce [34] and Ciel [109] are examples of execution engines with runtime systems that automatically handle configuration and failure management aspects, the latter supporting dynamic dataflow dependencies. If more fine-grained control is required, then separate deployment and configuration policies which are transparent to the programmer should be employed. In general, we believe that the role and the scope of orchestration languages [22] should be greatly expanded to this end.

RPC as a paradigm has been criticised for several other reasons: its execution model does not match the native call model, there is no good way of dealing with failure, and it is inherently inefficient [128]. By taking an abstract machine model in which RPCs behave exactly the same as local calls, by showing how a generic transaction mechanism can handle failure, and by implementing reasonably performant compilers we address all these problem head-on. We believe that we provide enough evidence for general native RPCs to be reconsidered in a more positive light.

In general, our main contribution is a theoretical firm starting point for the principled study of compilation targeting distributed architectures. It is our hope that future programming languages can use (extensions of) our abstract machines to get support for Remote Procedure Calls that are not added as an afterthought to the language, but act and feel *native*.

Bibliography

[1] *A garbage collector for C and C++*. `http://www.hboehm.info/gc/`. Last accessed: 13 March 2015.

[2] Saleh E. Abdullahi and Graem A. Ringwood. "Garbage Collecting the Internet: A Survey of Distributed Garbage Collection". In: *ACM Comput. Surv.* 30.3 (1998), pp. 330–373.

[3] Samson Abramsky, Radha Jagadeesan, and Pasquale Malacaria. "Full Abstraction for PCF". In: *Inf. Comput.* 163.2 (2000), pp. 409–470.

[4] Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. "Heartbeat: A Timeout-Free Failure Detector for Quiescent Reliable Communication". In: *Distributed Algorithms, 11th International Workshop, WDAG '97, Saarbrücken, Germany, September 24-26, 1997, Proceedings*. Vol. 1320. Lecture Notes in Computer Science. 1997, pp. 126–140.

[5] *Akka*. `http://akka.io`. Last accessed: 6 July 2015.

[6] Thorsten Altenkirch and Bernhard Reus. "Monadic Presentations of Lambda Terms Using Generalized Inductive Types". In: *Computer Science Logic, 13th International Workshop, CSL '99, 8th Annual Conference of the EACSL, Madrid, Spain, September 20-25, 1999, Proceedings*. Vol. 1683. Lecture Notes in Computer Science. 1999, pp. 453–468.

[7] Andrew W. Appel and David A. McAllester. "An indexed model of recursive types for foundational proof-carrying code". In: *ACM Trans. Program. Lang. Syst.* 23.5 (2001), pp. 657–683.

[8] Joe Armstrong, Robert Virding, and Mike Williams. *Concurrent programming in ERLANG*. Prentice Hall, 1993.

[9] Lennart Augustsson. "Compiling Pattern Matching". In: *FPCA*. 1985, pp. 368–381.

[10] John Backus. "The history of FORTRAN I, II, and III". In: *HOPL-1: The first ACM SIGPLAN conference on History of programming languages*. Los Angeles, CA, 1978, pp. 165–180.

[11] John W Backus et al. "The FORTRAN automatic coding system". In: *Papers presented at the February 26-28, 1957, western joint computer conference: Techniques for reliability*. ACM. 1957, pp. 188–198.

[12] Vincent Balat. "Ocsigen: typing web interaction with objective Caml". In: *Proceedings of the ACM Workshop on ML, 2006, Portland, Oregon, USA, September 16, 2006*. 2006, pp. 84–94.

[13] Jean-Pierre Banâtre, Anne Coutant, and Daniel Le Métayer. "Parallel Machines for Multiset Transformation and their Programming Style / Parallele Maschinen für die Multimengen-Transformation und deren Programmierstil". In: *it - Informationstechnik* 30.2 (1988), pp. 99–109.

[14] Nick Benton. "Embedded interpreters". In: *J. Funct. Program.* 15.4 (2005), pp. 503–542.

[15] Gérard Berry and Gérard Boudol. "The Chemical Abstract Machine". In: *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, California, USA, January 1990*. 1990, pp. 81–94.

[16] D. I. Bevan. "Distributed Garbage Collection Using Reference Counting". In: *PARLE, Parallel Architectures and Languages Europe, Volume II: Parallel Languages, Eindhoven, The Netherlands, June 15-19, 1987, Proceedings*. Vol. 259. Lecture Notes in Computer Science. 1987, pp. 176–187.

[17] Andrew Birrell and Bruce Jay Nelson. "Implementing Remote Procedure Calls". In: *ACM Trans. Comput. Syst.* 2.1 (1984), pp. 39–59.

[18] Ana Bove, Peter Dybjer, and Ulf Norell. "A Brief Overview of Agda - A Functional Language with Dependent Types". In: *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*. Vol. 5674. Lecture Notes in Computer Science. 2009, pp. 73–78.

[19] Silvia Breitinger et al. "DREAM: The DistRibuted Eden Abstract Machine". In: *Implementation of Functional Languages, 9th International Workshop, IFL'97, St. Andrews, Scotland, UK, September 10-12, 1997, Selected Papers*. Vol. 1467. Lecture Notes in Computer Science. 1997, pp. 250–269.

[20] N. G. de Bruijn. "Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem". In: *Indagationes Mathematicae (Proceedings)* (1972), pp. 381–392.

[21] Aloïs Brunel et al. "A Core Quantitative Coeffect Calculus". In: *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*. Vol. 8410. Lecture Notes in Computer Science. 2014, pp. 351–370.

[22] Nadia Busi et al. "Choreography and Orchestration: A Synergic Approach for System Design". In: *Service-Oriented Computing - ICSOC 2005, Third International Conference, Amsterdam, The Netherlands, December 12-15, 2005, Proceedings*. Vol. 3826. Lecture Notes in Computer Science. 2005, pp. 228–240.

[23] Luca Cardelli. "A Language with Distributed Scope". In: *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*. 1995, pp. 286–297.

[24] Adam Chlipala. "Ur: statically-typed metaprogramming with type-level record computation". In: *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*. 2010, pp. 122–133.

[25] Adam Chlipala. "Ur/Web: A Simple Model for Programming the Web". In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. 2015, pp. 153–165.

[26] Stephen Chong et al. "Secure web application via automatic partitioning". In: *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*. 2007, pp. 31–44.

[27] Raphael Collet. "The Limits of Network Transparency in a Distributed Programming Language". PhD thesis. Université catholique de Louvain, 2007.

[28] Ezra Cooper and Philip Wadler. "The RPC calculus". In: *Proceedings of the 11th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, September 7-9, 2009, Coimbra, Portugal*. 2009, pp. 231–242.

[29] Ezra Cooper et al. "Links: Web Programming Without Tiers". In: *Formal Methods for Components and Objects, 5th International Symposium, FMCO 2006, Amsterdam, The Netherlands, November 7-10, 2006, Revised Lectures*. Vol. 4709. Lecture Notes in Computer Science. 2006, pp. 266–296.

[30] Pierre-Louis Curien. "Notes on game semantics". Lecture Notes. 2006.

[31] Vincent Danos, Hugo Herbelin, and Laurent Regnier. "Game Semantics & Abstract Machines". In: *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science, New Brunswick, New Jersey, USA, July 27-30, 1996*. 1996, pp. 394–405.

[32] Vincent Danos and Laurent Regnier. "Reversible, Irreversible and Optimal lambda-Machines". In: *Theor. Comput. Sci.* 227.1-2 (1999), pp. 79–97.

[33] Olivier Danvy and Kevin Millikin. "A Rational Deconstruction of Landin's SECD Machine with the J Operator". In: *Logical Methods in Computer Science* 4.4 (2008).

[34] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: a flexible data processing tool". In: *Commun. ACM* 53.1 (2010), pp. 72–77.

[35] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters". In: *Commun. ACM* 51.1 (2008), pp. 107–113.

[36] Stephan Diehl, Pieter H. Hartel, and Peter Sestoft. "Abstract machines for programming language implementation". In: *Future Generation Comp. Syst.* 16.7 (2000), pp. 739–751.

[37] Jeff Epstein, Andrew P. Black, and Simon L. Peyton Jones. "Towards Haskell in the cloud". In: *Proceedings of the 4th ACM SIGPLAN Symposium on Haskell, Haskell 2011, Tokyo, Japan, 22 September 2011*. 2011, pp. 118–129.

[38] Matthias Felleisen and Daniel P. Friedman. "Control operators, the SECD-machine, and the lambda-calculus". In: *3rd Working Conference on the Formal Description of Programming Concepts*. Aug. 1986.

[39] Maribel Fernández and Ian Mackie. "Call-by-Value lambda-Graph Rewriting Without Rewriting". In: *Graph Transformation, First International Conference, ICGT 2002, Barcelona, Spain, October 7-12, 2002, Proceedings*. Vol. 2505. Lecture Notes in Computer Science. 2002, pp. 75–89.

[40] Fabrice Le Fessant. "Detecting distributed cycles of garbage in large-scale systems". In: *Proceedings of the Twentieth Annual ACM Symposium on Principles of Distributed Computing, PODC 2001, Newport, Rhode Island, USA, August 26-29, 2001*. 2001, pp. 200–209.

[41] A. J. Field and Peter G. Harrison. *Functional Programming*. Addison-Wesley, 1988.

[42] Cédric Fournet et al. "A Calculus of Mobile Agents". In: *CONCUR '96, Concurrency Theory, 7th International Conference, Pisa, Italy, August 26-29, 1996, Proceedings*. Vol. 1119. Lecture Notes in Computer Science. 1996, pp. 406–421.

[43] Olle Fredriksson. "Distributed call-by-value machines". In: *CoRR* abs/1401.5097 (2014).

[44] Olle Fredriksson and Dan R. Ghica. "Abstract Machines for Game Semantics, Revisited". In: *28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013, New Orleans, LA, USA, June 25-28, 2013*. 2013, pp. 560–569.

[45] Olle Fredriksson and Dan R. Ghica. "Krivine nets: a semantic foundation for distributed execution". In: *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*. 2014, pp. 349–361.

[46] Olle Fredriksson and Dan R. Ghica. "Seamless Distributed Computing from the Geometry of Interaction". In: *Trustworthy Global Computing - 7th International Symposium, TGC 2012, Newcastle upon Tyne, UK, September 7-8, 2012, Revised Selected Papers*. Vol. 8191. Lecture Notes in Computer Science. 2012, pp. 34–48.

[47] Olle Fredriksson, Dan R. Ghica, and Bertram Wheen. "Towards native higher-order remote procedure calls". In: *Proceedings of the 26th Symposium on Implementation and Application of Functional Languages, Boston, MA, USA, October 1-3, 2014*. 2014.

[48] Murdoch Gabbay and Dan R. Ghica. "Game Semantics in the Nominal Model". In: *Electr. Notes Theor. Comput. Sci.* 286 (2012), pp. 173–189.

[49] Murdoch Gabbay and Andrew M. Pitts. "A New Approach to Abstract Syntax Involving Binders". In: *14th Annual IEEE Symposium on Logic in Computer Science, Trento, Italy, July 2-5, 1999*. 1999, pp. 214–224.

[50] Philippa Gardner, Cosimo Laneve, and Lucian Wischik. "Linear forwarders". In: *Inf. Comput.* 205.10 (2007), pp. 1526–1550.

[51] Dan R. Ghica. "Applications of Game Semantics: From Program Analysis to Hardware Synthesis". In: *Proceedings of the 24th Annual IEEE Symposium on Logic in Computer Science, LICS 2009, 11-14 August 2009, Los Angeles, CA, USA*. 2009, pp. 17–26.

[52] Dan R. Ghica. "Function interface models for hardware compilation". In: *9th IEEE/ACM International Conference on Formal Methods and Models for Codesign, MEMOCODE 2011, Cambridge, UK, 11-13 July, 2011*. 2011, pp. 131–142.

[53] Dan R. Ghica. "Geometry of synthesis: a structured approach to VLSI design". In: *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*. 2007, pp. 363–375.

[54] Dan R. Ghica and Guy McCusker. "The regular-language semantics of second-order idealized ALGOL". In: *Theor. Comput. Sci.* 309.1-3 (2003), pp. 469–502.

[55] Dan R. Ghica and Andrzej S. Murawski. "Angelic semantics of fine-grained concurrency". In: *Ann. Pure Appl. Logic* 151.2-3 (2008), pp. 89–114.

[56] Dan R. Ghica, Andrzej S. Murawski, and C.-H. Luke Ong. "Syntactic control of concurrency". In: *Theor. Comput. Sci.* 350.2-3 (2006), pp. 234–251.

[57] Dan R. Ghica and Alex I. Smith. "Bounded Linear Types in a Resource Semiring". In: *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*. Vol. 8410. Lecture Notes in Computer Science. 2014, pp. 331–350.

[58] Dan R. Ghica and Alex I. Smith. "Geometry of Synthesis II: From Games to Delay-Insensitive Circuits". In: *Electr. Notes Theor. Comput. Sci.* 265 (2010), pp. 301–324.

[59] Dan R. Ghica and Alex I. Smith. "Geometry of synthesis III: resource management through type inference". In: *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*. 2011, pp. 345–356.

[60] Dan R. Ghica, Alex I. Smith, and Satnam Singh. "Geometry of synthesis iv: compiling affine recursion into static hardware". In: *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*. 2011, pp. 221–233.

[61] Jean-Yves Girard. "Geometry of interaction 1: Interpretation of System F". In: *Studies in Logic and the Foundations of Mathematics* 127 (1989), pp. 221–260.

[62] Jean-Yves Girard. "Linear Logic". In: *Theor. Comput. Sci.* 50 (1987), pp. 1–102.

[63] Georges Gonthier, Martín Abadi, and Jean-Jacques Lévy. "The Geometry of Optimal Lambda Reduction". In: *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 19-22, 1992*. 1992, pp. 15–26.

[64] Michael J. C. Gordon et al. "A Metalanguage for Interactive Proof in LCF". In: *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978*. 1978, pp. 119–130.

[65] William D Gropp, Ewing L Lusk, and Anthony Skjellum. *Using MPI: portable parallel programming with the message-passing interface*. Vol. 1. MIT Press, 1999.

[66] Gudmund Grov and Greg Michaelson. "Hume box calculus: robust system development through software transformation". In: *Higher-Order and Symbolic Computation* 23.2 (2010), pp. 191–226.

[67] Kevin Hammond and Greg Michaelson. "Hume: A Domain-Specific Language for Real-Time Embedded Systems". In: *Generative Programming and Component Engineering, Second International Conference, GPCE 2003, Erfurt, Germany, September 22-25, 2003, Proceedings*. Vol. 2830. Lecture Notes in Computer Science. 2003, pp. 37–56.

[68] John Hannan and Dale Miller. "From Operational Semantics to Abstract Machines". In: *Mathematical Structures in Computer Science* 2.4 (1992), pp. 415–459.

[69] Peter Henderson. *Functional programming - application and implementation*. Prentice Hall International Series in Computer Science. Prentice Hall, 1980.

[70] Daniel Hirschkoff, Damien Pous, and Davide Sangiorgi. "An efficient abstract machine for Safe Ambients". In: *J. Log. Algebr. Program.* 71.2 (2007), pp. 114–149.

[71] Eric Holk et al. "Kanor - A Declarative Language for Explicit Communication". In: *Practical Aspects of Declarative Languages - 13th International Symposium, PADL 2011, Austin, TX, USA, January 24-25, 2011. Proceedings*. Vol. 6539. Lecture Notes in Computer Science. 2011, pp. 190–204.

[72] Kohei Honda, Nobuko Yoshida, and Marco Carbone. "Multiparty asynchronous session types". In: *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*. 2008, pp. 273–284.

[73] Naohiko Hoshino. "A Modified GoI Interpretation for a Linear Functional Programming Language and Its Adequacy". In: *Foundations of Software Science and Computational Structures - 14th International Conference, FOSSACS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*. Vol. 6604. Lecture Notes in Computer Science. 2011, pp. 320–334.

[74] P. Hudak, S. Peyton Jones, and P. Wadler (editors). "Report on the Programming Language Haskell, A Non-strict Purely Functional Language (Version 1.2)". In: *ACM SIGPLAN Notices* 27.5 (May 1992).

[75] Paul Hudak and Lauren Smith. "Para-Functional Programming: A Paradigm for Programming Multiprocessor Systems". In: *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages, St. Petersburg Beach, Florida, USA, January 1986*. 1986, pp. 243–254.

[76] John Hughes. "A Novel Representation of Lists and its Application to the Function "reverse"". In: *Inf. Process. Lett.* 22.3 (1986), pp. 141–144.

[77] J. M. E. Hyland and C.-H. Luke Ong. "On Full Abstraction for PCF: I, II, and III". In: *Inf. Comput.* 163.2 (2000), pp. 285–408.

[78] J. M. E. Hyland and C.-H. Luke Ong. "Pi-Calculus, Dialogue Games and PCF". In: *FPCA*. 1995, pp. 96–107.

[79] Michael Isard et al. "Dryad: distributed data-parallel programs from sequential building blocks". In: *Proceedings of the 2007 EuroSys Conference, Lisbon, Portugal, March 21-23, 2007*. 2007, pp. 59–72.

[80] Radha Jagadeesan, Alan Jeffrey, and James Riely. "A Calculus of Untyped Aspect-Oriented Programs". In: *ECOOP 2003 - Object-Oriented Programming, 17th European Conference, Darmstadt, Germany, July 21-25, 2003, Proceedings*. Vol. 2743. Lecture Notes in Computer Science. 2003, pp. 54–73.

[81] Thomas Johnsson. "Lambda Lifting: Treansforming Programs to Recursive Equations". In: *FPCA*. 1985, pp. 190–203.

[82] Simon L. Peyton Jones. "Implementing Lazy Functional Languages on Stock Hardware: The Spineless Tagless G-Machine". In: *J. Funct. Program.* 2.2 (1992), pp. 127–202.

[83] Eric Jul et al. "Fine-Grained Mobility in the Emerald System". In: *ACM Trans. Comput. Syst.* 6.1 (1988), pp. 109–133.

[84] Gilles Kahn. "Natural Semantics". In: *STACS 87, 4th Annual Symposium on Theoretical Aspects of Computer Science, Passau, Germany, February 19-21, 1987, Proceedings*. Vol. 247. Lecture Notes in Computer Science. 1987, pp. 22–39.

[85] Paul H. Kelly. *Functional Programming for Loosely-Coupled Multiprocessors*. Cambridge, MA, USA: MIT Press, 1989.

[86] Jean-Louis Krivine. "A call-by-name lambda-calculus machine". In: *Higher-Order and Symbolic Computation* 20.3 (2007), pp. 199–207.

[87] Ramana Kumar et al. "CakeML: a verified implementation of ML". In: *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*. 2014, pp. 179–192.

[88] Yves Lafont. "Interaction Nets". In: *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, California, USA, January 1990*. 1990, pp. 95–108.

[89] James Laird. "Exceptions, Continuations and Macro-expressiveness". In: *Programming Languages and Systems, 11th European Symposium on Programming, ESOP 2002, held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*. Vol. 2305. Lecture Notes in Computer Science. 2002, pp. 133–146.

[90] John Lamping. "An Algorithm for Optimal Lambda Calculus Reduction". In: *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, California, USA, January 1990*. 1990, pp. 16–30.

[91] Leslie Lamport and Nancy A. Lynch. "Distributed Computing: Models and Methods". In: *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*. 1990, pp. 1157–1199.

[92] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. "The Byzantine Generals Problem". In: *ACM Trans. Program. Lang. Syst.* 4.3 (1982), pp. 382–401.

[93] Peter J. Landin. "The Mechanical Evaluation of Expressions". In: *Computer Journal* 6.4 (Jan. 1964), pp. 308–320.

[94] Xavier Leroy. "A Formally Verified Compiler Back-end". In: *J. Autom. Reasoning* 43.4 (2009), pp. 363–446.

[95] Xavier Leroy. "Formal certification of a compiler back-end or: programming a compiler with a proof assistant". In: *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*. 2006, pp. 42–54.

[96] Xavier Leroy. "MPRI course 2-4-2, Part II: abstract machines". University lecture. 2013-2014.

[97] Xavier Leroy. *The ZINC experiment: an economical implementation of the ML language*. Technical report 117. INRIA, 1990.

[98] Hans-Wolfgang Loidl et al. "Comparing Parallel Functional Languages: Programming and Performance". In: *Higher-Order and Symbolic Computation* 16.3 (2003), pp. 203–251.

[99] Rita Loogen, Yolanda Ortega-Mallén, and Ricardo Peña-Marí. "Parallel functional programming in Eden". In: *J. Funct. Program.* 15.3 (2005), pp. 431–475.

[100] Ian Mackie. "The Geometry of Interaction Machine". In: *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*. 1995, pp. 198–208.

[101] Simon Marlow, Alexey Rodriguez Yakushev, and Simon L. Peyton Jones. "Faster laziness using dynamic pointer tagging". In: *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007, Freiburg, Germany, October 1-3, 2007*. 2007, pp. 277–288.

[102] David May. "OCCAM". In: *SIGPLAN Notices* 18.4 (1983), pp. 69–79.

[103] Guy McCusker. "Games and Full Abstraction for FPC". In: *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science, New Brunswick, New Jersey, USA, July 27-30, 1996*. 1996, pp. 174–183.

[104] Robin Milner. *A Calculus of Communicating Systems*. Vol. 92. Lecture Notes in Computer Science. Springer, 1980.

[105] Robin Milner. "A Theory of Type Polymorphism in Programming". In: *J. Comput. Syst. Sci.* 17.3 (1978), pp. 348–375.

[106] Robin Milner. "Functions as Processes". In: *Automata, Languages and Programming, 17th International Colloquium, ICALP90, Warwick University, England, July 16-20, 1990, Proceedings*. Vol. 443. Lecture Notes in Computer Science. 1990, pp. 167–180.

[107] Robin Milner, Joachim Parrow, and David Walker. "A Calculus of Mobile Processes, I". In: *Inf. Comput.* 100.1 (1992), pp. 1–40.

[108] Yasuhiko Minamide, J. Gregory Morrisett, and Robert Harper. "Typed Closure Conversion". In: *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*. 1996, pp. 271–283.

[109] Derek Gordon Murray et al. "CIEL: A Universal Execution Engine for Distributed Data-Flow Computing". In: *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2011, Boston, MA, USA, March 30 - April 1, 2011*. 2011.

[110] Kensuke Narita and Shin-ya Nishizaki. "A Parallel Abstract Machine for the RPC Calculus". In: *Informatics Engineering and Information Science*. 2011, pp. 320–332.

[111] Ulf Norell. "Towards a practical programming language based on dependent type theory". PhD thesis. Department of Computer Science and Engineering, Chalmers University of Technology, Sept. 2007.

[112] Peter W. O'Hearn et al. "Syntactic Control of Interference Revisited". In: *Theor. Comput. Sci.* 228.1-2 (1999), pp. 211–252.

[113] Atsushi Ohori and Kazuhiko Kato. "Semantics for Communication Primitives in an Polymorphic Language". In: *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, USA, January 1993*. 1993, pp. 99–112.

[114] Benjamin C. Pierce and David N. Turner. "Pict: a programming language based on the Pi-Calculus". In: *Proof, Language, and Interaction, Essays in Honour of Robin Milner*. 2000, pp. 455–494.

[115] David Plainfossé and Marc Shapiro. "A Survey of Distributed Garbage Collection Techniques". In: *Memory Management, International Workshop IWMM 95, Kinross, UK, September 27-29, 1995, Proceedings*. Vol. 986. Lecture Notes in Computer Science. 1995, pp. 211–249.

[116] Gordon D. Plotkin. *A Structural Approach to Operational Semantics*. Tech. rep. DAIMI FN–19. Aarhus, Denmark: Computer Science Department, Aarhus University, Sept. 1981.

[117] Gordon D. Plotkin. "LCF Considered as a Programming Language". In: *Theor. Comput. Sci.* 5.3 (1977), pp. 223–255.

[118] Robert F. Pointon, Philip W. Trinder, and Hans-Wolfgang Loidl. "The Design and Implementation of Glasgow Distributed Haskell". In: *Implementation of Functional Languages, 12th International Workshop, IFL 2000, Aachen, Germany, September 4-7, 2000, Selected Papers*. Vol. 2011. Lecture Notes in Computer Science. 2000, pp. 53–70.

[119] John C. Reynolds. "Syntactic Control of Interference". In: *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978*. 1978, pp. 39–46.

[120] John C Reynolds. "The essence of Algol". In: *ALGOL-like Languages*. 1997, pp. 67–88.

[121] *RPyC — Transparent, Symmetric Distributed Computing*. `http://rpyc.readthedocs.org/en/latest/`. Last accessed: 19 March 2015.

[122] Davide Sangiorgi and David Walker. *The Pi-Calculus - a theory of mobile processes*. Cambridge University Press, 2001.

[123] Ulrich Schöpp. "On the Relation of Interaction Semantics to Continuations and Defunctionalization". In: *Logical Methods in Computer Science* 10.4 (2014).

[124] Manuel Serrano, Erick Gallesio, and Florian Loitsch. "Hop: a language for programming the web 2.0". In: *Companion to the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*. 2006, pp. 975–985.

[125] Peter Sewell, Pawel T. Wojciechowski, and Asis Unyapoth. "Nomadic pict: Programming languages, communication infrastructure overlays, and semantics for mobile computation". In: *ACM Trans. Program. Lang. Syst.* 32.4 (2010).

[126] James W. Stamos and David K. Gifford. "Remote Evaluation". In: *ACM Trans. Program. Lang. Syst.* 12.4 (1990), pp. 537–565.

[127] Ivan E. Sutherland. "Micropipelines". In: *Commun. ACM* 32.6 (1989), pp. 720–738.

[128] Andrew Stuart Tanenbaum and Robbert van Renesse. *A critique of the remote procedure call paradigm*. Vrije Universiteit, Subfaculteit Wiskunde en Informatica, 1987.

[129] Seved Torstendahl. "Open telecom platform". In: *Ericsson Review* 74.1 (1997), pp. 14–23.

[130] Prabhat Totoo, Pantazis Deligiannis, and Hans-Wolfgang Loidl. "Haskell vs. F# vs. Scala: a high-level language features and parallelism support comparison". In: *Proceedings of the 1st ACM SIGPLAN workshop on Functional high-performance computing*. FHPC '12. Copenhagen, Denmark, 2012, pp. 49–60.

[131] Philip W. Trinder, Hans-Wolfgang Loidl, and Robert F. Pointon. "Parallel and Distributed Haskells". In: *J. Funct. Program.* 12.4&5 (2002), pp. 469–510.

[132] Philip W. Trinder et al. "Algorithms + Strategy = Parallelism". In: *J. Funct. Program.* 8.1 (1998), pp. 23–60.

[133] Philip W. Trinder et al. "GUM: A Portable Parallel Implementation of Haskell". In: *Proceedings of the ACM SIGPLAN'96 Conference on Programming Language Design and Implementation (PLDI), Philadephia, Pennsylvania, May 21-24, 1996*. 1996, pp. 79–88.

[134] D. A. Turner. "Miranda: A Non-Strict Functional language with Polymorphic Types". In: *FPCA*. 1985, pp. 1–16.

[135] David N. Turner. "The polymorphic pi-calculus: Theory and implementation". PhD thesis. School of Informatics, College of Science and Engineering, University of Edinburgh, 1996.

[136] Vasco Thudichum Vasconcelos, Simon J. Gay, and António Ravara. "Type checking a multithreaded functional language with session types". In: *Theor. Comput. Sci.* 368.1-2 (2006), pp. 64–87.

[137] Tom Murphy VII, Karl Crary, and Robert Harper. "Type-Safe Distributed Programming with ML5". In: *Trustworthy Global Computing, Third Symposium, TGC 2007, Sophia-Antipolis, France, November 5-6, 2007, Revised Selected Papers*. Vol. 4912. Lecture Notes in Computer Science. 2007, pp. 108–123.

[138] Joseph Weizenbaum. "The FUNARG problem explained". Unpublished memorandum. 1968.

[139] Pawel T. Wojciechowski and Peter Sewell. "Nomadic Pict: language and infrastructure design for mobile agents". In: *IEEE Concurrency* 8.2 (2000), pp. 42–52.

[140] Nobuko Yoshida et al. "The Scribble Protocol Language". In: *Trustworthy Global Computing - 8th International Symposium, TGC 2013, Buenos Aires, Argentina, August 30-31, 2013, Revised Selected Papers*. Vol. 8358. Lecture Notes in Computer Science. 2013, pp. 22–41.

Appendix A

Proofs of theorems from Chapter 3

Proof of Proposition 3.2.2. By cases on the machine network step relation and construction of the Stack-Interaction-Control (SIC) machine's step relation.

Case SILENT **:** In this rule, $\mathcal{M}' = \mathcal{M}$, so $|\mathcal{M}'| = |\mathcal{M}|$. The SIC machine M that takes a step goes from **just** to **just** since all SIC machine step rules that do not send or receive messages are on that form, which also implies that $|active(N)| = |active(N')|$.

Case SEND **:** Here $\mathcal{M}' = \mathcal{M} \uplus [(p, e)]$ so $|\mathcal{M}'| = |\mathcal{M}| + 1$. The only SIC machine rule that applies here is the send rule, which takes the machine M from state **just** to **nothing**. So $|active(N')| = |active(N)| - 1$, and thus $|\mathcal{M}'| + |active(N')| = |\mathcal{M}| + 1 + |active(N)| - 1 = |\mathcal{M}| + |active(N)|$

Case RECEIVE **:** In this case, $\mathcal{M} = \mathcal{M}' \uplus [(p, e)]$ so $|\mathcal{M}'| = |\mathcal{M}| - 1$. The only SIC machine rule that applies is the receive rule, which takes the machine M from state **nothing** to **just**, meaning that $|active(N')| = |active(N)| + 1$. Thus $|\mathcal{M}'| + |active(N')| = |\mathcal{M}| - 1 + |active(N)| + 1 = |\mathcal{M}| + |active(N)|$.

□

Proof of Proposition 3.3.4. The first part is trivial. To show that the network N_1, N_2 is combinable, we have to consider its partitions. In the case where no subnetwork in the partition spans both N_1 and N_2, combinability follows from lifting the combinability of N_1 and N_2. If a subnetwork does span both N_1 and N_2, the interesting case is when there is communication between the part that stems from N_1, and the part that stems from N_2. Note that when that happens, it means that we have in the original N_1 network (the N_2 case is analogous):

$$(N_1, [(p, d)]) \rightarrow^* (N_1, [(p_1, d_1)])$$

where p is an input port and p_1 is an output port of N_1 but an internal port of N_1, N_2. By stack-neutrality of N_1 this means that the stacks of all machines are unchanged. The combination of the composition N_1, N_2 can simulate this behaviour, but will continue through the code that stems from N_2 when it reaches the shared port. Since the stack is unchanged at this point, its behaviour is the same as if it was not combined. □

Proof of Proposition 3.3.6. Stack-neutrality follows from stack-neutrality of N. The box stores state in the form of a stack element upon entrance, but any observable output of the network will go through an output port of N, which in turn will pop that stack element. Combinability follows from Lemma 3.3.5. Since entering the box is done first, we add an element to the stack of the combined machine containing the box, but those components will still have the same behaviour since they cannot inspect the shape of the stack. □

Appendix B

Proofs of theorems from Chapter 4

Proof of Proposition 4.1.2.

- Composition is well-defined, i.e. it preserves well-formedness.

  Let $f = (\overline{E}_f, \chi_f, A \Rightarrow B) : A \to B$ and $g = (\overline{E}_g, \chi_g, B' \Rightarrow C) : B' \to C$ be morphisms such that $\pi \vdash B =_{\mathbb{A}} B'$, and their composition $f;g = (\overline{E}_f \cup \overline{E}_g, \chi, A \Rightarrow C) : A \to C$ be as in the definition of composition. To prove that this is well-formed, we need to show that

  $$\chi \in sup((A \Rightarrow C)^{(\mathbf{O})} \otimes I_{fg}^{(\mathbf{P})}) \to sup((A \Rightarrow C)^{(\mathbf{P})} \otimes I_{fg}^{(\mathbf{O})}) =$$
  $$sup(A^{*(\mathbf{O})} \otimes C^{(\mathbf{O})} \otimes I_f^{(\mathbf{P})} \otimes I_g^{(\mathbf{P})}) \to sup(A^{*(\mathbf{P})} \otimes C^{(\mathbf{O})} \otimes I_f^{(\mathbf{O})} \otimes I_g^{(\mathbf{O})})$$

  where $I_{fg} = \otimes\{A \mid (A, P) \in \overline{E}_f \cup \overline{E}_g\}$, and that it is a bijection.

  We are given that

  $$\chi_f \in sup(A^{*(\mathbf{O})} \otimes B^{(\mathbf{O})} \otimes I_f^{(\mathbf{P})}) \to sup(A^{*(\mathbf{P})} \otimes B^{(\mathbf{P})} \otimes I_f^{(\mathbf{O})})$$
  $$\chi_g \in sup(B'^{*(\mathbf{O})} \otimes C^{(\mathbf{O})} \otimes I_g^{(\mathbf{P})}) \to sup(B'^{*(\mathbf{P})} \otimes C^{(\mathbf{P})} \otimes I_g^{(\mathbf{O})})$$
  $$\pi \in sup(B) \to sup(B')$$

  are bijections.

  It is relatively easy to see that the domains specified in the clauses of the definition of χ are mutually disjoint sets and that their union is the domain that we are after.

  Since χ is defined in clauses each of which defined using either χ_f or χ_g and/or π (which are bijections with disjoint domains and codomains), it is enough to show that the set of port names that χ_f is applied to in clause 1 and 4 are disjoint, and similarly for χ_g in clause 2 and 3:

  - In clause 4, we have $\chi_g(a) \in sup(B')$, and so $\pi^{-1}(\chi_g(a)) \in sup(B)$, which is disjoint from $sup(A^{*(\mathbf{O})} \otimes I_f^{(\mathbf{P})})$ in clause 1.
  - In clause 3, we have $\chi_f(a) \in sup(B)$, and so $\pi(\chi_f(a)) \in sup(B')$, which is disjoint from $sup(C^{(\mathbf{O})} \otimes I_g^{(\mathbf{P})})$ in clause 2.

- Composition is associative.

  Let

$$f = (\overline{E}_f, \chi_f, A \Rightarrow B) \ : \ A \to B,$$
$$g = (\overline{E}_g, \chi_g, B' \Rightarrow C) \ : \ B' \to C, \text{and}$$
$$h = (\overline{E}_h, \chi_h, C' \Rightarrow D) \ : \ C' \to D$$

  be nets such that $\pi_1 \vdash B =_{\mathbb{A}} B'$ and $\pi_2 \vdash C =_{\mathbb{A}} C'$. Then we have:

$$(f;g);h = (\overline{E}_f \cup \overline{E}_g \cup \overline{E}_h, \chi_{(f;g);h}, A \Rightarrow D)$$

  and

$$f;(g;h) = (\overline{E}_f \cup \overline{E}_g \cup \overline{E}_h, \chi_{f;(g;h)}, A \Rightarrow D)$$

  according to the definition of composition. We need to show that

$$\chi_{(f;g);h} = \chi_{f;(g;h)},$$

  which implies that $(f;g);h = f;(g;h)$.

  We do this by expanding the definitions, simplified using the following auxiliary function:

$$connect(c, A)(a) \stackrel{\Delta}{=} a \qquad \text{if } a \notin sup(A)$$
$$connect(c, A)(a) \stackrel{\Delta}{=} c(a) \qquad \text{if } a \in sup(A)$$

  $f;g = (\overline{E}_f \cup \overline{E}_g, \chi_{f;g}, A \Rightarrow C)$ and $g;h = (\overline{E}_g \cup \overline{E}_h, \chi_{g;h}, B' \Rightarrow D)$ where

$$\chi_{f;g}(a) \stackrel{\Delta}{=} connect(\chi_g \circ \pi_1, B)(\chi_f(a)) \text{ if } a \in sup(A^{*(\mathbf{O})} \otimes I_f^{(\mathbf{P})})$$
$$\chi_{f;g}(a) \stackrel{\Delta}{=} connect(\chi_f \circ \pi_1^{-1}, B')(\chi_g(a)) \text{ if } a \in sup(C^{(\mathbf{O})} \otimes I_g^{(\mathbf{P})})$$
$$\chi_{g;h}(a) \stackrel{\Delta}{=} connect(\chi_h \circ \pi_2, C)(\chi_g(a)) \text{ if } a \in sup(B'^{*(\mathbf{O})} \otimes I_g^{(\mathbf{P})})$$
$$\chi_{g;h}(a) \stackrel{\Delta}{=} connect(\chi_g \circ \pi_2^{-1}, C')(\chi_h(a)) \text{ if } a \in sup(D^{(\mathbf{O})} \otimes I_h^{(\mathbf{P})})$$

  Now $\chi_{(f;g);h}$ and $\chi_{f;(g;h)}$ are defined as follows:

$$\chi_{(f;g);h}(a) \stackrel{\Delta}{=} connect(\chi_h \circ \pi_2, C)(\chi_{f;g}(a)) \text{ if } a \in sup(A^{*(\mathbf{O})} \otimes I_{f;g}^{(\mathbf{P})})$$
$$\chi_{(f;g);h}(a) \stackrel{\Delta}{=} connect(\chi_{f;g} \circ \pi_2^{-1}, C')(\chi_h(a)) \text{ if } a \in sup(D^{(\mathbf{O})} \otimes I_h^{(\mathbf{P})})$$
$$\chi_{f;(g;h)}(a) \stackrel{\Delta}{=} connect(\chi_{g;h} \circ \pi_1, B)(\chi_f(a)) \text{ if } a \in sup(A^{*(\mathbf{O})} \otimes I_f^{(\mathbf{P})})$$
$$\chi_{f;(g;h)}(a) \stackrel{\Delta}{=} connect(\chi_f \circ \pi_1^{-1}, B')(\chi_{g;h}(a)) \text{ if } a \in sup(D^{(\mathbf{O})} \otimes I_{g;h}^{(\mathbf{P})})$$

One way to see that these two bijective functions are equal is to view them as case trees, and consider every case. There are 13 such cases to consider, out of which three are not possible.

We show three cases:

1. If $a \in sup(A^{*(\mathbf{O})} \otimes I_f^{(\mathbf{P})})$, $\chi_f(a) \notin sup(B)$, and $\chi_f(a) \notin sup(C)$, then

$$\begin{aligned}&\chi_{(f;g);h}(a)\\=&connect(\chi_h \circ \pi_2, C)(\chi_{f;g}(a))\\=&connect(\chi_h \circ \pi_2, C)(\chi_f(a))\\=&\chi_f(a)\end{aligned}$$

   and

$$\begin{aligned}&\chi_{f;(g;h)}(a)\\=&connect(\chi_{g;h} \circ \pi_1, B)(\chi_f(a))\\=&\chi_f(a)\end{aligned}$$

   and thus equal.

2. Consider the case where $a \in sup(A^{*(\mathbf{O})} \otimes I_f^{(\mathbf{P})})$, $\chi_f(a) \notin sup(B)$, and $\chi_f(a) \in sup(C)$. This case is not possible, since $sup(C)$ is not a subset of the codomain of $\chi_f(a)$, which is $sup(A^{*(\mathbf{P})} \otimes B^{(\mathbf{P})} \otimes I_f^{(\mathbf{O})})$.

3. If $a \in sup(D^{(\mathbf{O})} \otimes I_h^{(\mathbf{P})})$, $\chi_h(a) \in sup(C')$, $\pi_2^{-1}(\chi_h(a)) \in sup(C^{(\mathbf{O})} \otimes I_g^{(\mathbf{P})})$, and $\chi_g(\pi_2^{-1}(\chi_h(a))) \in sup(B')$, then

$$\begin{aligned}&\chi_{(f;g);h}(a)\\=&connect(\chi_{f;g} \circ \pi_2^{-1}, C')(\chi_h(a))\\=&\chi_{f;g}(\pi_2^{-1}(\chi_h(a)))\\=&connect(\chi_f \circ \pi_1^{-1}, B')(\chi_g(\pi_2^{-1}(\chi_h(a))))\\=&\chi_f(\pi_1^{-1}(\chi_g(\pi_2^{-1}(\chi_h(a))))))\end{aligned}$$

   and

$$\begin{aligned}&\chi_{f;(g;h)}(a)\\=&connect(\chi_f \circ \pi_1^{-1}, B')(\chi_{g;h}(a))\\=&connect(\chi_f \circ \pi_1^{-1}, B')(connect(\chi_g \circ \pi_2^{-1}, C')(\chi_h(a)))\\=&connect(\chi_f \circ \pi_1^{-1}, B')(\chi_g(\pi_2^{-1}(\chi_h(a))))\\=&\chi_f(\pi_1^{-1}(\chi_g(\pi_2^{-1}(\chi_h(a))))))\end{aligned}$$

   and thus equal.

The other cases are done similarly.

- id_A is well-formed. For any interface A,

$$id_A \triangleq (\varnothing, \chi, A \Rightarrow A')$$

for an A' such that $\pi \vdash A =_{\mathbb{A}} A'$ and

$$\chi(a) \triangleq \pi(a) \text{ if } a \in sup(A^{*(\mathbf{O})})$$
$$\chi(a) \triangleq \pi^{-1}(a) \text{ if } a \in sup(A'^{(\mathbf{O})}.)$$

according to the definition.

We need to show that χ is a bijection:

$$\begin{aligned}\chi \in sup((A \Rightarrow A')^{(\mathbf{O})}) &\to sup((A \Rightarrow A')^{(\mathbf{P})}) \\ = sup(A^{*(\mathbf{O})} \cup A'^{(\mathbf{O})}) &\to sup(A^{*(\mathbf{P})} \cup A'^{(\mathbf{P})})\end{aligned}$$

This is true since π is a bijection in $sup(A) \to sup(A')$.

- id_A is an identity. For any morphism $f : A \to B$ we observe that $id_A; f$ is structurally equivalent to f, so by Theorem 4.1.1, $[\![id_A; f]\!] =_{\mathbb{A}} [\![f]\!]$.

  The case for $f; id_B$ is similar. □

Proof of Proposition 4.1.3.

- The tensor product is well-defined, i.e. for two morphisms f, g, $f \otimes g$ is a well-formed net. This is easy to see since f and g are well-formed.

- The tensor product is a bifunctor:

  - $id_A \otimes id_B = (\varnothing, \chi_1 \otimes \chi_2, A \otimes B \Rightarrow A' \otimes B') = id_{A \otimes B}$ by the definition of $id_{A \otimes B}$.
  - $(f; g) \otimes (h; i) = f \otimes h; g \otimes i$ by the definition of composition and tensor on morphisms.

- The coherence conditions of the natural isomorphisms are trivial since the isomorphisms amount to identities. □

Proof of Theorem 4.1.7. We show that for any trace s, $s \in [\![S]\!]$ implies $s \in [\![S']\!]$ by induction on the length of the trace.

- Hypothesis. If $s \in [\![S]\!]$ and thus $\mathit{initial}(S) \xrightarrow{s} (\{(\overline{t}_1, h_1) : E_1, (\overline{t}_2, h_2) : E_2\}, \overline{m})$ for some sets of threads $\overline{t}_1$ and $\overline{t}_2$, heaps h_1 and h_2, and a multiset of messages $\overline{m}$, then $\mathit{initial}(S') \xrightarrow{s} (\{(\overline{t}_1 \cup \overline{t}_2 \cup \overline{t}_p, h_1 \cup h_2) : E_{12}\}, \overline{m}_p)$ where $\overline{t}_p$ is a set of threads and $\overline{m}_p$ is a multiset of messages such that:

  1. each $t \in \overline{t}_p$ is on the form $t = (\texttt{spark}\ a, \overline{d})$ with $\chi(a) \in \mathit{sup}(A_1 \otimes A_2)$, and
  2. $\overline{m} = \overline{m}_p \uplus \{(\chi(a), \mathit{msg}(\overline{d})) \mid (\texttt{spark}\ a, \overline{d}) \in \overline{t}_p\}$.

  Intuitively, the net where E_1 and E_2 have been combined into one engine will not have pending messages (in $\overline{m}$) for communications between E_1 and E_2, but it can match the behaviour of such messages by threads that are just about to spark.

- Base case. Since any net can take zero steps, the case when $s = \epsilon$ is trivial.

- Inductive step. If $s = s'{::}\alpha$ and the hypothesis holds for s', then we have

$$\begin{aligned}\mathit{initial}(S) &\xrightarrow{s'} (\{(\overline{t}_1, h_1) : E_1, (\overline{t}_2, h_2) : E_2\}, \overline{m}) \\ &\to^* \xrightarrow{\alpha} (\{(\overline{t}'_1, h'_1) : E_1, (\overline{t}'_2, h'_2) : E_2\}, \overline{m}') \\ \mathit{initial}(S') &\xrightarrow{s'} (\{(\overline{t}_1 \cup \overline{t}_2 \cup \overline{t}_p, h_1 \cup h_2) : E_{12}\}, \overline{m}_p)\end{aligned}$$

  with t_p and $\overline{m}'$ as in the hypothesis. We first show that S' can match the silent steps that S performs, by induction on the number of steps, using the same induction hypothesis as above:

  - Base case. Trivial.
  - Inductive step. Assume that we have

$$\begin{aligned}\mathit{initial}(S) &\xrightarrow{s'} \to^* (\{(\overline{t}_1, h_1) : E_1, (\overline{t}_2, h_2) : E_2\}, \overline{m}) \\ \mathit{initial}(S') &\xrightarrow{s'} \to^* (\{(\overline{t}_1 \cup \overline{t}_2 \cup \overline{t}_p, h_1 \cup h_2) : E_{12}\}, \overline{m}_p)\end{aligned}$$

    Such that the induction hypothesis holds. We need to show that any step

$$\begin{aligned}&(\{(\overline{t}_1, h_1) : E_1, (\overline{t}_2, h_2) : E_2\}, \overline{m}) \to \\ &(\{(\overline{t}'_1, h'_1) : E_1, (\overline{t}'_2, h'_2) : E_2\}, \overline{m}')\end{aligned}$$

    can be matched by (any number of) silent steps of the S' configuration, such that the induction hypothesis still holds.

* A thread of S performs a silent step. This is trivial, since the threads of the engine configuration of S' includes all threads of the configurations of S, and its heap is the union of those of S.
* A thread of S does an internal engine send step. Since $\overline{t}_1 \cup \overline{t}_2 \cup \overline{t}_p$ includes all threads of the S configuration, and for the port name a in question $\chi(a) \in A_1 \cup A_2 = A_1 \otimes A_2$, this can be matched by the configuration of S' such that the induction hypothesis still holds.
* A thread S does an external engine send. This means that there is a thread $t \in \overline{t}_1 \cup \overline{t}_2$ on the form $t = (\texttt{spark}\ a, \overline{d})$, which after the step will be removed, adding the message $(\chi(a), msg(\overline{d}))$ to its multiset of messages, i.e. $\overline{m}' = \overline{m} \uplus \{(\chi(a), msg(\overline{d}))\}$. If $\chi(a) \in A_1 \cup A_2$, then the configuration S' can take zero steps, and thus include t in the set of threads ready to spark. The induction hypothesis still holds, since

$$\begin{aligned}
\overline{m}' &= \overline{m} && \uplus \{(\chi(a), msg(\overline{d}))\} \\
&= \overline{m}_p && \uplus \{(\chi(a), msg(\overline{d})) \mid (\texttt{spark}\ a, \overline{d}) \in \overline{t}_p\} \\
& && \uplus \{(\chi(a), msg(\overline{d}))\} \\
&= \overline{m}_p && \uplus \{(\chi(a), msg(\overline{d})) \mid (\texttt{spark}\ a, \overline{d}) \in \overline{t}_p \cup \{t\}\}.
\end{aligned}$$

  If $\chi(a) \in I$, then the configuration of S' can match the step of S, removing the thread t from also its set of threads. It is easy to see that the induction hypothesis holds also in this case.
* An engine of S receives a message. Then $\overline{m} = \{(a, \overline{d})\} \uplus \overline{m}'$ for a message such that the port $(\mathbf{O}, a) \in A_1 \cup A_2 = A_1 \otimes A_2$. Then $(a, \overline{d})$ is in $\overline{m}_p$ or in $\{(\chi(a), msg(\overline{d})) \mid (\texttt{spark}\ a, \overline{d}) \in \overline{t}_p\}$. If it is the former, E_{12} can receive the message and start a thread equal to that started in the configuration of S. If it is the latter, there is a thread $t = (\texttt{spark}\ \chi^{-1}(a), \overline{d}') \in \overline{t}_p$ with $\overline{d} = msg(\overline{d}')$ that can first take a send m step, adding it to the multiset of pending messages of the configuration of S', and then it can be received as in S.

Next we show that the α step can be matched: Assume that we have

$$\begin{aligned}
initial(S) &\xrightarrow{s'}\!\!\rightarrow^* (\{(\overline{t}_1, h_1) : E_1, (\overline{t}_2, h_2) : E_2\}, \overline{m}) \\
initial(S') &\xrightarrow{s'}\!\!\rightarrow^* (\{(\overline{t}_1 \cup \overline{t}_2 \cup \overline{t}_p, h_1 \cup h_2) : E_{12}\}, \overline{m}_p)
\end{aligned}$$

Such that the induction hypothesis holds. We need to show that for any α, a step

$$(\{(\overline{t}_1, h_1) : E_1, (\overline{t}_2, h_2) : E_2\}, \overline{m}) \xrightarrow{\alpha} (\{(\overline{t}'_1, h'_1) : E_1, (\overline{t}'_2, h'_2) : E_2\}, \overline{m}')$$

can be matched by the S' configuration, such that the induction hypothesis still holds. We have two cases:

- The configuration of S performs a send step. That is $\overline{m} = \{m\} \uplus \overline{m}'$ for an $m = (a, \overline{d})$ such that $(\mathbf{P}, a) \in A$. Since $sup(A)$ is disjoint from $sup(A_1 \cup A_2)$, the message is also in $\overline{m}_p$, so the configuration of S' can match the step.
- The configuration of S performs a receive step. This case is easy, as S and S' have the same interface A. □

Proof of Lemma 4.2.28. We show that $s \in [\![\delta_{\pi_{12},\pi_{23}}; \Pi_1]\!]$ implies $s \in [\![\mathbb{C}_{\pi_{12},\mathfrak{A}_1,\mathfrak{A}_2}]\!]$ and the converse (the Π_2 case is analogous), by induction on the trace length. There is a simple relationship between the heap structures of the respective net configurations — they have the same structure but the diagonal stores additional integers for identifying what "side" a move comes from. □

Proof of Lemma 4.2.12. By cases on (x):

- If $(x) = \bullet$, then $\overline{e : E} = \{e : E\} \cup \overline{e_o : E_o}$, $e \xrightarrow[E,\chi]{(y)} e'$ for some (y), $\overline{e' : E} = \{e' : E\} \cup \overline{e'_o : E_o}$. We have three cases for (y):
  - If $(y) = \bullet$, then $e \xrightarrow[E,\chi]{} e'$ and $\overline{m}' = \overline{m}$. Then we also have $n_2 = (\{e : E\} \cup \overline{e_o : E_o}, \overline{m} \uplus \{m\}) \to (\{e' : E\} \cup \overline{e'_o : E_o}, \overline{m} \uplus \{m\}) = n'_2$.
  - If $(y) = (\mathbf{P}, m')$, then $e \xrightarrow[E,\chi]{m'} e'$ and $\overline{m}' = \{m'\} \cup \overline{m}$. Then we also have $n_2 = (\{e : E\} \cup \overline{e_o : E_o}, \overline{m} \uplus \{m\}) \to (\{e' : E\} \cup \overline{e'_o : E_o}, \{m'\} \uplus \overline{m} \uplus \{m\}) = n'_2$.
  - If $(y) = (\mathbf{O}, m')$, then $e \xrightarrow[E,\chi]{\overline{m'}} e'$ and $\overline{m} = \{m'\} \uplus \overline{m}'$. Then we also have $n_2 = (\{e : E\} \cup \overline{e_o : E_o}, \{m'\} \uplus \overline{m}' \uplus \{m\}) \to (\{e' : E\} \cup \overline{e'_o : E_o}, \overline{m}' \uplus \{m\}) = n'_2$.
- If $(x) = (\mathbf{P}, m')$, then $\overline{e' : E} = \overline{e : E}$ and $\overline{m} = \{m'\} \uplus \overline{m}'$. Then we also have $n_2 = (\overline{e : E}, \{m'\} \uplus \overline{m}' \uplus \{m\}) \xrightarrow{m'} (\overline{e : E}, \overline{m}' \uplus \{m\}) = n'_2$.
- If $(x) = (\mathbf{O}, m')$, where $m' = (a, p, p', d)$ then $\overline{e' : E} = \overline{e : E}$ and $\overline{m}' = \{(\chi(a), p, p', d)\} \uplus \overline{m}$. Then we also have $n_2 = (\overline{e : E}, \overline{m} \uplus \{m\}) \xrightarrow{\overline{m'}} (\overline{e : E}, \{(\chi(a), p, p', d)\} \uplus \overline{m} \uplus \{m\}) = n'_2$. □

Proof of Lemma 4.2.13.

1. $s = s_1{::}(l, m_1){::}(\mathbf{O}, m){::}s_2 \in [\![f]\!]$ means that

$$initial(f) \xrightarrow{s_1} \xrightarrow{(x)}{}^{*} n_1 \xrightarrow{(l,m_1)} n_2 \xrightarrow{(y)}{}^{*} \xrightarrow{(\mathbf{O},m)} n_3 \xrightarrow{(z)}{}^{*} \xrightarrow{s_2} n_4$$

for net configurations n_1, n_2, n_3, n_4. For clarity, we take $(x), (y), (z)$ to be "names" for the silent transitions. We show that there exist n_2' and (y') such that

$$initial(f) \xrightarrow{s_1} \xrightarrow{(x)}{}^{*} n_1 \xrightarrow{(\mathbf{O},m)} \xrightarrow{(l,m_1)} n_2' \xrightarrow{(y')}{}^{*} n_3 \xrightarrow{(z)} \xrightarrow{s_2} n$$

by induction on the length of $\xrightarrow{(y)}{}^{*}$:

- Base case. If $\xrightarrow{(y)}{}^{*}$ is the identity relation, then assume

$$n_1 \xrightarrow{(l,m_1)} n_2 \xrightarrow{(\mathbf{O},m)} n_3$$

  Let $n_1 = (\overline{e_1 : E}, \overline{m}_1)$, $n_2 = (\overline{e_2 : E}, \overline{m}_2)$, $m = (a, p, p', d)$, and $m' = (\chi(a), p, p', d)$. Then $n_3 = (\overline{e_2 : E}, \{m'\} \uplus \overline{m}_2)$ by the definition of $\rightarrow$. Since we have $(\mathbf{O}, a) \in I$, we also have $n_1 \xrightarrow{(\mathbf{O},m)} (\overline{e_1 : E}, \{m'\} \uplus \overline{m}_1)$. Also, since $n_1 \xrightarrow{(l,m_1)} n_2$ we have $(\overline{e_1 : E}, \{m'\} \uplus \overline{m}_2) \xrightarrow{(l,m_1)} n_3$ by Lemma 4.2.12. Composing the relations, we get

$$n_1 \xrightarrow{(\mathbf{O},m)} \xrightarrow{(l,m_1)} n_3$$

  which completes the base case.

- Inductive step. If $\xrightarrow{(y)}{}^{*} = \xrightarrow{(y_o)}{}^{*} \xrightarrow{\bullet}$ such that for any n_3'

$$n_1 \xrightarrow{(l,m_1)} n_2 \xrightarrow{(y_o)}{}^{*} \xrightarrow{(\mathbf{O},m)} n_3'$$

  implies that there exist n_2' and (y_o') with

$$n_1 \xrightarrow{(\mathbf{O},m)} \xrightarrow{(l,m_1)} n_2' \xrightarrow{(y_o')}{}^{*} n_3'$$

  then assume

$$n_1 \xrightarrow{(l,m_1)} n_2 \xrightarrow{(y_o)}{}^{*} n_{y_o} \xrightarrow{\bullet} n_y \xrightarrow{(\mathbf{O},m)} n_3$$

  Let $n_{y_o} = (\overline{e_{y_o} : E}, \overline{m}_{y_o})$, $n_y = (\overline{e_y : E}, \overline{m}_y)$, $m = (a, p, p', d)$, and $m' = (\chi(a), p, p', d)$. Then $n_3 = (\overline{e_y : E}, \{m'\} \uplus \overline{m}_y)$ by the

definition of $\rightarrow$. Since we have $(\mathbf{O}, a) \in I$, we also have $n_{y_o} \xrightarrow{(\mathbf{O},m)} (\overline{e_{y_o} : E}, \{m'\} \uplus \overline{m}_{y_o})$. Also, since $n_{y_o} \xrightarrow{\bullet} n_y$ by Lemma 4.2.12 we have $(\overline{e_{y_o} : E}, \{m'\} \uplus \overline{m}_{y_o}) \xrightarrow{\bullet} n_3$. Composing the relations, we get

$$n_1 \xrightarrow{(l,m_1)} n_2 \xrightarrow{(y_o)^*} n_{y_o} \xrightarrow{(\mathbf{O},m)} (\overline{e_{y_o} : E}, \{m'\} \uplus \overline{m}_{y_o}) \xrightarrow{\bullet} n_3$$

Applying the hypothesis, we finally get

$$n_1 \xrightarrow{(\mathbf{O},m)} \xrightarrow{(l,m_1)} n_2' \xrightarrow{(y_o')^*} \xrightarrow{\bullet} n_3$$

which completes the first part of the proof.

2. $s = s_1::(\mathbf{P}, m)::(l, m_1)::s_2 \in [\![f]\!]$ means that

$$initial(f) \xrightarrow{s_1} \xrightarrow{(x)^*} n_1 \xrightarrow{(\mathbf{P},m)} n_2 \xrightarrow{(y)^*} \xrightarrow{(l,m_1)} n_3 \xrightarrow{(z)^*} \xrightarrow{s_2} n_4$$

for net configurations n_1, n_2, n_3, n_4 and $(x), (y), (z)$ names for the silent transitions. We show that there exist (y') and n_2' such that

$$initial(f) \xrightarrow{s_1} \xrightarrow{(x)^*} n_1 \xrightarrow{(y')^*} n_2' \xrightarrow{(l,m_1)} \xrightarrow{(\mathbf{P},m)} n_3 \xrightarrow{(z)^*} \xrightarrow{s_2} n$$

by induction on the length of $\xrightarrow{(y)^*}$:

- Base case. If $\xrightarrow{(y)^*}$ is the identity relation, then assume

$$n_1 \xrightarrow{(\mathbf{P},m)} n_2 \xrightarrow{(l,m_1)} n_3$$

Let $n_2 = (\overline{e_2 : E}, \overline{m}_2)$, $n_3 = (\overline{e_3 : E}, \overline{m}_3)$, $m = (a, p, p', d)$ Then $n_1 = (\overline{e_2 : E}, \{m\} \uplus \overline{m}_2)$ by the definition of $\rightarrow$. Since $(\mathbf{P}, a) \in I$, $(\overline{e_3 : E}, \{m\} \uplus \overline{m}_3) \xrightarrow{(\mathbf{P},m)} n_3$. Also, since $n_2 \xrightarrow{(l,m_1)} n_3$ we have $n_1 \xrightarrow{(l,m_1)} (\overline{e_3 : E}, \{m\} \uplus \overline{m}_3)$ by Lemma 4.2.12. Composing the relations, we get

$$n_1 \xrightarrow{(l,m_1)} \xrightarrow{(\mathbf{P},m)} n_3$$

which completes the base case.

- Inductive step. If $\xrightarrow{(y)^*} = \xrightarrow{\bullet} \xrightarrow{(y_o)^*}$ such that for any n_1'

$$n_1' \xrightarrow{(\mathbf{P},m)} \xrightarrow{(y_o)^*} n_2 \xrightarrow{(l,m_1)} n_3$$

implies that there exist n_2' and (y_o') with

$$n_1' \xrightarrow{(y_o')}{}^{*} n_2' \xrightarrow{(l,m_1)} \xrightarrow{(\mathbf{P},m)} n_3$$

then assume

$$n_1 \xrightarrow{(\mathbf{P},m)} n_m \xrightarrow{\bullet} n_y \xrightarrow{(y_o)}{}^{*} n_2 \xrightarrow{(l,m_1)} n_3$$

Let $n_m = (\overline{e_m : E}, \overline{m}_m)$, $n_y = (\overline{e_y : E}, \overline{m}_y)$, and $m = (a, p, p', d)$. Then $n_1 = (\overline{e_m : E}, \{m\} \uplus \overline{m}_m)$ by the definition of $\rightarrow$. Since we have $(\mathbf{P}, a) \in I$, we have $(\overline{e_y : E}, \{m\} \uplus \overline{m}_y) \xrightarrow{(\mathbf{P},m)} n_y$. Also, since $n_m \xrightarrow{\bullet} n_y$ we have $n_1 \xrightarrow{\bullet} (\overline{e_y : E}, \{m\} \uplus \overline{m}_y)$ by Lemma 4.2.12. Composing the relations, we get

$$n_1 \xrightarrow{\bullet} (\overline{e_y : E}, \{m\} \uplus \overline{m}_y) \xrightarrow{(\mathbf{P},m)} n_y \xrightarrow{(y_o)}{}^{*} n_2 \xrightarrow{(l,m_1)} n_3$$

Applying the hypothesis, we finally get

$$n_1 \xrightarrow{\bullet} \xrightarrow{(y_o')}{}^{*} n_2' \xrightarrow{(l,m_1)} \xrightarrow{(\mathbf{P},m)} n_3$$

which completes the proof. □

Proof of Lemma 4.2.14. Induction on $\preccurlyeq$. The base case is trivial. Consider the case where $s = s_1 :: \alpha_2 :: \alpha_1 :: s_2$ and $s' = s_1 :: \alpha_1 :: \alpha_2 :: s_2$. Let $\alpha_1 = (l, (a_1, p_1, p_1', d_1))$ and $\alpha_2 = (l, (a_2, p_2, p_2', d_2))$.

1. Induction on the length of s_2. In the base case, we have (by associativity and commutatitivity of $\cup$): $enabled(s_1 :: \alpha_2 :: \alpha_1) = enabled(s_1) \cup \{(a, p_2') \mid a_2 \vdash_{\mathfrak{A}} a\} \cup \{(a, p_1') \mid a_1 \vdash_{\mathfrak{A}} a\} = enabled(s_1) \cup \{(a, p_1') \mid a_1 \vdash_{\mathfrak{A}} a\} \cup \{(a, p_2') \mid a_2 \vdash_{\mathfrak{A}} a\}$.

2. Induction on the length of s_2 as in 1.

3. Induction on the length of s_2. In the base case, we have (since by the def. of $\preccurlyeq$, $p_1 \neq p_2'$ and $p_2 \neq p_1'$):

$$\begin{aligned}
fp(s_1 :: \alpha_2 :: \alpha_1) &= \\
fp(s_1 :: \alpha_2) \cup (\{p_1\} \setminus bp(s_1 :: \alpha_2)) &= \\
fp(s_1) \cup (\{p_2\} \setminus bp(s_1)) \cup (\{p_1\} \setminus (bp(s_1) \cup \{p_2'\})) &= \\
fp(s_1) \cup (\{p_2\} \setminus (bp(s_1) \cup \{p_1'\})) \cup (\{p_1\} \setminus bp(s_1)) &= \\
fp(s_1) \cup (\{p_1\} \setminus bp(s_1)) \cup (\{p_2\} \setminus (bp(s_1) \cup \{p_1'\})) &= \\
fp(s_1 :: \alpha_1) \cup (\{p_2\} \setminus bp(s_1 :: \alpha_1)) &= \\
fp(s_1 :: \alpha_1 :: \alpha_2) &
\end{aligned}$$

□

Proof of Lemma 4.2.15. Induction on $\preccurlyeq$. The base case is trivial. We show the case of a single swapping. If $s' \preccurlyeq s$, we have $s = s_1{::}\alpha_2{::}\alpha_1{::}s_2$ and $s' = s_1{::}\alpha_1{::}\alpha_2{::}s_2$ for some $s_1, s_2, \alpha_1, \alpha_2$. Obviously, $s'{::}\alpha \preccurlyeq s{::}\alpha$.

We have to show that if $s{::}\alpha \in P_{\mathfrak{A}}$, then $s'{::}\alpha \in P_{\mathfrak{A}}$, i.e. that $s'{::}\alpha$ fulfils the legality conditions imposed by $P_{\mathfrak{A}}$:

- It is easy to see that $s'{::}\alpha$ has unique pointers and is correctly labelled.
- $s'{::}\alpha$ is justified since $enabled(s) = enabled(s')$ by Lemma 4.2.14.
- To see that $s'{::}\alpha$ strictly scoped, consider the ("worst") case when

  $$(l, (a, p, p', d)){::}s_3{::}\alpha \subseteq s'{::}\alpha \text{ and } a \in ans_{\mathfrak{A}}$$

  (i.e. we pick the segment that goes right up to the end of the trace). We consider the different possibilities of the position of this answer message:

  - If $(l, (a, p, p', d)) \subseteq s_1$, let $s_4' = (l, (a, p, p', d)){::}s_1'{::}\alpha_1{::}\alpha_2{::}s_2{::}\alpha \subseteq s'{::}\alpha$ and $s_4 = (l, (a, p, p', d)){::}s_1'{::}\alpha_2{::}\alpha_1{::}s_2{::}\alpha$. We also know that $p \notin fp(s_4)$ as $s{::}\alpha \in P_{\mathfrak{A}}$. Now, since $s_4' \preccurlyeq s_4$, we have $fp(s_4) = fp(s_4')$ by Lemma 4.2.14 and thus also $p \notin fp(s_4')$.
  - If $(l, (a, p, p', d)) = \alpha_2$. We know that $p \notin fp(s_2{::}\alpha)$ by $s{::}\alpha \in P_{\mathfrak{A}}$. Since $s' \in P_{\mathfrak{A}}$ we have $p \notin fp(\alpha_1)$ and can so conclude that $p \notin fp(\alpha_1{::}s_2{::}\alpha)$.
  - If $(l, (a, p, p', d)) = \alpha_1$ or $(l, (a, p, p', d)) \subseteq s_2$, $p \notin fp(s_2{::}\alpha)$ follows immediately from $s \in P_{\mathfrak{A}}$.
  - If $(l, (a, p, p', d)) = \alpha$, $p \notin fp(\epsilon) = \varnothing$ is trivially true.
- To see that $s'{::}\alpha$ is strictly nested, assume

  $$(l_1, (a_1, p, p', d_1)){::}s_1{::}(l_2, (a_2, p', p'', d_2)){::}s_2{::}(l_3, (a_3, p', p''', d_3)) \subseteq s'{::}\alpha$$

  for port names $a_1, a_2 \in qst_{\mathfrak{A}}$ and $a_3 \in ans_{\mathfrak{A}}$. We have to show that this implies $(l_4, (a_4, p'', -, d_4)) \subseteq s_2$, for a port name $a_4 \in ans_{\mathfrak{A}}$. We proceed by considering the possible positions of the last message in the segment:

  - If $(l_3, (a_3, p', p''', d_3)) \subseteq s'$, then the proof is immediate, by $s' \in P_{\mathfrak{A}}$ being strictly nested.
  - If $(l_3, (a_3, p', p''', d_3)) = \alpha$ we use the fact that $s{::}\alpha \in P_{\mathfrak{A}}$ is strictly nested. We assume that the implication (using the same names) as above holds but instead for $s{::}\alpha$, and show that any swappings that can have occurred in s' that reorder the a_1, a_2, a_4 moves would render s' illegal:

* If a_2 was moved before a_1, then s' would not be justified.
* If a_4 was moved before a_2, then s' would not be justified.

As the order is preserved, this shows that the swappings must be done in a way such that the implication holds for $s'::\alpha$. □

Proof of Lemma 4.2.17. For convenience, let $(f, \mathfrak{A} \Rightarrow \mathfrak{A}') = \mathbb{CC}_{\pi,\mathfrak{A},\mathfrak{A}'}$, $S_1 = cc^{st,alt}_{\mathfrak{A},\mathfrak{A}'}$ and $S_2 = [\![f]\!]$. We show that $s \in S_1$ implies $s \in S_2$, by induction on the length of s:

- Hypothesis. If s has even length, then $initial(f) \xrightarrow{s} (\{(\varnothing, h) : E\}, \varnothing)$ and h is exactly (nothing more than) a copycat heap for s over $\mathfrak{A} \Rightarrow \mathfrak{A}'$. In other words, there are no threads running and no pending messages and the heap is precisely specified.
- Base case. Immediate.
- Inductive step. At any point in the execution of the configuration of f, an **O**-labelled message can be received, so that case is rather uninteresting. Since the trace s is alternating, we consider two messages in each step:

  Assume $s = s'::(\mathbf{O}, (a_1, p_1, p_1', d_1))::(\mathbf{P}, (a_2, p_2, p_2', d_2)) \in S_1$ and that $s' \in S_2$. From the definition of cc we know that $a_2 = \tilde{\pi}_{\mathbb{A}}(a_1)$, $p_2 = \tilde{\pi}_{\mathbb{P}}(p_1)$, $p_2' = \tilde{\pi}_{\mathbb{P}}(p_2)$, and $d_1 = d_2$.

  We are given that $initial(f) \xrightarrow{s}{}' (\{(\varnothing, h) : E\}, \varnothing)$ as in the hypothesis. We have five cases for the port name a_1. We show the first three, as the others are similar. In each case our single engine will receive a message and start a thread:

  – If $a_1 \in ini_{\mathfrak{A}'}$, then (since s is justified) $p_2 = p_1'$ and (by the definition of $\pi'_{\mathbb{A}}$) $a_2 = \pi^{-1}_{\mathbb{A}}(a_1)$. The engine runs the first clause of the copycat definition, and chooses to create the pointer p_2 and then performs a send operation. We thus get:

  $$initial(f) \xrightarrow{s} (\{(\varnothing, h \cup \{p_2' \mapsto p_1'\})\}, \varnothing)$$

  It can easily be verified that the hypothesis holds for this new state.

  – If $a_1 \in (opp_{\mathfrak{A}'} \cap qst_{\mathfrak{A}'}) \setminus ini_{\mathfrak{A}'}$, then $a_2 = \pi^{-1}_{\mathbb{A}}(a_1)$. Since s is justified and strictly nested, there is a message $(\mathbf{P}, (a_3, p_3, p_1, d_3)) \sqsubseteq s'$ that is pending.

  By the hypothesis there is a message $(\mathbf{O}, (\pi'_{\mathbb{A}}(a_3), p_4, p_4', d_4)) \sqsubseteq s'$ with $h(p_1) = p_4'$, which means that the `ccq` instruction can be run, yielding the following:

  $$initial(f) \xrightarrow{s} (\{(\varnothing, h \cup \{p_2' \mapsto p_1'\})\}, \varnothing)$$

The hypothesis can easily be verified also in this new state.

– If $a_1 \in opp_{\mathfrak{A}'} \cap ans_{\mathfrak{A}'}$, then $a_2 = \pi_{\mathbb{A}}^{-1}(a_1)$. Since s is justified and strictly nested, there is a prefix $s_1{::}(\mathbf{P}, (a_3, p_3, p_1, d_3)) \preccurlyeq s'$ whose last message is a pending question. By the hypothesis s_1 is then on the form $s_1 = s_2{::}(\mathbf{O}, (\pi'_{\mathbb{A}}(a_3), \tilde{\pi}_{\mathbb{P}}(p_3), \tilde{\pi}_{\mathbb{P}}(p_1), d_4))$ with $h = h' \cup \{p_1 \mapsto \tilde{\pi}_{\mathbb{P}}(p_1)\}$, which means that the `cca` instruction can be run, yielding the following:

$$initial(f) \xrightarrow{s} (\{(\varnothing, h')\}, \varnothing)$$

The hypothesis is still true; the a_3 question is no longer pending and its pointer is removed from the heap (notice that $p_2 = \tilde{\pi}_{\mathbb{P}}(p_1)$). □

Proof of Lemma 4.2.18. By induction on $\preccurlyeq$.

- Base case. This means that $s = s_1{::}o{::}s_2 \in cc^{alt}_{\mathfrak{A},\mathfrak{A}'}$. But since $p \notin s_2$ and by the definition of the alternating copycat, $s_2 = \epsilon$. It is easy to check that $s{::}p \in cc^{alt}_{\mathfrak{A},\mathfrak{A}'}$ and that it is legal.
- Inductive step. Assume $s \preccurlyeq s'$ for an $s' \in P_{\mathfrak{A}\Rightarrow\mathfrak{A}'}$ such that $s'{::}p \in cc_{\mathfrak{A},\mathfrak{A}'}$. By Lemma 4.2.15, $s{::}p \in cc_{\mathfrak{A},\mathfrak{A}'}$. □

Proof of Lemma 4.2.21.

1. For convenience, we give the composition of silent steps a name, $n \xrightarrow{(x)}{}^{*} n'$. We proceed by induction on the length of (x):
   - Base case. Immediate.
   - Inductive step. If $n \rightarrow \xrightarrow{(x')}{}^{*} n'$, we analyse the first silent step, which means that a thread t of the engine in the net takes a step:
     – In the cases where an instruction that does not change or depend on the heap is run, the step cannot affect $ready(n)$.
     – In the case where the instruction is in $\{$`cci`, `ccq`, `exi`, `exq`$\}$, we note that the heap is not *changed*, but merely extended with a fresh mapping which can not have appeared earlier in the trace.
     – If the instruction is `cca`, since the trace s is strictly nested by assumption, the input message that this message stems from occurs in a position in the trace where it would later be illegal to mention the deallocated pointer again.
2. Immediate. □

Proof of Lemma 4.2.22. Induction on the length of s. The base case is immediate.

We need to show that if the theorem holds for a trace s, then it also holds for $s{::}\alpha$. We thus assume that there exists a permutation $\pi_{\mathbb{P}}$ such that the hypothesis holds for s and that $\mathit{initial}(\mathbb{C}) \xrightarrow{s} n \rightarrow^* \xrightarrow{\alpha} n'$.

1. If $\alpha = (\mathbf{P}, (\tilde{\pi}_{\mathbb{A}}(a), \tilde{\pi}_{\mathbb{P}}(p), \tilde{\pi}_{\mathbb{P}}(p'), d))$ then by (2) there must be a message $o = (\mathbf{O}, (a, p, p', d))$ such that $s = s_1{::}o{::}s_2$ and $\alpha \in \mathit{ready}(n)$. Since we "chose" $\pi_{\mathbb{P}}$ such that p can only be gotten from the thread spawned by o, we can proceed by cases as we did Lemma 4.2.17 to see that the heap structure is correct in each case.

2. - If $\alpha = (\mathbf{P}, (\tilde{\pi}_{\mathbb{A}}(a), \tilde{\pi}_{\mathbb{P}}(p), \tilde{\pi}_{\mathbb{P}}(p'), d))$ then by (2) there must be a message $o = (\mathbf{O}, (a, p, p', d))$ such that $s = s_1{::}o{::}s_2$ and $\alpha \in \mathit{ready}(n)$. By Lemma 4.2.21, $\mathit{ready}(n) = \mathit{ready}(n') \cup \{\alpha\}$. We can easily verify that (2) holds for n'.
   - If $\alpha = (\mathbf{O}, (a, p_1, p_1', d))$, then we can proceed as in Lemma 4.2.17 to see that a message $p = (\mathbf{P}, (\tilde{\pi}_{\mathbb{A}}(a), p_2, p_2', d)) \in \mathit{ready}(n')$. We then simply construct our extended permutation such that the hypothesis holds. □

Proof of Lemma 4.2.26. We show that $s' \in (S_f ;_{\mathfrak{B}} S_g)^{st,alt}$ implies that there exists a $\pi_{\mathbb{P}}$ such that $\pi_{\mathfrak{A},\mathfrak{C}} \cdot \pi_{\mathbb{P}} \cdot s' \in [\![f ;_{GAM} g]\!] = [\![\Lambda_A^{-1}(\Lambda_A(f) \otimes \Lambda_{B'}(g); K_{\mathfrak{A},\mathfrak{B},\mathfrak{C}})]\!] = [\![\Lambda_A(f)]\!] \otimes [\![\Lambda_{B'}(g)]\!]; [\![K_{\mathfrak{A},\mathfrak{B},\mathfrak{C}}]\!]$. Recall the definition of game composition:

$$S_f ;_{\mathfrak{B}} S_g \triangleq \{s \downarrow B \mid s \in \mathit{traces}_{A \otimes B \otimes C} \wedge s \downarrow C \in S_f \wedge \pi_{\mathfrak{B}} \cdot s^{*B} \downarrow A \in S_g\}$$

We proceed by induction on the length of such an s:

- Hypothesis. There exists an s_K such that $\mathit{initial}(K_{\mathfrak{A},\mathfrak{B},\mathfrak{C}}) \xrightarrow{s_K} n$ where $n = (\{(\varnothing, h) : E\}, \varnothing)$ and h is exactly (nothing more than) the union of a copycat heap for s_K over $\mathfrak{A}' \Rightarrow \mathfrak{A}$, a copycat heap for s_K over $\mathfrak{C} \Rightarrow \mathfrak{C}'$ and an extended copycat heap for s_K over $\mathfrak{B} \Rightarrow \mathfrak{B}'$.

  Let

  $$\begin{aligned}
  s_f &\triangleq s \downarrow C \\
  s_g &\triangleq \pi_{\mathfrak{B}} \cdot s^{*B} \downarrow A \\
  s_{f;g} &\triangleq s \downarrow B \\
  s_{Kf} &\triangleq s_K{-}A', B', C, C', \text{ the part of } s_K \text{ relating to } f \\
  s_{Kg} &\triangleq s_K{-}A, A', B, C', \text{ the part of } s_K \text{ relating to } g \\
  s_{Kf;g} &\triangleq s_K{-}A, B, B', C, \text{ the part of } s_K \text{ relating to the whole game net.}
  \end{aligned}$$

We require that s_K fulfils $s^*_{Kf} = s_f$, $s^*_{Kg} = s_g$, and $s_{Kf;g} = \pi_{\mathfrak{A},\mathfrak{C}} \cdot \pi_{\mathbb{P}} \cdot s_{f;g}$. Note that $s_{Kf;g}$ is the trace of $f ;_{GAM} g$, by the definition of trace composition.

- Base case. Immediate.
- Inductive step. Assume $s = s'::\alpha$ and that the hypothesis holds for s' and some $\pi'_{\mathbb{P}}$ and s'_K. We proceed by cases on the α message:
  - If $\alpha = (\mathbf{O}, (a, p, p', d))$, we have three cases:
    * If $a \in sup(A)$, intuitively this means that we are getting a message from outside the K engine, and need to propagate it via K to f. We construct s_K and $\pi_{\mathbb{P}}$, such that

      $$s_K = s'_K::(\mathbf{O}, (\pi_{\mathfrak{A}}(a), \tilde{\pi}_{\mathbb{P}}(p), \tilde{\pi}_{\mathbb{P}}(p'), d))::\alpha^*$$

      by further subcases on a ($\pi_{\mathbb{P}}$ will be determined by steps of the K configuration):
      · $a \in ini_{\mathfrak{A}}$ cannot be the case because an initial message in A must be justified by an initial (**O**-message) in C, and so must be a **P**-message.
      · If $a \in (qst_{\mathfrak{A}} \setminus ini_{\mathfrak{A}}) \cup ans_{\mathfrak{A}}$, this means that $s' \downarrow C::\alpha = (s'::\alpha) \downarrow C$ as the message must be justified by a message from $\mathfrak{A}$. As f is **O**-closed $s \downarrow C \in [\![\Lambda_A(f)]\!]$. This trace can be stepped to by n' just like how it was done in Lemma 4.2.17. We can verify that the parts of the hypothesis not in that theorem hold — in particular for this case we have $s_{Kf} = s'_{Kf}::\alpha^*$, so indeed $s^*_{Kf} = s_f$ as required.
    * $a \in sup(B)$:
      Intuitively this means that g is sending a message to f, which has to go through K. We construct s_K and $\pi_{\mathbb{P}}$, such that $s_K = s'_K::(\mathbf{O}, (a, \tilde{\pi}_{\mathbb{P}}(p), \tilde{\pi}_{\mathbb{P}}(p'), d))::\pi_{\mathfrak{B}} \cdot \alpha^*$, by further subcases on a ($\pi_{\mathbb{P}}$ will be determined by steps of the K configuration):
      · If $a \in ini_{\mathfrak{B}}$, there must be a pending **P**-message from $\mathfrak{C}$ justifying α in s', i.e. $(\mathbf{P}, (a_o, p_o, \tilde{\pi}_{\mathbb{P}}(p), d_o)) \subseteq s'$ and then by Definition 4.2.20 $h(\tilde{\pi}_{\mathbb{P}}(p)) = (p, \varnothing)$ (as $\tilde{\pi}_{\mathbb{P}}$ is its own inverse). This means that (running the `exi` instruction) we get:

        $$n' \xrightarrow{(\mathbf{O}, (\pi_{\mathfrak{B}}(a), \tilde{\pi}_{\mathbb{P}}(p), \tilde{\pi}_{\mathbb{P}}(p'), d))} \rightarrow^* \xrightarrow{\alpha^*}$$
        $$(\{(\varnothing, h \cup \{p' \mapsto (\tilde{\pi}_{\mathbb{P}}(p'), p)\}) : E\}, \varnothing) =n$$

Now $\pi_{\mathfrak{B}} \cdot \alpha^*$ is a new pending **P**-question in the trace that is initial in $\mathfrak{B} \Rightarrow \mathfrak{B}'$, but our new heap mapping fulfils clause (2) of Definition 4.2.25 as required.

· If $a \in (qst_{\mathfrak{B}} \setminus ini_{\mathfrak{B}}) \cup ans_{\mathfrak{B}}$, this is similar to the $\mathfrak{A}$ case (note that the extended copycat only differs from the ordinary copycat for initial messages).

* If $a \in sup(C)$.
Intuitively this means that we are getting a message from outside the K engine, and need to propagate it via K to g. We construct s_K and $\pi_{\mathbb{P}}$, such that:

$$s_K = s'_K :: (\mathbf{O}, (\pi_{\mathfrak{C}}(a), \tilde{\pi}_{\mathbb{P}}(p), \tilde{\pi}_{\mathbb{P}}(p'), d)) :: \alpha^*$$

In this case, the code that we will run is just that of $\mathbb{C}$, so we can proceed like in Lemma 4.2.17, easily verifying our additional assumptions.

– If $\alpha = (\mathbf{P}, (a, p, p', d))$, we have three cases:

* If $a \in sup(A)$, intuitively this means that we get a message from f and need to propagate it via K to the outside. By further subcases on a, we construct s_K and $\pi_{\mathbb{P}}$, such that:

$$s_K = s'_K :: \alpha^* :: (\mathbf{P}, (\pi_{\mathfrak{A}}(a), \tilde{\pi}_{\mathbb{P}}(p), \tilde{\pi}_{\mathbb{P}}(p'), d))$$

The pointer permutation $\pi_{\mathbb{P}}$ will be determined by steps of the K configuration.

· If $a \in ini_{\mathfrak{A}}$, then α must be justified in s' by a pending and initial **P**-question from $\mathfrak{B}$ by the definition of $\mathfrak{A} \Rightarrow \mathfrak{B}$ which must in turn be justified by a pending and initial **O**-question from $\mathfrak{C}$ by the definition of $\mathfrak{B} \Rightarrow \mathfrak{C}$. In s'_K, we have (since $s'_{Kf;g} = \pi_{\mathfrak{A},\mathfrak{C}} \cdot \pi_{\mathbb{P}} \cdot s'_{f;g}$)

$$s'_K = s_1 :: (\mathbf{O}, (a_{\mathfrak{C}'}, p_{\mathrm{o}}, p_{\mathfrak{C}'}, d_{\mathfrak{C}'})) :: s_2 :: (\mathbf{P}, (a_{\mathfrak{B}}, p_{\mathfrak{C}'}, p, d_{\mathfrak{C}'})) :: s_3$$

This means that clause (2) in Definition 4.2.25 applies, such that $h(p) = (\tilde{\pi}_{\mathbb{P}}(p), \tilde{\pi}_{\mathbb{P}}(p_{\mathrm{o}}))$ and that (running the exq instruction) we get:

$$n' \xrightarrow{\alpha^*} \xrightarrow{(\mathbf{P}, (\pi_{\mathfrak{A}}(a), \tilde{\pi}_{\mathbb{P}}(p), \tilde{\pi}_{\mathbb{P}}(p'), d))}$$
$$(\{(\varnothing, h \cup \{\tilde{\pi}_{\mathbb{P}}(p') \mapsto (p', d)\}) \,:\, E\}, \varnothing) = n$$

Clause (1) of Definition 4.2.25 applies to these new messages and trivially holds.

- When $a \in (qst_{\mathfrak{A}} \setminus ini_{\mathfrak{A}}) \cup ans_{\mathfrak{A}}$, the code that we will run is just that of $\mathbb{C}$, so we can proceed like in Lemma 4.2.17, also verifying our additional assumptions.

* If $a \in sup(B)$, intuitively this means that f is sending a message to g, which has to go through K.
  - $a \in ini_{\mathfrak{B}}$ cannot be the case for a **P**-message.
  - When $a \in (qst_{\mathfrak{B}} \setminus ini_{\mathfrak{B}}) \cup ans_{\mathfrak{B}}$, the code that we will run is just that of $\mathbb{C}$, so we can proceed like in Lemma 4.2.17, also verifying our additional assumptions.
* If $a \in sup(C)$, intuitively this means that we get a message from g and need to propagate it via K to the outside.
  - $a \in ini_{\mathfrak{C}}$ cannot be the case for a **P**-message.
  - When $a \in (qst_{\mathfrak{C}} \setminus ini_{\mathfrak{C}}) \cup ans_{\mathfrak{C}}$, the code that we will run is just that of $\mathbb{C}$, so we can proceed like in Lemma 4.2.17, also verifying our additional assumptions. □

Proof of Lemma 4.2.27. Similar to Lemma 4.2.22 and Lemma 4.2.26. We first identify the set $ready(n)$ with "uncopied" messages of a K net configuration n and show that these are legal according to the game composition. Then it follows by induction that, assuming a heap as in Lemma 4.2.26, the $ready(n)$ set is precisely those messages. □