Compiling Concurrency Correctly
Verifying Software Transactional Memory

Liyang HU

A Thesis submitted for the degree of Doctor of Philosophy

School of Computer Science

University of Nottingham

June 2012

Abstract

Concurrent programming is notoriously difficult, but with multi-core processors becoming the norm, is now a reality that every programmer must face. Concurrency has traditionally been managed using low-level mutual exclusion *locks*, which are error-prone and do not naturally support the compositional style of programming that is becoming indispensable for today's large-scale software projects.

A novel, high-level approach that has emerged in recent years is that of *software transactional memory* (STM), which avoids the need for explicit locking, instead presenting the programmer with a declarative approach to concurrency. However, its implementation is much more complex and subtle, and ensuring its correctness places significant demands on the compiler writer.

This thesis considers the problem of formally verifying an implementation of STM. Utilising a minimal language incorporating only the features that we are interested in studying, we first explore various STM design choices, along with the issue of compiler correctness via the use of automated testing tools. Then we outline a new approach to concurrent compiler correctness using the notion of bisimulation, implemented using the Agda theorem prover. Finally, we show how bisimulation can be used to establish the correctness of a low-level implementation of software transactional memory.

Acknowledgements

Many have had a part to play in this production, and I cannot hope to enumerate them exhaustively. Nevertheless, I would like to begin by thanking everyone at the Functional Programming Laboratory in Nottingham who have made it such an interesting place, academically and socially. Conor McBride deserves a special mention for his multitudes of infectious ideas that started me on this dependently-typed journey, as do Ulf Norell and Nils Anders Danielsson for the years they have put into Agda and its standard library that underpins a large part of this work.

There were plenty of ups and downs in the process. I am eternally grateful to my flatmate Rebecca who had a large part in maintaining my sanity, my muse Ana whose company kept my spirits up through those seemingly endless hours of writing, and my friend Tom for sharing his inexhaustible enthusiasm with me. The numerous thoughtful gifts from Star and Cosmo are also warmly received. I am very fortunate to have my parents, whose support made my aspirations possible. Thank you all.

I appreciate the effort of Andy Gordon and Thorsten Altenkirch in undertaking the rewardless task of my examination. Tsuru Capital—as well as being an excellent place to work—afforded me much time and flexibility with which to complete my corrections. Last but certainly not least, I would like to express my gratitude towards my supervisor Graham Hutton for his guidance and enduring patience through all these years, without whose insights and encouragement this thesis would certainly have found itself in perpetual limbo.

Contents

Chapter 1

Introduction

In this chapter we set the scene for this thesis. We begin with a brief background on the history of concurrent computing, and the concepts and issues involved in developing software that takes advantage of the additional computational capability. We then describe a number of mainstream approaches to constructing concurrent software, along with the transactional memory approach that is the focus of this thesis, illustrated with some simple examples. We conclude with a synopsis of each chapter and a summary of our primary contributions.

1.1 Background

1.1.1 A Brief Note on Moore's Law

Since the invention of the integrated circuit over 50 years ago and the subsequent development of the microprocessor, the number of transistors that engineers can manufacture on a single silicon chip for the same cost has been increasing at an exponential pace, roughly doubling every two years. This growth has been remained consistent, so much so that it has been informally codified as 'Moore's Law' [Moo65]. The related

statement[1] that "microprocessors *performance* roughly doubles every 18 months" has also held true, once we factor in the improved performance of individual transistors.

The popular understanding of Moore's Law tends to be simplified to "computer speed doubles roughly every 18 months." Until half a decade ago, this interpretation sufficed, because in order to pack more transistors next to each other, each one had to be made smaller. This in turn meant faster signal propagation between components, and so faster switching (or clock speeds), increasing performance. The implication of this is that one could expect the same piece of software to run twice as fast on the available hardware, 18 months down the line.

1.1.2 The End of the Free Lunch

Moore's Law had become self-perpetuating as the industry assumed its truth to make projections for their technology roadmaps. By shrinking the size of individual transistors, not only were silicon manufacturers able to increase how many transistors can be economically placed on a single piece of silicon, they were also able to clock their processors at progressively higher frequencies due to reduced switching and signal propagation delays.

Sadly, miniaturisation has some undesirable side-effects: on the sub-micron scales of a modern microprocessor, current leakage due to quantum tunnelling effects across the on-chip insulation is very much detrimental to signal integrity: the smaller the features, the more power the integrated circuit needs to counteract these side-effects. This additional power must be dissipated in the form of waste heat, limiting the extent to which we can simply increase the clock speed. Indeed, some recent desktop processors expend up to a third [LS08] of their power solely to ensure accurate clock signal distribution to outlying areas of the silicon die, and expel as much as 150W of excess heat in an area less than 15mm^2.

[1] Due to David House—an Intel executive at the time—as claimed by Moore: `http://news.cnet.com/2100-1001-984051.html`

Given the restriction that we cannot reasonably clock individual processors at increasingly higher speeds, how could we pack more computing power onto each silicon die? With the extra transistors afforded to us by Moore's Law, the most obvious and easiest solution is to resort to *symmetric multiprocessing* (SMP), by fabricating multiple independent processors on the same die that share access to the same memory.

1.1.3 A Brief Look at Parallel and Concurrent Computing

The concept of SMP had been put to practice as early as the 1960s with the introduction of the Burroughs B5500 mainframe [2]. In the decades that followed, the entire computing industry resorted one by one to some form of parallelism in order to achieve their stated performance. First steps in this dirction included the development of vector processors, where each instruction simultaneously operate on tens or sometimes hundreds of words of data. In today's parlance, we refer to such architectures as *single instruction multiple data* (SIMD).

In contrast, a *multiple instruction multiple data* (MIMD) architecture comprises a number of independent processing units, each concurrently executing its own sequence of instructions. However, programming for multiprocessing systems is a task fraught with pitfalls, as Seymour Cray was alleged to have once quipped: "If you were ploughing a field, which would you rather use: two strong oxen or 1024 chickens?" His remark alludes to the challenge of synchronising a large number of independent processors with each one working on a small part of a larger problem while sharing the same working memory. It is much easier for people to work with a few oxen than to try and herd a large number of chickens.

Such systems with multiple, independent processors are therefore suited to domains involving very large datasets and have intrinsically parallelisable solutions that

[2] `https://wiki.cc.gatech.edu/folklore/index.php/Burroughs_Third-Generation_Computers`

do not require much synchronisation or communication between individual processors. This correlates closely with many scientific computation and simulation problems, and the insatiable demand for computational power in these domains drove the development of massively parallel computers in the decades that followed. Even Cray eventually conceded to the fact that multiprocessing was inevitable, as the costs and resources required to breed and feed the two hypothetical oxen became prohibitive compared to breeding, feeding and herding 1024 chickens.

Meanwhile, as multiprocessor computers grew increasingly larger, it became difficult to maintain fast access to the same shared memory for all processor nodes. Cutting-edge systems therefore moved towards a more *non-uniform memory architecture* (NUMA), where each node had fast access to local memory, but access to globally shared or non-local data took correspondingly longer. The lessons learnt have strongly influenced the design today's high-performance hardware, even in the context of personal computing, as seen with the recent development of *general-purpose graphical processing units* (GPGPUs). On a more distributed and larger scale, Beowulf clusters of networked computers may be seen as a looser interpretation of the NUMA paradigm. Such systems typically form the workhorse behind many of today's large-scale scientific simulations, at least in part due to the fact that they can be and often are built from cheap and readily available commodity hardware.

1.1.4 A Supercomputer on Every Desk

Moving back to the sphere of personal computing, it was not until the last decade that shared memory SMP computers managed to establish a presence. The explanation for this is twofold, yet complementary: on the one hand, the cost of motherboards that can accommodate multiple processor packages were significantly more expensive, and so were only sought after by users with specialist needs. On the other, the inability for predominantly single-threaded software—such as a game or a word-processor—to

take advantage of multiple processors, meant that the vast majority of users had no interest in resorting to SMP: simply waiting another 18 months had been sufficient.

However as raw processor speeds have plateaued in recent years, we can no longer afford to dismiss multiprocessor systems as being too difficult to program. Traditional supercomputing problems—large-scale, loosely-coupled computations such as physical simulations or graphical rendering—have been well-catered for, but they encompass only a small fraction of our day-to-day software usage. We need to figure out how to make concurrency accessible and manageable for everyday software.

1.2 Approaches to Concurrent Software

We resort to concurrency with the hope that the more computational power we have at our disposal, the faster our programs will run. How successfully this is in practice depends on how much concurrency we can exploit in our programs. How we model concurrency has a large influence on how we—as software engineers—think and reason about our concurrent programs, which in turn influences the ease with which we can exploit concurrency. This chapter demonstrates using a few examples why concurrent programming has such a reputation for being difficult, then review some of the basic models of concurrency.

1.2.1 Concurrency versus Parallelism

In general computing literature, the terms 'concurrency' and 'parallelism' are often taken as synonyms and used interchangeably by many, while others make a clear distinction between the two. We will afford a few sentences to clarify what we mean by each, in the context of this thesis.

When we say *parallelism*, we mean the extraction of better *performance* from a program by inferring computations that do not interact with each other then simul-

taneously carrying them out, and the writing of programs in such a way as to make this process easy or possible for the compiler. *Concurrency* on the other hand means the use of interdependent threads of execution as a means of *structuring* the control flow of programs. The focus of this thesis is on explicit concurrency.

1.2.2 Counting: easy as 0, 1, 2...

Consider the problem of incrementing a counter, represented in Haskell using a mutable reference:

```
type Counter_IORef = IORef Integer

makeCounter_IORef :: IO Counter_IORef
makeCounter_IORef = new_IORef 0
```

The $\mathsf{increment}_{\mathsf{IORef}}$ program could then be implemented as follows:

```
increment_IORef :: Counter_IORef → IO ()
increment_IORef counter = do
  n ← read_IORef counter
  write_IORef counter (n + 1)
```

When only a single instance of $\mathsf{increment}_{\mathsf{IORef}}$ is executing, the above code behaves as expected. Suppose however, that two instances of $\mathsf{increment}_{\mathsf{IORef}}$ were executing at the same time. This results in four possible interleavings of the two $\mathsf{read}_{\mathsf{IORef}}$ and $\mathsf{write}_{\mathsf{IORef}}$ operations, not all of which would have the intended effect of incrementing the counter twice. For example, the following interleaving would only increment the counter by one:

Thread A	Thread B	*counter*
$n_A \leftarrow \mathsf{read}_{\mathsf{IORef}}\ counter$		0
	$n_B \leftarrow \mathsf{read}_{\mathsf{IORef}}\ counter$	0

	$\mathsf{write}_{\mathsf{IORef}}\ counter\ (n_B + 1)$	1
$\mathsf{write}_{\mathsf{IORef}}\ counter\ (n_A + 1)$		1

Typically, reading from and writing to mutable variables are relatively fast primitive operations. When they occur in immediate succession, the probability of Thread A being interleaved by Thread B in the above manner is very small, and can easily slip through seemingly thorough empirical testing. Such errors are termed *race conditions*, and can occur whenever there is the possibility of concurrent access to any shared resource.

1.2.3 Shared Memory and Mutual Exclusion

The most widely used approach to prevent the kind of race conditions we have seen in the previous section is to simply prevent concurrent accesses to the shared resource, via a selection of related techniques—locks, semaphores, critical sections, mutexes—all of which are based on the principle of mutual exclusion.

Without discussing implementation details, let us assume the existence of two primitives operations—*lock* and *release*—with the following behaviour: *lock* attempts to acquire exclusive access to a given mutable variable; if the variable is already locked, *lock* waits until it becomes available before proceeding. Its counterpart *release* relinquishes the exclusivity previously obtained.

We can now eliminate the earlier race condition as follows:

```
increment_lock :: Counter_lock → IO ()
increment_lock counter = do
  lock counter
  increment_lock counter
  release counter
```

Even if Thread A were interleaved mid-way, Thread B cannot proceed past the lock primitive until Thread A releases *counter*, ruling out the earlier race condition:

Thread A	Thread B	*counter*
lock *counter*		0
$n_A \leftarrow \mathsf{read}_{\mathsf{IORef}}$ *counter*		0
	lock *counter*	0
$\mathsf{write}_{\mathsf{IORef}}$ *counter* $(n_A + 1)$	⋮	1
release *counter*	blocked on *counter*	1
	⋮	
	$n_B \leftarrow \mathsf{read}_{\mathsf{IORef}}$ *counter*	1
	$\mathsf{write}_{\mathsf{IORef}}$ *counter* $(n_B + 1)$	2
	release *counter*	2

Such two-state locks can easily be generalised to n states with *counting semaphores* where some limited concurrent sharing may take place.

1.2.4 Example: Deadlock

Let us consider a slightly more interesting example: we are required to implement a procedure to increment two given counters in lock-step. A first attempt may be as follows:

```
increment'pair :: Counterlock → Counterlock → IO ()
increment'pair c0 c1 = do
  incrementlock c0
  incrementlock c1
```

However, this does not have the desired effect, because there is an intermediate state between the two calls to $\mathsf{increment}_{\mathsf{lock}}$ when c_0 has been incremented but c_1 has not, yet neither is locked. A better implementation might lock both counters before incrementing:

```
increment_pair :: Counter_lock → Counter_lock → IO ()
increment_pair c0 c1 = do
  lock c0
  lock c1
  increment_lock c0
  increment_lock c1
  release c0
  release c1
```

While this version ensures that the two counters are updated together, it however suffers from a more subtle problem. If two threads A and B both attempt to increment the same pair of counters passed to $\mathsf{increment}_{\mathsf{pair}}$ in a different order, a potential deadlock situation can occur:

A: $\mathsf{increment}_{\mathsf{pair}}\ c_0\ c_1$	B: $\mathsf{increment}_{\mathsf{pair}}\ c_1\ c_0$
$\mathsf{lock}\ c_0$	$\mathsf{lock}\ c_1$
$\mathsf{lock}\ c_1$	$\mathsf{lock}\ c_0$
⋮	⋮
blocked on c_1	blocked on c_0
⋮	⋮

Neither thread can make progress, as they attempt to acquire a lock on the counter which is being held by the other. This could be solved by always acquiring locks in a specific order, but enforcing this is not always straightforward.

Correctness considerations aside, there are the issues of code reuse and scalability to consider. In terms of reuse, ideally we would not want to expose *increment*, as only $\mathsf{increment}_{\mathsf{lock}}$ is safe for concurrent use. On the other hand, to build more complex operations on top of the basic 'increment' operation, chances are that we would need access to the unsafe *increment* implementation. Unfortunately exposing this breaks

the abstraction barrier, with nothing to enforce the safe use of *increment* other than trust and diligence on the part of the programmer.

On the issue of scalability, there is also some tension regarding lock granularity, inherent to mutual-exclusion. Suppose we have a large shared data structure, and our program makes use of as many threads as there are processors. In order to allow concurrent access to independent parts of the data structure, we would need to associate a lock with each constituent part. However acquiring a large number of locks has unacceptable overheads; particularly noticeable when there are only a small number of threads contending for access to the shared data. On the other hand, increasing lock granularity would reduced the number of locks required, and in turn the overheads associated with taking the locks, but this would also rule out some potential for concurrency.

1.2.5 Message Passing and Implicit Synchronisation

The message passing paradigm focuses on the sending of messages between threads in the computation as a primitive, rather than the explicit use of shared memory and mutual exclusion as a medium and protocol for communication. Conceptually, this is a higher-level notion which abstracts the act of sending a message from the how, leaving it to the run-time system to choose the appropriate means. As a result, programs written using this approach have the potential to scale from single processors to distributed networks of multiprocessor computers.

Established languages and frameworks supporting message passing concurrency include Erlang [AVWW96], the Parallel Virtual Machine (PVM) [GBD+94] and the Message Passing Interface (MPI) [GLS99]. In Haskell, we can implement our previous counter example using channels, where Chan α is the polymorphic type of channels carrying messages of type α:

```
data Action = Increment | Get (Chan Integer)
```

```
type Counter_Chan = Chan Action
```

Here we have defined a new datatype Action enumerating the operations the counter supports. A counter is then represented by a channel accepting such Actions, to which we can either send an Increment command, or another channel on which to return the current count via the Get command.

The makeCounter$_{Chan}$ function returns a channel, to which other threads may send Actions to increment or query the counter:

```
makeCounter_Chan :: IO Counter_Chan
makeCounter_Chan = do
  counter ← new_Chan
  value ← new_IORef 0
  fork_IO ∘ forever $ do
    action ← read_Chan counter
    n ← read_IORef value
    case action of
      Increment → write_IORef value (n + 1)
      Get result → write_Chan result n
  return counter
```

Even though we make use of an IORef to store the current count, we have implicitly avoided the mutual exclusion problem by only allowing the forked thread access, essentially serialising access to the mutable variable. Implementing an increment$_{Chan}$ operation is now straightforward:

```
increment_Chan :: Counter_Chan → IO ()
increment_Chan counter = write_Chan counter Increment
```

If concurrent threads invoke increment$_{Chan}$ on the same counter, the atomicity of the write$_{Chan}$ primitive rules out any unwanted interleavings.

Unfortunately, just like the previous mutual exclusion-based solution, it is not trivial to build upon or to reuse $\mathsf{increment}_{\mathsf{Chan}}$—say—to increment two counters in lock-step.

1.2.6 Software Transactional Memory

The popularity of mutual exclusion could be partially attributed to the fact that its implementation is relatively easy to comprehend. On the other hand, managing and composing lock-based code is rather error-prone in practice.

Automatic garbage collection frees the programmer from having to manually manage memory allocation. Laziness in functional programming allows us to write efficient higher-level programs without having to manually schedule the order of computation. In a similar vein [Gro07], software transactional memory (STM) [ST97] allows us to write programs in a compositional style in the presence of concurrency, without requiring us to manually manage undesired interleavings of operations in a shared memory environment.

The idea of using *transactions* to tackle concurrency originated in the context of concurrent databases, which face similar issues of undesirable interleavings of basic operations when different clients attempt to access the same database at the same time. Rather than explicitly locking any requisite resources before proceeding with some sequence of operations on shared data, the client simply informs the database server that the operations are to be treated as a single transaction. From the perspective of other database clients, the server ensures that none of the intermediate states of the transaction is visible, as if the entire transaction took place as a single indivisible operation. Should it fail for whatever reason, the outside perspective would be as if the transaction hadn't taken place at all.

STM implements the same concept, but with shared memory being the 'database' and individual program threads taking the rôle of the 'clients'. STM takes an op-

timistic approach to concurrency: transactions are allowed to overlap in their execution, making the most of the available hardware. Conflicts between transactions arise only when an earlier transaction commits a change to a shared variable which a later transaction depended on. Should this happen, the later one is aborted and retried. In particular, a transaction is only aborted when another one has successfully committed, thereby ensuring overall progress and the absence of deadlocks.

Under the Haskell implementation of STM, transactional computations returning a value of type α have the type STM α. We can give an almost identical implementation of $\mathsf{increment}_{\mathsf{STM}}$ as that of $\mathsf{increment}_{\mathsf{IORef}}$, but uses TVars (*transactional variables*) instead of IORefs, and results in an STM action rather than an arbitrary IO action:

```
type Counter_STM = TVar Integer

increment_STM :: Counter_STM → STM ()
increment_STM counter = do
  n ← read_TVar counter
  write_TVar counter (n + 1)
```

To effect a transaction, we have at our disposal an atomically primitive, which takes an STM α and returns a runnable IO α action. The following program fragment increments the given counter twice.

```
do
  counter ← atomically (new_TVar 0)
  fork_IO (atomically (increment_STM counter))
  fork_IO (atomically (increment_STM counter))
```

In particular, the atomically primitive guarantees that the two instances of $\mathsf{increment}_{\mathsf{STM}}$ do not interleave each other in any way of consequence.

STM makes it straightforward to reuse existing code: simply sequencing two transactions one after another creates a larger composite transaction that increments both

counters atomically when executed:

```
incrementBoth :: Counter_STM → Counter_STM → STM ()
incrementBoth c0 c1 = do
  increment_STM c0
  increment_STM c1
```

This section presented but a brief overview of STM. We will examine and discuss it in more depth in Chapter 2.

1.3 Thesis Overview

The remaining chapters of this thesis comprise of the following:

Chapter 2 provides additional background on the use and implementation of transactional memory, followed by a brief primer on STM Haskell.

Chapter 3 reviews the notions of denotational, big- and small-step operational semantics along with some reasoning techniques, illustrated using a small language. We then present a compiler for this language and its corresponding virtual machine, to show the essence of a compiler correctness proof.

Chapter 4 implements executable semantics for the above language as a Haskell program. We demonstrate the use of QuickCheck in conjunction with the Haskell Program Coverage toolkit for randomised testing of the results established in the previous chapter.

Chapter 5 puts the above empirical approach into practice, on a simplified subset of STM Haskell with a high-level stop-the-world semantics, linked by a compiler to a virtual machine with a log-based implementation of transactions.

Chapter 6 gently introduces the Agda proof assistant, for the purpose of constructing formal machine-checked proofs, and culminates in a verified formalisation of the results of chapter 3.

Chapter 7 extends the notion of compiler correctness to non-deterministic languages using our new notion of combined machines, illustrated using the simple Zap language, for which a complete compiler correctness result is produced and discussed.

Chapter 8 scales our new technique to include explicit concurrency, by introducing a 'fork' primitive and threads. We replay the compiler correctness proof of the previous chapter in this new setting with the help of a few extra concurrency lemmas.

Chapter 9 develops the concept of consistency between transaction logs and the heap, which we use to establish the correctness of a log-based implementation of software transactional memory in the presence of arbitrary interference by an external agent.

Chapter 10 concludes with a summary of this thesis, and a list of various future research directions.

The reader is assumed to be familiar with functional programming and their type systems; knowledge of Haskell would be a bonus. Relevant ideas are introduced when appropriate, with references for further reading.

1.4 Contributions

The contributions of this thesis are as follows:

- Identification of a simplified subset of STM Haskell with a high-level stop-the-world semantics for transactions.

- A virtual machine for this language, in which transactions are implemented following a low-level log-based approach, along with a semantics for this machine.

- A compiler linking the language to the virtual machine with a statement of compiler correctness, empirically tested using QuickCheck and HPC.

- The core idea of a combined machine and semantics, that allows us to establish a direct bisimulation between the high-level language and the low-level virtual machine.

- Putting the above technique into practice using the Agda proof assistant, giving a formal compiler correctness proof for a language with a simple notion of nondeterminism.

- Showing that our technique scales to a language with explicit concurrency, complete with a formal proof.

- A formal correctness proof for a transactional language that shows the equivalence of the log-based approach and the stop-the-world semantics for transactions.

Earlier accounts of some of these have been published in the following papers,

- Liyang HU and Graham Hutton [HH08]. "Towards a Verified Implementation of Software Transactional Memory". In *Proceedings of the Symposium on Trends in Functional Programming*. Nijmegen, The Netherlands, May 2008.

- Liyang HU and Graham Hutton [HH09]. "Compiling Concurrency Correctly: Cutting Out the Middle Man". In *Proceedings of the Symposium on Trends in Functional Programming*. Komárno, Slovakia, June 2009.

but have since been refined and expanded upon in this thesis.

The complete source code to this thesis, in the form of literate Haskell and Agda documents, may be found online at `http://liyang.hu/#thesis` .

Chapter 2

Software Transactional Memory

While mutual exclusion is the dominant paradigm for shared memory concurrent programming, it can be difficult to reason about and is error-prone in practice. Taking a cue from the distributed databases community, software transactional memory applies the concept of transactions to shared memory concurrent programming. In this chapter, we introduce the notion of transactions and transactional memory, along with high-level overviews of how transactional memory can potentially be implemented. We then give a brief history of the development of the approach up to the present day, followed by a primer to the implementation as found in the Glasgow Haskell Compiler.

2.1 Database Transactions

Consider a database server that supports multiple concurrent clients. Each client may need to carry out some complex sequence of operations, and it is up to the database management server to ensure that different clients do not make conflicting changes to the data store. As with any concurrent software, the clients could obtain exclusive access to part of the database by taking a lock, but without careful coordination between clients this can result in deadlock situations, as we have seen in the previous

chapter. This problem is only exacerbated by the possibility of failure on the part of the client software or hardware: a failed client could be holding a lock to critical parts of the database, thus preventing others from making progress.

The alternative is to make use of the concept of *transactions* [Dat95], which provides a higher-level approach to managing concurrency than explicit locking or mutial exclusion. A client starts a transaction by issuing a *begin* command to the server. Thereafter, all subsequent operations until the final *commit* command are considered part of the transaction. The intended behaviour of such transactions is informally captured by the following four properties, abbreviated by the term 'ACID' [Gra81]:

Atomicity The sequence of operations take place as if they were a single indivisible operation, ensuring that transactions follow a simple 'all-or-nothing' semantics: if any of its constituent operations fail, or if the transaction is aborted for whatever reason, the server guarantees that the resulting state of the database is as if none of the operations took place at all.

Consistency The sequence of operations cannot put the data store into a state that is inconsistent with pre-defined invariants of the database. Were this the case, the server would cause the commit operation to fail. Typically, such invariants would be specific to the particular database application, and serve as a safety net to catch client-side bugs.

Isolation As a transaction is running, other clients cannot observe any of its intermediate states. Conversely, until the transaction has completed and been committed to the data store, it cannot influence the behaviour other concurrent clients.

Durability Once the database server has accepted a transaction as being committed, the effects of the operations on the database store should persist, even in the event of system failure.

This approach to concurrency significantly simplifies client implementation. If a transaction fails—say because another client successfully committed a conflicting change—the original client will be notified and may simply retry the transaction at a later point. Atomicity ensures that the partially completed transaction is automatically 'rolled back': the client need not carefully undo the changes it had made so far to avoid affecting subsequent transactions, while isolation ensures that potentially inconsistent intermediate states of the transaction is not visible to others.

Furthermore, this approach is intended to be *optimistic* in the sense that we can always proceed with any given transaction, as there are no locks to acquire, making deadlock a non-issue. The trade-off is that the transaction may later fail or be unable to commit, should a different transaction commit a conflicting change in the meantime. Nevertheless, the system as a whole has made progress.

2.2 Transactional Memory

Transactional memory applies the idea of the previous section to concurrent software, with shared memory playing the rôle of the database store and individual program threads acting as the clients. Clearly there will be some differences: with shared random-access memory being volatile, we may not be able to satisfy the durability aspect of the ACID properties in the event of a power failure, for example. In addition, consistency is more an issue of sequential program correctness and hence largely orthogonal to our main concern of concurrency. The focus of transactional memory is therefore on providing atomicity and isolation in the presence of concurrency.

In this section, we give a brief overview of various developments leading up to the current state of the art in the application of transactional memory to concurrent software.

2.2.1 Hardware Transactional Memory

While it is possible to implement synchronisation between multiple parties using only primitive read and write operations, for example with Lamport's bakery algorithm [Lam74], such software-only techniques do not scale well with the addition of further participants. Rather, most modern architectures feature in one form or another a *compare-and-swap* (CAS) instruction that compares a given value to a word in memory and conditionally swaps in a new value if the comparison yields true. The key property of a CAS instruction is that it does so in an atomic and isolated manner, enforced at the hardware level. This is a powerful primitive on which more sophisticated synchronisation constructs can be readily built.

In a sense, a compare-and-swap primitive already allows us to implement simple word-sized transactions on the hardware level: the CAS instruction can detect interference by comparing the current value of the word with what we had read during the transaction, and if they match, commit the new value to memory. Features resembling transactions are more readily observed on other architectures—such as the DEC Alpha [JHB87]—that provided pairs of *load-linked* and *store-conditional* (LL and SC) primitives. The load-linked instruction—as well as fetching a word from memory—additionally places a watch on the system memory bus for the address in question. The subsequent store-conditional instruction proceeds only when no writes to the address has occurred in the meantime. Of course, the programmer must still check for its success, and either manually retry the operation, or attempt an alternative route.

Herlihy and Moss [HM93] later extended this approach to explicitly support multi-word transactions, building upon existing cache-coherency protocols for multiprocessor architectures. In effect, a *transactional cache* local to the processor buffers any tentative writes, which are only propagated to the rest of the system after a successfully commit. They leverage existing cache-coherency protocols to efficiently guard against potential interference. As the size of the transactional cache is limited by

hardware, this sets an upper bound on the size of the transactions that can occur. To this end, Herlihy and Moss suggest virtualisation as a potential solution, transparently falling back to a software-based handler in much the same way as virtual memory gives the illusion of a practically infinite memory space.

2.2.2 Software Transactional Memory

While some form of hardware support beyond the basic compare-and-swap would be desirable for the implementation of transactional memory, Shavit and Touitou [ST97] propose a software-only version of Herlihy and Moss's approach, which can be efficiently implemented on existing architectures that support CAS or LL/SC instructions. In purely pragmatic terms, it does not require the level of initial investment required by a hardware-assisted solution.

In his thesis, Fraser [Fra03] demonstrated that non-trivial data structures based on his implementation of STM had comparable performance to other lock-based or intricately crafted lock-free designs, running on a selection of existing modern hardware. In this sense, STM could be considered practical for everyday use.

However, even though transactional algorithms can be derived from existing sequential ones by simply replacing memory accesses with calls to the STM library, the impedance mismatch of having to use library calls for mutating variables makes programming in the large somewhat impractical in many of today's mainstream languages. Furthermore, it was not possible to prevent a programmer from directly accessing shared data and circumventing the atomicity and isolation guarantees bestowed by the transaction.

Harris and Fraser [HF03] experimented with transactional extensions to the Java language, along with an STM run-time library. Simply wrapping an existing block of code within an `atomic` construct executes it within a transaction. Upon reaching the end of such a block, the run-time system would attempt to commit its changes,

and should this fail, retries the transaction from the beginning.

Harris and Fraser's Java bytecode translator traps in-language read and writes to the program heap, replacing them with transaction-safe alternatives. However, this does not prevent Java's native methods from surreptitiously modifying the heap or performing irreversible side-effects such as input or output, which would be problematic given that a transaction may execute more than once before it successfully commits. Arbitrary Java code is permitted within `atomic` blocks, but calls to foreign methods within a transaction would raise run-time exceptions as these have the potential to void any atomicity or isolation guarantees of the system. At present, transactional memory has yet to be accepted by the wider Java community, although there is much pioneering work in both commercial [Goe06] and academic contexts [HLM06, KSF10].

Following on from this work, Harris et al. [HMPJH05] presented an implementation of software transactional memory for the Glasgow Haskell Compiler (GHC) as part of their paper entitled *Composable Memory Transactions*. Contributions of this work include the use of Haskell's type system to ensure that only those operations that can be rolled back are used in transactions, along with an operator for composing pairs of alternative transactions. In later work, they introduce 'data invariants' [HPJ06] for enforcing consistency.

More recent work has brought transactional memory to other programming languages [Tan05, Nodir0o0], as well as more efficient [Enn05] or alternative low-level implementations [ATS09].

2.3 Implementing Transactions

The high-level view of transactions is that each one is executed atomically and in isolation from other concurrent threads, as if the entire transaction took place in a

single instant without any of its intermediate states being visible by concurrently running threads. Indeed, an implementation could simply suspend all other threads upon starting a transaction, and resume them after the transaction has ended. Pragmatically, this *stop-the-world* view can be easily achieved by ensuring that only one transaction is ever executing at any point in time, say using a global mutual-exclusion lock. While this approach would be easy to implement, it prevents transactions from proceeding concurrently, and would not be make good use of multi-core hardware.

2.3.1 Log-Based Transactions

A concurrent implementation of transactions might make use of the notion of a *transaction log*. During the execution of a transaction, its associated log serves to isolate it from other concurrently running threads. Rather than immediately acting on any side-effecting operations, the run-time makes a record of these in the log, applying them globally only when the transaction successfully commits.

In the case of shared mutable variables, the log essentially acts as a transaction-local buffer for read and write operations: for each variable, only the first read would come from the shared heap, and only the last value written goes to the shared heap; all intermediate reads and writes operate solely on the log. At the end of the transaction, if any undesired interference has occurred, say due to another transaction having completed in the meantime, the run-time system need only discard the log and re-run the transaction, since no globally-visible changes have been made. Therefore it would be only be appropriate to allow operations that can be buffered in some suitable manner to be recorded in the transaction log, such as changes to shared mutable variables; external side-effects—such as launching missiles [HMPJH05]—cannot be undone, and should be prevented. As long as the transaction log corresponds to a consistent state of the current shared heap on the other hand, the new values recorded in the log can then be applied to the global heap in an atomic manner.

A simple implementation of the commit operation need only ensure that globally, only one commit is taking place at any time, say using a global lock. Even then, we still make gains on multicore systems, as the bodies of transactions still run concurrently. A more sophisticated implementation might allow transactions to commit concurrently, for example by making use of specialised lock-free data structures. While concurrent commits would be trickier to implement, additional complexities are restricted to the run-time system implementer rather than the proletarian programmer

2.3.2 Alternative Approaches to Atomicity

Atomicity is undoubtedly a useful tool for writing programs in a modular, reusable manner in the face of the challenges posed by concurrency. As well as using the notion of a transaction log, there are a number of alternative approaches for creating an illusion of atomicity:

Compensating transactions [Gra81, KLS90] may be thought of as a conflict-detection framework, in which the programmer manually specifies how the whole system can be returned to its initial state, should a transaction need to be retried. A transactions could abort for a variety of reasons, and the roll-back code must be able to deal with any failure mode. The as the roll-back operation is manually defined on a case-by-case basis, there is the additional challenge of ascertaining its correctness. The advantage on the other hand is that we may perform arbitrary I/O, provided we can undo them in an undetectable way.

Lock inference [FFL05, CGE08] on the other hand attempts to automatically insert the fine-grained locks that a programmer might have used, via various code analysis techniques. For reasons of safety, the inference of such locks must necessarily be conservative, and does not always allow for optimal concurrency. Additionally, as code analysis techniques are generally ideally performed on whole programs, we might

lose modularity on the level of object files and/or require the use of sophisticated type-systems [CPN98]. Since roll-back is no longer necessary in this approach, we can allow arbitrary I/O side-effects, but isolation would only be afforded to mutable variables.

2.4 Haskell and Sequential Computation

In this section we will revisit some basic aspects of Haskell required for the understanding of the implementation of STM given by Harris et al. [HMPJH05]. The material should be accessible to the reader with a general understanding of functional programming; no working knowledge of Haskell in particular is required. To aid readability, we will also make use of the following colour scheme for different syntactic classes:

Syntactic Class	Examples
Keywords	**type**, **data**, **let**, **do**...
Types	(), Bool, Integer, IO...
Constructors	False, True, Just, Nothing...
Functions	return, $\mathsf{get}_{\mathsf{Char}}$, $\mathsf{read}_{\mathsf{TVar}}$...
Literals	0, 1, "hello world"...

2.4.1 Monads for Sequential Computation

The Haskell programming language [PJ03a, Mar10]—named after the logician Haskell B. Curry—can be characterised by its three key attributes: functional, pure, and lazy. Functional programming languages are rooted in Church's λ-calculus [Chu36, Bar84], and emphasise the evaluation of mathematical functions rather than the manipulation of state. The core λ-calculus ideas of abstraction and application are typically given prominent status in such languages. In turn, purity means functions depend only on their arguments, eschewing state or mutable variables, akin to the mathematical

notion of functions. The same program expression will always evaluate to the same result regardless of its context, and replacing an expression with its value leaves the meaning of the program unchanged. In other words, the language is *referentially transparent* [Sab98]. The laziness aspect of Haskell means that expressions are only evaluated when their values are required, thus the evaluation order—or even whether something is evaluated at all—is not necessarily immediately apparent from the program text. Together, these properties meant that for some time, it was not clear how to write programs that are more naturally expressed in an imperative, sequential style, or to deal with input and output.

A solution was found in the form of *monads*, which Moggi [Mog89] and Wadler [Wad92] adopted from category theory [ML98]. In the context of computer science, a monad could be viewed as a 'container' for some general notion of computation, together with an operation for combining such computations. As it turns out, sequential computation is just one instance of a monad, as we shall see in the following section.

2.4.2 Modelling Mutable State

Since Haskell is referentially transparent, we cannot directly work with mutable variables, but we can model them. Let us consider the case of a single mutable variable: the basic approach involves passing around the current value of the mutable variable—say, of type σ—as an extra argument. Thus, rather than implementing a function of type $\alpha \to \beta$ which cannot access any mutable state, we instead write one of type $\alpha \to \sigma \to (\sigma, \beta)$. This takes the current value of the mutable variable as an extra argument and returns a new value of the variable, together with the original result of type β.

As we will frequently make use of similar types that mutate some given state, it is convenient to define the following State synonym:

$$\textbf{type}\ \mathsf{State}\ \sigma\ \alpha = \sigma \to (\sigma, \alpha)$$

A value of type State σ α can be regarded as a computation involving some mutable σ state that delivers an α result. Thus we can write the following definitions of read and write, corresponding to our intuition of a mutable variable:

```
read :: State σ σ
read = λs → (s, s)
write :: σ → State σ ()
write s' = λs → (s', ())
```

The read computation results in a value that is the current state, without changing it in any way. On the other hand, the write computation replaces the current state s with the supplied s', giving an information-free result of the singleton () type.

With these two primitives, we can implement an increment operation as follows:

```
increment :: State Integer ()
increment = λs →
  let (s', n) = read s
  in write (n + 1) s'
```

The result of read is bound to the name n, then the state is updated with $n+1$ by the subsequent write. The initial state s we have been given is passed to read—potentially resulting in a different state s'—which is then passed along to write.

In the above instance we know that read does not change the state, but in general any State σ α computation could, therefore we must carefully thread it through each computation in order to maintain the illusion of mutable state. The following definition increments the counter twice:

```
twice :: State Integer ()
twice = λs →
  let (s', _) = increment s
  in increment s'
```

Were we to inadvertently pass the initial s to the second invocation of increment as well, we would have made a copy of the initial state, having discarded the updated s'. The resulting twice would only appear to increment the counter once, despite its two invocations of increment. Explicit state threading is not only rather tedious, small errors can also silently lead to very unexpected results.

Fortunately, the act of threading state around is sufficiently regular that we can implement a general purpose operator that hides away the details of the plumbing:

```
(>>=) :: State σ α → (α → State σ β) → State σ β
ma >>= fmb = λs → let (s', a) = ma s in fmb a s'
```

The $\gg\!=$ operator—pronounced 'bind'—takes on its left a computation delivering an α; its right argument is a function from α to a second computation delivering a β. Bind then passes the result of its left argument—along with the modified state s'—and threads both through its right argument, delivering a stateful computation of type β. Using this operation, we can rewrite increment and twice as follows:

```
increment' :: State Integer ()
increment' =
  read >>= λn →
  write (n + 1)
twice' :: State Integer ()
twice' =
  increment' >>= λ_ →
  increment'
```

In the above definitions, we no longer need to explicitly thread state around as this is handled automatically by the $\gg\!=$ operation, and the resulting code has a much more imperative appearance. In fact, Haskell provides a few helper functions as well

as some lightweight syntactic sugar to support exactly this style of programming, allowing the following succinct definition:

```
increment″ :: State Integer ()
increment″ = do
  n ← read
  write (n + 1)
twice″ :: State Integer ()
twice″ = do
  increment″
  increment″
```

Here, we may think of the expression $n \leftarrow$ read in the definition of increment″ as binding the result of read to the name n, and in fact desugars to the same code as that of increment′. Should we merely want to run a computation for its side-effects, as we do in the definition of twice″, we simply omit both the $\leftarrow$ operator and the resulting name.

To prevent direct manipulation or duplication of the threaded state, we can make State an opaque data type to its users, hiding the above implementations, and offer only read and write as primitives for accessing the state.

2.4.3 Monadic Properties

So far in this section we have deliberately avoided using the mathematical term 'monad'. In fact, some members of the Haskell community have jokingly remarked that they would rather have used the phrase 'warm fuzzy thing' [PJ03b] instead. The $\gg=$ operator above already constitutes the primary component of the definition of the State σ monad, and we need only one further function to complete it.

$$\mathsf{return} :: \alpha \to \mathsf{State}\ \sigma\ \alpha$$

$$\mathsf{return}\ a = \lambda s \to (s, a)$$

Here, the return function produces a trivial computation that results in the value of its given argument. Were they to agree with the mathematical definition of a monad, our definitions of bind and return for the State σ monad must satisfy certain properties, which are as follows:

$$\mathsf{return}\ a \gg\!= fmb \quad \equiv \quad fmb\ a \qquad \text{(ident-left)}$$

$$ma \gg\!= \mathsf{return} \quad \equiv \quad ma \qquad \text{(ident-right)}$$

$$(ma \gg\!= fmb) \gg\!= fmc \quad \equiv \quad ma \gg\!= (\lambda a \to fmb\ a \gg\!= fmc) \qquad \text{(assoc)}$$

The first two specify that return is a left as well as right-identity for $\gg\!=$, while the third says that the $\gg\!=$ operator is associative, modulo the binding operation inherent in the use of $\gg\!=$. Using the power of equational reasoning afforded to us in this pure functional setting, we can show that our definition of $\gg\!=$ and return for the State σ monad satisfies the above laws by simply expanding the relevant definitions. For example, the (ident-left) property can be shown as follows:

$$\begin{aligned}
&\mathsf{return}\ a \gg\!= fmb \\
\equiv\{&\quad \text{definition of } \gg\!= \quad\} \\
&\lambda s \to \mathbf{let}\ (s', a') = \mathsf{return}\ a\ s\ \mathbf{in}\ fmb\ a'\ s' \\
\equiv\{&\quad \text{definition of return} \quad\} \\
&\lambda s \to \mathbf{let}\ (s', a') = (s, a)\ \mathbf{in}\ fmb\ a'\ s' \\
\equiv\{&\quad \text{substitute for } a \text{ and } s \quad\} \\
&\lambda s \to fmb\ a\ s \\
\equiv\{&\quad \eta\text{-contract} \quad\} \\
&fmb\ a
\end{aligned}$$

2.4.4 Input, Output and Control Structures

Now that we have shown how we can sequence operations on mutable state, what about input and output? In a sense, we can conceptually think of I/O as mutating the outside world, and indeed this is the approach used in Haskell. By threading a token representing the state of the real world through a program in a similar way to the State σ monad, we ensure that real-world side-effects occur in a deterministic order. For example, the IO type in Haskell could be defined as follows,

```
type IO α = State RealWorld α
```

where RealWorld is the opaque type of the token, inaccessible to the end-programmer. Assuming two primitves $\mathsf{get}_{\mathsf{Char}}$ :: IO Char and $\mathsf{put}_{\mathsf{Char}}$ :: Char → IO () for interacting with the user, we can implement an echo procedure as follows:

```
echo :: IO ()
echo = do
  c ← getChar
  putChar c
```

In Haskell, monadic actions such as $\mathsf{get}_{\mathsf{Char}}$ or echo are first-order, and when we write a program, we are in fact just composing values—evaluating echo for example does not prompt the user for a character nor print one out, it merely results in a value of type IO () corresponding to the composition of $\mathsf{get}_{\mathsf{Char}}$ and $\mathsf{put}_{\mathsf{Char}}$. The only way to make echo actually perform input and output is to incorporate it into the definition of the system-invoked *main* :: IO () action.

Being able to manipulate monadic actions is a very powerful concept, and allows us to create high-level control structures within the language itself. For example, there's no need for Haskell to have a built-in *for-loop* construct, because we can implement it ourselves:

```
for :: Integer → Integer → (Integer → State σ ()) → State σ ()
for m n body = case m < n of
  False → return ()
  True → do
    body m
    for (m + 1) n body
```

The for function invokes *body* with successive integers arguments, starting at m and stopping before n. While the type of for explicitly mentions the State σ monad, IO is a particular instance of this, so the expression for 0 10 ($\lambda\_ \rightarrow$ echo) corresponds to an IO action that echoes 10 characters entered by the user.

Haskell's typeclasses permits a form of ad-hoc polymorphism, which allows us to give type-specific instances of $\gg=$ and return, so the above definition of for works in any monad we care to define. However a discussion of the topic [WB89] is beyond the scope of—and not required for—this thesis.

2.5 Haskell and Concurrent Computation

While the Haskell language is pure and lazy, occasionally we still need to make use of certain imperative features [PJ01]. By keeping such features within the IO monad—where a token of the external world state is implicitly threaded through each IO action—not only can we then guarantee a particular execution order, we also preserve the purity of the rest of the language.

For example, in those cases where the only known efficient solution to a problem is explicitly imperative, Haskell's standard library provides true mutable variables in the form of the IORef datatype, where IORef α is a reference to values of type α. Its basic interface is given below:

```
newIORef :: α → IO (IORef α)
```

```
read_IORef  :: IORef α → IO α
write_IORef :: IORef α → α → IO ()
```

For multiprocessor programming, Parallel Haskell [THLPJ98] provides the par :: $\alpha \rightarrow \beta \rightarrow \beta$ combinator, which instructs to the run-time system that it may be worth evaluating its first argument in parallel (cf. section 1.2.1), and otherwise acting as the identity function on its second argument. As is evident from its type, the par combinator is pure and cannot perform any side-effects, nor can there be any interaction between its arguments even if they are evaluated in parallel. In fact, it would be perfectly sound for an implementation of par to simply ignore its first argument.

However, explicit concurrency is a necessity as well as a convenience when used as a mechanism for structuring many real-world programs. Concurrent Haskell [PJGF96] introduced the fork_{IO} :: IO () → IO () primitive, which provides a mechanism analogous to the Unix `fork()` system call, sparking a separate thread to run its argument IO () action. Forking is considered impure as threads can interact with each other via a variety of mechanisms, and this fact is correspondingly reflected in the return type of fork_{IO}. With the mutability provided by IORefs, we can create concurrent applications in the same imperative manner as other lower-level programming languages. For example, the following program launches a secondary thread to repeatedly print the letter '`y`', while $\text{main}_{\text{Char}}$ carries on to print '`n`'s:

```
main_Char :: IO ()
main_Char = do
  fork_IO (forever (put_Char 'y'))
  forever (put_Char 'n')
```

The user would observe an unending stream of '`y`'s and '`n`'s, interleaved in an unspecified manner.

To demonstrate concurrency with interaction, the following program launches two

threads, both repeatedly incrementing a shared counter, as well as an individual one. The $\mathsf{main}_{\mathsf{IORef}}$ function meanwhile checks that the shared counter indeed equals the sum of the two thread-local ones.

```
type Counter_IORef = IORef Integer

increment_IORef :: Counter_IORef → IO ()
increment_IORef c = do
  n ← read_IORef c
  write_IORef c (n + 1)

incBoth_IORef :: Counter_IORef → Counter_IORef → IO ()
incBoth_IORef sum local = do
  increment_IORef sum
  increment_IORef local

main_IORef :: IO ()
main_IORef = do
  sum ← new_IORef 0
  a ← new_IORef 0
  b ← new_IORef 0
  fork_IO (forever (incBoth_IORef sum a))
  fork_IO (forever (incBoth_IORef sum b))
  forever (do
    n_sum ← read_IORef sum
    n_a ← read_IORef a
    n_b ← read_IORef b
    when (n_sum ≢ n_a + n_b) (do
      putStrLn "oh dear."))
```

Such a program, while seemingly straightforward in intent, can leave the programmer

with exponential number of possibilities to consider as it scales; it would simply be impractical to apply sequential reasoning to each potential interleaving. Worse still is the fact that the unwanted interleavings are often the least likely to occur, and can easily slip through otherwise thorough empirical testing.

The above program has a number of potentially rare and unexpected behaviours. Firstly, the two forked-off children both increment the *sum* counter, and it is quite possible for one thread's execution of $\mathsf{increment}_{\mathsf{IORef}}$ to interleave the $\mathsf{read}_{\mathsf{IORef}}$ and $\mathsf{write}_{\mathsf{IORef}}$ of the other thread—as we have witnessed in section 1.2.2—losing counts in the process. Requiring our implementation of increment to follow a locking discipline for each counter in question would eliminate this particular race condition. Even with this fix in place, another issue remains: each thread first increments *sum*, followed by its own specific counter; meanwhile, the main thread may interleave either child in-between the two aforementioned steps, and observe a state in which the value of *sum* disagrees with the sum of the values of a and b.

As a concurrent program increases in size, race conditions and deadlock can become much more subtle and difficult to debug. Transactional memory—amongst other high-level approaches—aims to avoid such bugs, while retaining the speed benefits of concurrency.

2.6 Haskell and Software Transactional Memory

The previous section outlined the standard approach to concurrency in Haskell, which makes use of explicit threading and mutable variables via $\mathsf{fork}_{\mathsf{IO}}$ and IORefs within the IO monad. In an analogous fashion, STM Haskell provides mutable *transactional variables* of type TVar α, with the following interface:

$\mathsf{new}_{\mathsf{TVar}} :: \alpha \to \mathsf{STM}\ (\mathsf{TVar}\ \alpha)$

$\mathsf{read}_{\mathsf{TVar}} :: \mathsf{TVar}\ \alpha \to \mathsf{STM}\ \alpha$

$\mathsf{write}_{\mathsf{TVar}} :: \mathsf{TVar}\ \alpha \to \alpha \to \mathsf{STM}\ ()$

The $\mathsf{new}_{\mathsf{TVar}}$ function creates a transactional variable initialised to some given value, while $\mathsf{read}_{\mathsf{TVar}}$ and $\mathsf{write}_{\mathsf{TVar}}$ inspect and mutate TVars. The type $\mathsf{STM}\ \alpha$ may be read as *a transaction which delivers a result of type* α. We may combine the above three primitives to form more elaborate transactions, using the following monadic sequencing operators:

$(\gg\!=) :: \mathsf{STM}\ \alpha \to (\alpha \to \mathsf{STM}\ \beta) \to \mathsf{STM}\ \beta$

$\mathsf{return} :: \alpha \to \mathsf{STM}\ \alpha$

The definition of *bind* for the STM monad composes transactions in a sequential manner, while return takes a given value to a trivial transaction resulting in the same. Transactions are converted to runnable IO actions via the atomically primitive,

$\mathsf{atomically} :: \mathsf{STM}\ \alpha \to \mathsf{IO}\ \alpha$

which when run, performs the while transaction as if it were a single indivisible step. The intention is that when implementing some data structure for example, we need only expose the basic operations as $\mathsf{STM}\ \alpha$ actions, without the need to anticipate all the potential ways in which a user may wish to compose said operations in the future. The end-programmer may compose these primitives together in any desired combination, wrapped in an outer call to atomically. Concurrent transactions are achieved through explicit threading, using $\mathsf{fork}_{\mathsf{IO}}$ as before, while STM run-time takes care of the book-keeping necessary to guarantee that each composite transaction takes place in an atomic and isolated manner.

STM Haskell makes use of the notion of a transaction log (as we mentioned previously in section 2.3.1) and may automatically re-run transactions when conflicts are detected. Therefore it is important that STM actions only make changes to transactional variables—which can be encapsulated within its corresponding log—rather than arbitrary and possibly irrevocable IO actions. This an easy guarantee

because the Haskell type system strictly and statically differentiates between IO α and STM α, and there is no facility for actually performing an IO action while inside the STM monad. Of course, a transaction can always manipulate and return IO actions as first-order values, to be performed post-commit. In any case, as we idiomatically perform the bulk of computations in Haskell using only pure functions, these are necessarily free from side-effects. Thus they do not need to be kept track of by the transaction implementation and may simply be discarded in the event of a conflict. The ability to statically make this three-fold distinction between irrevocable (namely IO) and revocable (or STM) side-effecting computations—used relatively infrequently in practice—alongside pure ones, makes Haskell an ideal environment for an implementation of STM.

Let us now revisit the example of the previous section, with two threads competing to incrementing a shared counter. Using STM, we can make the previous program behave in the intended manner as follows, with only minor changes to its structure:

```
type Counter_TVar = TVar Integer

increment_TVar :: Counter_TVar → STM ()
increment_TVar c = do
  n ← read_TVar c
  write_TVar c (n + 1)

incBoth_TVar :: Counter_TVar → Counter_TVar → STM ()
incBoth_TVar sum local = do
  increment_TVar sum
  increment_TVar local

main_TVar :: IO ()
main_TVar = do
  sum ← atomically (new_TVar 0)
```

```
a ← atomically (newTVar 0)
b ← atomically (newTVar 0)
forkIO (forever (atomically (incBothTVar sum a)))
forkIO (forever (atomically (incBothTVar sum b)))
forever (do
  (n_sum, n_a, n_b) ← atomically (do
    n_sum ← readTVar sum
    n_a ← readTVar a
    n_b ← readTVar b
    return (n_sum, n_a, n_b))
  when (n_sum ≢ n_a + n_b) (do
    putStrLn "oh dear."))
```

That is, the counter is now represented as an integer TVar rather than an IORef. Correspondingly, the $\mathsf{increment}_{\mathsf{TVar}}$ primitive and the $\mathsf{incBoth}_{\mathsf{TVar}}$ function now result in STM rather than IO actions. Finally, $\mathsf{main}_{\mathsf{TVar}}$ atomically samples the three counters inside a single transaction to avoid potential race conditions.

While the sequencing of transactions provides a convenient and composable way to access shared data structures, a concurrency framework ought to also provide efficient ways to perform coordination between threads, say to wait on some collection of resources to become available before proceeding. With mutual exclusion, waiting on a number of objects could be implemented by waiting on each one in turn, taking care to avoid deadlocks. However, there are often cases where we might want to proceed whenever *any* one of some collection of objects becomes ready. For example, Haskell's standard concurrency library offers generalised counting semaphores, which could be used for coordination between multiple threads. Similarly, most flavours of Unix provides a `select(2)` system call, which takes a set of file descriptors and

blocks until at least one is ready to be read from, or written to. Unfortunately, these techniques do not scale: for example in the latter case, all the file descriptors being waited upon must be collated up to a single top-level `select()`, which runs contrary to the idea of modular software development.

STM Haskell answers this problem with a pair of primitives for blocking and composing alternative transactions. The first primitive,

```
retry :: STM α
```

forces the current transaction to fail and retry. This gives a flexible, programmatic way to signal that the transaction is not yet ready to proceed, unlike traditional approaches in which the requisite conditions must be specified upfront using only a restricted subset of the language, such as e.g. Hoare's *conditional critical regions* [Hoa02].

Armed with the retry primitive, we can demonstrate how a $\mathsf{Counter}_{\mathsf{TVar}}$ could be used as a counting semaphore [Dij65]. The $\mathsf{decrement}_{\mathsf{TVar}}$ function below behaves as the `wait` primitive, decrementing the counter only when its value is strictly positive, and blocking otherwise. Correspondingly the earlier $\mathsf{increment}_{\mathsf{TVar}}$ defined above behaves as `signal`, incrementing the count.

```
decrementTVar :: CounterTVar → STM ()
decrementTVar c = do
  n_c ← readTVar c
  unless (n_c > 0)
    retry
  writeTVar c (n_c − 1)
```

The retry statement conceptually discards any side-effects performed so far and restarts the transaction from the beginning. However, the control flow within the transaction is influenced only by the TVars read up until the retry, so if none of these have

been modified by another concurrent thread, the transaction will only end up at the same retry statement, ending up in a busy-waiting situation and wasting processor cycles. The STM run-time can instead suspend the current thread, rescheduling it only when one or more of the TVars read has changed, thus preserving the semantics of retry; the TVars involved in the decision to retry are conveniently recorded within the transaction log.

Suppose we now wish to implement a function to decrement the *sum* variable of the earlier example. In order to maintain the invariant $a + b = sum$, we must also decrement either one of a or b. Knowing that $\mathsf{decrement}_{\mathsf{TVar}}$ blocks when the counter is zero, we may conclude that if $\mathsf{decrement}_{\mathsf{TVar}}\ sum$ succeeds, then a and b cannot both be zero, and we ought to be able to decrement one of the two without blocking. But how do we choose? View $\mathsf{Counter}_{\mathsf{TVar}}$ as a counting semaphore, it is not possible to wait on multiple semaphores unless such a primitive is provided by the system. STM Haskell provides a second primitive,

$$\mathsf{orElse} :: \mathsf{STM}\ \alpha \to \mathsf{STM}\ \alpha \to \mathsf{STM}\ \alpha$$

for composing alternative transactions. With this we may implement the function described above:

$$\begin{aligned}
&\mathsf{decEither}_{\mathsf{TVar}} :: \mathsf{Counter}_{\mathsf{TVar}} \to \mathsf{Counter}_{\mathsf{TVar}} \to \mathsf{Counter}_{\mathsf{TVar}} \to \mathsf{STM}\ () \\
&\mathsf{decEither}_{\mathsf{TVar}}\ sum\ a\ b = \mathbf{do} \\
&\quad \mathsf{decrement}_{\mathsf{TVar}}\ sum \\
&\quad \mathsf{decrement}_{\mathsf{TVar}}\ a\ \text{`}\mathsf{orElse}\text{`}\ \mathsf{decrement}_{\mathsf{TVar}}\ b
\end{aligned}$$

The orElse combinator—written above using infix notation—allows us to choose between alternative transactions: the expression t `orElse` u corresponds to a transaction that runs one of t or u. It is left-biased, in the sense that t is run first: if it retries, any changes due to t is rolled back, and u is attempted instead. Only when both t and u cannot proceed, would the transaction as a whole retry. The final line of the

above fragment would decrement a preferentially over b, and blocking when neither can proceed. (In practice, the latter case can never arise in the above program.) Note that orElse need not be explicitly told which variables the transactions depend on—this is inferred from their respective transaction logs by the run-time system.

Using orElse for composing alternative transactions also allow us to elegantly turn blocking operations into non-blocking ones, for example:

```
decrement'TVar :: CounterTVar → STM Bool
decrement'TVar c = (do decrementTVar c; return True) `orElse` return False
```

This non-blocking $\mathsf{decrement}'_{\mathsf{TVar}}$ operation attempts to decrement the given counter using the original $\mathsf{decrement}_{\mathsf{TVar}}$ and return a boolean True to indicate success. Should that retry or fail to commit, orElse immediately attempts the alternative transaction, which returns False instead.

By design, retry and orElse satisfy the following rather elegant properties:

$$\mathsf{retry}\ `\mathsf{orElse}`\ u \quad \equiv \quad u \qquad \text{(ident-left)}$$

$$t\ `\mathsf{orElse}`\ \mathsf{retry} \quad \equiv \quad t \qquad \text{(ident-right)}$$

$$(t\ `\mathsf{orElse}`\ u)\ `\mathsf{orElse}`\ v \quad \equiv \quad t\ `\mathsf{orElse}`\ (u\ `\mathsf{orElse}`\ v) \qquad \text{(assoc)}$$

In other words, the type STM α of transactions forms a monoid, with orElse as the associative binary operation and retry as the unit.

2.7 Conclusion

In this chapter, we have reviewed the concept of transactions in the context of databases, followed by an overview of the development of transactional memory in both hardware and software, together with how transactions can be used as a high-level concurrency primitive. In section 2.3, we examined a log-based approach to

implementing transactions, contrasted with some alternatives. Section 2.4 introduced the Haskell language, in particular how monads are used to model mutable state in a purely functional context. The penultimate section (§2.5) presented primitives for mutable state and concurrency in Haskell, and we finished with a primer on STM Haskell—in particular a novel form of composing alternative transactions—in order to motivate the study of STM.

Chapter 3

Semantics for Compiler Correctness

In the context of computer science, the primary focus of *semantics* is the study of the meaning of programming languages. Having a mathematically rigorous definition of a language allows us to reason about programs written in the language in a precise manner. In this chapter, we begin by reviewing different ways of giving a formal semantics to a language, and various techniques for proving properties of these semantics. We conclude by presenting a compiler for a simple expression language, exploring what it means for this compiler to be correct, and how this may be proved.

3.1 Semantics

3.1.1 Natural Numbers and Addition

To unambiguously reason about what any given program means, we need to give a mathematically rigorous definition of the language in which it is expressed. To this end, let us consider the elementary language of natural numbers and addition [HW04,

HW06, HW07].

$$\mathsf{Expression} ::= \mathbb{N} \qquad \text{(Exp-}\mathbb{N}\text{)}$$

$$\mid \mathsf{Expression} \oplus \mathsf{Expression} \qquad \text{(Exp-}\oplus\text{)}$$

That is, an Expression is either simply a natural number, or a pair of Expressions, punctuated with the $\oplus$ symbol to represent the operation of addition. We will adhere to a naming convention of $m, n \in \mathbb{N}$ and $a, b, e \in \mathsf{Expression}$.

Although seemingly simplistic, this language has sufficient structure to illustrate two fundamental aspects of computation, namely that of sequencing computations and combining their results. We shall shortly expand on this in section 3.1.5.

3.1.2 Denotational Semantics

Denotational semantics attempts to give an interpretation of a source language in some suitable existing formalism that we already understand. More specifically, the denotation of a program is a representation of what the program means in the vocabulary of the chosen formalism, which could be the language of sets and functions, the λ-calculus, or perhaps one of the many process calculi. Thus, to formally give a denotational semantics for a language is to define a mapping from the source language into some underlying semantic domain. For example, we can give the following semantics for our earlier Expression language, denoted as a natural number:

$$[\![\_]\!] : \mathsf{Expression} \to \mathbb{N}$$

$$[\![m]\!] = m \qquad \text{(denote-val)}$$

$$[\![a \oplus b]\!] = [\![a]\!] + [\![b]\!] \qquad \text{(denote-plus)}$$

Here, a numeric Expression is interpreted as just the number itself. The expression $a \oplus b$ is denoted by the sum of the denotations of its sub-expressions a and b; alternatively, we could say that the denotation of the $\oplus$ operator is the familiar $+$ on natural numbers. This illustrates the essential compositional aspect of denotational semantics, that the meaning of an expression is given in terms of the meaning of its parts. The expression $[\![(1 \oplus 2) \oplus (4 \oplus 8)]\!]$ say, has the denotation 15 by repeatedly applying the above definition:

$$\begin{aligned}
[\![(1 \oplus 2) \oplus (4 \oplus 8)]\!] &= [\![1 \oplus 2]\!] + [\![4 \oplus 8]\!] \\
&= ([\![1]\!] + [\![2]\!]) + ([\![4]\!] + [\![8]\!]) \\
&= (1 + 2) + (4 + 8) \\
&= 15
\end{aligned}$$

3.1.3 Big-Step Operational Semantics

The notion of big-step operational semantics is concerned with the overall result of a computation. Formally, we define a relation $\Downarrow \subseteq \mathsf{Expression} \times \mathbb{N}$ between Expressions and their final values, given below in a natural deduction style:

$$\frac{}{m \Downarrow m} \quad \textsf{(big-val)}$$

$$\frac{a \Downarrow m \quad b \Downarrow n}{a \oplus b \Downarrow m + n} \quad \textsf{(big-plus)}$$

The first (big-val) rule says that a simple numeric Expression evaluates to the number itself. The second (big-plus) rule states that, if a evaluates to m and b evaluates to n, then $a \oplus b$ evaluates to the sum $m + n$. Thus according to this semantics, we can

show that $(1 \oplus 2) \oplus (4 \oplus 8) \Downarrow 15$ by the following derivation:

$$\dfrac{\dfrac{\dfrac{}{1 \Downarrow 1}\,(\textsf{big-val}) \quad \dfrac{}{2 \Downarrow 2}\,(\textsf{big-val})}{1 \oplus 2 \Downarrow 3}\,(\textsf{big-plus}) \quad \dfrac{\dfrac{}{4 \Downarrow 4}\,(\textsf{big-val}) \quad \dfrac{}{8 \Downarrow 8}\,(\textsf{big-val})}{4 \oplus 8 \Downarrow 12}\,(\textsf{big-plus})}{(1 \oplus 2) \oplus (4 \oplus 8) \Downarrow 15}\,(\textsf{big-plus})$$

For this simple language, the big-step operational semantics happens to be essentially the same as the denotational semantics, expressed in a different way. However, one advantage of a relational operational semantics is that the behaviour can be non-deterministic, in the sense that each expression could potentially evaluate to multiple distinct values. In contrast, a denotational semantics deals with non-determinism in the source language by mapping it to a potentially different notion of non-determinism in the underlying formalism. For example, should we require our expression language to be non-deterministic, we would need to switch the semantic domain of the previous semantics to the power set of natural numbers, rather than just the set of natural numbers.

3.1.4 Small-Step Operational Semantics

Small-step semantics on the other hand is concerned with how a computation proceeds as a sequence of steps. Both big-step and small-step semantics are 'operational' in the sense that the meaning of a program is understood through how it operates to arrive at the result. However, in this case each reduction step is made explicit, which is particularly apt when we wish to consider computations that produce side-effects. Again we formally define a relation $\mapsto \subseteq \mathsf{Expression} \times \mathsf{Expression}$, but between pairs

of Expressions, rather than between expressions and their values:

$$\frac{}{m \oplus n \mapsto m + n} \quad \text{(small-plus)}$$

$$\frac{b \mapsto b'}{m \oplus b \mapsto m \oplus b'} \quad \text{(small-right)}$$

$$\frac{a \mapsto a'}{a \oplus b \mapsto a' \oplus b} \quad \text{(small-left)}$$

The first rule (small-plus) deals with the case where the expressions on both sides of $\oplus$ are numerals: in a single step, it reduces to the sum $m + n$. The second (small-right) rule applies when the left argument of $\oplus$ is a numeral, in which case the right argument can make a single reduction, while (small-left) reduces the left argument of $\oplus$ if this is possible. There is no rule corresponding to a lone numeric Expression as no further reductions are possible in this case.

As each $\mapsto$ step corresponds to a primitive computation, it will often be more convenient to refer to it via its reflexive, transitive closure, defined as follows:

$$\frac{}{a \mapsto^\star a} \text{(small-nil)} \qquad \frac{a \mapsto a' \qquad a' \mapsto^\star b}{a \mapsto^\star b} \text{(small-cons)}$$

For example, the full reduction sequence of $(1 \oplus 2) \oplus (4 \oplus 8) \mapsto^\star 15$ would begin by evaluating the $1 \oplus 2$ sub-expression,

$$\dfrac{\dfrac{\text{(small-plus)}\ \dfrac{}{1 \oplus 2 \mapsto 3}}{\text{(small-left)}\ (1 \oplus 2) \oplus (4 \oplus 8) \mapsto 3 \oplus (4 \oplus 8)}}{}$$

followed by $4 \oplus 8$,

$$\dfrac{\dfrac{}{4 \oplus 8 \mapsto 12}\,\textsf{(small-plus)}}{3 \oplus (4 \oplus 8) \mapsto 3 \oplus 12}\,\textsf{(small-right)}$$

before delivering the final result:

$$\dfrac{}{3 \oplus 12 \mapsto 15}\,\textsf{(small-plus)}$$

3.1.5 Modelling Sequential Computation with Monoids

It would be perfectly reasonable to give a right-to-left, or even a non-deterministic interleaved reduction strategy for the small-step semantics of our Expression language. However, we enforce a left-to-right order in order to model the sequential style of computation as found in the definition of the State α monad from §2.4.2.

In the case where the result type of monadic computations form a monoid, such computations themselves can also be viewed as a monoid. Concretely, suppose we are working in some monad M computing values of type $\mathbb{N}$. Using the monoid of sums $(\mathbb{N},\ +,\ 0)$, the following definition of $\circledast$:

$$\begin{aligned} \circledast \quad &: \quad \mathsf{M}\mathbb{N} \to \mathsf{M}\mathbb{N} \to \mathsf{M}\mathbb{N} \\ a \circledast b \quad &= \quad a \gg\!= \lambda m \to \\ &\qquad\quad b \gg\!= \lambda n \to \\ &\qquad\quad \mathit{return}\ (m + n) \end{aligned}$$

gives the monoid $(\mathsf{M}\ \mathbb{N},\ \circledast,\ \mathit{return}\ 0)$. We can easily verify that the identity and associativity laws hold for this monoid via simple equational reasoning proofs, as we had done in §2.4.3. Therefore, we can view monoids as a degenerate model of monads.

The expression languages of this thesis only computes values of natural numbers, so rather than work with monadic computations of type $\mathsf{M}\ \mathbb{N}$, we may work directly with the underlying $(\mathbb{N},\ +,\ 0)$ monoid, since it shares the same monoidal structure. This simplification allows us to avoid the orthogonal issues of variable binding and substitution. By enforcing a left-to-right evaluation order for $\oplus$ in our expression language to mirror that of the $\gg=$ operator, we are able to maintain a sequential order for computations, which is the key aspect of the monads that we are interested in.

3.2 Equivalence Proofs and Techniques

Now that we have provided precise definitions for the semantics of the language, we can proceed to show various properties of the Expression language in a rigorous manner. One obvious questions arises, on the matter of whether the semantics we have given in the previous section—denotational, big-step and small-step—agree in some manner. This section reviews the main techniques involved.

3.2.1 Rule Induction

The main proof tool at our disposal is that of *well-founded induction*, which can be applied to any well-founded structure. For example, we can show that the syntax of the Expression language satisfies the condition of well-foundedness when paired with the following sub-expression ordering:

$$a \sqsubset a \oplus b \qquad b \sqsubset a \oplus b \tag{Exp-$\sqsubset$}$$

The partial order given by the transitive closure of $\sqsubset$ is well-founded, since any $\sqsubset$-descending chain of expressions must eventually end in a numeral at the leaves of the finite expression tree. This particular ordering arises naturally from the induc-

tive definition of Expression: the inductive case (Exp-$\oplus$) allows us to build a larger expression $a \oplus b$ given two existing expressions a and b, while the base case (Exp-$\mathbb{N}$) constructs primitive expressions out of any natural number. In this particular case, to give a proof that some property $P(e)$ holds for all $e \in \mathsf{Expression}$, it suffices by the well-founded induction principle to show instead that:

$$\forall b \in \mathsf{Expression}.\ (\forall a \in \mathsf{Expression}.\ a \sqsubset b \to P(a)) \to P(b)$$

More explicitly, we are provided with the hypothesis that $P(a)$ already holds for all sub-expressions $a \sqsubset b$ when proving $P(b)$; in those cases when b has no sub-expressions, we must show that $P(b)$ holds directly.

The application of well-founded induction to the structure of an inductive definition is called *structural induction*: to prove that a property $P(x)$ holds for all members x of an inductively defined structure X, it suffices to initially show that $P(x)$ holds in all the base cases in the definition of X, and that $P(x)$ holds in the inductive cases assuming that $P(x')$ holds for any immediate substructure x' of x.

Our earlier reduction rules $\Downarrow$ along with $\mapsto$ and its transitive closure $\mapsto^\star$ are similarly inductively defined, and therefore admits the same notion of structural induction. These instances will be referred to as *rule induction*.

3.2.2 Proofs of Semantic Equivalence

We shall illustrate the above technique with some examples.

Theorem 3.1. *Denotational semantics and big-step operational semantics coincide:*

$$\forall e \in \mathit{Expression},\ m \in \mathbb{N}.\quad [\![e]\!] \equiv m \ \leftrightarrow\ e \Downarrow m$$

Proof. We consider each direction of the $\leftrightarrow$ equivalence separately. To show $[\![e]\!] \equiv$

$m \to e \Downarrow m$, we ought to proceed by induction on the definition of the $[\![\_]\!]$ function. As it happens to be structurally recursive on its argument, we may equivalently proceed by structural induction on e, giving us two cases to consider:

Case $e \equiv n$**:** Substituting e, this base case specialises to showing that:

$$[\![n]\!] \equiv m \to n \Downarrow m$$

By (denote-val) in the definition of $[\![\_]\!]$, the hypothesis evaluates to $n \equiv m$. This allows us to substitute m for n in the conclusion, which is trivially satisfied by instantiating (big-val) with m in the definition of $\Downarrow$.

Case $e \equiv a \oplus b$**:** Substituting e as before, we need to show that:

$$[\![a \oplus b]\!] \equiv m \to a \oplus b \Downarrow m$$

Applying (denote-plus) once to the hypothesis, we obtain that $[\![a]\!] + [\![b]\!] \equiv m$. Substituting for m, the conclusion becomes $a \oplus b \Downarrow [\![a]\!] + [\![b]\!]$. Instantiate the induction hypothesis twice with the trivial equalities $[\![a]\!] \equiv [\![a]\!]$ and $[\![b]\!] \equiv [\![b]\!]$ to yield proofs of $a \Downarrow [\![a]\!]$ and $b \Downarrow [\![b]\!]$, which are precisely the two antecedents required by (big-plus) to obtain $a \oplus b \Downarrow [\![a]\!] + [\![b]\!]$.

The second half of the proof requires us to show that $[\![e]\!] \equiv m \leftarrow e \Downarrow m$. We may proceed by rule induction directly on our assumed hypothesis of $e \Downarrow m$, which must match either (big-val) or (big-plus) in the definition of $\Downarrow$:

Rule (big-val)**:** Matching $e \Downarrow m$ with the consequent of (big-val), we may conclude that there exists an n such that $e \equiv n$ and $m \equiv n$. Substituting n for e and m in $[\![e]\!] \equiv m$ and applying (denote-val) once, the desired conclusion becomes $n \equiv n$, which is trivially satisfied by the reflexivity of $\equiv$.

Rule (big-plus)**:** Again by matching $e \Downarrow m$ with the consequent of (big-plus), there exists a, b, n_a and n_b where $e \equiv a \oplus b$ and $m \equiv n_a + n_b$, such that $a \Downarrow n_a$ and $b \Downarrow n_b$. Substituting for e and m, the conclusion becomes $[\![a \oplus b]\!] \equiv n_a + n_b$, which reduces to:

$$[\![a]\!] + [\![b]\!] \equiv n_a + n_b$$

by applying (denote-plus) once. Instantiating the induction hypothesis twice with $a \Downarrow n_a$ and $b \Downarrow n_b$ yields the equalities $[\![a]\!] \equiv n_a$ and $[\![b]\!] \equiv n_b$ respectively, which allows us to rewrite the conclusion as $[\![a]\!] + [\![b]\!] \equiv [\![a]\!] + [\![b]\!]$ by substituting n_a and n_b. The desired result is now trivially true by reflexivity of $\equiv$.

Thus we have shown both directions of the theorem. □

Theorem 3.2. *Big-step and small-step operational semantics coincide. That is,*

$$\forall e \in \mathit{Expression},\ m \in \mathbb{N}.\quad e \Downarrow m \ \leftrightarrow\ e \mapsto^\star m$$

Proof. We shall consider each direction separately as before. To show the forward implication, we proceed by rule induction on the assumed $e \Downarrow m$ hypothesis:

Rule (big-val)**:** There exists an n such that $e \equiv n$ and $m \equiv n$, by matching $e \Downarrow m$ with the consequent of (big-val). Substituting n for both e and m, we can readily conclude that $n \mapsto^\star n$ via (small-nil).

Rule (big-plus)**:** There exists a, b, n_a and n_b where $e \equiv a \oplus b$ and $m = n_a + n_b$, such that $a \Downarrow n_a$ and $b \Downarrow n_b$. After substituting for e and m, the desired conclusion becomes:

$$a \oplus b \mapsto^\star n_a + n_b$$

Instantiating the induction hypothesis with $a \Downarrow n_a$ and $b \Downarrow n_b$ gives us evidence of $a \mapsto^\star n_a$ and $b \mapsto^\star n_b$ respectively. With the former, we can apply $\_ \oplus b$ to

each of the terms and (small-left) to obtain a proof of $a \oplus b \mapsto^\star n_a \oplus b$, while with the latter, we obtain $n_a \oplus b \mapsto^\star n_a \oplus n_b$ by applying $n_a \oplus \_$ and (small-right).

By the transitivity of $\mapsto^\star$, these two small-step reduction sequences combine to give $a \oplus b \mapsto^\star n_a \oplus n_b$, to which we need only append an instance of (small-plus) to arrive at the conclusion.

We proceed by induction over the definition of $\mapsto^\star$ and using an additional lemma that we state and prove afterwards. Given $e \mapsto^\star m$,

Rule (small-nil)**:** If the reduction sequence is empty, then it follows that $e \equiv m$. In this case, we can trivially satisfy the conclusion of $m \Downarrow m$ with (big-val).

Rule (small-cons)**:** For non-empty reduction sequences, there exists an e' such that $e \mapsto e'$ and $e' \mapsto^\star m$. Invoke lemma 3.3 below with $e \mapsto e'$ and $e' \Downarrow m$, where the latter is given by the induction hypothesis, to obtain proof of $e \Downarrow m$.

Pending the proof of lemma 3.3 below, we have thus shown the equivalence of big- and small-step semantics for the Expression language. □

Lemma 3.3. *A single small-step reduction preserves the value of expressions with respect to the big-step semantics:*

$$\forall e, e' \in \textit{Expression},\ m \in \mathbb{N}.\quad e \mapsto e' \ \rightarrow\ e' \Downarrow m \ \rightarrow\ e \Downarrow m$$

Proof. Assume the two premises $e \mapsto e'$ and $e' \Downarrow m$, and proceed by induction on the structure of the first:

Rule (small-plus)**:** There are n_a and n_b such that $e \equiv n_a \oplus n_b$ and $e' \equiv n_a + n_b$. As e' is a numeric expression, the only applicable rule for $e' \Downarrow m$ is (big-val), which implies $m \equiv n_a + n_b$. Thus the desired conclusion of $e \Downarrow m$—after substituting

for e and m—may be satisfied as follows:

$$\dfrac{\dfrac{}{n_a \Downarrow n_a}\ (\textsf{big-val}) \qquad \dfrac{}{n_b \Downarrow n_b}\ (\textsf{big-val})}{n_a \oplus n_b \Downarrow n_a + n_b}\ (\textsf{big-plus})$$

Rule (small-right)**:** There exists n_a, b and b' such that $b \mapsto b'$ with $e \equiv n_a \oplus b$ and $e' \equiv n_a \oplus b'$. Substituting for e', the second assumption becomes $n_a \oplus b' \Downarrow m$, with (big-plus) as the only matching rule. This implies the existence of the premises $n_a \Downarrow n_a$ and $b' \Downarrow n_b$,

$$\dfrac{\dfrac{}{n_a \Downarrow n_a} \qquad \dfrac{\vdots}{b' \Downarrow n_b}}{n_a \oplus b' \Downarrow n_a + n_b}$$

for some n_b such that $m \equiv n_a + n_b$. Invoking the induction hypothesis with $b \mapsto b'$ and the above derivation of $b' \Downarrow n_b$, we obtain a proof of $b \Downarrow n_b$. The conclusion is satisfied by the following derivation:

$$\dfrac{\dfrac{}{n_a \Downarrow n_a} \qquad \dfrac{\vdots}{b \Downarrow n_b}\ (\textsf{IH})}{n_a \oplus b \Downarrow n_a + n_b}$$

Rule (small-left)**:** This case proceeds in a similar manner to the previous rule, but with a, a' and b such that $a \mapsto a'$, where $e \equiv a \oplus b$ and $e' \equiv a' \oplus b$. Substituting for e and e' in the second assumption and inspecting its premises, we observe

that $a' \Downarrow n_a$ and $b \Downarrow n_b$ for some n_a and n_b where $m \equiv n_a + n_b$:

$$\dfrac{\dfrac{\vdots}{a' \Downarrow n_a} \qquad \dfrac{\vdots}{b \Downarrow n_b}}{a' \oplus b \Downarrow n_a + n_b}$$

Instantiating the induction hypothesis with $a \mapsto a'$ and $a' \Downarrow n_a$ delivers evidence of $a \Downarrow n_a$. Reusing the second premise of $b \Downarrow n_b$ verbatim, we can then derive the conclusion of $a \oplus b \Downarrow n_a + n_b$:

$$\dfrac{(\mathsf{IH})\ \dfrac{\vdots}{a \Downarrow n_a} \qquad \dfrac{\vdots}{b \Downarrow n_b}}{a \oplus b \Downarrow n_a + n_b}$$

This completes the proof of $e \mapsto e' \to e' \Downarrow m \to e \Downarrow m$. □

3.3 Compiler Correctness

Now that we have established the equivalence of our three semantics for the expression language, we consider how this language may be compiled for a simple stack-based machine, what it means for such a compiler to be correct, and how this may be proved.

3.3.1 A Stack Machine and Its Semantics

Unlike the previously defined high-level semantics—which operate directly on Expressions themselves—real processors generally execute a *linear* sequences of instructions, each mutating the state of the machine in some primitive way. In order to give such a low-level implementation of our Expression language, we will make use of a stack-based

virtual machine.

Our stack machine has a stack of natural numbers as its sole form of storage, and the state of the Machine at any point may be conveniently represented as the pair of the currently executing Code, along with the current Stack,

$$\begin{aligned} \mathsf{Machine} &::= \langle \mathsf{Code},\ \mathsf{Stack} \rangle \\ \mathsf{Code} &::= [] \mid \mathsf{Instruction} :: \mathsf{Code} \\ \mathsf{Stack} &::= [] \mid \mathbb{N} :: \mathsf{Stack} \end{aligned}$$

where Code comprises a sequence of Instructions, and Stack a sequence of values.

Due to the simple nature of the Expression language, the virtual machine only requires two Instructions, both of which operate on the top of the stack:

$$\mathsf{Instruction} ::= \texttt{PUSH}\ \mathbb{N} \mid \texttt{ADD}$$

The $\texttt{PUSH}\ m$ instruction places the number m on top of the current stack, while ADD replaces the top two values with their sum. Formally, the semantics of the virtual machine is defined by the $\rightarrowtail$ reduction relation, given below:

$$\langle \texttt{PUSH}\ m :: c,\ \sigma \rangle \rightarrowtail \langle c,\ m :: \sigma \rangle \qquad \text{(vm-push)}$$

$$\langle \texttt{ADD} :: c,\ n :: m :: \sigma \rangle \rightarrowtail \langle c,\ m + n :: \sigma \rangle \qquad \text{(vm-add)}$$

As with the previous definition of $\mapsto^\star$, we shall write $\rightarrowtail^\star$ for the transitive, reflexive closure of $\rightarrowtail$:

$$\frac{}{t \rightarrowtail^\star t}\ \text{(vm-nil)} \qquad \frac{a \rightarrowtail a' \qquad a' \rightarrowtail^\star b}{a \rightarrowtail^\star b}\ \text{(vm-cons)}$$

Informally, the difference between the semantics of a virtual machine versus a small-step operational semantics is that the reduction rules for the former is simply a collection of transition rules between pairs of states, and does not make use of any premises.

3.3.2 Compiler

Given an Expression, a compiler in this context produces some Code that when executed according to the semantics of the virtual machine just defined, computes the value of the Expression, leaving the result on top of the current stack. To avoid the need to define concatenation on instruction sequences and the consequent need to prove various distributive properties, the definition of *compile* below accepts an extra *code continuation* argument, to which the code for the expression being compiled is prepended. To compile a top-level expression, we simply pass in the empty sequence `[]`. This both simplifies reasoning and results in more efficient compilers [Hut07]. A numeric expression m is compiled to a `PUSH` m instruction, while $a \oplus b$ involves compiling the sub-expressions a and b in turn, followed by an `ADD` instruction:

$$\begin{aligned}
&\mathit{compile} : \mathsf{Expression} \to \mathsf{Code} \to \mathsf{Code} \\
&\mathit{compile}\ m\ c = \texttt{PUSH}\ m :: c && \text{(compile-val)} \\
&\mathit{compile}\ (a \oplus b)\ c = \mathit{compile}\ a\ (\mathit{compile}\ b\ (\texttt{ADD} :: c)) && \text{(compile-add)}
\end{aligned}$$

For example, $\mathit{compile}\ ((1 \oplus 2) \oplus (4 \oplus 8))\ \texttt{[]}$ produces the code below,

$$\texttt{PUSH}\ 1 :: \texttt{PUSH}\ 2 :: \texttt{ADD} :: \texttt{PUSH}\ 4 :: \texttt{PUSH}\ 8 :: \texttt{ADD} :: \texttt{ADD} :: \texttt{[]}$$

which when executed with an empty initial stack, reduces as follows:

$$
\begin{aligned}
&\langle \texttt{PUSH}\ 1 :: \texttt{PUSH}\ 2 :: \ldots,\ [] \rangle \\
&\rightarrowtail \langle \texttt{PUSH}\ 2 :: \texttt{ADD} :: \ldots,\ 1 :: [] \rangle \\
&\rightarrowtail \langle \texttt{ADD} :: \texttt{PUSH}\ 4 :: \ldots,\ 2 :: 1 :: [] \rangle \\
&\rightarrowtail \langle \texttt{PUSH}\ 4 :: \texttt{PUSH}\ 8 :: \ldots,\ 3 :: [] \rangle \\
&\rightarrowtail \langle \texttt{PUSH}\ 8 :: \texttt{ADD} :: \ldots,\ 4 :: 3 :: [] \rangle \\
&\rightarrowtail \langle \texttt{ADD} :: \texttt{ADD} :: [],\ 8 :: 4 :: 3 :: [] \rangle \\
&\rightarrowtail \langle \texttt{ADD} :: [],\ 12 :: 3 :: [] \rangle \\
&\rightarrowtail \langle [],\ 15 :: [] \rangle
\end{aligned}
$$

3.3.3 Compiler Correctness

By compiler correctness, we mean that given an expression e which evaluates to m according to a high-level semantics, compiling e and executing the resultant code on the corresponding virtual machine must compute the same m. Earlier in the chapter, we had shown the equivalence of our denotational, big-step, and small-step semantics. While we may freely choose any of these as our high-level semantics, we shall adopt the big-step semantics, as it makes our later proofs much shorter.

Using these ideas, the correctness of our compiler can now be formalised by the following equivalence:

$$e \Downarrow m \quad \leftrightarrow \quad \langle \mathit{compile}\ e\ [],\ \sigma \rangle \rightarrowtail^{\star} \langle [],\ m :: \sigma \rangle$$

The $\rightarrow$ direction corresponds to a notion of *completeness*, and states that the machine must be able to compute any m that the big-step semantics permits. Conversely, the $\leftarrow$ direction corresponds to *soundness*, and states that the machine can only produce

values permitted by the big-step semantics. For this proof, we will need to generalise the virtual machine on the right hand side to an arbitrary code continuation and stack.

Theorem 3.4 (Compiler Correctness)**.**

$$e \Downarrow m \quad \leftrightarrow \quad \forall c, \sigma.\ \langle \mathit{compile}\ e\ c,\ \sigma\rangle \rightarrowtail^{\star} \langle c,\ m :: \sigma\rangle$$

Proof. We shall consider each direction of the double implication separately. In the forward direction, we assume $e \Downarrow m$ and proceed on its structure:

Rule (big-val)**:** There exists an n such that $e \equiv n$ and $m \equiv n$. Substituting n for both e and m, the conclusion becomes:

$$\langle \mathit{compile}\ n\ c,\ \sigma\rangle \rightarrowtail^{\star} \langle c,\ n :: \sigma\rangle \quad ,$$
$$\text{or} \quad \langle \texttt{PUSH}\ n :: c,\ \sigma\rangle \rightarrowtail^{\star} \langle c,\ n :: \sigma\rangle$$

by (compile-val) in the definition of *compile*. The conclusion is satisfied by simply applying (vm-cons) to (vm-push) and (vm-nil):

$$\dfrac{\dfrac{}{\langle \texttt{PUSH}\ n :: c,\ \sigma\rangle \rightarrowtail \langle c,\ n :: \sigma\rangle}\ (\text{vm-push}) \qquad \dfrac{}{\langle c,\ n :: \sigma\rangle \rightarrowtail^{\star} \langle c,\ n :: \sigma\rangle}\ (\text{vm-nil})}{\langle \texttt{PUSH}\ n :: c,\ \sigma\rangle \rightarrowtail^{\star} \langle c,\ n :: \sigma\rangle}\ (\text{vm-cons})$$

Rule (big-plus)**:** By matching the assumed $e \Downarrow m$ with the consequent of (big-plus), we see that there exists a, b, n_a and n_b where $e \equiv a \oplus b$ and $m \equiv n_a + n_b$, such that $a \Downarrow n_a$ and $b \Downarrow n_b$. Substituting for e and m, the conclusion becomes

$$\langle \mathit{compile}\ (a \oplus b)\ c,\ \sigma\rangle \rightarrowtail^{\star} \langle c,\ n_a + n_b :: \sigma\rangle \text{ , or}$$
$$\langle \mathit{compile}\ a\ (\mathit{compile}\ b\ (\texttt{ADD} :: c)),\ \sigma\rangle \rightarrowtail^{\star} \langle c,\ n_a + n_b :: \sigma\rangle$$

by expanding *compile*. Instantiating the induction hypothesis with $a \Downarrow n_a$ and $b \Downarrow n_b$ yields proofs of

$$\forall c_a, \sigma_a.\ \langle \mathit{compile}\ a\ c_a,\ \sigma_a \rangle \rightarrowtail^\star \langle c_a,\ n_a :: \sigma_a \rangle \text{ , and}$$
$$\forall c_b, \sigma_b.\ \langle \mathit{compile}\ b\ c_b,\ \sigma_b \rangle \rightarrowtail^\star \langle c_b,\ n_b :: \sigma_b \rangle$$

respectively. Note that crucially, these two hypotheses are universally quantified over c and σ, written with distinct subscripts above to avoid ambiguity. Now substitute $c_b = \texttt{ADD} :: c$, $c_a = \mathit{compile}\ b\ c_b$, $\sigma_a = \sigma$, $\sigma_b = n_a :: \sigma_a$ and we obtain via the transitivity of $\rightarrowtail^\star$:

$$\begin{aligned}\forall c, \sigma.\ &\langle \mathit{compile}\ a\ (\mathit{compile}\ b\ (\texttt{ADD} :: c)),\ \sigma \rangle \\ &\rightarrowtail^\star \langle (\mathit{compile}\ b\ (\texttt{ADD} :: c),\ n_a :: \sigma \rangle \\ &\rightarrowtail^\star \langle \texttt{ADD} :: c,\ n_b :: n_a :: \sigma \rangle\end{aligned}$$

A second application of transitivity to (vm-add) instantiated as follows,

$$\langle \texttt{ADD} :: c,\ n_b :: n_a :: \sigma \rangle \rightarrowtail \langle c,\ n_a + n_b :: \sigma \rangle$$

gives the required conclusion of:

$$\forall c, \sigma.\ \langle \mathit{compile}\ a\ (\mathit{compile}\ b\ (\texttt{ADD} :: c)),\ \sigma \rangle \rightarrowtail \langle c,\ n_a + n_b :: \sigma \rangle$$

For the backward direction, we proceed on the structure of e:

Case $e \equiv n$**:** Substituting e with n, the base case becomes:

$$\forall c, \sigma.\ \langle \mathit{compile}\ n\ c,\ \sigma \rangle \rightarrowtail^\star \langle c,\ m :: \sigma \rangle \rightarrow n \Downarrow m \text{ , or}$$
$$\forall c, \sigma.\ \langle \texttt{PUSH}\ n :: c,\ \sigma \rangle \rightarrowtail^\star \langle c,\ m :: \sigma \rangle \rightarrow n \Downarrow m$$

Assume the hypothesis and set both c and σ to `[]` to obtain $\langle \texttt{PUSH}\ n :: [],\ [] \rangle \rightarrowtail^\star \langle [],\ m :: [] \rangle$, which must be a single reduction corresponding to (vm-push). Therefore m and n must be one and the same, and the conclusion of $n \Downarrow n$ is trivially satisfied by (big-val).

Case $e \equiv a \oplus b$: Substituting e with $a \oplus b$ and expanding the definition of *compile*, we need to show that:

$$\forall c, \sigma.\ \langle \mathit{compile}\ a\ (\mathit{compile}\ b\ (\texttt{ADD} :: c)),\ \sigma \rangle \rightarrowtail^\star \langle c,\ m :: \sigma \rangle \rightarrow a \oplus b \Downarrow m$$

Now, for both a and b, we know that there exists n_a and n_b such that:

$$\forall c_a, \sigma_a.\ \langle \mathit{compile}\ a\ c_a,\ \sigma_a \rangle \rightarrowtail^\star \langle c_a,\ n_a :: \sigma_a \rangle \text{ , and}$$
$$\forall c_b, \sigma_b.\ \langle \mathit{compile}\ b\ c_b,\ \sigma_b \rangle \rightarrowtail^\star \langle c_b,\ n_b :: \sigma_b \rangle$$

Substituting for the subscripted c_a, c_b, σ_a and σ_b as we had done in the (big-plus) case of the first half of this proof, we obtain:

$$\forall c, \sigma.\ \langle \mathit{compile}\ a\ (\mathit{compile}\ b\ (\texttt{ADD} :: c)),\ \sigma \rangle \rightarrowtail^\star \langle c,\ n_a + n_b :: \sigma \rangle$$

Contrast this with the hypothesis:

$$\forall c, \sigma.\ \langle \mathit{compile}\ a\ (\mathit{compile}\ b\ (\texttt{ADD} :: c)),\ \sigma \rangle \rightarrowtail^\star \langle c,\ m :: \sigma \rangle$$

Since the $\rightarrowtail$ reduction relation is deterministic, it must be the case that m and $n_a + n_b$ are the same. Substituting in $n_a + n_b$ for m, the conclusion becomes $a \oplus b \Downarrow n_a + n_b$—an instance of (big-plus)—whose premises of $a \Downarrow n_a$ and $b \Downarrow n_b$ are in turn satisfied by instantiating the induction hypothesis with $\forall c_a, \sigma_a.\ \langle \mathit{compile}\ a\ c_a,\ \sigma_a \rangle \rightarrowtail^\star \langle c_a,\ n_a :: \sigma_a \rangle$ and $\forall c_b, \sigma_b.\ \langle \mathit{compile}\ b\ c_b,\ \sigma_b \rangle \rightarrowtail^\star$

$\langle c_b,\ n_b :: \sigma_b \rangle$.

This completes the proof of the compiler correctness theorem. □

3.4 Conclusion

In this chapter, we have shown by example what it means to give the semantics of simple language in denotational, big-step and small-step styles. We justified the use of a monoidal model of natural numbers and addition—with left-to-right evaluation order—as simplification of monadic sequencing. We then proved the three given semantics to be equivalent, and demonstrate the use of well-founded induction on the structure of the reduction rules (that is, rule induction) and of the syntax. Finally, we defined a stack-based virtual machine and a compiler for running programs of the Expression language, and presented a proof of compiler correctness.

Chapter 4

Randomised Testing in Haskell

During the initial development of a language, it is useful to be able to check that its behaviour agrees with our intuitive expectations. Formal proofs require a significant investment of effort, and are not always straightforward to revise in light of any underlying changes. Using Haskell, we can implement our semantics as an executable program: its high-level expressiveness leads to a much narrower semantic gap between the mathematical definitions and the implementation, giving us much greater confidence in the fidelity of the latter, in contrast to more traditional programming languages.

In turn, if we state the expected properties as Boolean functions in Haskell, these can then be subjected to randomly generated inputs using the QuickCheck tool [CH00], which displays any counterexamples found. While successful runs of hundreds and thousands of such tests do not comprise a formal proof of the corresponding property, they do however corroborate the existence of one. Additionally, the use of the Haskell Program Coverage (HPC) toolkit offers confidence in the validity of these tests by highlighting any unexercised fragments of the implementation.

4.1 Executable Semantics

Let us begin by implementing the syntax of the expression language of the previous chapter:

```
data Expression = Val Integer | Expression ⊕ Expression
```

The Expression algebraic data type defined above has two constructors Val and $(\oplus)$, corresponding to numerals and addition.

4.1.1 Denotational Semantics

The denotational semantics for our language given in the previous chapter map expressions to its underlying domain of numbers:

$$[\![\_]\!] : \text{Expression} \to \mathbb{N}$$

$$[\![m]\!] = m \qquad \text{(denote-val)}$$

$$[\![a \oplus b]\!] = [\![a]\!] + [\![b]\!] \qquad \text{(denote-plus)}$$

This can be directly implemented as the following denot function, mapping any given Expression to an Integer:

```
denot :: Expression → Integer
denot (Val m) = m
denot (a ⊕ b) = denot a + denot b
```

4.1.2 Big-Step Operational Semantics

While denotations can be implemented directly as functions, there is in general no corresponding notion in Haskell for the transition relations given for a definition of

operational semantics, since transitions need not necessarily be deterministic. Nevertheless, we can accommodate non-deterministic transitions by implementing them as set-valued functions, that return a set of possible reducts for each expression.

Let us first define two type synonyms REL and Rel corresponding to heterogeneous and homogeneous relations respectively:

```
type REL α β = α → Set β
type Rel α = REL α α
```

That is, a heterogeneous relation between α and β can be realised as a function from α to a set of possible βs.

For convenience, we also define some auxiliary functions not found in the Haskell standard library. Firstly $\mathsf{join}_{\mathsf{Set}}$ flattens nested sets,

```
joinSet :: Set (Set α) → Set α
joinSet = Set.fold (∪) {}
```

by folding the binary set union operation ($\cup$) over the outer set. Note that we have typeset Set.union and Set.empty as their usual mathematical notations.

The productWith function computes a generalised Cartesian product of two sets, combining the elements from each set with the function f:

```
productWith :: (α → β → γ) → Set α → Set β → Set γ
productWith f as bs = joinSet ((λa → f a `Set.map` bs) `Set.map` as)
```

Let us remind ourselves of the big-step $\Downarrow$ relation given in the previous chapter:

$$\frac{}{m \Downarrow m} \quad \text{(big-val)}$$

$$\frac{a \Downarrow m \quad b \Downarrow n}{a \oplus b \Downarrow m + n} \quad \text{(big-plus)}$$

Using our previously defined helpers, we can now implement the $\Downarrow$ relation as the following bigStep function, where Set.singleton m is rendered as $\{m\}$:

```
bigStep :: REL Expression Integer
bigStep (Val m) = {m}
bigStep (a ⊕ b) = productWith (+) (bigStep a) (bigStep b)
```

The first case corresponds to the (big-val) rule, with Val m reducing to m. The second case of $a \oplus b$ recursively computes the possible reducts for a and b, then combines the two sets using productWith $(+)$ to obtain the possible values corresponding to $m + n$.

4.1.3 Small-Step Operational Semantics

In the previous chapter, we had defined the small-step reduction relation $\mapsto$ as follows:

$$\frac{}{m \oplus n \mapsto m + n} \quad \text{(small-plus)}$$

$$\frac{b \mapsto b'}{m \oplus b \mapsto m \oplus b'} \quad \text{(small-right)}$$

$$\frac{a \mapsto a'}{a \oplus b \mapsto a' \oplus b} \quad \text{(small-left)}$$

The above rules are implemented as the three cases of the smallStep function below, corresponding to (small-plus), (small-right) and (small-left) in that order:

```
smallStep :: Rel Expression
smallStep (Val m ⊕ Val n) = {Val (m + n)}
smallStep (Val m ⊕ b    ) = (λb' → Val m ⊕ b') `Set.map` smallStep b
smallStep (a     ⊕ b    ) = (λa' → a'     ⊕ b ) `Set.map` smallStep a
```

The three patterns for smallStep above are not exhaustive, as we are missing a case for Val m. Since such expressions cannot reduce any further, we return an empty set to indicate the lack of such transitions:

```
smallStep (Val m)          = {}
```

In a small-step semantics, the eventual result of a computation is given by repeated application of the small-step reduction relation to the initial expression. The following reduceUntil—parametrised on the reduction relation—performs this task:

```
reduceUntil :: (α → Maybe β) → Rel α → REL α β
reduceUntil p reduce init = step ({init}, {}) where
  step :: (Set α, Set β) → Set β
  step (running, finished) = case Set.null running of
    True  → finished
    False → step (first (join_Set ∘ Set.map reduce)
                        (Set.fold partition ({}, finished) running))
  partition :: α → (Set α, Set β) → (Set α, Set β)
  partition e = case p e of
    Nothing → first (Set.insert e)
    Just n  → second (Set.insert n)
```

The step helper takes a pair of *running* and *finished* sets of states, accumulating those that satisfy p into the finished set for the next iteration with the aid of partition, and repeatedly applies *reduce* to the set of remaining running states until it becomes exhausted.

Together with the auxiliary isVal function,

```
isVal :: Expression → Maybe Integer
isVal (Val n) = Just n
isVal _       = Nothing
```

we obtain an executable implementation of $\mapsto^\star$ as the following Haskell function:

```
smallStepStar :: REL Expression Integer
smallStepStar = reduceUntil isVal smallStep
```

4.1.4 Virtual Machine

The virtual machine presented previously is implemented in a similarly straightforward manner. We begin with a trio of type synonyms defining what comprises a virtual machine:

```
type Machine = (Code, Stack)
type Code = [Instruction]
type Stack = [Integer]
```

We make use of Haskell's built-in list type for Code and Stack rather than giving our own definitions. Instructions are defined as the following algebraic data type, with one constructor for each corresponding instruction:

```
data Instruction = PUSH Integer | ADD
```

Intuitively, a PUSH m instruction places the number m onto the top of the stack, while the ADD instruction replaces the top two numbers on the stack with their sum:

$$\langle \texttt{PUSH}\ m :: c,\ \sigma \rangle \rightarrowtail \langle c,\ m :: \sigma \rangle \qquad \text{(vm-push)}$$

$$\langle \texttt{ADD} :: c,\ n :: m :: \sigma \rangle \rightarrowtail \langle c,\ m + n :: \sigma \rangle \qquad \text{(vm-add)}$$

The (vm-push) and (vm-add) rules are directly implemented as the first two rules of the following stepVM function:

```
stepVM :: Rel Machine
stepVM (PUSH m : c,        σ) = {(c, m : σ)}
```

```
stepVM (ADD     : c, n : m : σ) = {(c, m + n : σ)}
stepVM (c       ,        σ)     = {}
```

Since no other transitions are possible for the virtual machine, we return the empty set in the final catch-all case.

The virtual machine is considered halted when its sequence of instructions is exhausted, and the stack consists of only a single number corresponding to the result of the computation:

```
isHalted :: Machine → Maybe Integer
isHalted ([], n : []) = Just n
isHalted _            = Nothing
```

In the same way as small-step semantics, we may make use of our earlier reduceUntil function to repeatedly iterate stepVM until the virtual machine halts:

```
stepVMStar :: REL Machine Integer
stepVMStar = reduceUntil isHalted stepVM
```

Finally, the compiler is more-or-less a direct transliteration of our previous definition:

```
compile :: Expression → Code → Code
compile (Val n) c = PUSH n : c
compile (a ⊕ b) c = compile a (compile b (ADD : c))
```

4.2 Randomised Testing with QuickCheck and HPC

QuickCheck is a system for testing properties of Haskell programs with randomly-generated inputs. Informally, properties are specified as Haskell functions that return

a boolean result. For example, to assert that $(2 \times n) \div 2 \equiv n$ holds in virtually all cases for floating-point numbers, we may interactively invoke the following call to quickCheck:

```
*Main> quickCheck (λn → (2 × n) ÷ 2 ≡ n)
+++ OK, passed 100 tests.
```

It is important to highlight the fact that unlike certain model-checking tools, QuickCheck does not attempt to exhaustively generate all possible inputs. Thus even given many successful repeated invocations of quickCheck, some rare corner cases may remain unprobed. For example, even discounting the possibility of *NaN* and *infinity*, the following expression evaluates to False for 32-bit IEEE-754 floating-point numbers, due to its limited range and finite precision:

```
*Main> (λn → (2 × n) ÷ 2 ≡ n) (3e38 :: Float)
False
```

QuickCheck does not obviate the need for formal proofs. However, it is nevertheless very helpful while the implementation is still in a state of flux, allowing us to detect many flaws ahead of committing to more rigorous and laborious analyses.

4.2.1 Generating Arbitrary Expressions

While QuickCheck comes with existing generators for many built-in Haskell data types, custom generators can also be seamlessly added for new data types defined in our own programs. This is achieved using Haskell's *type class* mechanism—a form of ad-hoc polymorphism based on type-directed dispatch that in essence allows us to overload function names for more than one type.

The following code defines the generator for an arbitrary expression, by instantiating the Arbitrary type class for Expressions:

```
instance Arbitrary Expression where
  arbitrary :: Gen Expression
  arbitrary = oneof
    [Val ⟨$⟩ arbitrary
    ,(⊕) ⟨$⟩ arbitrary ⟨*⟩ arbitrary]
```

The oneof :: [(Gen α)] $\rightarrow$ Gen α combinator above picks one of the listed generators with equal probability. In the first case, Val is applied to an Integer generated by invoking a different instance of arbitrary, while the latter recursively generate the two sub-expression arguments to (⊕). We can test this with the following incantation:

```
*Main> sample (arbitrary :: Gen Expression)
Val 5 ⊕ (Val 8 ⊕ Val 4)
Val 1
...
```

However, we soon find that this generates some unacceptably large expressions. Writing $\mathbb{E}[|e(n)|]$ for the *expected size* of the generated expression, we see why this is the case:

$$\mathbb{E}[|e(n)|] = \frac{1}{2} + \frac{1}{2}(\mathbb{E}[|e(n)|] + \mathbb{E}[|e(n)|]) = \infty$$

To bring the size of the generated expressions under control, we can use the sized combinator as follows, to allow QuickCheck some influence over the size of the resulting Expression:

```
instance Arbitrary Expression where
  arbitrary :: Gen Expression
  arbitrary = sized (λn → frequency
    [(n + 1, Val ⟨$⟩ arbitrary)
    ,(n,     (⊕) ⟨$⟩ arbitrary ⟨*⟩ arbitrary)])
```

The frequency combinator behaves analogously to oneof, but chooses each alternative with a probability proportional to the accompanying weight. That is, the above definition of arbitrary produces a Val constructor with probability $(n+1)/(2n+1)$, and an $(\oplus)$ constructor with probability $n/(2n+1)$. Applying the same analysis as above, we expect the generated expressions to be much more manageable:

$$\mathbb{E}[|e(n)|] = \frac{(n+1) + 2n\mathbb{E}[|e(n)|]}{2n+1} = n+1$$

When QuickCheck finds a counterexample to the proposition in question, we often find that it is rarely the smallest such case, which makes it difficult to understand exactly where and how the problem arises. QuickCheck provides a mechanism to automate the search for smaller counterexamples, via the shrink method of the Arbitrary type class:

```
shrink :: Expression → [Expression]
shrink e = case e of
  Val m → Val ⟨$⟩ shrinkIntegral m
  a ⊕ b → [a, b]
```

QuickCheck expects our definition of shrink to return a list of similar values that are 'smaller' in some sense. This is implemented in a straightforward manner for Expressions by simply returning a list of direct sub-expressions for the $(\oplus)$ constructor. For values, we use the built-in shrinkIntegral to obtain a list of 'smaller' candidates.

4.2.2 Semantic Equivalence and Compiler Correctness

While we have already given a formal proof of the equivalence between denotational, big-step and small-step semantics in the previous chapter, we shall illustrate how we could have informally asserted our theorems using QuickCheck with a much smaller investment of effort.

Let us recall theorem 3.1, which states that our denotational and big-step semantics are equivalent:

$$\forall e \in \mathsf{Expression},\ m \in \mathbb{N}.\quad [\![e]\!] \equiv m \leftrightarrow e \Downarrow m$$

That is, for all expressions whose denotation is m, the same expression also evaluates under our big-step semantics to m. Written literally, this corresponds to the following Haskell predicate,

```
prop_DenotBig' :: Expression → Integer → Bool
prop_DenotBig' e m = (denot e ≡ m) ≡ (bigStep e ∋ m)
```

where ∋ is the flipped Set membership operator. This can be checked as follows:

```
*Main> quickCheck prop_DenotBig'
+++ OK, passed 100 tests.
```

There are some subtleties involved: in the above test, QuickCheck generates a random Integer as well as an Expression, and checks that the implication holds in both directions. However, given the unlikelihood of some unrelated m coinciding with the value of e according to either semantics, both sides of the outer ≡ will be False a majority of the time. This is clearly not a fruitful exploration of the test space.

We can write the same test much more efficiently by using only a single random Expression, by rephrasing the property as below:

```
prop_DenotBig :: Expression → Bool
prop_DenotBig e = {denot e} ≡ bigStep e
```

That is to say, denot e is the unique value that e can evaluate to under the big-step semantics. When fed to QuickCheck, the response is as we would expect:

```
*Main> quickCheck prop_DenotBig
+++ OK, passed 100 tests.
```

Theorem 3.2—that is, the correspondence between big-step and small-step semantics—can be approached in the same way:

$$\forall e \in \mathsf{Expression},\ m \in \mathbb{N}.\quad e \Downarrow m \leftrightarrow e \mapsto^{\star} m$$

Transliterated as a QuickCheck property, this corresponds to the following definition:

```
prop_BigSmall' :: Expression → Integer → Bool
prop_BigSmall' e m = bigStep e ∋ m ≡ smallStepStar e ∋ m
```

However, the above is just an instance of what it means for two sets to be equal. Thus we may elide the m argument and just use Set's built-in notion of equality:

```
prop_BigSmall :: Expression → Bool
prop_BigSmall e = bigStep e ≡ smallStepStar e
```

Again, QuickCheck does not find any counterexamples:

```
*Main> quickCheck prop_BigSmall
+++ OK, passed 100 tests.
```

Finally, we revisit our previous statement of compiler correctness:

$$\forall e \in \mathsf{Expression},\ m \in \mathbb{N}.\quad e \Downarrow m \quad \leftrightarrow \quad \langle \mathit{compile}\ e\ [],\ [] \rangle \rightarrowtail^{\star} \langle [],\ m :: [] \rangle$$

Taking into account what we discussed in the previous paragraphs, the above can be defined as the following property:

```
prop_Correctness :: Expression → Bool
prop_Correctness e = smallStepStar e ≡ stepVMStar (compile e [], [])
```

Feeding this to QuickCheck yields no surprises:

```
*Main> quickCheck prop_Correctness
+++ OK, passed 100 tests.
```

4.2.3 Coverage Checking with HPC

With any testing process, it is important to ensure that all relevant parts of the program has been exercised during testing. The Haskell Program Coverage (HPC) toolkit [GR07] supports just this kind of analysis. It instruments every (sub-)expression in the given Haskell source file to record if it has been evaluated, and accumulates this information to a `.tix` file each time the program is executed. Analysis of the resulting `.tix` file enables us to quickly identify unevaluated fragments of code.

Using HPC is a straightforward process, as it is included with all recent releases of the Glasgow Haskell Compiler. We do however need to turn our implementation so far into a fully-fledged program, namely by implementing a main function:

```
main :: IO ()
main = do
  quickCheck prop_DenotBig
  quickCheck prop_BigSmall
  quickCheck prop_Correctness
```

This *literate Haskell* source file is then compiled with HPC instrumentation—then executed—as follows:

```
$ ghc -fhpc testing.lhs --make
[1 of 1] Compiling Main  ( testing.lhs, testing.o )
Linking testing ...
$ ./testing
+++ OK, passed 100 tests.
```

```
+++ OK, passed 100 tests.
+++ OK, passed 100 tests.
```

Running the `hpc report` command on the resulting `testing.tix` file displays the following report,

```
  94% expressions used (181/192)
  82% alternatives used (19/23)
  50% local declarations used (2/4)
  89% top-level declarations used (26/29)
```

which is not quite the 100% coverage we might have expected. To see where, we may use the `hpc draft` command to produce a set of missing 'ticks', corresponding to unevaluated expressions in our code. (Alternatively, we could have visualised the results using the `hpc markup` command, which generates a highlighted copy of the source code for our perusal.) The salient[1] ones are listed below:

```
module "Main" {
  inside "smallStep"  { tick "Set.empty" on line 201; }
  inside "stepVM"     { tick "Set.empty" on line 295; }
  tick function "shrink" on line 433;
}
```

The first two missing ticks refer to the use of Set.empty—otherwise typeset as {}—in the cases where smallStep and stepVM are invoked on expressions or virtual machine states that cannot reduce any further, which would imply the existence of a 'stuck' state. The third tick refers to the shrink function, which QuickCheck only invokes when it finds a counterexample for any of the properties under test. Thus incomplete coverage in these cases is not unexpected.

[1] The omitted ticks correspond to compiler-generated helper definitions.

4.3 Conclusion

The combination of QuickCheck for randomised testing and HPC to confirm complete code coverage was first pioneered in-the-large by the XMonad project [SJ07], providing a high level of assurance of implementation correctness with minimal effort. Leveraging these same tools and techniques as a means of verifying semantic properties, we have conferred similar levels of confidence that our implementation satisfies the compiler correctness property. Given the small semantic gap between our implementation and the mathematical definitions of the previous chapter, it would be reasonable to interpret this as evidence towards the correctness of theorem 3.4 for our expression language.

Chapter 5

A Model of STM

In this chapter, we identify a simplified subset of STM Haskell that is suitable for exploring design and implementation issues, and define a high-level stop-the-world semantics for this language. We then define a low-level virtual machine for this language in which transactions are made explicit, along with a semantics for this machine. Finally, we relate the two semantics using a compiler correctness theorem, and test the validity of this theorem using QuickCheck and HPC on an executable implementation this language.

5.1 A Simple Transactional Language

In chapter §2, we introduced STM Haskell, which provides a small set of primitives for working with transactional variables,

$$\begin{aligned}
\mathsf{new}_{\mathsf{TVar}} &:: \alpha \to \mathsf{STM}\ (\mathsf{TVar}\ \alpha) \\
\mathsf{read}_{\mathsf{TVar}} &:: \mathsf{TVar}\ \alpha \to \mathsf{STM}\ \alpha \\
\mathsf{write}_{\mathsf{TVar}} &:: \mathsf{TVar}\ \alpha \to \alpha \to \mathsf{STM}\ ()
\end{aligned}$$

along with a primitive for running transactions, and another for explicit concurrency:

$$\mathsf{atomically} :: \mathsf{STM}\ \alpha \to \mathsf{IO}\ \alpha$$
$$\mathsf{fork}_{\mathsf{IO}} :: \mathsf{IO}\ () \to \mathsf{IO}\ ()$$

The STM and IO types are both monads (§2.4), so we may use $\gg\!=$ and return on both levels for sequencing effectful computations.

5.1.1 Syntax

As a first step towards a verified implementation of STM, let us consider a simplified model of the above, to allow us to focus on the key issues. The language we consider has a two-level syntax—mirroring that of the STM Haskell primitives—which can be represented as the following Tran and Proc data types in Haskell:

$$\mathbf{data}\ \mathsf{Tran} = \mathsf{Val}_{\mathsf{T}}\ \mathsf{Integer} \mid \mathsf{Tran} \oplus_{\mathsf{T}} \mathsf{Tran} \mid \mathsf{Read}\ \mathsf{Var} \mid \mathsf{Write}\ \mathsf{Var}\ \mathsf{Tran}$$
$$\mathbf{data}\ \mathsf{Proc} = \mathsf{Val}_{\mathsf{P}}\ \mathsf{Integer} \mid \mathsf{Proc} \oplus_{\mathsf{P}} \mathsf{Proc} \mid \mathsf{Atomic}\ \mathsf{Tran} \mid \mathsf{Fork}\ \mathsf{Proc}$$

The two types correspond to actions in the STM and IO monads respectively. The language is intentionally minimal, because issues such as name binding and performing general-purpose computations are largely orthogonal to our goal of verifying an implementation of STM. Thus, we replace the $\gg\!=$ and return of both monads with the monoid of addition and integers, as motivated in section §3.1.5 of the previous chapter. In our language, $\mathsf{Val}_{\mathsf{T}}$ and $\mathsf{Val}_{\mathsf{P}}$ correspond to return for the STM and IO monads, while $\oplus_{\mathsf{T}}$ and $\oplus_{\mathsf{P}}$ combine Tran and Proc computations in an analogous way to *bind*. By enforcing left-to-right reduction semantics for $\oplus_{\mathsf{T}}$ and $\oplus_{\mathsf{P}}$, we nevertheless retain the fundamental idea of using monads to sequence computations and combine their results.

The remaining constructs emulate the STM and IO primitives provided by STM Haskell: Read and Write correspond to $\mathsf{read}_{\mathsf{TVar}}$ and $\mathsf{write}_{\mathsf{TVar}}$, where Var represents a finite collection of transactional variables. Due to the lack of name binding, we omit an analogue of $\mathsf{new}_{\mathsf{TVar}}$ from our language, and assume all variables are initialised to

zero. Atomic runs a transaction to completion, delivering a value, while Fork spawns off its argument as a concurrent process in the style of $\mathsf{fork}_{\mathsf{IO}}$.

For simplicity, we do not consider orElse or retry, as they are not required to illustrate the basic implementation of a log-based transactional memory system.

Example

Let us revisit the transactional counter example from section 2.6:

```
type Counter_TVar = TVar Integer

increment_TVar :: Counter_TVar → STM ()
increment_TVar c = do
  n ← read_TVar c
  write_TVar c (n + 1)
```

In our STM model, the corresponding increment function would be written as follows:

```
increment :: Var → Tran
increment c = Write c (Read c ⊕_T Val_T 1)
```

To increment the same counter twice using concurrent threads, we would write:

```
incTwice :: Var → Proc
incTwice c = Fork (Atomic (increment c)) ⊕_P Fork (Atomic (increment c))
```

5.1.2 Transaction Semantics

We specify the meaning of transactions in this language using a small-step operational semantics, following the approach of [HMPJH05]. Formally, we give a reduction relation $\mapsto_{\mathsf{T}}$ on pairs $\langle h,\ e\rangle$ consisting of a heap $h :: \mathit{Heap}$ and a transaction $e ::$ Tran. In this section we explain each of the rules defining $\mapsto_{\mathsf{T}}$, and simultaneously describe its implementation in order to highlight their similarity.

First, we model the heap as a map from variable names to their values—initialised to zero—and write $h\,?\,v$ to denote the value of variable v in the heap h. This may be implemented in Haskell using the standard Map datatype:

```
type Heap = Map Var Integer

(?) :: Heap → Var → Integer
h ? v = Map.findWithDefault 0 v h
```

While the $\mapsto_{\mathsf{T}}$ relation cannot be implemented directly, we may nevertheless model it as a set-valued function where each state reduces to a *set of possible results*:

```
type REL α β = α → Set β    -- Heterogeneous binary relations
type Rel α = REL α α        -- Homogeneous binary relations

reduce_Tran :: Rel (Heap, Tran)
```

Reading a variable v looks up its value in the heap,

$$\langle h,\ \mathsf{Read}\ v\rangle \mapsto_{\mathsf{T}} \langle h,\ \mathsf{Val}_{\mathsf{T}}\ (h\ ?\ v)\rangle \qquad \text{(Read)}$$

which is implemented by the following code, where we have written Set.singleton x as $\{x\}$ for clarity of presentation:

```
reduce_Tran (h, Read v) = {h, Val_T (h ? v)}
```

Writing to a variable is taken care of by two rules: (Write$\mathbb{Z}$) updates the heap with the new value for a variable in the same manner as the published semantics of STM Haskell [HMPJH05], while (WriteT) allows its argument expression to be repeatedly

reduced until it becomes a value,

$$\langle h,\ \mathsf{Write}\ v\ (\mathsf{Val}_\mathsf{T}\ n)\rangle \mapsto_\mathsf{T} \langle h \uplus \{v \mapsto n\},\ \mathsf{Val}_\mathsf{T}\ n\rangle \qquad (\mathsf{Write}\mathbb{Z})$$

$$\frac{\langle h,\ e\rangle \mapsto_\mathsf{T} \langle h',\ e'\rangle}{\langle h,\ \mathsf{Write}\ v\ e\rangle \mapsto_\mathsf{T} \langle h',\ \mathsf{Write}\ v\ e'\rangle} \qquad (\mathsf{WriteT})$$

writing $h \uplus \{v \mapsto n\}$ to denote the heap h with the variable v updated to n.

We implement these two rules by inspecting the subexpression e whose value we wish to update the heap with. In the the former case, e is just a plain number—corresponding to (WriteℤT)—and we update the heap with the new value of v accordingly. The latter case implements (WriteT) by recursively reducing the subexpression e, then reconstructing $\mathsf{Write}\ v\ e'$ by mapping $\mathsf{second}\ (\mathsf{Write}\ v)$ over the resulting set of (h', e'):

```
reduceTran (h, Write v e) = case e of
  ValT n → {h ⊎ {v ↦ n}, ValT n}
  _      → second (Write v) `Set.map` reduceTran (h, e)
```

As we replace the act of sequencing computations with addition in our language, it is therefore important to enforce a sequential evaluation order. The final group of three rules define reduction for $\oplus_\mathsf{T}$, and ensure the left argument is evaluated to completion, before starting on the right hand side:

$$\langle h,\ \mathsf{Val}_\mathsf{T}\ m \oplus_\mathsf{T} \mathsf{Val}_\mathsf{T}\ n\rangle \mapsto_\mathsf{T} \langle h,\ \mathsf{Val}_\mathsf{T}\ (m + n)\rangle \qquad (\mathsf{Add}\mathbb{Z}_\mathsf{T})$$

$$\frac{\langle h,\ b\rangle \mapsto_\mathsf{T} \langle h',\ b'\rangle}{\langle h,\ \mathsf{Val}_\mathsf{T}\ m \oplus_\mathsf{T} b\rangle \mapsto_\mathsf{T} \langle h',\ \mathsf{Val}_\mathsf{T}\ m \oplus_\mathsf{T} b'\rangle} \qquad (\mathsf{AddR}_\mathsf{T})$$

$$\frac{\langle h,\ a\rangle \mapsto_\mathsf{T} \langle h',\ a'\rangle}{\langle h,\ a \oplus_\mathsf{T} b\rangle \mapsto_\mathsf{T} \langle h',\ a' \oplus_\mathsf{T} b\rangle} \qquad (\mathsf{AddL}_\mathsf{T})$$

Our implementation of $\oplus_\mathsf{T}$ mirrors the above rules, as below:

```
reduceTran (h, a ⊕T b) = case (a, b) of
  (ValT m, ValT n) → {h, ValT (m + n)}
  (ValT m, _     ) → second (ValT m ⊕T  ) `Set.map` reduceTran (h, b)
  (_     , _     ) → second (     ⊕T b) `Set.map` reduceTran (h, a)
```

To complete the definition of $\mathsf{reduce}_\mathsf{Tran}$, we require a further case,

```
reduceTran (h, ValT m) = {}
```

where we return the empty set for $\mathsf{Val}_\mathsf{T}\ m$, as it has no associated reduction rules.

Because $\mapsto_\mathsf{T}$ only describes a single reduction step, we also need to implement a helper function to run a given initial expression to completion for our executable model. Let us first define $\mathsf{join}_\mathsf{Set}$, which flattens nested Sets:

```
joinSet :: Set (Set α) → Set α
joinSet = Set.fold (∪) {}
```

Here, Set.union and Set.empty are written as $(\cup)$ and $\{\}$ respectively.

The following definition of reduceUntil—parameterised over a relation *reduce*—reduces the given *init* state to completion, according to the predicate p:

```
reduceUntil :: (α → Maybe β) → Rel α → REL α β
reduceUntil p reduce init = step ({init}, {}) where
  step :: (Set α, Set β) → Set β
  step (running, finished) = case Set.null running of
    True  → finished
    False → step (first (joinSet ∘ Set.map reduce)
                        (Set.fold partition ({}, finished) running))
  partition :: α → (Set α, Set β) → (Set α, Set β)
```

```
partition e = case p e of
  Nothing → first (Set.insert e)
  Just n  → second (Set.insert n)
```

The step helper takes a pair of *running* and *finished* sets of states, accumulating those that satisfy p into the finished set for the next iteration with the aid of partition, and repeatedly applies *reduce* to the set of remaining running states until it becomes exhausted.

Finally, given the following $\mathsf{isVal}_\mathsf{T}$ predicate,

```
isVal_T :: (Heap, Tran) → Maybe (Heap, Integer)
isVal_T (h, Val_T n) = Just (h, n)
isVal_T (h, _      ) = Nothing
```

the expression reduceUntil $\mathsf{isVal}_\mathsf{T}$ $\mathsf{reduce}_\mathsf{Tran}$ then corresponds to an implementation of $\mapsto^\star_\mathsf{T}$, which produces a set of (Heap, Integer) pairs from an initial (Heap, Tran).

5.1.3 Process Soup Semantics

The reduction relation $\mapsto_\mathsf{P}$ for processes acts on pairs $\langle h,\ s\rangle$ consisting of a heap h as before, and a 'soup' s of running processes [PJ01]. While the soup itself is to be regarded as a multi-set, here we use a more concrete representation, namely a list of Procs.

The reduction rules for process are in general defined by matching on the first process in the soup. However, we begin by giving the (Preempt) rule, which allows the rest of the soup to make progress, giving rise to non-determinism in the language:

$$\frac{\langle h,\ s\rangle \mapsto_\mathsf{P} \langle h',\ s'\rangle}{\langle h,\ p : s\rangle \mapsto_\mathsf{P} \langle h',\ p : s'\rangle} \qquad \text{(Preempt)}$$

Our implementation of $\mapsto_\mathsf{P}$ comprise a pair of mutually-recursive definitions,

$$\begin{aligned}&\mathsf{reduce_P} :: \mathsf{Proc} \to \mathsf{Rel}\ (\mathsf{Heap}, [\mathsf{Proc}])\\&\mathsf{reduce_S} :: \qquad \mathsf{Rel}\ (\mathsf{Heap}, [\mathsf{Proc}])\end{aligned}$$

where $\mathsf{reduce_P}$ performs a single-step reduction of a particular Proc in the context of the given heap and soup, and $\mathsf{reduce_S}$ corresponds to the general case. We begin with the definition of $\mathsf{reduce_S}$:

$$\begin{aligned}&\mathsf{reduce_S}\ (h, [\,]\ \ ) = \{\}\\&\mathsf{reduce_S}\ (h, p : s) = (\mathsf{second}\ (p :)\ \text{`Set.map`}\ \mathsf{reduce_S}\ (h, s))\\&\quad \cup\, \mathsf{reduce_P}\ p\ (h, s)\end{aligned}$$

That is, when the soup is empty, no reduction is possible, so we return a empty set.

When the soup is not empty, we can either apply (Preempt) to reduce the rest rest of the soup s, or reduce only the first process p using $\mathsf{reduce_P}$. These two sets of reducts are combined using $(\cup)$.

In turn, for the definition of $\mathsf{reduce_P}$, it is not possible for values to reduce any further in our semantics, so we return an empty set when p is a $\mathsf{Val_P}$.

$$\mathsf{reduce_P}\ (\mathsf{Val_P}\ n) \quad (h, s) = \{\}$$

Executing $\mathsf{Fork}\ p$ adds p to the process soup, and evaluates to $\mathsf{Val_P}\ 0$ (which corresponds to $\mathsf{return}\ ()$ in Haskell) as the result of this action:

$$\langle h,\ \mathsf{Fork}\ p : s\rangle \mapsto_{\mathsf{P}} \langle h,\ \mathsf{Val_P}\ 0 : p : s\rangle \qquad \text{(Fork)}$$

This is handled by the following case in the definition of $\mathsf{reduce_P}$:

$$\mathsf{reduce_P}\ (\mathsf{Fork}\ p) \quad (h, s) = \{h, \mathsf{Val_P}\ 0 : p : s\}$$

Next, the (Atomic) rule has a premise which evaluates the given expression until it reaches a value (where $\mapsto_{\mathsf{T}}^{\star}$ denotes the reflexive, transitive closure of $\mapsto_{\mathsf{T}}$), and a

conclusion which encapsulates this as a single transition on the process level:

$$\frac{\langle h,\ e\rangle \mapsto_{\mathsf{T}}^{\star} \langle h',\ \mathsf{Val}_{\mathsf{T}}\ n\rangle}{\langle h,\ \mathsf{Atomic}\ e : s\rangle \mapsto_{\mathsf{P}} \langle h',\ \mathsf{Val}_{\mathsf{P}}\ n : s\rangle} \qquad \text{(Atomic)}$$

In this manner we obtain a *stop-the-world* semantics for atomic transactions, preventing interference from other concurrently executing processes. Note that while the use of $\mapsto_{\mathsf{T}}^{\star}$ may seem odd in a small-step semantics, it expresses the intended meaning in a clear and concise way [HMPJH05], namely that the transaction executes as if it were a single atomic step.

Our model of the (Atomic) rule implements the same stop-the-world semantics using reduceUntil defined in the previous section. The values resulting from the execution of t are then placed back into the soup:

```
reduceP (Atomic t) (h, s) = second (λn → ValP n : s)
  `Set.map` reduceUntil isValT reduceTran (h, t)
```

Finally, it is straightforward to handle $\oplus_{\mathsf{P}}$ on the process level using three rules, in an analogous manner to $\oplus_{\mathsf{T}}$ on the transaction level:

$$\langle h,\ \mathsf{Val}_{\mathsf{P}}\ m \oplus_{\mathsf{P}} \mathsf{Val}_{\mathsf{P}}\ n : s\rangle \mapsto_{\mathsf{P}} \langle h,\ \mathsf{Val}_{\mathsf{P}}\ (m + n) : s\rangle \qquad \text{(Add}\mathbb{Z}_{\mathsf{P}}\text{)}$$

$$\frac{\langle h,\ b : s\rangle \mapsto_{\mathsf{P}} \langle h',\ b' : s'\rangle}{\langle h,\ \mathsf{Val}_{\mathsf{P}}\ m \oplus_{\mathsf{P}} b : s\rangle \mapsto_{\mathsf{P}} \langle h',\ \mathsf{Val}_{\mathsf{P}}\ m \oplus_{\mathsf{P}} b' : s'\rangle} \qquad \text{(AddR}_{\mathsf{P}}\text{)}$$

$$\frac{\langle h,\ a : s\rangle \mapsto_{\mathsf{P}} \langle h',\ a' : s'\rangle}{\langle h,\ a \oplus_{\mathsf{P}} b : s\rangle \mapsto_{\mathsf{P}} \langle h',\ a' \oplus_{\mathsf{P}} b : s'\rangle} \qquad \text{(AddL}_{\mathsf{P}}\text{)}$$

The corresponding implementation mirrors that of $\oplus_{\mathsf{T}}$, evaluating expressions in a left-to-right order:

```
reduceP (a ⊕P b) (h, s) = case (a, b) of
```

```
  (Val_P m, Val_P n) → {h, Val_P (m + n) : s}
  (Val_P m, b)       → second (mapHead (Val_P m ⊕_P  )) `Set.map` reduce_S (h, b : s)
  (a,       b)       → second (mapHead (       ⊕_P b)) `Set.map` reduce_S (h, a : s)
  where mapHead f (p : s) = f p : s
```

In a similar way to our earlier definition for $\mathsf{isVal}_\mathsf{T}$, we define an $\mathsf{isVal}_\mathsf{S}$ predicate to determine when an entire soup has finished reducing.

```
isVal_S :: (Heap, [Proc]) → Maybe (Heap, [Integer])
isVal_S (h, s) = case traverse isVal_P s of
  Nothing → Nothing
  Just ns → Just (h, ns)
  where
  isVal_P :: Proc → Maybe Integer
  isVal_P (Val_P n) = Just n
  isVal_P _         = Nothing
```

This completes our executable model of the high-level semantics: in particular, the term $\mathsf{reduceUntil}\ \mathsf{isVal}_\mathsf{S}\ \mathsf{reduce}_\mathsf{S}$ then corresponds to an implementation of $\mapsto_\mathsf{P}$, which produces a set of $(\mathsf{Heap}, [\mathsf{Integer}])$ pairs from an initial $(\mathsf{Heap}, [\mathsf{Proc}])$.

In summary, the above semantics for transactions and processes mirror those for STM Haskell, but for a simplified language. Moreover, while the original semantics uses evaluation contexts to identify the point at which transition rules such as $(\mathsf{Add}\mathbb{Z}_\mathsf{P})$ can be applied, our language is sufficiently simple to allow the use of explicit structural rules such as $(\mathsf{AddL}_\mathsf{P})$ and $(\mathsf{AddR}_\mathsf{P})$, which for our purposes have the advantage of being directly implementable.

5.2 A Simple Transactional Machine

The (Atomic) rule of the previous section simply states that the evaluation sequence for a transaction may be seen as a single indivisible transition with respect to other concurrent processes. However, to better exploit the available multi-core hardware, an actual implementation of this rule would have to allow multiple transactions to run concurrently, while still maintaining the illusion of atomicity. In this section we consider how this notion of concurrent transactions can be implemented, and present a compiler and virtual machine for our language.

5.2.1 Instruction Set

Let us consider compiling expressions into code for execution on a stack machine, in which Code comprises a sequence of Instructions:

```
type Code        = [Instruction]
data Instruction = PUSH Integer | ADD | READ Var | WRITE Var
                 | BEGIN | COMMIT | FORK Code
```

The PUSH instruction leaves its argument on top of the stack, while ADD replaces the top two numbers with their sum. The behaviour of the remaining instructions is more complex in order to maintain atomicity, but conceptually, READ pushes the value of the named variable onto the stack, while WRITE updates the variable with the topmost value. In turn, BEGIN and COMMIT mark the start and finish of a transaction, and FORK executes the given code concurrently.

5.2.2 Compiler

We define the compile_T and compile_P functions to provide translations from Tran and Proc to Code, both functions taking an additional Code argument to be appended

to the instructions produced by the compilation process, as in chapter 3. In both cases, integers and addition are compiled into PUSH and ADD instructions, while the remaining language constructs map directly to their analogous machine instructions. The intention is that executing a compiled transaction or process always leaves a single result value on top of the stack.

```
compile_T :: Tran → Code → Code
compile_T e c = case e of
  Val_T i       → PUSH i : c
  x ⊕_T y       → compile_T x (compile_T y (ADD : c))
  Read v        → READ v : c
  Write v e'    → compile_T e' (WRITE v : c)
compile_P :: Proc → Code → Code
compile_P e c = case e of
  Val_P i       → PUSH i : c
  x ⊕_P y       → compile_P x (compile_P y (ADD : c))
  Atomic e'     → BEGIN : compile_T e' (COMMIT : c)
  Fork x        → FORK (compile_P x []) : c
```

For example, invoking $\mathsf{compile_P}$ (incTwice *counter*) [] delivers the following code:

```
[FORK [BEGIN, READ counter, PUSH 1, ADD, WRITE counter, COMMIT]
, FORK [BEGIN, READ counter, PUSH 1, ADD, WRITE counter, COMMIT]
, ADD]
```

5.2.3 Implementing Transactions

The simplest method of implementing transactions would be to suspend execution of all other concurrent processes on encountering a BEGIN, and carry on with the current process until we reach the following COMMIT. In essence, this is the approach

used in the high-level semantics presented in the previous section. Unfortunately, this does not allow transactions to execute concurrently, one of the key aspects of transactional memory. This section introduces the log-based approach to implementing transactions, and discusses a number of design issues.

Transaction Logs

In order to allow transactions to execute concurrently, we utilise the notion of a *transaction log*. Informally such a log behaves as a cache for read and write operations on transactional variables. Only the first read from any given variable accesses the heap, and only the last value written can potentially modify the heap; all intermediate reads and writes operate solely on the log. Upon reaching the end of the transaction, and provided that that no other concurrent process has 'interfered' with the current transaction, the modified variables in the log can then be committed to the heap. Otherwise, the log is discarded and the transaction is restarted afresh.

Note that restarting a transaction relies on the fact that it executes in complete isolation, in the sense that all its side-effects are encapsulated within the log, and hence can be revoked by simply discarding the log. For example, it would not be appropriate to 'launch missiles' [HMPJH05] during a transaction.

Interference

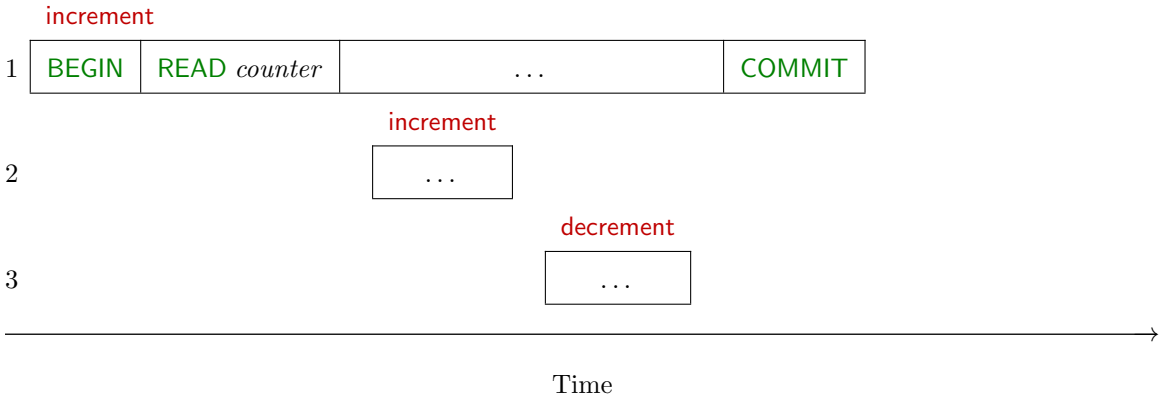
But what constitutes *interference*? When a transaction succeeds and commits its log to the heap, all of its side-effects are then made visible in a single atomic step, as if it had been executed in its entirety at that point with a stop-the-world semantics. Thus when a variable is read for the first time and its value logged, the transaction is essentially making the following bet: at the end of the transaction, the value of the variable in the heap will still be the same as that in the log.

In this manner, interference arises when any such bet fails, as the result of other

concurrent processes changing the heap in a way that invalidates the assumptions about the values of variables made in the log. In this case, the transaction fails and is restarted. Conversely, the transaction succeeds if the logged values of all the variables read are 'equal' to their values in the heap at the end of the transaction.

Equality

But what constitutes *equality*? To see why this is an important question, and what the design choices are, let us return to our earlier example of a transaction that increments a given counter. Consider the following timeline:



Suppose the *counter* starts at zero, which is read by the first transaction and logged. Prior to its final COMMIT, a second concurrent transaction successfully increments the *counter*, which is subsequently decremented by a third transaction. When the first finally attempts to commit, the count is back to zero as originally logged, even though it has changed in the interim. Is this acceptable? That is, are the two zeros 'equal'? We can consider a hierarchy of notions of equality, in increasing order of permissiveness:

- The most conservative choice is to increment a global counter every time the heap is updated. Under this scheme, a transaction fails if the heap is modified at any point during its execution, reflected by a change in the counter, even if this does not actually interfere with the transaction itself.

- A more refined approach is provided by the notion of *version equality*, where a separate counter is associated with each variable, and is incremented each time the variable is updated. In this case, our example transaction would still fail to commit, since the two zeros would be have different version numbers, and hence considered different.

- For a pure language such as Haskell, in which values are represented as pointers to immutable structures, *pointer equality* can be used as an efficient but weaker form of version equality. In this case, whether the two zeros are considered equal or not depends on whether the implementation created a new instance of zero, or reused the old zero by sharing.

- We can also consider *value equality*, in which two values are considered the same if they have the same representation. In this case, the two zeros are equal and the transaction succeeds.

- The most permissive choice would be a *user-defined equality*, beyond that built-in to the programming language itself, in order to handle abstract data structures in which a single value may have several representations, e.g. sets encoded as lists. Haskell provides this capability via the Eq typeclass.

Which of the above is the appropriate notion of equality when committing transactions? Recall that under a stop-the-world semantics, a transaction can be considered to be executed in its entirely at the point when it successfully commits, and any prior reads are effectively bets on the state of the heap at the commit point. Any intermediate writes that may have been committed by other transactions do not matter, as long as the final heap is consistent with the bets made in the log. Hence in our instance, there is no need at commit time to distinguish between the two zeroes in our example, as they are equal in the high-level expression language.

From a semantics point of view, therefore, value or user-defined equality are the best choices. Practical implementations may wish to adopt a more efficient notion of equality (e.g. STM Haskell utilises pointer equality), but for the purposes of this thesis, we will use value equality.

5.2.4 Virtual Machine

The state of the virtual machine is given by a pair $\langle h,\ s\rangle$, comprising a heap h mapping variables to integers, and a soup s of concurrent *threads*. A Thread is represented as a tuple of the form (c, σ, f, r, w), where c is the code to be executed, σ is the thread-local stack, f gives the code to be rerun if a transaction fails to commit, and finally, r and w are two logs (partial maps from variables to integers) acting as read and write caches between a transaction and the heap.

```
type Thread = (Code, Stack, Code, Log, Log)
type Stack  = [Integer]
type Log    = Map Var Integer
```

We specify the behaviour of the machine using a transition relation $\mapsto_{\mathsf{M}}$ between machine states, defined via a collection of transition rules that proceed by case analysis on the first thread in the soup. As with the previous semantics, we begin by defining a (PREEMPT) rule to allow the rest of the soup to make progress, giving rise to non-determinism in the machine:

$$\frac{\langle h,\ s\rangle \mapsto_{\mathsf{M}} \langle h',\ s'\rangle}{\langle h,\ t : s\rangle \mapsto_{\mathsf{M}} \langle h',\ t : s'\rangle} \qquad \text{(PREEMPT)}$$

This rule corresponds to an idealised scheduler that permits context switching at every instruction, as our focus is on the implementation of transactions rather than scheduling policies. We return to this issue when we consider the correctness of our

compiler later on in this chapter.

We implement $\mapsto_{\mathsf{M}}$ using a pair of mutually recursive functions $\mathsf{step}_{\mathsf{M}}$ and $\mathsf{step}_{\mathsf{T}}$ in a similar fashion to that of $\mapsto_{\mathsf{P}}$ earlier. The former implements reduction between arbitrary soups of threads:

```
step_M :: Rel (Heap, [Thread])
step_M (h, []   ) = {}
step_M (h, t : s) = (second (t : ) `Set.map` step_M (h, s)) ∪ step_T t (h, s)
```

The first case handles empty soups, returning an empty set of resulting states. The second case takes the first thread t in the soup, and implements the (PREEMPT) rule by reducing the rest of the soup s before placing t back at the head. These are then combined with the states resulting from a single step of t, implemeted by $\mathsf{step}_{\mathsf{T}}$:

```
step_T :: Thread → Rel (Heap, [Thread])
step_T ([],   σ, f, r, w) (h, s) = {}
step_T (i : c, σ, f, r, w) (h, s) = step_I i where
  ∼(n : σ1@∼(m : σ2)) = σ
  step_I :: Instruction → Set (Heap, [Thread])
  ... defined below
```

A thread with an empty list of instructions cannot make any transitions, so we return an empty set. When there is at least one instruction remaining, we use the $\mathsf{step}_{\mathsf{I}}$ helper function to handle each particular instruction. The above code also brings into scope the names c, σ, f, r and w as detailed previously, as well as σ_1 and σ_2 for the current stack with one or two values popped.

Let us detail the semantics of each instruction in turn. Firstly, executing FORK adds a new thread t to the soup, comprising the given code c' with an initially empty

stack, restart code and read and write logs:

$$\langle h,\ (\mathsf{FORK}\ c' : c,\ \sigma,\ f,\ r,\ w) : s\rangle \mapsto_{\mathsf{M}} \langle h,\ (c,\ 0 : \sigma,\ f,\ r,\ w) : t : s\rangle \qquad \text{(FORK)}$$

where $t = (c',\ [],\ [],\ \emptyset,\ \emptyset)$

The above transition may be implemented directly, as follows:

$$\begin{aligned}&\mathsf{step_I}\ (\mathsf{FORK}\ c') \ = \{h,\ (c,\ \ 0 : \sigma,\ f,\ \ r,\ \ w) : t : s\}\\ &\quad \textbf{where}\ t = (c', [], [], \{\}, \{\})\end{aligned}$$

The PUSH instruction places its argument n on top of the stack, while ADD takes the top two integer from the stack and replaces them with their sum:

$$\langle h, (\mathsf{PUSH}\ n, c, \sigma, f, r, w) : s\rangle \mapsto_{\mathsf{M}} \langle h, (c, n : \sigma, f, r, w) : s\rangle \qquad \text{(PUSH)}$$

$$\langle h, (\mathsf{ADD}, c, n : m : \sigma, f, r, w) : s\rangle \mapsto_{\mathsf{M}} \langle h, (c, m + n : \sigma, f, r, w) : s\rangle \qquad \text{(ADD)}$$

The corresponding cases in the definition of $\mathsf{step_I}$ are almost identical:

$$\begin{aligned}&\mathsf{step_I}\ (\mathsf{PUSH}\ n) \ = \{h,\ (c,\ \ n : \sigma,\ f,\ \ r,\ \ w) \ \ : s\}\\ &\mathsf{step_I}\ \mathsf{ADD} \ \ = \{h,\ (c, m + n : \sigma_2, f,\ \ r,\ \ w) \ \ : s\}\end{aligned}$$

Executing BEGIN starts a transaction, which involves clearing the read and write logs, while making a note of the code to be executed if the transaction fails:

$$\langle h,\ (\mathsf{BEGIN} : c,\ \sigma,\ f,\ r,\ w) : s\rangle \mapsto_{\mathsf{M}} \langle h,\ (c,\ \sigma,\ \mathsf{BEGIN} : c,\ \emptyset,\ \emptyset) : s\rangle \qquad \text{(BEGIN)}$$

Accordingly, $\mathsf{step_I}$ sets up the retry code and initialises both read and write logs:

$$\mathsf{step_I}\ \mathsf{BEGIN} \ \ = \{h,\ (c,\ \ \sigma, \mathsf{BEGIN} : c, \{\}, \{\}) \ \ : s\}$$

Next, READ places the appropriate value for the variable v on top of the stack. The instruction first consults the write log. If the variable has not been written to, the

read log is then consulted. Otherwise, if the variable has not been read from either, its value is looked up from the heap and the read log updated accordingly:

$$\langle h,\ (\mathsf{READ}\ v : c,\ \sigma,\ f,\ r,\ w) : s\rangle \mapsto_{\mathsf{M}} \langle h,\ (c,\ n : \sigma,\ f,\ r',\ w) : s\rangle \qquad \text{(READ)}$$

$$\text{where } \langle n,\ r'\rangle = \begin{cases} \langle w(v),\ r\rangle & \text{if } v \in \mathrm{dom}(w) \\ \langle r(v),\ r\rangle & \text{if } v \in \mathrm{dom}(r) \\ \langle h(v),\ r \uplus \{v \mapsto h\ ?\ v\}\rangle & \text{otherwise} \end{cases}$$

The transliteration of the (READ) rule to our implementation is as follows:

```
stepI (READ v)   = {h, (c,      n : σ, f,        r', w)    : s}
  where (n, r') = case (Map.lookup v w, Map.lookup v r, h ? v) of
    (Just n', _,          _ ) → (n', r)
    (Nothing, Just n', _ ) → (n', r)
    (Nothing, Nothing, n') → (n', r ⊎ {v ↦ n'})
```

In turn, WRITE simply updates the write log for the variable v with the value on the top of the stack, without changing the heap or the stack:

$$\langle h,\ (\mathsf{WRITE}\ v : c,\ n : \sigma,\ f,\ r,\ w) : s\rangle \mapsto_{\mathsf{M}} \langle h,\ (c,\ n : \sigma,\ f,\ r,\ w') : s\rangle \qquad \text{(WRITE)}$$

$$\text{where } w' = w \uplus \{v \mapsto n\}$$

The (WRITE) rule has a similarly straightforward implementation:

```
stepI (WRITE v) = {h, (c,      n : σ1, f,       r,  w')   : s} where
  w' = w ⊎ {v ↦ n}
```

Finally, COMMIT checks the read log r for consistency with the current heap h, namely that the logged value for each variable read is equal to its value in the heap. According to the above definitions of (READ) and (WRITE), variables written to

before being read from during the same transaction will not result in a read log entry, since the corresponding value in the global heap cannot influence the results of the transaction. Therefore, we do not need to perform consistency checks for variables that occur only in the write log.

Using our representation of logs and heaps, the consistency condition can be concisely stated as $r \subseteq h$. That is, if they are consistent, then the transaction has succeeded, so we may commit its write log w to the heap. This update is expressed in terms of the overriding operator on maps as $h \uplus w$. Otherwise the transaction has failed, in which case the heap is not changed, the result on the top of the stack is discarded, and the transaction is restarted at f:

$$\langle h,\ (\mathsf{COMMIT} : c,\ n : \sigma,\ f,\ r,\ w) : s\rangle \mapsto_{\mathsf{M}} \langle h',\ (c',\ \sigma',\ f,\ r,\ w) : s\rangle$$

$$\text{where } \langle h',\ c',\ \sigma'\rangle = \begin{cases} \langle h \uplus w,\ c,\ n : \sigma\rangle & \text{if } r \subseteq h \\ \langle h,\ f,\ \sigma\rangle & \text{otherwise} \end{cases} \qquad \text{(COMMIT)}$$

There is no need to explicitly clear the logs in the above rule, since this is already taken care of by the fact that the first instruction of f is always a BEGIN.

```
step_I COMMIT   = {h', (c',        σ', f,        r,  w)    : s}
  where (h', c', σ') = case (r ∩ h) ⊆ h of
    True  → (h ⊎ w, c, n : σ_1)
    False → (h,     f,   σ)
```

Finally we define a $\mathsf{halted}_{\mathsf{M}}$ function on virtual machines to discriminate between completed and running threads.

```
halted_M :: (Heap, [Thread]) → Maybe (Heap, [Integer])
halted_M (h, s) = case traverse halted_T s of
  Just ns → Just (h, ns)
```

```
Nothing → Nothing
where
haltedT :: Thread → Maybe Integer
haltedT ([], n : [], _, _, _) = Just n
haltedT _ = Nothing
```

5.3 Correctness of the Implementation

As we have seen, the high-level semantics of atomicity is both clear and concise, comprising a single inference rule (Atomic) that wraps up a complete evaluation sequence as a single transition. On the other hand, the low-level implementation of atomicity using transactions is rather more complex and subtle, involving the management of read and write logs, and careful consideration of the conditions that are necessary in order for a transaction to commit. How can we be sure that these two different views of atomicity are consistent? Our approach to establishing the correctness of the low-level implementation is to formally relate it to the high-level semantics via a compiler correctness theorem.

5.3.1 Statement of Correctness

In order to formulate our correctness result, we utilise a number of auxiliary definitions. First of all, because our semantics is non-deterministic, we define a relation $\Downarrow_{\mathsf{P}}$ that encapsulates the idea of completely evaluating a process using our high-level semantics:

$$\langle h,\ ps\rangle \Downarrow_{\mathsf{P}} \langle h',\ ps'\rangle \quad \leftrightarrow \quad \langle h,\ ps\rangle \mapsto_{\mathsf{P}}^{\star} \langle h',\ ps'\rangle \not\mapsto_{\mathsf{P}}$$

That is, a process soup $ps :: [\mathsf{Proc}]$ with the initial heap h can evaluate to any heap h' and soup ps' that results from completely reducing ps using our high-level semantics,

where $\not\mapsto_{\mathsf{P}}$ expresses that no further transitions are possible. We may implement the $\Downarrow_{\mathsf{P}}$ relation as the following eval function:

$$\begin{aligned}&\mathsf{eval} :: \mathsf{REL}\ (\mathsf{Heap}, [\mathsf{Proc}])\ (\mathsf{Heap}, [\mathsf{Integer}])\\&\mathsf{eval} = \mathsf{reduceUntil}\ \mathsf{isVal}_{\mathsf{S}}\ \mathsf{reduce}_{\mathsf{S}}\end{aligned}$$

Similarly, we define a relation $\Downarrow_{\mathsf{M}}$ that encapsulates complete execution of a thread soup $ts :: [\mathsf{Thread}]$ with the initial heap h using our virtual machine, resulting in a heap h' and a thread soup ts':

$$\langle h,\ ts\rangle \Downarrow_{\mathsf{M}} \langle h',\ ts'\rangle \quad \leftrightarrow \quad \langle h,\ ts\rangle \mapsto^{\star}_{\mathsf{M}} \langle h',\ ts'\rangle \not\mapsto_{\mathsf{M}}$$

Likewise, we may implement $\Downarrow_{\mathsf{M}}$ as the following exec function:

$$\begin{aligned}&\mathsf{exec} :: \mathsf{REL}\ (\mathsf{Heap}, [\mathsf{Thread}])\ (\mathsf{Heap}, [\mathsf{Integer}])\\&\mathsf{exec} = \mathsf{reduceUntil}\ \mathsf{halted}_{\mathsf{M}}\ \mathsf{step}_{\mathsf{M}}\end{aligned}$$

Next, we define a function load that converts a process into a corresponding thread for execution, which comprises the compiled code for the process, together with an empty stack, restart code and read and write logs:

$$\begin{aligned}&\mathsf{load} :: [\mathsf{Proc}] \to [\mathsf{Thread}]\\&\mathsf{load} = \mathsf{map}\ (\lambda p \to (\mathsf{compile}_{\mathsf{P}}\ p\ [], [], [], \{\}, \{\}))\end{aligned}$$

Using these definitions, the correctness of our compiler can now be expressed by the following property:

$$\begin{aligned}&\forall p \in \mathsf{Proc}, h \in \mathsf{Heap}, s \in [\mathsf{Integer}].\\&\qquad \langle \{\},\ p : []\rangle \Downarrow_{\mathsf{P}} \langle h,\ s\rangle \leftrightarrow \langle \{\},\ \mathsf{load}\ (p : [])\rangle \Downarrow_{\mathsf{M}} \langle h,\ s\rangle\end{aligned}$$

That is, evaluating a process p starting with an initial heap using our high-level stop-the-world process semantics is equivalent to compiling and loading the process, and

executing the resulting thread using the interleaved virtual machine semantics. For the purposes of proving the result, we generalise the above over a process soup rather than a single process, as well as an arbitrary initial heap:

Theorem 5.1 (Compiler Correctness)**.**

$$\forall ps \in [\mathsf{Proc}], h, h' \in \mathsf{Heap}, s \in [\mathsf{Integer}].$$
$$\langle h,\ ps\rangle \Downarrow_{\mathsf{P}} \langle h',\ s\rangle \leftrightarrow \langle h,\ \mathsf{load}\ ps\rangle \Downarrow_{\mathsf{M}} \langle h',\ s\rangle$$

The above $\leftrightarrow$ equivalence can also be considered separately, where the $\rightarrow$ direction corresponds to soundness, and states that the compiled code will always produce a result that is permitted by the semantics. Dually, the $\leftarrow$ direction corresponds to completeness, and states that the compiled code can indeed produce every result permitted by the semantics.

In practice, some language implementations are not complete with respect to the semantics for the language by design, because implementing every behaviour that is permitted by the semantics may not be practical or efficient. For example, a real implementation may utilise a scheduler that only permits a context switch between threads at fixed intervals, rather than after every transition as in our semantics, because doing so would be prohibitively expensive.

5.3.2 Validation of Correctness

Proving the correctness of programs in the presence of concurrency is notoriously difficult. Ultimately we would like to have a formal proof, but the randomised testing approach—using QuickCheck and HPC, as described in Chapter 4—can provide a high level of assurance with relatively minimal effort. In the case of theorem 5.1, we can transcribe it as the following property:

```
prop_Correctness :: Heap → [Proc] → Bool
prop_Correctness h ps = eval (h, ps) ≡ exec (h, load ps)
```

In other words, from any initial heap h and process soup ps, the stop-the-world semantics produces the same set of possible outcomes as that from executing the compiled thread soup using a log-based implementation of transactions.

Given suitable Arbitrary instances for *Prop* and Heap, we can use QuickCheck to generate a large number of random test cases, and check that the theorem holds in each and every one:

```
*Main> quickCheck prop_Correctness
OK, passed 100 tests.
```

Having performed many thousands of tests in this manner, we can be highly confident in the validity of our compiler correctness theorem. However, as with any testing process, it is important to ensure that all the relevant parts of the program have been exercised in the process. Repeating the coverage checking procedure described in §4.2.3, we obtain the following report of unevaluated expressions[1]:

```
module "Main" {
  inside "reduceTran" {
    tick "Set.empty" on line 303;
  }
}
```

The {} corresponds to the Val_T case of the $\mathsf{reduce}_{\mathsf{Tran}}$ function, which remains unused simply because we never ask it for the reduct of $\mathsf{Val}_\mathsf{T}\ m$ expressions.

[1] As before, unused expressions corresponding to compiler-generated instances have been omitted.

5.4 Conclusion

In this chapter we have shown how to implement software transactional memory correctly, for a simplified language inspired by STM Haskell. Using QuickCheck and HPC, we tested a low-level, log-based implementation of transactions with respect to a high-level, stop-the-world semantics, by means of a compiler and its correctness theorem. This appears to be the first time that the correctness of a compiler for a language with transactions has been mechanically tested.

The lightweight approach provided by QuickCheck and HPC was indispensable in allowing us to experiment with the design of the language and its implementation, and to quickly check any changes. Our basic definitions were refined many times during the development of this work, both as a result of correcting errors, and streamlining the presentation. Ensuring that our changes were sound was simply a matter of re-running QuickCheck and HPC.

On the other hand, it is important to recognise the limitations of this approach. First of all, randomised testing does not constitute a formal proof, and the reliability of QuickCheck depends heavily on the quality of the test-case generators. Secondly, achieving 100% code coverage with HPC does not guarantee that all possible interactions between parts of the program have been tested. Nonetheless, we have found the use of these tools to be invaluable in our work.

Chapter 6

Machine-Assisted Proofs in Agda

To give a formal proof of the correctness property posited in the previous chapter, we may make use of a mechanised proof assistant. Agda [Nor07, The10] is a dependently-typed functional programming language based on Martin-Löf intuitionistic type theory [ML80, NPS90]. Via the Curry-Howard correspondence—that is, viewing types as propositions and programs as proofs—it is also used as a proof assistant for constructive mathematics. In this chapter, we shall provide a gentle introduction to the language, and demonstrate how we can formalise statements of compiler correctness by means of machine-checked proofs, culminating in a verified formalisation of the proofs of chapter 3.

6.1 Introduction to Agda

The current incarnation of Agda has a syntax similar to that of Haskell, and should look familiar to readers versed in the latter. As in previous chapters, we will adopt a colouring convention for ease of readability:

Syntactic Class	Examples
Keywords	**data** , **where** , **with** . . .

Types	ℕ, List, Set...
Constructors	zero, suc, tt, []...
Functions	id, _+_, Star.gmap...

Semantically, Agda is distinguished by its foundation on *dependent types*, and is closely related to systems such as Epigram [M+08, McB05] and Coq [The08]. Dependent types systems are so-called because they allow for types to depend on values, in addition to the usual parametrisation by other types as seen in languages such as Haskell. This provides us with a much richer vocabulary of discourse for not only stating the properties of our programs, but also to be able to prove such properties within the same system. We will begin to explore how this is facilitated by dependent types from section 6.1.2 onwards.

6.1.1 Data and Functions

We start our introduction to Agda with some simple data and function definitions. The language itself does not specify any primitive data types, and it serves as a good introduction to see how some of these may be defined in its standard library [Dan10b]. For example, we may define the Peano numbers as follows:

```
data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ
```

This is syntactically similar to Haskell's generalised abstract data type (GADT) declarations [PJWW04] with a few minor differences. Firstly, arbitrary Unicode characters may be used in identifiers, and we do not use upper and lower case letters to distinguish between values and constructors[1]. Secondly, we write : to mean *has-type-of*,

[1]The implication here is that the processes of syntax highlighting and type-checking are inextricably linked, and that syntax colours provides more information for the reader.

and write Set for the *type of types*[2].

Thus, the above defines ℕ as a new data type inhabiting Set, with a nullary constructor zero as the base case and an unary constructor suc as the inductive case. These correspond to two of the Peano axioms that define the natural numbers: zero is a natural number, and every number n has a successor suc n.

We may define addition on the natural numbers as follows, by induction on its first argument:

```
_+_ : ℕ → ℕ → ℕ
zero    + n = n
suc m + n = suc (m + n)
```

Agda allows the use of arbitrary mixfix operators, where underscores '_' denote the positions of arguments. Another difference is that all functions in Agda must be total. For example, omitting the zero case of _+_ would lead to an error during the typechecking process, rather than a warning as in Haskell. Additionally, only structural recursion is permitted. In the definition of _+_ above, we recurse on the definition of ℕ: the first argument is strictly smaller on the right hand side, i.e. m rather than suc m. While more general forms of recursion are possible, Agda requires us to explicitly prove that the resulting definitions are total.

6.1.2 Programs as Proofs and Types as Predicates

The Curry-Howard correspondence refers to the initial observation by Curry that types—in the sense familiar to functional programmers—correspond to axiom-schemes for intuitionistic logic, while Howard later noted that proofs in formal systems such as natural deduction can be directly interpreted as terms in a model of computation such as the typed lambda calculus.

[2]Agda in fact has countably infinite levels of Sets, with Set : Set_1 : Set_2 : .... This stratification prevents the formation of paradoxes that would lead to inconsistencies in the system.

The intuitionistic approach to logic only relies on constructive methods, disallowing notions from classical logic such as the law of the excluded middle ($P \vee \neg P$) or double-negation elimination ($\neg\neg P \to P$). For example, intuitionist reject $P \vee \neg P$ because there exists a statement P in any sufficiently powerful logic that can neither be proved nor disproved within the system, by Gödel's incompleteness theorems. In other words, intuitionism equates the truth of a statement P with the possibility of constructing a proof object that satisfies P, therefore a proof of $\neg\neg P$, refuting the non-existence of P, does not imply P itself.

What does this mean for the proletarian programmer? Under the Curry-Howard correspondence, the type $A \to B$ is interpreted as the logical statement 'A *implies* B', and vice-versa. Accordingly, a program $p : A \to B$ corresponds to a proof of 'A *implies* B', in that executing p constructs a witness of B as output, given a witness of A as input. Thus in a suitable type system, programming is the same as constructing proofs in a very concrete sense.

In a traditional strongly typed programming language such as Haskell, the type system exists to segregate values of different types. On the other hand, distinct values of the same type all look the same to the type-checker, which means we are unable to form types corresponding to propositions about particular values. Haskell's GADTs break down this barrier in a limited sense, by allowing the constructors of a parametric type to target particular instantiations of the return type. While this allows us to exploit a Haskell type checker that supports GADTs as a proof-checker in some very simple cases, it comes at the cost of requiring 'counterfeit type-level copies of data' [McB02].

6.1.3 Dependent Types

Dependent type systems follow a more principled approach, being founded on Martin-Löf intuitionistic type theory [NPS90]. These have been studied over several decades,

and the current incarnation of Agda [Nor07, The10] is one example of such a system.

Coming from a Hindley-Milner background, the key distinction of dependent types is that values can influence the types of other, subsequent values. Let us introduce the notion by considering an example of a dependent type:

```
data Fin : ℕ → Set where
  fz : {n : ℕ}          → Fin (suc n)
  fs : {n : ℕ} → Fin n → Fin (suc n)
```

This defines a data type Fin—similar to ℕ above—that is additionally *indexed* by a natural number. Its two constructor are analogues of the zero and suc of ℕ. The fz constructor takes an argument of type ℕ named n, that is referred to in its resultant type of Fin (suc n). The braces { and } indicate that n is an implicit parameter, which we may omit at occurrences of fz, provided that the value of n can be automatically inferred. We can see how this might be possible in the case of the fs constructor: its explicit argument has type Fin n, from which we may deduce n.

The above Fin represents a family of types, where each type Fin n has exactly n distinct values. This is apparent in the resultant types of the constructors, for example: neither fz and fs can inhabit Fin zero, as they both target Fin (suc n) for some n. The only inhabitant of Fin (suc zero) is fz, since fs requires an argument of type Fin zero, which would correspond to a proof that Fin zero is inhabited.

A routine application of the Fin family is in the safe lookup of lists or vectors, where a static guarantee on the bounds of the given position is required. The following definition defines the type of vectors, indexed by their length:

```
data Vec (X : Set) : ℕ → Set where
  []  :                          Vec X zero
  _::_ : {n : ℕ} → X → Vec X n → Vec X (suc n)
```

Unsurprisingly the empty list [] corresponds to a Vec X of length zero, while the _::_

constructor prepends an element of X to an existing Vec X of length n, to give a Vec X of length suc n.

A lookup function can then be defined as follows:

```
lookup : {X : Set} {n : ℕ} → Fin n → Vec X n → X
lookup fz     (x :: xs) = x
lookup (fs i) (x :: xs) = lookup i xs
```

The first argument gives the position in the vector where we wish to extract an element, with the second argument being the vector itself.

Earlier we mentioned that all definitions in Agda must be total, yet the lookup function seemingly does not consider the case of the empty vector []. This is because in a dependently-typed language, pattern matching on one argument can potentially influence the types of other arguments: matching on either fz or fs i forces n to suc n' for some n', and in both cases the type of the vector is refined to Vec X (suc n'). As [] can only inhabit Vec X zero, we needn't explicitly list this case. Had we pattern matched the vector with [] first on the other hand, the type of the position then becomes Fin zero, which is uninhabited. In this case, we may use the 'impossible' pattern () for the first argument, to indicate that the case is unreachable:

```
lookup ()     []
```

In such cases, the right hand side of the definition is simply omitted.

This is only an elementary demonstration of the power of dependent types. Being able to form any proposition in intuitionistic logic as a type gives us a powerful vocabulary with which to state and verify the properties of our programs. Conversely, by interpreting a type as its corresponding proposition, we have mapped the activity of constructing mathematical proofs to 'just' programming, albeit in a more rigorous fashion than usual.

6.1.4 Equality and its Properties

The previous section introduced dependent types from a programming and data types point-of-view. We shall now take a look at how we can use dependent types to state logical propositions and to construct their proofs.

A simple and commonplace construct is that of an equality type, corresponding to the proposition that two elements of the same type are definitionally equal. The following definition encodes the Martin-Löf equality relation:

```
data _≡_ {X : Set} : X → X → Set where
  refl : {x : X} → x ≡ x
```

Agda does not provide the kind of Hindley-Milner polymorphism as seen in Haskell, although we can simply take an additional parameter of the type we wish to be polymorphic over. In the above definition of equality, the variable X corresponds to the type of the underlying values, which can often be inferred from the surrounding context. By marking this parameter as implicit, we effectively achieve the same end result in its usage.

The above definition of _≡_ is indexed by two explicit parameters of type X. Its sole constructor is refl, that inhabits the type $x \equiv x$ given an argument $x : X$. Logically, refl corresponds to the axiom of reflexivity $\forall x.\ x \equiv x$. In cases where the type of the argument—such as x here—can be inferred form the surrounding context, the $\forall$ keyword allows us to write the type in a way that better resembles the corresponding logical notation, for example:

```
refl : ∀ {x} → x ≡ x
```

In general, Agda syntax allows us to write '_' in place of expressions—including types—that may be automatically inferred. The above is in fact syntactic sugar for the following:

```
refl : {x : _} → x ≡ x
```

Agda includes an interactive user interface for the Emacs [Sta10] operating system that supports incremental development by the placement of 'holes' where arbitrary expressions are expected. Incomplete programs with holes can be passed to the type checker, which then informs the user of the expected type. Thus, writing proofs in Agda typically involves a two-way dialogue between the user and the type checker.

Given the above definition of reflexivity as an axiom, we may go on to prove that $\equiv$ is also symmetric and transitive. The former is the proposition that given any x and y, a proof of $x \equiv y$ implies that $y \equiv x$. This is implemented as the following sym function,

```
sym : {X : Set} (x y : X) → x ≡ y → y ≡ x
sym x y x≡y = ?
```

where the '?' mark indicates an as-yet unspecified expression. Multiple arguments of the same type are simply separated with spaces, while arrows between pairs of named arguments can be omitted, since there cannot be any ambiguity.

Successfully type-checking the above turns the '?' into a *hole* { }, inside which we may further refine the proof. The expected type of the hole is displayed, and we can ask the type-checker to enumerate any local names that are in scope. In this case, the hole is expected to be of type $y \equiv x$, with the arguments $x\ y : X$ and $x{\equiv}y : x \equiv y$ in scope.

At this point, we can ask the system to perform case-analysis on $x{\equiv}y$. Being an equality proof, this argument must be the refl constructor. But something magical also happens during the case-split operation:

```
sym x .x refl = { }
```

Since refl only inhabits the type $x \equiv x$, the type checker concludes that the first two arguments x and y must be the same, and rewrites the second as $.x$ to reflect

this fact. Whereas pattern matching in Haskell is an isolated affair, in a dependently typed context it can potentially cause interactions with other arguments, revealing more information about the them that can be checked and enforced by the system.

Accordingly, y is no longer in-scope, and the goal type becomes $x \equiv x$, which is satisfied by refl:

```
sym x .x refl = refl
```

As we do not explicitly refer to x on the right hand side, we could have made the x and y arguments implicit too, leading to a more succinct definition:

```
sym : {X : Set} {x y : X} → x ≡ y → y ≡ x
sym refl = refl
```

We prove transitivity in a similar fashion,

```
trans : ∀ {X : Set} {x y z : X} → x ≡ y → y ≡ z → x ≡ z
trans refl refl = refl
```

where pattern-matching the first explicit argument with refl unifies y with x, refining the type of the second argument to $x \equiv z$; in turn, matching this with refl then unifies z with x. The resulting goal of $x \equiv x$ is met on the right hand side with simply refl.

6.1.5 Existentials and Dependent Pairs

In the previous section, we had already surreptitiously modelled the universal quantifier $\forall$ as dependent functions—that is, functions where values of earlier arguments may influence later types. Dually, we can model the existential quantifier $\exists$ using *dependent pairs*. This is typically defined in terms of the Σ type[3]:

[3]In this thesis, I deviate from the Agda standard library by writing _∧_ instead of _,_, to avoid excessive visual overloading.

```
data Σ (X : Set) (P : X → Set) : Set where
  _∧_ : (x : X) → (p : P x) → Σ X P
```

We can interpret the type $\Sigma\ X\ (\lambda\ x\ \rightarrow\ P\ x)$ as the existentially quantified statement that $\exists x \in X.\ P(x)$. Correspondingly, a proof of this comprises a pair $x\ \wedge\ p$, where the latter is a proof of the proposition $P\ x$. Unlike classical proofs of existence which may not necessarily be constructive, a proof of the existence of some x satisfying P necessarily requires us to supply such an x. Conversely, we can always extract an x given such a proof.

As X can often be inferred from the type of the predicate P, we may define a shorthand $\exists$ that accepts X as an implicit argument:

```
∃ : {X : Set} (P : X → Set) → Set
∃ {X} = Σ X
```

The above definition of $\exists$ allows us to write $\exists\ \lambda\ x\ \rightarrow\ P\ x$, which better resembles the corresponding logical proposition of $\exists x.\ P(x)$.

Of course, the predicate P need not necessarily depend on the first element of a pair. In such cases, the resulting type is a non-dependent pair, which corresponds to a logical conjunction. This can be recovered as a degenerate case of the above Σ type, in which we simply ignore the value of the first type:

```
_×_ : Set → Set → Set
X × Y = Σ X (λ _ → Y)
```

Putting the above into practice, we present below a definition of splitAt that splits a vector of length $m\ +\ n$ at position m into left and right vectors of length m and n respectively. Unlike the eponymous Haskell function, we can also witness that the contatenation of the resulting vectors coincides with the input:

```
splitAt : (m : ℕ) {n : ℕ} {X : Set} (xs : Vec X (m + n)) →
```

```
    Σ (Vec X m) λ ys → Σ (Vec X n) λ zs → xs ≡ ys ++ zs
splitAt zero xs = [] ∧ xs ∧ refl
```

For the base case where the position is zero, we simply return the empty list as the left part and the entirety of xs as the right. Since $[] \mathbin{+\!\!+} xs$ reduces to just xs, a simple appeal to reflexivity completes this clause.

In the inductive case of suc m, the input vector must contain at least one element, followed by xs. We wish to split xs recursively at m and prepend x to the left result. In Agda, we can pattern match on intermediate results using the magic ‘ **with** ’ keyword, which could be thought of as the dependently-typed analogue to Haskell’s **case**:

```
splitAt (suc m) (x :: xs)              with splitAt m xs
splitAt (suc m) (x :: .(ys ++ zs)) | ys ∧ zs ∧ refl = x :: ys ∧ zs ∧ refl
```

When we case-split on the proof of $xs \equiv ys \mathbin{+\!\!+} zs$, Agda sees that ≡.refl is the only possible constructor, and correspondingly rewrites the tail of the input vector to the dotted pattern $.(ys \mathbin{+\!\!+} zs)$. This is simply a more sophisticated instance of what we saw when case-splitting sym in the previous section.

6.1.6 Reflexive Transitive Closure

Before we conclude this brief introduction to Agda, we shall introduce the *reflexive transitive closure* of McBride, Norell and Jansson [McB07], which generalises the notion of sequences in a dependently-typed context. This general construct will prove useful later when working with sequences of small-step reductions.

We begin by defining binary relations parametrised on their underlying types:

```
Rel : Set → Set₁
Rel X = X → X → Set
```

Rel has Set_1 as its codomain in order to avoid the Set : Set inconsistency. We use the Rel X shorthand to denote the type of binary relations on X. In fact, our earlier definition of propositional equality could equivalently have been written as follows:

```
data _≡_ {X : Set} : Rel X where
  refl : {x : X} → x ≡ x
```

The definition of Star is parametrised on a set of indices I and a binary relation R on I. The type Star $\{I\}$ R is itself another binary relation, indexed on the same I:

```
data Star {I : Set} (R : Rel I) : Rel I where
  ε   : {i : I} → Star R i i
  _◁_ : {i j k : I} → (x : R i j) → (xs : Star R j k) → Star R i k
```

Here, ε gives a trivial witness of reflexivity for any i. The $\_\triangleleft\_$ constructor provides a heterogeneous definition of transitivity, accepting a proof of $R\ i\ j$ and an R-chain relating j and k as its two arguments. Thus Star R defines the type of the reflexive transitive closure of R. Being data, we may take apart an element of Star $R\ i\ k$ and inspect each step of the R-chain.

Alternatively the constructors ε and $\_\triangleleft\_$ could be thought of as generalisations of the *nil* and *cons* constructors of the list data type for proofs of the form $R\ i\ k$, with the constraint that two adjacent elements $x : R\ i\ j$ and $y : R\ j\ k$ must share a common index j.

In the degenerate case of a constant relation $(\lambda\ \_\ \_ \to X)$: Rel $\top$ whose indices provide no extra information, we recover the usual definition of lists of elements of type X:

```
List : Set → Set
List X = Star (λ _ _ → X) tt tt
```

Here tt is the unique constructor of the unit type $\top$, with ε and $\_\triangleleft\_$ taking the rôles

of *nil* and *cons* respectively. For example, we could write the two-element list of 0 and 1 as follows:

```
two : List ℕ
two = zero ◁ suc zero ◁ ε
```

As it is a generalisation of lists, Star admits many of the usual list-like functions. For example, while the _◁◁_ function below provides a proof of transitivity for Star R, it could equally be considered the generalisation of *append* on lists, as suggested by the structure of its definition:

```
_◁◁_ : {I : Set} {R : Rel I} {i j k : I} →
       Star R i j → Star R j k → Star R i k
ε       ◁◁ ys = ys
(x ◁ xs) ◁◁ ys = x ◁ (xs ◁◁ ys)
```

Similarly, we can define an analogue of *map* on lists:

```
gmap : {I I′ : Set} {R : Rel I} {S : Rel I′} (f : I → I′) →
       ({x y : I} →     R x y →     S (f x) (f y)) →
       {i j : I} → Star R i j → Star S (f i) (f j)
gmap f g ε       = ε
gmap f g (x ◁ xs) = g x ◁ gmap f g xs
```

As well as a function g that is mapped over the individual elements of the sequence, gmap also takes a function f that allows for the indices to change.

To conclude, let us consider a simple use case for Star. Take the *predecessor* relation on natural numbers, defined as _<₁_ below:

```
data _<₁_ : Rel ℕ where
  lt₁ : (n : ℕ) → n <₁ suc n
```

Here, $\mathsf{lt}_1\ n$ witnesses n as the predecessor of $\mathsf{suc}\ n$. We can then conveniently define the reflexive transitive closure of $\_<_1\_$ as follows:

```
_≤_ : Rel ℕ
_≤_ = Star _<₁_
```

Of course, this is just the familiar *less than or equal* relation, for which we can easily prove some familiar properties, such as the following:

```
0≤n : (n : ℕ) → zero ≤ n
0≤n zero     = ε
0≤n (suc n) = 0≤n n ◅◅ (lt₁ n ◅ ε)
```

Note that a proof of $\mathsf{zero} \leq n$ as produced by the above $\mathsf{0{\leq}n}$ function actually consists of a *sequence* of proofs in the style of "*0 is succeeded by 1, which is succeeded by 2, ... which is succeeded by n*" that can be taken apart and inspected one by one. We will be doing exactly this when considering reduction sequences in our later proofs.

6.2 Agda for Compiler Correctness

In this section, we shall revisit the language of numbers and addition from chapter 3, and demonstrate how the previous compiler correctness result can be formalised using Agda.

6.2.1 Syntax and Semantics

As in chapter 5, we can encode the syntax of our language as a simple algebraic data type:

```
data Expression : Set where
  #_ : (m : ℕ) → Expression
  _⊕_ : (a b : Expression) → Expression
```

The two constructors #_ and _⊕_ correspond to the (Exp-ℕ) and (Exp-⊕) rules in our original definition of the Expression language in chapter 3. Its denotational semantics—being a mapping from the syntax to the underlying domain of ℕ—can be simply implemented as a function:

```
⟦_⟧ : Expression → ℕ
⟦ # m ⟧     = m
⟦ a ⊕ b ⟧ = ⟦ a ⟧ + ⟦ b ⟧
```

The two cases in the definition of ⟦_⟧ correspond to (denote-val) and (denote-plus) respectively. A numeric expression is interpreted as just its value, while the ⊕ operator translates to the familiar + on natural numbers.

We had previously modelled the big-step semantics of our language as a binary relation ⇓ between expressions and numbers. In Agda, such a relation can be implemented as a dependent data type, indexed on Expression and ℕ:

```
data _⇓_ : REL Expression ℕ where
  ⇓-ℕ : ∀ {m} → # m ⇓ m
  ⇓-⊕ : ∀ {a b m n} → a ⇓ m → b ⇓ n → a ⊕ b ⇓ m + n
```

The ⇓-ℕ constructor corresponds to the (big-val) rule: an expression # m evaluates to just the value m. The ⇓-⊕ constructor takes two arguments of type $a \Downarrow m$ and $b \Downarrow n$, corresponding to the two premises of (big-plus).

The small-step semantics for Expressions is implemented in the same fashion,

```
data _↦_ : Rel Expression where
  ↦-ℕ : ∀ {m n}   →          # m ⊕ # n ↦ # (m + n)
  ↦-L : ∀ {a a' b} → a ↦ a' →   a ⊕ b   ↦   a' ⊕ b
  ↦-R : ∀ {m b b'} → b ↦ b' → # m ⊕ b   ↦ # m ⊕ b'
```

with (small-val), (small-left) and (small-right) represented as the ↦-ℕ, ↦-L and ↦-R

constructors. Using Star defined earlier in this chapter, we obtain the reflexive transitive closure of _↦_, along with proofs of the usual properties, for free:

```
_↦⋆_ : Rel Expression
_↦⋆_ = Star _↦_
```

6.2.2 Semantic Equivalence

Let us move swiftly on to the semantic equivalence theorems of section 3.2.2. Our previous technique of rule induction essentially boils down to induction on the structure of inductively-defined data types.

The proof for the forward direction of theorem 3.1—which captures the equivalence of the denotational and big-step semantics—is implemented as denote→big, proceeding by case analysis on its argument e : Expression:

```
denote→big : ∀ {e m} → ⟦ e ⟧ ≡ m → e ⇓ m
denote→big {# n}     ≡.refl = ⇓-ℕ
denote→big {a ⊕ b} ≡.refl = ⇓-⊕ (denote→big ≡.refl) (denote→big ≡.refl)
```

For the $e \equiv \# n$ case, matching the proof of $\llbracket \# n \rrbracket \equiv m$ with ≡.refl convinces the typechecker that m and n are equal, so $e \Downarrow n$ is witnessed by ⇓-ℕ. Agda considers terms equal up to β-reduction, so the $\llbracket a \oplus b \rrbracket \equiv m$ argument is equivalently a proof of $\llbracket a \rrbracket + \llbracket b \rrbracket \equiv m$, by the definition of $\llbracket\_\rrbracket$. The goal type of $a \oplus b \Downarrow \llbracket a \rrbracket + \llbracket b \rrbracket$ is therefore met with ⇓-⊕, whose premises of $a \Downarrow \llbracket a \rrbracket$ and $b \Downarrow \llbracket b \rrbracket$ are obtained by recursively invoking denote→big with two trivial witnesses to $\llbracket a \rrbracket \equiv \llbracket a \rrbracket$ and $\llbracket b \rrbracket \equiv \llbracket b \rrbracket$.

The backwards direction of theorem 3.1 is given by the following big→denote, which uses structural induction on the definition of _⇓_:

```
big→denote : ∀ {e m} → e ⇓ m → ⟦ e ⟧ ≡ m
big→denote ⇓-ℕ = ≡.refl
```

```
big→denote (⇓-⊕ a⇓m b⇓n) with big→denote a⇓m | big→denote b⇓n
big→denote (⇓-⊕ a⇓m b⇓n) | ≡.refl | ≡.refl = ≡.refl
```

For the base case of ⇓-ℕ, it must follow that $e \equiv \# \; m$, so the goal type after β-reduction of $[\![\_]\!]$ becomes $m \equiv m$, which is satisfied trivially. The ⇓-⊕ inductive case brings with it two proofs of $a \Downarrow m$ and $b \Downarrow n$, which can be used to obtain proofs of $[\![\; a \;]\!] \equiv m$ and $[\![\; b \;]\!] \equiv n$ via the induction hypothesis. The goal of $[\![\; a \;]\!] + [\![\; b \;]\!] \equiv m + n$ after β-reduction is then trivially satisfied, thus completing the proof of theorem 3.1.

The **with** keyword provides an analogue of Haskell's **case** ... **of** construct that allows us to pattern match on intermediate results. Agda requires us to repeat the left hand side of the function definition when using a **with** -clause, since dependent pattern matching may affect the other arguments, as noted earlier in this chapter.

We go on to prove theorem 3.2—the equivalence between the big-step and small-step semantics—in a similar manner:

```
big→small : ∀ {e m} → e ⇓ m → e ↦⋆ # m
big→small ⇓-ℕ = ε
big→small (⇓-⊕ {a} {b} {m} {n} a⇓m b⇓n) =
  Star.gmap (λ a′ → a′ ⊕ b) ↦-L (big→small a⇓m) ◅◅
  Star.gmap (λ b′ → # m ⊕ b′) ↦-R (big→small b⇓n) ◅◅
  ↦-ℕ ◅ ε
```

The goal in the case of ⇓-ℕ is trivially satisfied by the empty reduction sequence, since $e \equiv \# \; m$. In the ⇓-⊕ case, recursively invoking big→small with $a \Downarrow m$ and $b \Downarrow n$ gives reduction sequences for $a \mapsto^\star \# \; m$ and $b \mapsto^\star \# \; n$ respectively. We can map over the former using ↦-L over the witnesses and $\lambda \; a' \to a' \oplus b$ over the indices to obtain the reduction sequence $a \oplus b \mapsto^\star \# \; m \oplus b$, and likewise for the second to obtain $\# \; m \oplus b \mapsto^\star \# \; m \oplus \# \; n$. By appending these two resulting sequences, followed by

a final application of the $\mapsto$-$\mathbb{N}$ rule—or equivalently invoking transitivity—we obtain the desired goal of $a \oplus b \mapsto^\star \# m + n$.

The proof for the reverse direction of 3.2 proceeds by 'folding'[4] lemma 3.3—implemented as the sound helper function below—over the $e \mapsto^\star \# m$ reduction sequence:

```
small→big : ∀ {e m} → e ↦⋆ # m → e ⇓ m
small→big ε                    = ⇓-ℕ
small→big (e↦e′ ◁ e′↦⋆m) = sound e↦e′ (small→big e′↦⋆m) where
  sound : ∀ {e e′ m} → e ↦ e′ → e′ ⇓ m → e ⇓ m
  sound ↦-ℕ          ⇓-ℕ              = ⇓-⊕ ⇓-ℕ ⇓-ℕ
  sound (↦-L a↦a′) (⇓-⊕ a′⇓m b⇓n)  = ⇓-⊕ (sound a↦a′ a′⇓m) b⇓n
  sound (↦-R b↦b′) (⇓-⊕ m⇓m b′⇓n)  = ⇓-⊕ m⇓m (sound b↦b′ b′⇓n)
```

The sound helper performs case analysis on the structure of the $e \mapsto e'$ small-step reduction. In the case of $\mapsto$-$\mathbb{N}$, it must be the case that $e \equiv \# m \oplus \# n$ and $e' \equiv \# m + n$ respectively, which in turn forces the $e' \Downarrow m$ argument to be $\Downarrow$-$\mathbb{N}$. The goal type of $\# m \oplus \# n \Downarrow m + n$ is accordingly witnessed by $\Downarrow$-$\oplus$ $\Downarrow$-$\mathbb{N}$ $\Downarrow$-$\mathbb{N}$.

For the two remaining $\mapsto$-L and $\mapsto$-R rules, it must necessarily be the case that $e' \equiv a' \oplus b$ or $e' \equiv \# m \oplus b'$ respectively. Thus the $e' \Downarrow m$ argument contains a witness of either $a' \Downarrow m$ or $b' \Downarrow n$, which we use to recursively obtain a proof of $a \Downarrow m$ or $b \Downarrow n$.

6.2.3 Stack Machine, Compiler, and Correctness

In section 3.3.1 we used a stack machine as our low-level semantics, with the machine modelled as a pair of a list of instructions and a stack of values. This translates to the following Agda data declaration:

[4] Unfortunately we cannot use Star.gfold from Agda's standard library, as _⇓_ : REL Expression ℕ is not a homogeneous relation.

```
data Machine : Set where
  ⟨_,_⟩ : (c : List Instruction) → (σ : List ℕ) → Machine
```

The List type from in Agda's standard library implements the familiar *nil* and *cons* lists as found in Haskell. In turn, the virtual machine's instruction set comprises the standard PUSH and ADD for stack machines:

```
data Instruction : Set where
  PUSH : ℕ → Instruction
  ADD  : Instruction
```

The two reduction rules (vm-push) (vm-add) for the virtual machine are realised as the ⟼-PUSH and ⟼-ADD constructors of the _⟼_ relation:

```
data _⟼_ : Rel Machine where
  ⟼-PUSH : ∀ {m   c σ} → ⟨ PUSH m :: c ,      σ ⟩ ⟼ ⟨ c , m       :: σ ⟩
  ⟼-ADD  : ∀ {m n c σ} → ⟨ ADD :: c , n :: m :: σ ⟩ ⟼ ⟨ c , m + n :: σ ⟩

_⟼*_ : Rel Machine
_⟼*_ = Star _⟼_
```

Again, we define _⟼*_ as Star _⟼_, and receive the usual properties of a reflexive transitive closure absolutely free.

The compiler is defined identically to that of section 3.3.2,

```
compile : Expression → List Instruction → List Instruction
compile (# m)   c = PUSH m :: c
compile (a ⊕ b) c = compile a (compile b (ADD :: c))
```

which compiles a number # m to PUSH m, and a sum $a \oplus b$ to the concatenation of code that computes the value of a and b, followed by an ADD instruction.

The following definition of _⟼*#_ provides a convenient synonym for what it means when we say that executing the compiled code for an expression e computes the result m:

```
_⟼*#_ : REL Expression ℕ
e ⟼*# m = ∀ {c σ} → ⟨ compile e c , σ ⟩ ⟼* ⟨ c , m :: σ ⟩
```

Note that the above proposition is quantified over any code continuation c and initial stack σ, and a proof of compiler correctness (theorem 3.4) amounts to functions in both directions between $e \Downarrow m$ and $e \rightarrowtail^{\star}\# m$. In the forward direction—that is, from the high-level/big-step semantics to the low-level/virtual machine semantics—this is implemented by induction on the structure of $e \Downarrow m$:

```
big→machine : ∀ {e m} → e ⇓ m → e ⟼*# m
big→machine ⇓-ℕ = ⟼-PUSH ◁ ε
big→machine (⇓-⊕ a⇓m b⇓n) =
  big→machine a⇓m ◁◁ big→machine b⇓n ◁◁ ⟼-ADD ◁ ε
```

In the case of ⇓-ℕ we have $e \equiv \#\ m$, and so ⟼-PUSH ◁ ε witnesses the reduction sequence ∀ {c σ} → ⟨ compile (# m) c , σ ⟩ ⟼* ⟨ c , m :: σ ⟩. In the second case, the recursive terms big→machine $a⇓m$ and big→machine $b⇓n$ are of the following types:

```
big→machine a⇓m : ∀ {c_a σ_a} → ⟨ compile a c_a , σ_a ⟩ ⟼* ⟨ c_a , m :: σ_a ⟩
big→machine b⇓n : ∀ {c_b σ_b} → ⟨ compile b c_b , σ_b ⟩ ⟼* ⟨ c_b , n :: σ_b ⟩
```

The right hand side requires a proof of:

```
∀ {c σ} → ⟨ compile (a ⊕ b) c , σ ⟩ ⟼* ⟨ c , m + n :: σ ⟩
```

which can be obtained by instantiating c_a = compile b (ADD :: c), c_b = ADD :: c, σ_a = σ, σ_b = m :: σ, and concatenating the resulting reduction sequences,

followed by a final application of the $\rightarrowtail$-ADD rule. As these values can be automatically inferred by Agda, we do not need to make them explicit in the definition of big→machine.

The backwards direction of the compiler correctness proof requires us to compute a witness of $e \Downarrow m$ from one of $e \rightarrowtail^\star \# m$. We first need to implement a pair of lemmas, however. The exec lemma simply states that that for any expression e, there exists a number m for which executing compile e computes m:

```
exec : ∀ e → ∃ λ m → e ⇀⋆# m
exec (# m) = m ∧ λ {c} {σ} → ⇀-PUSH ◁ ε
exec (a ⊕ b) with exec a | exec b
exec (a ⊕ b) | n_a ∧ a⇀⋆#n_a | n_b ∧ b⇀⋆#n_b = n_a + n_b ∧
  λ {c} {σ} → a⇀⋆#n_a ◁◁ b⇀⋆#n_b ◁◁ ⇀-ADD ◁ ε
```

When $e \equiv \# m$, this number clearly has to be m, with $\rightarrowtail$-PUSH $\triangleleft \varepsilon$ serving as the witness. Note that due to Agda's handling of implicit arguments, we must explicitly generalise over c and σ. In the $e \equiv a \oplus b$ case, we simply recurse on the a and b subexpressions and concatenate the resulting reduction sequences with an $\rightarrowtail$-ADD.

The unique lemma states that given two reduction sequences from the same initial state, the resulting stacks must coincide:

```
unique : ∀ {c σ σ′ σ″} → ⟨ c , σ ⟩ ⇀⋆ ⟨ [] , σ′ ⟩ →
                         ⟨ c , σ ⟩ ⇀⋆ ⟨ [] , σ″ ⟩ → σ′ ≡ σ″
unique {[]}      (() ◁ c′⇀⋆σ′) c⇀⋆σ″
unique {[]}      ε             (() ◁ c′⇀⋆σ″)
unique {[]}      ε             ε             = ≡.refl
```

We proceed by recursion on the code c: the first group of cases above deal with an empty c. Both sequences must be ε, since there no reduction is possible from the machine state $\langle\ [] , \sigma\ \rangle$. In Agda, we identify such cases with the '*impossible*'

pattern—written as ()—and accordingly the first two cases do not have a corresponding right hand side. In the third case, the proof of $\sigma' \equiv \sigma''$ is trivial, since matching on the two ε constructors for reflexivity has already unified the σ, σ' and σ'' variables.

The next two cases deal with the PUSH and ADD instructions. In the first instance, we obtain the proof of $\sigma' \equiv \sigma''$ by recursion on $c' \rightarrowtail^\star \sigma'$ and $c' \rightarrowtail^\star \sigma''$, both starting from the machine state $\langle\ c'\ ,\ m :: \sigma\ \rangle$:

```
unique {PUSH m :: c'}          (⟶-PUSH ◁ c'⟶⋆σ')
                               (⟶-PUSH ◁ c'⟶⋆σ'') = unique c'⟶⋆σ' c'⟶⋆σ''
unique {ADD :: c'} {      []} (()    ◁ c'⟶⋆σ') c⟶⋆σ''
unique {ADD :: c'} {  m :: []} (()    ◁ c'⟶⋆σ') c'⟶⋆σ''
unique {ADD :: c'} {n :: m :: σ} (⟶-ADD ◁ c'⟶⋆σ')
                               (⟶-ADD ◁ c'⟶⋆σ'') = unique c'⟶⋆σ' c'⟶⋆σ''
```

The reduction rule for ADD requires at least two numbers on top of the stack, which is ruled out by the first two cases in the latter group. The final case proceeds by recursion on the remainder of the reduction sequence, both of which start from the same $\langle\ c'\ ,\ m\ +\ n :: \sigma\ \rangle$ machine state.

Returning to the second half of our compiler correctness theorem, it remains for us to show that $e \rightarrowtail^\star\# m$ implies $e \Downarrow m$. The following definition of machine→big provides the proof, which proceeds by case analysis on the expression e:

```
machine→big : ∀ {e m} → e ⟶⋆# m → e ⇓ m
machine→big {# n} n⟶⋆#m with n⟶⋆#m {[]} {[]}
machine→big {# n} n⟶⋆#m | ⟶-PUSH ◁ ε = ⇓-ℕ
machine→big {# n} n⟶⋆#m | ⟶-PUSH ◁ () ◁ n'⟶⋆#m
```

The $n \rightarrowtail^\star\#m$ argument is in fact a function that takes two implicit arguments, having

the type:

$$\forall \{c\ \sigma\} \to \langle\ \mathsf{compile}\ (\#\ n)\ c\ ,\ \sigma\ \rangle \rightarrowtail^{\star} \langle\ c\ ,\ m :: \sigma\ \rangle$$

To this we apply an empty code continuation and stack, then pattern match the resulting reduction sequence against $\rightarrowtail$-PUSH $\lhd$ ε. This unifies n and the implicit argument m, allowing $\Downarrow$-$\mathbb{N}$ to witness the goal type of $\#\ m \Downarrow m$. The second case handles the case of longer reduction sequences, which is impossible, since compile $(\#\ n)$ outputs only a single PUSH instruction.

For expressions of the form $e \equiv a \oplus b$, we first use the exec helper to obtain the two reduction sequences $a \rightarrowtail^{\star}\# n_a$ and $b \rightarrowtail^{\star}\# n_b$, making use of a **with** clause:

```
machine→big {a ⊕ b} a⊕b⟶⋆#m with exec a       | exec b
machine→big {a ⊕ b} a⊕b⟶⋆#m | n_a ∧ a⟶⋆#n_a | n_b ∧ b⟶⋆#n_b
  with unique {σ = []} a⊕b⟶⋆#m (a⟶⋆#n_a ◅◅ b⟶⋆#n_b ◅◅ ⟶-ADD ◅ ε)
machine→big {a ⊕ b} a⊕b⟶⋆#m | n_a ∧ a⟶⋆#n_a | n_b ∧ b⟶⋆#n_b
  | ≡.refl = ⇓-⊕ (machine→big a⟶⋆#n_a) (machine→big b⟶⋆#n_b)
```

The concatenation $a{\rightarrowtail^{\star}\#}n_a \mathbin{◅◅} b{\rightarrowtail^{\star}\#}n_b \mathbin{◅◅} \rightarrowtail$-ADD $\lhd$ ε produces a reduction sequence of $a \oplus b \rightarrowtail^{\star}\# n_a + n_b$. Using the previously defined unique lemma, we may match each step of the sequence with those of $a{\oplus}b{\rightarrowtail^{\star}\#}m$ to yield a proof of $n_a + n_b \equiv m$. Examining this proof using a second **with** clause, the goal becomes $a \oplus b \Downarrow n_a + n_b$. This calls for an instance of $\Downarrow$-$\oplus$, whose two premises of $a \Downarrow n_a$ and $b \Downarrow n_b$ are obtained by recursion.

6.3 Conclusion

In this chapter we have given a tutorial of some basic Agda suitable for our needs, and have produced a fully machine-verified version of the results of chapter 3 as an

example. Finally we concluded with a brief to using coinductive data types in Agda.

On reflection, formalising such results in Agda has provided a number of advantages: firstly, it clarifies thinking by forcing us to be very explicit with regards to numerous low-level details that are often elided in pen-and-paper proofs. Testing out new hypotheses also becomes easier, using the type checker to guide proof development; in case of any invalid assumptions, Agda helps to pinpoint precisely where these occur.

A final benefit of using a mechanised proof system such as Agda is that it aids the incremental evolution of this work: often we would prove a simplified form of a theorem to gain some insight. At this point, we may then change, extend, or generalise some core definitions towards our target. Strictly-speaking, all the proofs leading up to the final result would need to be proved again, but in the majority of cases only minor tweaks to the simplified proofs are required. Agda helps out by pointing out where these changes need to be made. This process would not be as straightforward using a script-based theorem prover, or even possible in a pen-and-paper-based approach.

Chapter 7

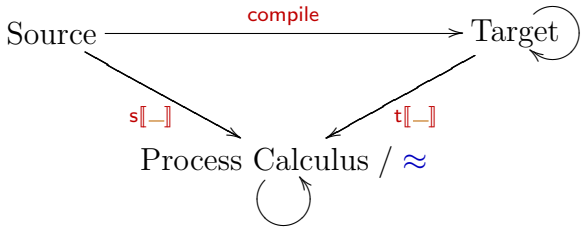
Compiling Non-Determinism Correctly

The standard approach [Wan95] to proving compiler correctness for concurrent languages requires the use of multiple translations into an underlying process calculus. In this chapter, we present a simpler approach that avoids the need for such an underlying language, using a new method that allows us to directly establish a bisimulation between the source and target languages. We illustrate this technique on a small non-deterministic language, using the Agda system to present and formally verify our compiler correctness proofs.

7.1 Existing Approach

The standard approach [Wan95] to proving compiler correctness for concurrent languages requires the use of multiple translations into an intermediate process calculus.

This methodology is captured by the following diagram:



That is, for some compiler from a source language to a target language, we define separate denotational semantics $\mathsf{s}[\![\_]\!]$ and $\mathsf{t}[\![\_]\!]$ for both languages in terms of an underlying process calculus with a suitable notion of bisimulation, or observational equivalence. The compiler is said to be correct when $\mathsf{s}[\![p]\!]$ and $\mathsf{t}[\![\mathsf{compile}\ p]\!]$ are bisimilar for all programs p. The advantage of using a traditional process calculus is that we may reuse existing theories and techniques, and perform our reasoning in a single, homogeneous framework.

However, there are two drawbacks to this method: firstly, the source languages is defined by a map $\mathsf{s}[\![\_]\!]$ into an underlying process calculus, which adds an extra level of indirection when reasoning about the operational behaviour of source programs. Secondly, the target language and process calculus are given separate *operational* semantics — represented by the two circular arrows in the above diagram — with a semantic function $\mathsf{t}[\![\_]\!]$ mapping the former to the latter. Thus we need to further show that the operational semantics of the target language is *adequate* with respect to that of the process calculus via the translation $\mathsf{t}[\![\_]\!]$.

7.2 Related Work

The formal notion of *compiler correctness* dates back to 1967, when McCarthy and Painter [MP67] proved the correctness of a compiler from arithmetic expressions to a register machine. Their stated goal was "*to make it possible to use a computer to*

check proofs that compilers are correct"; we aim to do precisely this for a concurrent language.

In the four decades since, a large body of pen-and-paper correctness proofs have been produced for various experimental languages. (See [Dav03] for a detailed bibliography.) However it is only recently—by making essential use of a formal theorem prover—that it has become possible to fully verify a realistic compiler. In particular, Leroy [Ler06] has produced a certified compiler for a C-like core language in the Coq [The08] framework, relating a series of intermediate languages, eventually targeting PowerPC assembly code.

While compiler correctness for sequential languages has been well explored by many researchers, the issue of concurrency has received relatively little attention, with most of the progress being made by Wand and his collaborators. In the early 1980s, Wand [WS95] initially suggested a methodology for sequential languages: by giving the *denotational* semantics of both source and target languages in a common domain, the correctness proofs relating the *operational* behaviour of the source and target languages may then be carried out within the same domain. Wand then adapted this paradigm to the concurrent setting [Wan95] as summarised in our introduction, which is further elaborated by Gladstein [GW96, Gla94].

Our work in this chapter follows on from Hutton and Wright [HW07], who recently considered the issue of compiler correctness for a simple non-deterministic language, by relating a denotational source semantics to an operational target semantics, based on the extensional notion of comparing final results. As noted in [HW07], the addition of effects and concurrency requires an intensional notion of comparing intermediate actions via a suitable notion of bisimulation. The purpose of this chapter is to explore this idea, while retaining the approach of directly relating the source and target without the need for an intermediate language.

7.3 A Non-Deterministic Language

In order to focus on the essence of this problem, we abstract from the details of a real language and keep to a simple expression language consisting of integers and addition [HW04, HW06, HW07] as we had done in previous chapters. We give its operational semantics using a labelled transition system, together with an extra '*zap*' rule to introduce a form of non-determinism. A virtual machine, and a compiler linking the two completes the definition. We present and justify a novel technique for proving compiler correctness in the presence of non-determinism.

We begin by recalling the syntax of our expression language—comprising natural numbers and addition—from the previous chapter:

```
data Expression : Set where
  #_   : (m : ℕ)          → Expression
  _⊕_  : (a b : Expression) → Expression
```

As we had previously noted in §3.1.5, the monoid $(\mathbb{N}, 0, \oplus)$ may be seen as a degenerate monad, allowing us to avoid orthogonal issues such as binding and substitution. The sequencing aspect of monads is retained by giving our language a left-to-right evaluation order.

Rather than defining a reduction relation between pairs of states as in chapter 6, we generalise to a labelled transition system. For this we define a shorthand LTS, parametrised on the type of labels and the underlying state:

```
LTS : Set → Set → Set₁
LTS L X = X → L → X → Set
```

We use Labels to indicate the nature of the transition:

```
data Action : Set where
  ⊞  : Action
```

```
  ↯   : Action
  □_  : (m : ℕ) → Action
data Label : Set where
  τ   : Label
  !_  : (α : Action) → Label
```

Each transition either emits (denoted by '!') one of the ⊞, ↯ or □ actions (pronounced 'add', 'zap' and 'result' respectively), or has the silent label τ. We make a two-level distinction between Labels and Actions in this language so that potentially silent and non-silent transitions may be identified by their types in the Agda proofs.

Transitions on expressions are defined as the following data type:

```
data _↦<_>_ : LTS Label Expression where
  ↦-⊞ : ∀ {m n} → # m ⊕ # n ↦< ! ⊞ > # m + n
  ↦-↯ : ∀ {m n} → # m ⊕ # n ↦< ! ↯ > # 0
  ↦-R : ∀ {m b b′ Λ} (b↦b′ : b ↦< Λ > b′) →
                     # m ⊕ b  ↦< Λ >  # m ⊕ b′
  ↦-L : ∀ {a a′ b Λ} (a↦a′ : a ↦< Λ > a′) →
                     a  ⊕ b  ↦< Λ >  a′  ⊕ b
```

By convention, we will use the letters m and n for natural numbers, a, b and e for expressions, α for actions and Λ for labels. Using Agda's facility for mixfix operators, the proposition that e reduces in a single Λ-labelled step to e' is written quite naturally as follows: $e \mapsto\!< \Lambda >\! e'$.

Each constructor of the above definition corresponds to a transition rule. Let us consider the two base rules, covering expression of the form $\#\ m \oplus \#\ n$. When reducing $\#\ m \oplus \#\ n$, one of two things can happen: either the two numbers are summed as usual by the ↦-⊞ rule, or they are 'zapped' to zero by ↦-↯; the two possible transitions are labelled accordingly. The inclusion of the ↦-↯ rule introduces

a simple form of non-determinism, as a first step towards moving from a sequential, deterministic language, to a concurrent, non-deterministic one. The two remaining rules $\mapsto$-R and $\mapsto$-L ensure a left-biased reduction order, as mentioned previously.

7.3.1 Choice of Action Set

How was the set of actions chosen? As we shall see later in §7.5, we wish to distinguish between different choices in the reduction path a given expression can take. In the instance of this Zap language, we need to know which of the $\mapsto$-$\boxplus$ or $\mapsto$-$\lightning$ rules were applied, hence the use of the distinct actions $\boxplus$ and $\lightning$ respectively. Where there is only one unique transition from some given state, we label it with a silent τ. Later in §7.6 we also wish to distinguish between different final results for an expression, which are revealed using the $\square$ action.

7.4 Compiler, Virtual Machine and its Semantics

The virtual machine for this language has a simple stack-based design, with the same two instructions as we had in the previous chapter:

```
data Instruction : Set where
  PUSH : (m : ℕ) → Instruction
  ADD : Instruction
```

A program comprises a list of such instructions. The compiler for our expression language is the same as the previous chapter, and is repeated below. In order to make our proofs more straightforward, we take a code continuation as an additional argument [Hut07], which corresponds to writing the compiler in an accumulator-passing style:

```
compile : Expression → List Instruction → List Instruction
compile (# m) c = PUSH m :: c
compile (a ⊕ b) c = compile a (compile b (ADD :: c))
```

To execute a program c : List Instruction, we pair it with a stack σ : List $\mathbb{N}$. This is precisely how we represent the state of a virtual machine:

```
data Machine : Set where
  ⟨_,_⟩ : (c : List Instruction) (σ : List ℕ) → Machine
```

Finally, we specify the operational semantics of the virtual machine through the _⟼<_>_ relation:

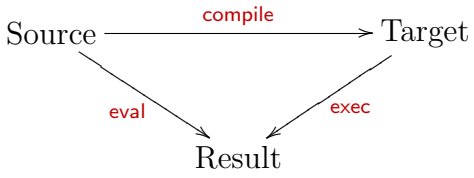
```
data _⟼<_>_ : LTS Label Machine where
  ⟼-PUSH : ∀ {c σ m} → ⟨ PUSH m :: c , σ ⟩ ⟼< τ > ⟨ c , m :: σ ⟩
  ⟼-ADD : ∀ {c σ m n} →
    ⟨ ADD :: c , n :: m :: σ ⟩ ⟼< ! ⊞ > ⟨ c , m + n :: σ ⟩
  ⟼-ZAP : ∀ {c σ m n} →
    ⟨ ADD :: c , n :: m :: σ ⟩ ⟼< ! ↯ > ⟨ c , 0 :: σ ⟩
```

That is, the PUSH instruction takes a numeric argument m and pushes it onto the stack σ, with a silent label τ. In turn, the ADD instruction replaces the top two numbers on the stack with either their sum, or zero—labelled respectively with $\boxplus$ or $\lightning$—in correspondence with the ⟼-ADD and ⟼-ZAP rules.

7.5 Non-Deterministic Compiler Correctness

In general, a compiler correctness theorem asserts that for any source program, the result of executing the corresponding compiled target code on its virtual machine will

coincide with that of evaluating the source using its high-level semantics:



With a deterministic language and virtual machine—such as our Zap language without the two 'zap' rules—it would be natural to use a high-level denotational or big-step semantics for the expression language, which we can realise as an interpreter

$$\mathsf{eval} \;:\; \mathsf{Expression} \to \mathbb{N}$$

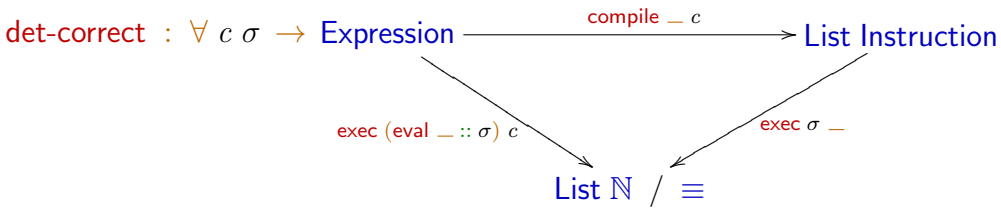
In turn, the low-level operational or small-step semantics for the virtual machine can be realised as a function

$$\mathsf{exec} \;:\; \mathsf{List}\;\mathbb{N} \to \mathsf{List\;Instruction} \to \mathsf{List}\;\mathbb{N}$$

that takes an initial stack along with a list of instructions and returns the final stack. Compiler correctness is then the statement that:

$$\mathsf{det\text{-}correct} \;:\; \forall\; c\;\sigma\;e \to \mathsf{exec}\;\sigma\;(\mathsf{compile}\;e\;c) \equiv \mathsf{exec}\;(\mathsf{eval}\;e :: \sigma)\;c$$

Equivalently, we can visualise this as the following commuting diagram:



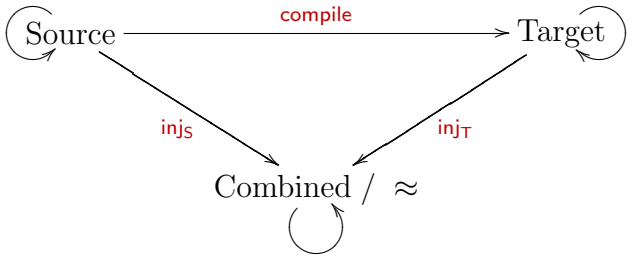
That is to say, compiling an expression a and then executing the resulting code together with a code continuation c gives the same result—up to definitional equality—as executing c with the value of a pushed onto the original stack σ.

The presence of non-determinism requires a more refined approach, due to the possibility that different runs of the same program may give different results. One approach is to realise the interpreter and virtual machine as set-valued functions [HW07], restating the above equality on final values in terms of *sets* of final values. A more natural approach however, is to define the high-level semantics as a relation rather than a function, using a small-step operational semantics. Moreover, the small-step approach also allows us to consider the *intensional* (or local) notion of what choices are made in the reduction paths, in contrast to the *extensional* (or global) notion of comparing final results. In our Zap language, the available choices are reflected in our selection of transition labels, and we weaken the above definitional equality to a suitable notion of branching equivalence on intermediate states. This is just the familiar notion of bisimilarity [Mil89], which we shall make concrete in §7.7. As we shall see, the local reasoning afforded by this approach also leads to simpler and more natural proofs.

7.6 Combined Machine and its Semantics

In this section, we introduce our key idea of a 'combined machine', which we arrive at by considering the small-step analogue of the compiler correctness statement for big-step deterministic languages. The advantage of the combined machine is that it lifts source expressions and target virtual machine states into the same domain, which avoids the need for an intermediate process calculus [Wan95] and allows us to directly establish a bisimulation between the source and target languages.

Our approach to non-deterministic compiler correctness is illustrated below,



making use of such a combined machine, where the lifting by $\mathsf{inj_S}$ and $\mathsf{inj_T}$ are trivial enough to be 'obviously correct'. Rather than considering final results, we consider combined machines up to a suitable notion of bisimilarity.

In the case of our Zap language, a combined machine x : Combined has three distinct phases of execution,

```
data Combined : Set where
  ⟨_,_⟩ : (e : Expression) (t : Machine) → Combined
  ⟨_⟩ : (t : Machine) → Combined
  ⟨⟩ : Combined
```

whose semantics is defined by the following transition relation:

```
data _⇛<_>_ : LTS Label Combined where
  ⇛-↦ : ∀ {e e′ t Λ} (e↦e′ : e ↦< Λ > e′) → ⟨ e , t ⟩ ⇛< Λ > ⟨ e′ , t ⟩
  ⇛-↣ : ∀ {t t′ Λ} (t↣t′ : t ↣< Λ > t′) → ⟨ t ⟩ ⇛< Λ > ⟨ t′ ⟩
  ⇛-switch : ∀ {m c σ} → ⟨ # m , ⟨ c , σ ⟩ ⟩ ⇛< τ > ⟨ ⟨ c , m :: σ ⟩ ⟩
  ⇛-done : ∀ {m} → ⟨ ⟨ [] , m :: [] ⟩ ⟩ ⇛< ! □ m > ⟨⟩
```

The first constructor $\langle \_,\_ \rangle$ of Combined pairs an expression with a virtual machine continuation. In this initial phase, a combined machine $\langle\ e\ ,\ \langle\ c\ ,\ \sigma\ \rangle\ \rangle$ can be understood as the small-step analogue of the right side of the det-correct statement—exec (eval a :: σ) c—which begins by effecting the reduction of a. The applicable

reductions are exactly those of the Expression language, inherited via the →»-↦ rule above.

When the expression a eventually reduces to a value m, the →»-switch rule pushes m onto the stack σ, switching the combined machine to its second phase of execution, corresponding to the $\langle \_ \rangle$ constructor. This can be thought of as the small-step analogue of pushing the result $m \equiv \mathsf{eval}\ a$ onto the stack, again following the right side of det-correct, namely $\mathsf{exec}\ (m :: \sigma)\ c$.

The second Combined constructor $\langle \_ \rangle$ lifts a virtual machine into a combined machine, which then effects the reduction of the former via the →»-↣ rule. This corresponds to the small-step analogue of $\mathsf{exec}\ \sigma\ c$, which matches the left side of det-correct, and also the right side after the evaluation of the embedded expression has completed.

Lastly, the →»-done rule embeds the computed result in a □ action, and terminates with the empty $\langle\rangle$ state. This construction allows us to compare final result values using the existing bisimulation machinery.

7.7 Weak Bisimilarity

Now we can give a concrete definition to our notion of bisimilarity. More specifically, we shall be defining 'weak bisimilarity', as we are not concerned with silent transitions. First of all, it is convenient to define a 'visible transition' _⤇<_>_ where only Actions are exposed, in terms of the _→»<_>_ relation from the previous section,

```
data _⤇<_>_ : LTS Action Combined where
  ⤇-→» : ∀ {x x₀ x₁ x′ α} (x→»τ⋆x₀ : x →»<τ>⋆ x₀)
    (x₀→»x₁ : x₀ →»< ! α > x₁) (x₁→»τ⋆x′ : x₁ →»<τ>⋆ x′) →
    x ⤇< α > x′
```

where we write _↠<τ>*_ as a shorthand for the reflexive transitive closure of ↠< τ >, defined as follows:

```
_↠<τ>_ : Rel Combined
x ↠<τ> y = x ↠< τ > y

_↠<τ>*_ : Rel Combined
_↠<τ>*_ = Star _↠<τ>_
```

Two states x and y are now defined to be 'weakly bisimilar' if and only if whatever visible transition x can make, y is able to follow with the same action, resulting in states x' and y' that are also bisimilar, and vice versa. Since this is symmetric in both directions, we will define a helper _≼_ for ease of readability:

```
_≼_ : Rel Combined
x ≼ y = ∀ x' {α} → x ↦< α > x' → ∃ λ y' → y ↦< α > y' × x' ≈' y'
```

That is, we write $x \preccurlyeq y$ iff for whatever state x' can be reached from x while emitting the action α, y can reach a corresponding state y' such that x' and y' are bisimilar. In turn, a relation $x \approx y$ is simply a conjunction of $x \preccurlyeq y$ and $y \preccurlyeq x$:

```
data _≈_ : Rel Combined where
  _&_ : ∀ {x y} (x≼y : ∞ (x ≼ y)) (y≼x : ∞ (y ≼ x)) → x ≈ y
```

The types of both $x \preccurlyeq y$ and $y \preccurlyeq x$ arguments are coinductive, as indicated by the use of the ∞ type, since bisimilarity is a coinductive notion [Mil89, San09].

Note that in the above definition of _≼_, we demand a proof of $x' \approx' y'$ rather than a direct proof of $x' \approx y'$, in order to 'beat Agda's productivity checker' [Dan10a]. The _≈'_ data type can be seen as the syntax for an embedded language, where each constructor corresponds to an operation that we wish to perform on bisimilarity proofs. In this instance we only require symmetry and transitivity, along with a constructor ≈'-≈ that embeds _≈_ proofs:

```
data _≈′_ : Rel Combined where
  ≈′-≈ : _≈_ ⇒ _≈′_
  ≈′-sym : Symmetric _≈′_
  ≈′-trans : Transitive _≈′_
```

We write $\_\Rightarrow\_$ as a synomym for relational implication, which is—along with Symmetric and Transitive—defined in the standard library in the obvious manner. With this technique, we are obliged to show that $\_\approx'\_$ implies $\_\approx\_$, i.e. provide an interpretor from this embedded language to an actual $\_\approx\_$ proof. However, as far as comprehension is concerned, the reader can simply treat the use of $\_\approx'\_$ in the definition of $\_\preccurlyeq\_$ as if it were $\_\approx\_$.

Given the above, it is straightforward to prove that $\_\approx\_$ is an equivalence relation on Combined. Reflexivity is shown by the following two mutual definitions:

```
mutual
  ≼-refl : Reflexive _≼_
  ≼-refl x′ x↦x′ = x′ ∧ x↦x′ ∧ ≈′-≈ ≈-refl
  ≈-refl : Reflexive _≈_
  ≈-refl = ♯ ≼-refl & ♯ ≼-refl
```

The type of ≼-refl is synonymous to $\forall\ \{x\} \to x \preccurlyeq x$, which in turn expands by the definition of $\_\preccurlyeq\_$ to the following:

$$\forall\ \{x\}\ x'\ \{\alpha\} \to x \mapsto\!\!< \alpha >\!\!\!\!\rightarrow x' \to \exists\ \lambda\ x'' \to x \mapsto\!\!< \alpha >\!\!\!\!\rightarrow x'' \times x' \approx' x''$$

We prove the existence of an x'' such that $x \mapsto\!\!< \alpha >\!\!\!\!\rightarrow x''$ and $x' \approx' x''$ by returning the same x' and the witness of $x \mapsto\!\!< \alpha >\!\!\!\!\rightarrow x'$ as we were given. Proof of $x' \approx' x'$ is obtained by embedding the result of a corecursive call to $\approx$-refl $: x' \approx x'$.

The proof of Reflexive $\_\approx\_$ involves two symmetric invocations to ≼-refl. Since both corecursive instances of ≈-refl are guarded by an unbroken chain of $\_\&\_$, $\sharp\_$,

_∧_ and ≈′-≈ constructors with no intervening function invocations, Agda accepts the above definition as productive.

Symmetry on the other hand is much more straightforward,

```
≈-sym : Symmetric _≈_
≈-sym (x≼y & y≼x) = y≼x & x≼y
```

as we need only swap the two halves of the bisimilarity proof. Finally to show transitivity of _≈_, we again make use of the symmetric nature of bisimilarity, with the help of a mutually corecursive definition ≼-trans:

```
mutual
  ≼-trans : Transitive _≼_
  ≼-trans x≼y y≼z x′ x⇔x′ with x≼y x′ x⇔x′
  ≼-trans x≼y y≼z x′ x⇔x′ | y′ ∧ y⇔y′ ∧ x′≈′y′ with y≼z y′ y⇔y′
  ≼-trans x≼y y≼z x′ x⇔x′ | y′ ∧ y⇔y′ ∧ x′≈′y′ | z′ ∧ z⇔z′ ∧ y′≈′z′
    = z′ ∧ z⇔z′ ∧ ≈′-trans x′≈′y′ y′≈′z′

  ≈-trans : Transitive _≈_
  ≈-trans (x≼y & y≼x) (y≼z & z≼y)
    = ♯ ≼-trans (♭ x≼y) (♭ y≼z) & ♯ ≼-trans (♭ z≼y) (♭ y≼x)
```

The ≼-trans proof—given $x \preccurlyeq y$, $y \preccurlyeq z$, x' and $x \Leftrightarrow x'$—has a goal of:

```
∃ λ z′ → z ⇔< α > z′ × x′ ≈′ z′
```

We can use $x \preccurlyeq y$ and $y \preccurlyeq z$ in two steps to construct evidence of y' and z' such that,

```
y ⇔< α > y′ × x′ ≈′ y′    and    z ⇔< α > z′ × y′ ≈′ z′
```

with the witness to $x' \approx' z'$ obtained by ≈′-trans. The proof for Transitive _≈_ proceeds in the same manner as ≈-refl, with two corecursive calls to ≼-trans for each half of the property.

Earlier we stated that we are obliged to show that $\_\approx'\_$ implies $\_\approx\_$. Having now implemented all of the functions corresponding to the terms of the embedded $\_\approx'\_$ language, we can proceed quite simply as follows:

```
≈′→≈ : _≈′_ ⇒ _≈_
≈′→≈ (≈′-≈ x≈y) = x≈y
≈′→≈ (≈′-sym y≈′x) = ≈-sym (≈′→≈ y≈′x)
≈′→≈ (≈′-trans x≈′z z≈′y) = ≈-trans (≈′→≈ x≈′z) (≈′→≈ z≈′y)
```

Here it is clear that $\approx'\!\to\!\approx$ is structurally recursive on its $x \approx' y$ argument, and therefore must be total.

We conclude this section by noting that $\_\approx\_$ is an equivalence relation on Combined machines, which enables us to use the equational (actually pre-order) reasoning combinators from the standard library module Relation.Binary.PreorderReasoning. The plumbing details have been omitted in this presentation for brevity, however.

7.8 The elide-τ Lemma

A key lemma used throughout our correctness proofs states that a silent transition between two states x and y implies that they are bisimilar:

```
elide-τ : _-↠<τ>_ ⇒ _≈_
elide-τ {x} {y} x-↠τy = ♯ x≼y & ♯ y≼x where
  y≼x : y ≼ x
  y≼x y′ (⊩-↠ y-↠τ*y₀ y₀-↠y₁ y₁-↠τ*y′)
    = y′ ∧ ⊩-↠ (x-↠τy ◁ y-↠τ*y₀) y₀-↠y₁ y₁-↠τ*y′ ∧ ≈′-refl
```

In the y≼x direction, the proof is trivial: whatever y does, x can always match it by first making the given $x\!-\!\!\twoheadrightarrow\!\tau y$ transition, after which it can follow y exactly.

In the x≼y direction, the proof relies on the fact that wherever there is a choice in the reduction of any give state, each possible transition is identified with a distinct and non-silent label, which we mentioned in §7.3.1. Conversely given a silent transition $x \twoheadrightarrow \tau y \ : \ x \ \twoheadrightarrow\!\langle \tau \rangle \ y$, it must in fact be the *unique* transition from x, which we can show by the following unique lemma:

```
unique : ∀ {x y Λ y′} → x ↠<τ> y → x ↠< Λ > y′ → Λ ≡ τ × y′ ≡ y
unique (↠-↦ e↦e′) x↠y′ = ⊥-elim (¬↦<τ> e↦e′)
unique (↠-↣ ↣-PUSH) (↠-↣ ↣-PUSH) = ≡.refl ∧ ≡.refl
unique ↠-switch (↠-↦ ())
unique ↠-switch ↠-switch = ≡.refl ∧ ≡.refl
```

Using unique, the x≼y direction of elide-τ is shown in two parts; the first shows that x cannot make a non-silent transition,

```
x≼y : x ≼ y
x≼y x′ (↦-↠ ε x↠x₀ x₀↠τ⋆x′) with unique x↠τy x↠x₀
x≼y x′ (↦-↠ ε x↠x₀ x₀↠τ⋆x′) | () ∧ x₀≡y
```

while the second shows that the first silent transition x makes must coincide with $x \twoheadrightarrow \tau y$, in which case y can transition to x' by following the subsequent transitions:

```
x≼y x′ (↦-↠ (x↠x₀ ◁ x₀↠τ⋆x₁) x₁↠x₂ x₂↠τ⋆x′) with unique x↠τy x↠x₀
x≼y x′ (↦-↠ (x↠τy ◁ y↠τ⋆x₁) x₁↠x₂ x₂↠τ⋆x′) | ≡.refl ∧ ≡.refl
  = x′ ∧ ↦-↠ y↠τ⋆x₁ x₁↠x₂ x₂↠τ⋆x′ ∧ ≈′-refl
```

Since _≈_ is transitive and reflexive, we can generalise the above result to handle silent transition sequences of arbitrary length, as well as a symmetric variant:

```
elide-τ⋆ : _↠<τ>⋆_ ⇒ _≈_
elide-τ⋆ = Star.fold _≈_ ≈-trans ≈-refl ∘ Star.map elide-τ
```

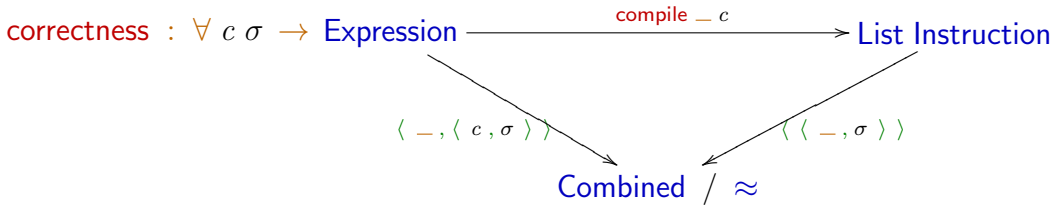
```
elide-τ*′ : Sym _↠<τ>*_ _≈_
elide-τ*′ = ≈-sym ∘ elide-τ*
```

7.9 Compiler Correctness

Now we have enough machinery to formulate the compiler correctness theorem, which states that given a code continuation c and an initial stack σ, execution of the compiled code for an expression e followed by c is weakly bisimilar to the reduction of the expression e followed by the machine continuation $\langle\ c\ ,\ \sigma\ \rangle$,

```
correctness : ∀ e c σ → ⟨ e , ⟨ c , σ ⟩ ⟩ ≈ ⟨ ⟨ compile e c , σ ⟩ ⟩
```

or equivalently as the following commuting diagram:



In particular, instantiating c and σ to empty lists results in the corollary that for any expressions e, the bisimilarity $\langle\ e\ ,\ \langle\ []\ ,\ []\ \rangle\ \rangle \approx \langle\ \langle\ \text{compile}\ e\ []\ ,\ []\ \rangle\ \rangle$ holds.

7.9.1 Proof of correctness

We proceed to prove correctness by structural induction on the expression e. Two additional lemmas corresponding to the following propositions are required, which we will prove along the way:

```
eval-left  : ⟨   a  ⊕ b , ⟨ c , σ ⟩ ⟩ ≈ ⟨ a , ⟨ compile b (ADD :: c) ,      σ ⟩ ⟩
eval-right : ⟨ # m ⊕ b , ⟨ c , σ ⟩ ⟩ ≈ ⟨ b , ⟨              ADD :: c  , m :: σ ⟩ ⟩
```

First, let us consider the base case of correctness, where $e \equiv \# m$:

```
correctness (# m) c σ =
  begin
    ⟨ # m , ⟨ c , σ ⟩ ⟩
  ≈⟨ elide-τ ↠-switch ⟩
    ⟨ ⟨ c , m :: σ ⟩ ⟩
  ≈⟨ elide-τ (↠-↣ ↣-PUSH) ⁻¹⟩
    ⟨ ⟨ PUSH m :: c , σ ⟩ ⟩
  ≡⟨ ≡.refl ⟩
    ⟨ ⟨ compile (# m) c , σ ⟩ ⟩
  □
```

As we mentioned in the conclusion of chapter 6, the equational reasoning combinators defined in the standard library [Dan10b] allow us to present the proof in a simple calculational style. In the code above, proofs of bisimilarity (or definitional equality) are supplied as the second argument of the _≈⟨_⟩_ operator; the _≈⟨_⁻¹⟩_ operator is a symmetric variant, while _≡⟨_⟩_ takes a proof of definitional equality instead. The proofs are combined using the appropriate reflexivity, symmetry and transitivity properties. That is, the above could have been equivalently written:

```
correctness (# m) c σ = ≈-trans (elide-τ ↠-switch)
  (≈-trans (≈-sym (elide-τ (↠-↣ ↣-PUSH))) (≡→≈ ≡.refl))
```

However, the extra annotations in the equational reasoning version makes the proof easier to read and understand.

Moving on to the inductive case, where $e \equiv a \oplus b$:

```
correctness (a ⊕ b) c σ =
  begin
```

```
    ⟨ a ⊕ b , ⟨ c , σ ⟩ ⟩
  ≈⟨ eval-left a b c σ ⟩
    ⟨ a , ⟨ compile b (ADD :: c) , σ ⟩ ⟩
  ≈⟨ correctness a (compile b (ADD :: c)) σ ⟩
    ⟨ ⟨ compile (a ⊕ b) c , σ ⟩ ⟩
  □
```

The first bisimilarity is given by the eval-left lemma, while the second uses the inductive hypothesis for the subexpression a, instantiating the code continuation with compile b (ADD :: c).

7.9.2 The eval-left Lemma

The lemma eval-left has essentially a coinductive proof on the possible transitions $\exists\ \lambda\ \alpha\ \to\ \exists\ \lambda\ a'\ \to\ a\ \mapsto\!<\ \alpha\ >\ a'$ starting from a, with a base case—when no transitions are possible—that is mutually inductively defined with correctness. In the instance of this Zap language however, it suffices to proceed by case analysis on a, as the two alternatives happen to coincide with whether a has any possible transitions. For the base case where a is a numeral # m, no further transitions are possible, and we can reason equationally as follows:

```
eval-left : ∀ a b c σ →
  ⟨ a ⊕ b , ⟨ c , σ ⟩ ⟩ ≈ ⟨ a , ⟨ compile b (ADD :: c) , σ ⟩ ⟩
eval-left (# m) b c σ =
  begin
    ⟨ # m ⊕ b , ⟨ c , σ ⟩ ⟩
  ≈⟨ eval-right m b c σ ⟩
    ⟨ b , ⟨ ADD :: c , m :: σ ⟩ ⟩
  ≈⟨ correctness b (ADD :: c) (m :: σ) ⟩
```

```
    ⟨ ⟨ compile b (ADD :: c) , m :: σ ⟩ ⟩
≈⟨ elide-τ ↠-switch ⁻¹⟩
    ⟨ # m , ⟨ compile b (ADD :: c) , σ ⟩ ⟩
□
```

The proof above makes use of the inductive hypothesis for the subexpression b (of $e \equiv a \oplus b$) in the mutual definition of correctness, as well as the eval-right lemma. In the coinductive case, we consider the x≼y and y≼x halves of the bisimilarity separately,

```
eval-left (a^l ⊕ a^r) b c σ = ♯ x≼y & ♯ y≼x where
  a = a^l ⊕ a^r
```

As we mentioned earlier, this part of the proof is coinductive on the possible reductions rather than structural on the expression a, so the fact that $a \equiv a^l \oplus a^r$ is besides the point. We define a as a synonym for $a^l \oplus a^r$ for clarity as we do not need to refer to a^l or a^r in the proof. We also adopt the convention of writing x for the left, and y for the right hand sides of the $\_\approx\_$ bisimilarity.

In the forward direction, we are given a witness to $x \mapsto\!< \alpha >\! x'$, and must show that y can follow with the same action to some y', such that the resulting x' and y' are bisimilar[1]. The coinduction hypothesis tells us that a can make some visible transition to an a' while emitting an action α. Since the evaluation of $a \oplus b$ is left-biased, it must be the case that $a \oplus b$ reduces under the ↦-L rule. Therefore, we can extract the witness $a{\mapsto}a'$ and repack it to witness the following:

```
⟨ a , ⟨ compile b (ADD :: c) , σ ⟩ ⟩ ↦< α > ⟨ a' , ⟨ compile b (ADD :: c) , σ ⟩ ⟩
```

[1]Recall that the type synonym $\_\preccurlyeq\_$ was defined in §7.7 as follows:

```
_≼_ : Rel Combined
x ≼ y = ∀ x' {α} → x ↦< α > x' → ∃ λ y' → y ↦< α > y' × x' ≈' y'
```

That is, $x \preccurlyeq y$ is a function that given an x', an implicit α and a witness of $x \mapsto\!< \alpha >\! x'$, must return a triple comprising y', a witness of $y \mapsto\!< \alpha >\! y'$, along with a proof of $x' \approx' y'$.

The proof below does not explicitly mention the action α (among others identifiers) as this is already implied by its type, and is automatically inferred.

$$\begin{aligned}
&\mathsf{x{\preccurlyeq}y} \;:\; \langle\ a \oplus b\ ,\ \langle\ c\ ,\ \sigma\ \rangle\ \rangle \preccurlyeq \langle\ a\ ,\ \langle\ \mathsf{compile}\ b\ (\mathsf{ADD} :: c)\ ,\ \sigma\ \rangle\ \rangle \\
&\mathsf{x{\preccurlyeq}y}\ _\ (\Mapsto\!\!\!\!\rightarrow\!\!\!\!\twoheadrightarrow\ \varepsilon\ (\twoheadrightarrow\!\!\!\!\mapsto\ (\mapsto\!\!\!\!\mathsf{L}\ \{a' = a'\}\ a{\mapsto}a'))\ \varepsilon) \\
&\quad = \langle\ a'\ ,\ \langle\ \mathsf{compile}\ b\ (\mathsf{ADD} :: c)\ ,\ \sigma\ \rangle\ \rangle \\
&\quad \wedge\ \Mapsto\!\!\!\!\rightarrow\!\!\!\!\twoheadrightarrow\ \varepsilon\ (\twoheadrightarrow\!\!\!\!\mapsto\ a{\mapsto}a')\ \varepsilon \\
&\quad \wedge\ \approx'\!\!\!\!\approx\ (\mathsf{eval\text{-}left}\ a'\ b\ c\ \sigma)
\end{aligned}$$

The above clause only matches visible transitions with empty sequences of silent combined transitions (i.e. ε). Alternative cases are not possible since $\_{\mapsto}\!<\!\_\!>\!\_$ transitions cannot be silent, and are eliminated using the $\neg{\mapsto}\!<\!\tau\!>$ lemma:

$$\begin{aligned}
&\mathsf{x{\preccurlyeq}y}\ x'\ (\Mapsto\!\!\!\!\rightarrow\!\!\!\!\twoheadrightarrow\ \varepsilon\ (\twoheadrightarrow\!\!\!\!\mapsto\ (\mapsto\!\!\!\!\mathsf{L}\ a{\mapsto}a_0))\ (\twoheadrightarrow\!\!\!\!\mapsto\ a_0{\oplus}b{\mapsto}a_1{\oplus}b\ \lhd\ x_1{\twoheadrightarrow}\tau^\star x')) \\
&\quad = \bot\text{-}\mathsf{elim}\ (\neg{\mapsto}\!<\!\tau\!>\ a_0{\oplus}b{\mapsto}a_1{\oplus}b) \\
&\mathsf{x{\preccurlyeq}y}\ x'\ (\Mapsto\!\!\!\!\rightarrow\!\!\!\!\twoheadrightarrow\ (\twoheadrightarrow\!\!\!\!\mapsto\ a{\oplus}b{\mapsto}a_0{\oplus}b\ \lhd\ x_0{\twoheadrightarrow}\tau^\star x_1)\ x_1{\twoheadrightarrow}x_2\ x_2{\twoheadrightarrow}\tau^\star x') \\
&\quad = \bot\text{-}\mathsf{elim}\ (\neg{\mapsto}\!<\!\tau\!>\ a{\oplus}b{\mapsto}a_0{\oplus}b)
\end{aligned}$$

In the opposite direction, the combined machine y makes a visible transition to $\langle\ a_0\ ,\ \langle\ \mathsf{compile}\ b\ (\mathsf{ADD} :: c)\ ,\ \sigma\ \rangle\ \rangle$ (henceforth denoted by y_0), from which we can extract a witness $a{\mapsto}a_0$. This is followed by some sequence of silent transitions $y_0{\twoheadrightarrow}\tau^\star y'$. In response, x can make a transition to $\langle\ a_0 \oplus b\ ,\ \langle\ c\ ,\ \sigma\ \rangle\ \rangle$ (x' from here on) emitting the same action, that is bisimilar to y_0 via the coinductive hypothesis on a_0. Finally, $\mathsf{elide}\text{-}\tau^{\star\prime}\ y_0{\twoheadrightarrow}\tau^\star y'$ provides a proof of $y'\ \approx'\ y_0$, with which we can obtain $y'\ \approx'\ x'$ by transitivity:

$$\begin{aligned}
&\mathsf{y{\preccurlyeq}x} \;:\; \langle\ a\ ,\ \langle\ \mathsf{compile}\ b\ (\mathsf{ADD} :: c)\ ,\ \sigma\ \rangle\ \rangle \preccurlyeq \langle\ a \oplus b\ ,\ \langle\ c\ ,\ \sigma\ \rangle\ \rangle \\
&\mathsf{y{\preccurlyeq}x}\ y'\ (\Mapsto\!\!\!\!\rightarrow\!\!\!\!\twoheadrightarrow\ \varepsilon\ (\twoheadrightarrow\!\!\!\!\mapsto\ \{e' = a_0\}\ a{\mapsto}a_0)\ y_0{\twoheadrightarrow}\tau^\star y') \\
&\quad = \langle\ a_0 \oplus b\ ,\ \langle\ c\ ,\ \sigma\ \rangle\ \rangle \\
&\quad \wedge\ \Mapsto\!\!\!\!\rightarrow\!\!\!\!\twoheadrightarrow\ \varepsilon\ (\twoheadrightarrow\!\!\!\!\mapsto\ (\mapsto\!\!\!\!\mathsf{L}\ a{\mapsto}a_0))\ \varepsilon
\end{aligned}$$

```
    ∧ ≈′-trans (≈′-≈ (elide-τ⋆′ y₀↠τ⋆y′)) (≈′-sym (≈′-≈ (eval-left a₀ b c σ)))
y≼x y′ (⊨-↠ (↠-↦ a↦a₀ ◁ y₀↠τ⋆y₁) y₁↠y₂ y₂↠τ⋆y₁)
    = ⊥-elim (¬↦<τ> a↦a₀)
```

The second clause eliminates the impossible case of a making a τ transition, which completes the proof of eval-left.

7.9.3 The eval-right Lemma

The eval-right lemma proceeds in a similar manner by coinduction on the possible _↦<_>_ transitions from b. In the base case, b cannot make any further such transitions (i.e. is some number # n), and we need to show:

```
⟨ # m ⊕ # n , ⟨ c , σ ⟩ ⟩ ≈ ⟨ # n , ⟨ ADD :: c , m :: σ ⟩ ⟩
```

As the right hand side can make a silent ↠-switch transition, we can factor this part out to make the proof a little neater:

```
eval-right : ∀ m b c σ → ⟨ # m ⊕ b , ⟨ c , σ ⟩ ⟩ ≈ ⟨ b , ⟨ ADD :: c , m :: σ ⟩ ⟩
eval-right m (# n) c σ =
  begin
    ⟨ # m ⊕ # n , ⟨ c , σ ⟩ ⟩
  ≈⟨ ⊕≈ADD ⟩
    ⟨ ⟨ ADD :: c , n :: m :: σ ⟩ ⟩
  ≈⟨ elide-τ ↠-switch ⁻¹⟩
    ⟨ # n , ⟨ ADD :: c , m :: σ ⟩ ⟩
  □ where
  ⊕≈ADD : ⟨ # m ⊕ # n , ⟨ c , σ ⟩ ⟩ ≈ ⟨ ⟨ ADD :: c , n :: m :: σ ⟩ ⟩
  ⊕≈ADD = ♯ x≼y & ♯ y≼x where
```

The ⊕≈ADD lemma is where we encounter the non-determinism in our Zap language. The proof for the two halves x≼y and y≼x are as follows: in the first instance,

m ⊕ # n can transition with either the ↦-⊞ or ↦-↯ rule, and we must show that ⟨ ADD :: c , $n :: m :: \sigma$ ⟩ can follow with the same action:

```
x≼y : ⟨ # m ⊕ # n , ⟨ c , σ ⟩ ⟩ ≼ ⟨ ⟨ ADD :: c , n :: m :: σ ⟩ ⟩
x≼y x′ (⊨⇒-⇝ ε (⇝-↦ ↦-⊞) x₀⇝τ⋆x′)
  = ⟨ ⟨ c , m + n :: σ ⟩ ⟩
  ∧ ⊨⇒-⇝ ε (⇝-↣ ↣-ADD) ε
  ∧ ≈′-trans (≈′-≈ (elide-τ⋆′ x₀⇝τ⋆x′)) (≈′-≈ (elide-τ ⇝-switch))
x≼y x′ (⊨⇒-⇝ ε (⇝-↦ ↦-↯) x₀⇝τ⋆x′)
  = ⟨ ⟨ c , 0 :: σ ⟩ ⟩
  ∧ ⊨⇒-⇝ ε (⇝-↣ ↣-ZAP) ε
  ∧ ≈′-trans (≈′-≈ (elide-τ⋆′ x₀⇝τ⋆x′)) (≈′-≈ (elide-τ ⇝-switch))
x≼y x′ (⊨⇒-⇝ ε (⇝-↦ (↦-R ())) x₀⇝τ⋆x′)
x≼y x′ (⊨⇒-⇝ ε (⇝-↦ (↦-L ())) x₀⇝τ⋆x′)
x≼y x′ (⊨⇒-⇝ (⇝-↦ e↦e₀ ◁ x₀⇝τ⋆x₁) x₁⇝x₂ x₂⇝τ⋆x′)
  = ⊥-elim (¬↦<τ> e↦e₀)
```

This is sketched in figure 7.1—read from the top down—where the left and right branches correspond to the first two clauses above. The third and fourth clauses handle the fact that neither # m nor # n by themselves can reduce any further, while the last deals with silent _↦<_>_ transitions.

The other direction of ⊕≈ADD is illustrated by reading figure 7.1 from the bottom up, and proceeds in the same manner:

```
y≼x : ⟨ ⟨ ADD :: c , n :: m :: σ ⟩ ⟩ ≼ ⟨ # m ⊕ # n , ⟨ c , σ ⟩ ⟩
y≼x y′ (⊨⇒-⇝ ε (⇝-↣ ↣-ADD) y₀⇝τ⋆y′)
  = ⟨ ⟨ c , m + n :: σ ⟩ ⟩
  ∧ ⊨⇒-⇝ ε (⇝-↦ ↦-⊞) (⇝-switch ◁ ε)
  ∧ ≈′-≈ (elide-τ⋆′ y₀⇝τ⋆y′)
```

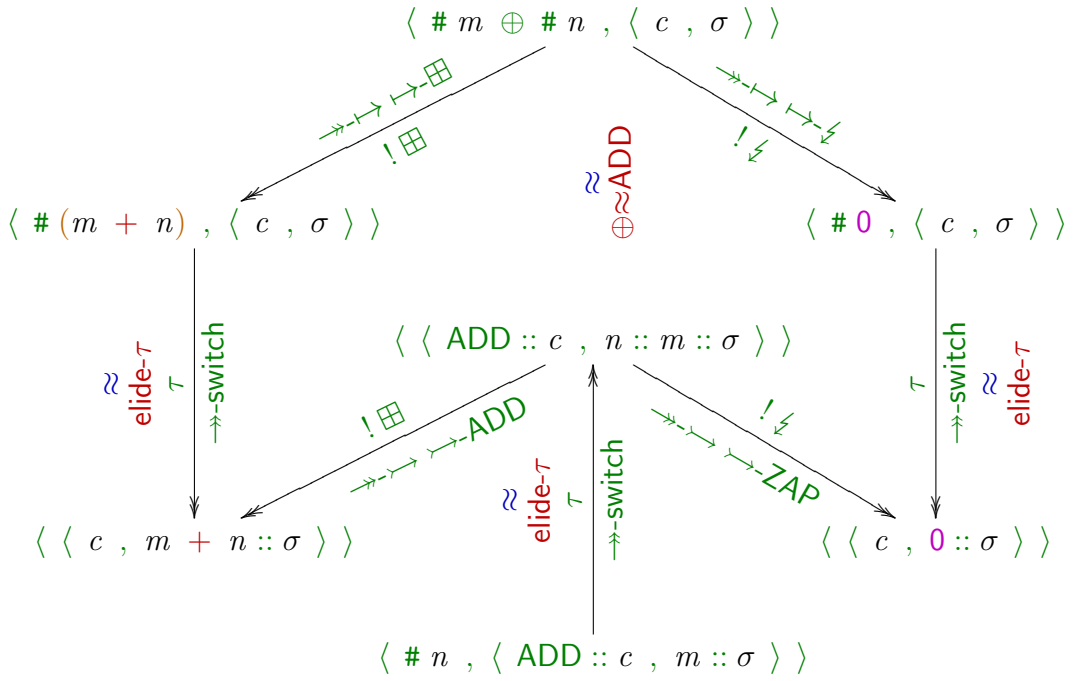


Figure 7.1: Proof sketch for the base case of the eval-right lemma.

$$
\begin{aligned}
&\mathsf{y}\preccurlyeq\mathsf{x}\ y'\ (\Mapsto\!\!\text{-}\twoheadrightarrow\ \varepsilon\ (\twoheadrightarrow\!\!\text{-}\rightarrowtail\ \rightarrowtail\!\!\text{-}\mathsf{ZAP})\ y_0 \twoheadrightarrow \tau^\star y') \\
&\quad = \langle\ \langle\ c\ ,\ 0 :: \sigma\ \rangle\ \rangle \\
&\quad \wedge\ \Mapsto\!\!\text{-}\twoheadrightarrow\ \varepsilon\ (\twoheadrightarrow\!\!\text{-}\mapsto\ \mapsto\!\!\text{-}\lightning)\ (\twoheadrightarrow\!\!\text{-switch}\ \triangleleft\ \varepsilon) \\
&\quad \wedge\ \approx'\!\text{-}\approx\ (\mathsf{elide}\text{-}\tau^{\star\prime}\ y_0 \twoheadrightarrow \tau^\star y') \\
&\mathsf{y}\preccurlyeq\mathsf{x}\ y'\ (\Mapsto\!\!\text{-}\twoheadrightarrow\ (\twoheadrightarrow\!\!\text{-}\rightarrowtail\ ()\ \triangleleft\ y_0 \twoheadrightarrow \tau^\star y_1)\ y_1 \twoheadrightarrow y_2\ y_2 \twoheadrightarrow \tau^\star y_1)
\end{aligned}
$$

The impossible pattern () in the final clause corresponds to the fact that there is no transition in which an ADD instruction emits a silent action.

The coinductive case of the eval-right lemma follows a similar structure to that of eval-left, on the possible transitions from b:

$$
\begin{aligned}
&\mathsf{eval\text{-}right}\ m\ (b^l \oplus b^r)\ c\ \sigma\ =\ \sharp\,\mathsf{x}\preccurlyeq\mathsf{y}\ \&\ \sharp\,\mathsf{y}\preccurlyeq\mathsf{x}\ \textbf{where} \\
&\quad b\ =\ b^l \oplus b^r
\end{aligned}
$$

In the x≼y direction, $\langle\ \#\ m\ \oplus\ b\ ,\ \langle\ c\ ,\ \sigma\ \rangle\ \rangle$ must reduce according to $b \mapsto b_0$, therefore y can follow by transitioning to $\langle\ b_0\ ,\ \langle\ \mathsf{ADD} :: c\ ,\ m :: \sigma\ \rangle\ \rangle$:

```
x≼y : ⟨ # m ⊕ b , ⟨ c , σ ⟩ ⟩ ≼ ⟨ b , ⟨ ADD :: c , m :: σ ⟩ ⟩
x≼y x′ (⊨-↠ ε (↠-↦ (↦-R {b′ = b₀} b↦b₀)) x₀↠τ⋆x′)
  = ⟨ b₀ , ⟨ ADD :: c , m :: σ ⟩ ⟩
  ∧ ⊨-↠ ε (↠-↦ b↦b₀) ε
  ∧ ≈′-trans (≈′-≈ (elide-τ⋆′ x₀↠τ⋆x′)) (≈′-≈ (eval-right m b₀ c σ))
x≼y x′ (⊨-↠ ε (↠-↦ (↦-L ())) x₀↠τ⋆x′)
x≼y x′ (⊨-↠ (↠-↦ m⊕b↦m⊕b₀ ◁ x₀↠τ⋆x₁) x₁↠x₂ x₂↠τ⋆x′)
  = ⊥-elim (¬↦<τ> m⊕b↦m⊕b₀)
```

The second clause shows that # m $\mapsto\!<$ Λ $>$ a' is uninhabited, while the third uses the fact that # m $\oplus$ b cannot make a silent $\_\mapsto\!<\_>\_$ transition, to show that both cases are impossible.

For the y≼x direction, the proof simply runs in reverse, with x following y by transitioning to $\langle$ # m $\oplus$ b_0 , $\langle$ c , σ $\rangle$ $\rangle$:

```
y≼x : ⟨ b , ⟨ ADD :: c , m :: σ ⟩ ⟩ ≼ ⟨ # m ⊕ b , ⟨ c , σ ⟩ ⟩
y≼x y′ (⊨-↠ ε (↠-↦ {e′ = b₀} b↦b₀) y₀↠τ⋆y′)
  = ⟨ # m ⊕ b₀ , ⟨ c , σ ⟩ ⟩
  ∧ ⊨-↠ ε (↠-↦ (↦-R b↦b₀)) ε
  ∧ ≈′-trans (≈′-≈ (elide-τ⋆′ y₀↠τ⋆y′)) (≈′-sym (≈′-≈ (eval-right m b₀ c σ)))
y≼x y′ (⊨-↠ (↠-↦ b↦b₀ ◁ y₀↠τ⋆y₁) y₁↠y₂ y₂↠τ⋆y₁)
  = ⊥-elim (¬↦<τ> b↦b₀)
```

The final case deals with the impossibility of silent $b\mapsto b_0$ transitions. This completes the proof of the eval-right and eval-left lemmas, and in turn the correctness theorem for our Zap language.

7.10 Conclusion

In this chapter we introduced a new technique for handling non-determinism in the context of compiler correctness proofs, which we illustrated using the Zap language. By carefully choosing silent and visible actions to distinguish between non-deterministic choices in the reduction of expressions and virtual machines, we were able to show the crucial elide-τ lemma used in the compiler correctness proof that follows. Finally, our notion of a combined machine allowed us to directly establish a bisimulation between our source and target languages without the need for an underlying process calculus.

Chapter 8

Compiling Concurrency Correctly

In the previous chapter, we introduced our methodology of using the notion of bisimilarity on combined machines for tacking compiler correctness for a simple non-deterministic language. In this chapter, we shall demonstrate that this technique is scalable to a concurrent setting, by extending the language with a simple fork primitive that introduces explicit concurrency into our system.

8.1 The Fork Language

8.1.1 Syntax and Virtual Machine

As with the Zap language of the previous chapter, we base this Fork language on that of natural numbers and addition. The inclusion of an extra fork primitive introduces a simple and familiar approach to explicit concurrency:

```
data Expression : Set where
  #_   : (m : ℕ) → Expression
  _⊕_  : (a b : Expression) → Expression
  fork : (e : Expression) → Expression
```

An expression fork e will begin evaluation of e in a new thread, immediately returning # 0, in a manner reminiscent of Haskell's forkIO primitive. The collection of concurrent threads in the system is modelled as a 'thread soup' [PJ01], defined later in §8.2. Similarly, we extend the virtual machine with a FORK instruction, which spawns a given sequence of instructions in a new thread:

```
data Instruction : Set where
  PUSH : (m : ℕ) → Instruction
  ADD  : Instruction
  FORK : (c : List Instruction) → Instruction
```

The compiler includes an extra case for fork, but remains otherwise unchanged from the definition in §7.4:

```
compile : Expression → List Instruction → List Instruction
compile (# m)     c = PUSH m :: c
compile (a ⊕ b) c = compile a (compile b (ADD :: c))
compile (fork e)  c = FORK (compile e []) :: c
```

As before, each virtual machine thread comprises a list of Instructions along with a stack of natural numbers:

```
data Machine : Set where
  ⟨_,_⟩ : (c : List Instruction) (σ : List ℕ) → Machine
```

8.1.2 Actions

We extend the set of actions by $^{+}\ e$, to indicate the spawning of a new thread e, and the action $\ldots\ \alpha$ to indicate preemption of the foremost thread:

```
data Action (L : Set) : Set where
  τ : Action L
```

```
  ⊞ : Action L
  □_ : (m : ℕ) → Action L
  ⁺_ : (x : L) → Action L
  ..._ : (α : Action L) → Action L
```

The above definition of Action is parametrised over the type of spawned threads, either Expressions or Machines. As we now have explicit concurrency in the language, we no longer require the 'zap' action or its associated semantics.

With the Zap language, a τ label sufficed to identify silent actions, because its semantics does not diverge at points where silent transitions occurred. With the Fork language, we have a 'soup' of concurrent threads, of which more than one may be able to make a silent transition at any given point. Previously we mandated that distinct choices in the reduction path must be labelled with distinct actions: in this case, we have folded the τ label into the definition of Action, such that e.g. both τ and $\ldots\,\tau$ are considered to be silent, yet they remain distinct.

This choice does complicate matters somewhat: in the two-level definition of labels and actions in the Zap language, we could simply pattern match a 'label' with τ to determine if a transition was silent; in the same way, we know *a priori* that 'actions' cannot be silent. With the Fork language, we must use a more elaborate scheme:

```
  data _≃τ {l} : Action l → Set where
    is-τ  :                    τ  ≃τ
    is-... : ∀ {α} → α ≃τ → (... α) ≃τ
```

The above type functions as a predicate on Actions: $\alpha \simeq\tau$ is inhabited precisely when α is considered silent. Conversely, the negation of $\_\simeq\tau$ serves the same rôle for non-silent actions, defined as follows:

```
  _≄τ : ∀ {l} → Action l → Set
  α ≄τ = ¬ α ≃τ
```

8.1.3 Semantics

It remains for us to give the semantics for expressions and virtual machines. As per §7.3, expression follow a left-biased reduction semantics given by the ↦-⊞, ↦-R and ↦-L rules:

```
data _↦<_>_ : LTS (Action Expression) Expression where
  ↦-⊞    : ∀ {m n} → # m ⊕ # n ↦< ⊞ > # (m + n)
  ↦-R    : ∀ {m b b′ α} (b↦b′ : b ↦< α > b′) → # m ⊕ b ↦< α > # m ⊕ b′
  ↦-L    : ∀ {a a′ b α} (a↦a′ : a ↦< α > a′) →   a ⊕ b ↦< α > a′ ⊕ b
  ↦-fork : ∀ {e} → fork e ↦< ⁺ e > # 0
```

Spawning of new threads is handled by the ↦-fork rule, which embeds the expression in an $^{+}\ e$ action. The expression fork e immediately reduces to # 0, in a manner reminiscent of Haskell's forkIO :: IO () → IO () primitive.

In turn, the virtual machine for our Fork language inherits the ⟼-PUSH and ⟼-ADD rules from that of the Zap language, given in §7.4:

```
data _⟼<_>_ : LTS (Action Machine) Machine where
  ⟼-PUSH : ∀ {c σ m} →
    ⟨ PUSH m :: c , σ ⟩ ⟼< τ > ⟨ c , m :: σ ⟩
  ⟼-ADD : ∀ {c σ m n} →
    ⟨ ADD :: c , n :: m :: σ ⟩ ⟼< ⊞ > ⟨ c , m + n :: σ ⟩
  ⟼-FORK : ∀ {c c′ σ} →
    ⟨ FORK c′ :: c , σ ⟩ ⟼< ⁺ ⟨ c′ , [] ⟩ > ⟨ c , 0 :: σ ⟩
```

In this instance, we have added a ⟼-FORK rule that handles the case of a FORK c' instruction: given a sequence of instructions c', we emit a newly initialised virtual machine embedded in an $^{+}\ \langle\ c'\ ,\ []\ \rangle$ action, leaving 0 on top of the stack.

8.2 Combined Machines and Thread Soups

Our definition of a combined machine remains unchanged from the Zap language, with the constructors $\langle \_,\_ \rangle$, $\langle \_ \rangle$ and $\langle\rangle$ corresponding to the three phases of executions:

```
data Combined : Set where
  ⟨_,_⟩ : (e : Expression) (t : Machine) → Combined
  ⟨_⟩ : (t : Machine) → Combined
  ⟨⟩ : Combined
```

So far, the semantics of the Fork language have been given in terms of individual expression or virtual machine threads. Since the notion of a 'thread soup' is common to both cases, we simply matters by introducing concurrency at the level of combined machines. It suffices to model our thread soups as Lists of combined machines, and we define labelled transitions between them as follows:

```
data _↠<_>_ : LTS (Action Combined) (List Combined) where
  ↠-↦ : ∀ {e e′ t s α} →
    (e↦e′ : e ↦< α > e′) → let α′ = E⁺ <$> α in
      ⟨ e , t ⟩ :: s ↠< α′ > ⟨ e′ , t ⟩ :: α′ ⁺:: s
  ↠-↣ : ∀ {t t′ s α} →
    (t↣t′ : t ↣< α > t′) → let α′ = M⁺ <$> α in
          ⟨ t ⟩ :: s ↠< α′ > ⟨ t′ ⟩      :: α′ ⁺:: s
  ↠-done : ∀ {m s} →
    ⟨ ⟨ [] , m :: [] ⟩ ⟩ :: s ↠< □ m > ⟨⟩ :: s
  ↠-switch : ∀ {m c σ s} →
    ⟨ # m , ⟨ c , σ ⟩ ⟩ :: s ↠< τ > ⟨ ⟨ c , m :: σ ⟩ ⟩ :: s
  ↠-preempt : ∀ {x s s′ α} → (s↠s′ : s ↠< α > s′) →
    x :: s ↠< ... α > x :: s′
```

As with the Zap language, the $\twoheadrightarrow$-$\mapsto$ and $\twoheadrightarrow$-$\rightarrowtail$ rules lift transitions on expression and virtual machine threads to soups of combined machines. The trivial E^+ and M^+ helpers lift Expression and Machine into Combined, given as follows,

$$\mathsf{E}^+ : \mathsf{Expression} \to \mathsf{Combined}$$
$$\mathsf{E}^+\ e = \langle\ e\ ,\ \langle\ []\ ,\ []\ \rangle\ \rangle$$
$$\mathsf{M}^+ : \mathsf{Machine} \to \mathsf{Combined}$$
$$\mathsf{M}^+ = \langle \_ \rangle$$

while $\_^+::\_$ inserts any spawned threads into the thread soup, defined below:

$$\_^+::\_ : \mathsf{Action\ Combined} \to \mathsf{List\ Combined} \to \mathsf{List\ Combined}$$
$$\tau\ \ ^+::\ s = s$$
$$\boxplus\ \ ^+::\ s = s$$
$$\square\ m\ ^+::\ s = s$$
$$^+\ x\ \ ^+::\ s = x :: s$$
$$\ldots\ \alpha\ \ ^+::\ s = s$$

Aside from the generalisation to thread soups, the $\twoheadrightarrow$-done and $\twoheadrightarrow$-switch rules are otherwise identical to those defined for the Zap language.

Finally, we allow arbitrary thread interleaving via the $\twoheadrightarrow$-preempt rule. As our focus is not on the subtleties of different scheduling algorithms, we therefore do not place any restrictions on what thread the 'scheduler' may choose to execute next.

8.3 Silent and Visible Transitions

For our later proofs, it will be convenient to have a canonical definition of silent and non-silent transitions. We regard a silent transition between r and s as a triple comprising an action α, a proof of $\alpha \simeq \tau$, along with the transition $r \twoheadrightarrow\!\!< \alpha >\ s$:

```
_↠τ_ : ∀ r s → Set
r ↠τ s = ∃ λ α → α ≃τ × r ↠< α > s
```

Conversely, a non-silent transition carries a proof of $\alpha \not\simeq\tau$ instead:

```
_↠≄τ_ : ∀ r s → Set
r ↠≄τ s = ∃ λ α → α ≄τ × r ↠< α > s
```

Finally we may write $\_\twoheadrightarrow\tau^\star\_$ for the reflexive, transitive closure of $\_\twoheadrightarrow\tau\_$, using the following definition:

```
_↠τ*_ : ∀ r s → Set
_↠τ*_ = Star _↠τ_
```

When we are only interested in the transitions of a single thread, the following synonyms are helpful for stating any relevant properties:

```
_↠τ₁_ : ∀ x y → Set
x ↠τ₁ y = ∀ s → x :: s ↠τ y :: s
_↠τ₁*_ : ∀ x y → Set
x ↠τ₁* y = ∀ s → x :: s ↠τ* y :: s
```

We subscript the above transitions with '$_1$' as a reminder that the propositions are $\forall$-quantified over the rest of the thread soup. For $\_\twoheadrightarrow\not\simeq\tau_1\_$, we must concatenate the resulting x'^+ : List Combined to the rest of the soup,

```
_↠≄τ₁_ : ∀ x x'⁺ → Set
x ↠≄τ₁ x'⁺ = ∀ s → x :: s ↠≄τ x'⁺ ++ s
```

as non-silent transitions may potentially spawn new threads. Finally, the $\_\triangleleft_1\_$ function allows us to conveniently combine $\_\twoheadrightarrow\tau_1\_$ sequences, in the same manner as the $\_\triangleleft\_$ constructor of the Star type:

```
_◁₁_ : ∀ {x x′ y} → x →↠τ₁ x′ → x′ →↠τ₁⋆ y → x →↠τ₁⋆ y
x↠τ₁x′ ◁₁ x′↠τ₁⋆y = λ s → x↠τ₁x′ s ◁ x′↠τ₁⋆y s
```

Given the above definitions of silent and non-silent transitions, our notion of a visible transition is identical in essence to that of the Zap language, given back in §7.7:

```
data _⤇<_>_ : LTS (Action ⊤) (List Combined) where
  ⤇-↠ : ∀ {s s₀ s₁ s′}
    (s↠τ⋆s₀    : s  →↠τ⋆  s₀)
    (s₀↠≢τs₀ : s₀ →↠≢τ s₁)
    (s₁↠τ⋆s′   : s₁ →↠τ⋆  s′) →
    s ⤇< ⟦ s₀↠≢τs₀ ⟧ > s′
```

As we do not have direct access to the action emitted by the non-silent $s_0 \twoheadrightarrow_{\not\equiv\tau} s_0$ transition, we require a helper $[\![\_]\!]$ to extract the visible action:

```
visible : ∀ {α : Action Combined} → α ≢τ → Action ⊤
visible {τ}     α≢τ = ⊥-elim (α≢τ is-τ)
visible {⊞}     α≢τ = ⊞
visible {□ m}   α≢τ = □ m
visible {⁺ x}   α≢τ = ⁺ tt
visible {… α}   …α≢τ = visible (…α≢τ ∘ is-…)

⟦_⟧ : ∀ {s s′} → s →↠≢τ s′ → Action ⊤
⟦_⟧ (α ∧ α≢τ ∧ s↠s′) = visible α≢τ
```

By this point, any spawned threads will have been inserted into the thread soup already, so we are no longer interested in its particulars, other than that a fork has taken place. Correspondingly, $[\![\_]\!]$ returns an Action ⊤—where ⊤ is the singleton 'unit' type—rather than an Action Combined.

8.4 Bisimilarity

The definition of bisimilarity remains identical to that of the Zap language given in §7.7, save for a change in the carrier set from Combined to List Combined:

```
_≼_ : Rel (List Combined)
x ≼ y = ∀ x′ {α} → x ⇛< α > x′ → ∃ λ y′ → y ⇛< α > y′ × x′ ≈′ y′

data _≈_ (x : List Combined) : List Combined → Set where
  _&_ : ∀ {y} → (x≼y : ∞ (x ≼ y)) → (y≼x : ∞ (y ≼ x)) → x ≈ y
```

The embedded language of $\_\approx'\_$ gains an extra symbol $\approx'\text{-cong}_2$ that allows us to combine two pairs of bisimilar thread soups. Formally, it corresponds to the proposition that $\_\approx'\_$ is a congruence relation on the monoid (List Combined, $+\!\!+$, []),

```
data _≈′_ : Rel (List Combined) where
  ≈′-≈     : _≈_ ⇒ _≈′_
  ≈′-sym   : Symmetric _≈′_
  ≈′-trans : Transitive _≈′_
  ≈′-cong₂ : _++_ Preserves₂ _≈′_ ⟶ _≈′_ ⟶ _≈′_
```

where the type of the $\approx'\text{-cong}_2$ constructor expands to:

```
≈′-cong₂ : ∀ {r^l s^l r^r s^r} → r^l ≈′ s^l → r^r ≈′ s^r → r^l ++ r^r ≈′ s^l ++ s^r
```

The same proofs from §7.7 suffices to show that $\_\approx\_$ forms an equivalence relation on thread soups. The obligation to show that $\_\approx'\_$ implies $\_\approx\_$ will be fulfilled at the end of §8.7, as it depends on the proof that $\_\approx\_$ is also a congruence relation on (List Combined, $+\!\!+$, []). Before we can do that however, we have a few more lemmas to establish.

8.5 Properties of Thread Soups

In this section, we will highlight various lemmas concerning thread soups that are used towards the final correctness theorem for this Fork language. For brevity, we will omit the proofs, and instead hint at the proof method; the full Agda source code may be found on my website.

8.5.1 Soups Concatenation Preserves Silent Transitions

Concatenation of thread soups preserve silent transition sequences,

```
↠τ*-++ : _++_ Preserves₂ _↠τ*_ ⟶ _↠τ*_ ⟶ _↠τ*_
```

or equivalently, given $r \twoheadrightarrow_\tau^\star r'$ and $s \twoheadrightarrow_\tau^\star s'$, we can produce a silent transition sequence $r \mathbin{+\!\!+} s \twoheadrightarrow_\tau^\star r' \mathbin{+\!\!+} s'$:

```
↠τ*-++ : ∀ {r r' s s'} → r ↠τ* r' → s ↠τ* s' → r ++ s ↠τ* r' ++ s'
```

We can proceed by structural induction on the first thread soup argument, which is $x :: r$ in the inductive case. Using the fact that forking cannot be silent, we can decapitate the x from the thread soup to obtain a pair of transitions $x \twoheadrightarrow_{\tau_1}^\star x'$ and $r \twoheadrightarrow_\tau^\star r'$. The first of these can be instantiated to one half of the goal, that is:

$$x :: r \mathbin{+\!\!+} s \twoheadrightarrow_\tau^\star x' :: r \mathbin{+\!\!+} s$$

The induction hypothesis on the other hand uses the second $r \twoheadrightarrow_\tau^\star r'$ to give the sequence $r \mathbin{+\!\!+} s \twoheadrightarrow_\tau^\star r' \mathbin{+\!\!+} s'$, which we can map ↠-preempt over to arrive at the other half of the goal:

$$x' :: r \mathbin{+\!\!+} s \twoheadrightarrow_\tau^\star x' :: r' \mathbin{+\!\!+} s'$$

Concatenation of the two halves completes the proof.

8.5.2 Partitioning Silent Transitions

Conversely, the thread soups of a silent transition sequence $r \mathbin{+\!\!+} s \twoheadrightarrow\!\tau^\star\ r's'$ can be partitioned into $r \twoheadrightarrow\!\tau^\star\ r'$ and $s \twoheadrightarrow\!\tau^\star\ s'$:

$$\twoheadrightarrow\!\tau^\star\text{-split} : \forall\ r\ s\ \{r's'\} \to r \mathbin{+\!\!+} s \twoheadrightarrow\!\tau^\star\ r's' \to \exists_2\ \lambda\ r'\ s' \to r' \mathbin{+\!\!+} s' \equiv r's' \times r \twoheadrightarrow\!\tau^\star\ r' \times s \twoheadrightarrow\!\tau^\star\ s'$$

Again, the proof uses structural induction on the first thread soup argument; decapitating x from the $x :: r$ in the inductive case—as per the proof for $\twoheadrightarrow\!\tau^\star$-$+\!\!+$—produces a pair of transitions $x \twoheadrightarrow\!\tau_1^\star\ x'$ and $r \twoheadrightarrow\!\tau^\star\ r'$. The induction hypothesis delivers the $r \twoheadrightarrow\!\tau^\star\ r'$ needed to construct the sequence $x :: r \twoheadrightarrow\!\tau^\star\ x' :: r \twoheadrightarrow\!\tau^\star\ x' :: r'$, which completes the proof.

A useful corollary of $\twoheadrightarrow\!\tau^\star$-split allows us to focus our attention on a single thread, by dissecting its transitions out of a transition sequence on the entire thread soup:

$$\twoheadrightarrow\!\tau^\star\text{-dissect} : \forall\ r^l\ r^r\ \{x\ r'\} \to r^l \mathbin{+\!\!+} x :: r^r \twoheadrightarrow\!\tau^\star\ r' \to \exists_2\ \lambda\ r^{l\prime}\ r^{r\prime} \to \exists\ \lambda\ x' \to r' \equiv r^{l\prime} \mathbin{+\!\!+} x' :: r^{r\prime} \times r^l \twoheadrightarrow\!\tau^\star\ r^{l\prime} \times r^r \twoheadrightarrow\!\tau^\star\ r^{r\prime} \times x \twoheadrightarrow\!\tau_1^\star\ x'$$

In other words, given a silent transition sequence starting from the thread soup $r^l \mathbin{+\!\!+} x :: r^r$, there exists $r^{l\prime}$, $r^{r\prime}$ and x' satisfying $r^l \twoheadrightarrow\!\tau^\star\ r^{l\prime}$, $r^r \twoheadrightarrow\!\tau^\star\ r^{r\prime}$ and $x \twoheadrightarrow\!\tau_1^\star\ x'$ respectively.

8.5.3 Partitioning a Non-Silent Transition

We can also partition the thread soup for a non-silent transition, although the situation is a little more involved:

$$\twoheadrightarrow\!\not\equiv\!\tau\text{-split} : \forall\ r\ s\ \{r's'\} \to (rs \twoheadrightarrow r's' : r \mathbin{+\!\!+} s \twoheadrightarrow\!\not\equiv\!\tau\ r's') \to (\exists\ \lambda\ r' \to r' \mathbin{+\!\!+} s \equiv r's' \times \Sigma\ (r \twoheadrightarrow\!\not\equiv\!\tau\ r')\ \lambda\ r \twoheadrightarrow r' \to [\![\ r \twoheadrightarrow r'\ ]\!] \equiv [\![\ rs \twoheadrightarrow r's'\ ]\!])\ \uplus$$

$$(\exists\, \lambda\ s'\ \to\ r\ +\!\!+\ s'\ \equiv\ r's'\ \times\ \Sigma\ (s\ \twoheadrightarrow\!\!\not\equiv\tau\ s')$$
$$\lambda\ s\twoheadrightarrow s'\ \to\ [\![\ s\twoheadrightarrow s'\ ]\!]\ \equiv\ [\![\ rs\twoheadrightarrow r's'\ ]\!])$$

Partitioning the thread soup in a non-silent transition has two possible outcomes, as the active thread responsible for the transition could be in either one of r or s. The proof proceeds by structural induction on the soup r as before, by inspecting each of its threads. If found, we construct and return the transition $r\ \twoheadrightarrow\!\!\not\equiv\tau\ r'$; otherwise none of the r threads are responsible for the non-silent transition, so by elimination it must be in s, and we can respond with the transition $s\ \twoheadrightarrow\!\!\not\equiv\tau\ s'$. In both cases, we construct a proof that the action emitted by the original $rs\twoheadrightarrow r's'$ transition is the same as either $[\![\ r\twoheadrightarrow r'\ ]\!]$ or $[\![\ s\twoheadrightarrow s'\ ]\!]$, as appropriate.

In the same way that $\twoheadrightarrow\tau^\star$-split has its $\twoheadrightarrow\tau^\star$-dissect corollary, we can show the following $\twoheadrightarrow\!\!\not\equiv\tau$-dissect corollary for $\twoheadrightarrow\!\!\not\equiv\tau$-split:

$$\twoheadrightarrow\!\!\not\equiv\tau\text{-dissect}\ :\ \forall\ r^l\ r^r\ \{x\ x'\ r'\}\ \to$$
$$x\ \twoheadrightarrow\tau_1\ x'\ \to\ (r^l x r^r\twoheadrightarrow r'\ :\ r^l\ +\!\!+\ x :: r^r\ \twoheadrightarrow\!\!\not\equiv\tau\ r')\ \to$$
$$(\exists\, \lambda\ r^{l\prime}\ \to\ r'\ \equiv\ r^{l\prime}\ +\!\!+\ x :: r^r\ \times\ \Sigma\ (r^l\ \twoheadrightarrow\!\!\not\equiv\tau\ r^{l\prime})$$
$$\lambda\ r^l\twoheadrightarrow r^{l\prime}\ \to\ [\![\ r^l\twoheadrightarrow r^{l\prime}\ ]\!]\ \equiv\ [\![\ r^l x r^r\twoheadrightarrow r'\ ]\!])\ \uplus$$
$$(\exists\, \lambda\ r^{r\prime}\ \to\ r'\ \equiv\ r^l\ +\!\!+\ x :: r^{r\prime}\ \times\ \Sigma\ (r^r\ \twoheadrightarrow\!\!\not\equiv\tau\ r^{r\prime})$$
$$\lambda\ r^r\twoheadrightarrow r^{r\prime}\ \to\ [\![\ r^r\twoheadrightarrow r^{r\prime}\ ]\!]\ \equiv\ [\![\ r^l x r^r\twoheadrightarrow r'\ ]\!])$$

Here we are given an additional hypothesis that the thread x makes a silent initial transition, which is unique by our choice of actions. Therefore the thread responsible for the non-silent transition $r^l x r^r\twoheadrightarrow r'$ must reside in either one of r^l or r^r, giving rise to the two alternatives of either $r^l\ \twoheadrightarrow\!\!\not\equiv\tau\ r^{l\prime}$ or $r^r\ \twoheadrightarrow\!\!\not\equiv\tau\ r^{r\prime}$.

8.5.4 Dissecting a Visible Transition

Recall that a visible transition comprises of two sequences of silent transitions either side of a single non-silent transition. Therefore combining the previous results allows

us to dissect a visible transition in much the same way:

$$\begin{aligned}
&\mapsto\text{-dissect} \;:\; \forall\, r^l\; r^r\; \{x\; x'\; r'\; \alpha\} \;\to \\
&\quad x \twoheadrightarrow_{\tau_1} x' \to r^l \mathbin{+\!\!+} x :: r^r \mapsto\!\!< \alpha >\!\!\!\cdot\; r' \to \\
&\quad r^l \mathbin{+\!\!+} x' :: r^r \mapsto\!\!< \alpha >\!\!\!\cdot\; r' \uplus \\
&\quad \exists_2\, \lambda\; r^{l\prime}\; r^{r\prime} \to r' \equiv r^{l\prime} \mathbin{+\!\!+} x :: r^{r\prime} \times \\
&\qquad ((r^l \mapsto\!\!< \alpha >\!\!\!\cdot\; r^{l\prime} \times r^r \twoheadrightarrow_{\tau^\star} r^{r\prime}) \uplus (r^l \twoheadrightarrow_{\tau^\star} r^{l\prime} \times r^r \mapsto\!\!< \alpha >\!\!\!\cdot\; r^{r\prime}))
\end{aligned}$$

Given a witness of $x \twoheadrightarrow_{\tau_1} x'$ and the visible transition $r^l \mathbin{+\!\!+} x :: r^r \mapsto\!\!< \alpha >\!\!\!\cdot\; r'$, there are two possibilities regarding the thread x: either $x \twoheadrightarrow_{\tau_1} x'$ takes place somewhere within the visible transition, so that removing it results in a witness of $r^l \mathbin{+\!\!+} x' :: r^r \mapsto\!\!< \alpha >\!\!\!\cdot\; r'$; or that x remains inactive throughout while r^l and r^r make transitions to some $r^{l\prime}$ and $r^{r\prime}$ respectively. Depending on which of r^l or r^r the thread responsible for the non-silent action is found in, we provide witnesses of either $r^l \mapsto\!\!< \alpha >\!\!\!\cdot\; r^{l\prime}$ and $r^r \twoheadrightarrow_{\tau^\star} r^{r\prime}$, or $r^l \twoheadrightarrow_{\tau^\star} r^{l\prime}$ and $r^r \mapsto\!\!< \alpha >\!\!\!\cdot\; r^{r\prime}$ respectively.

The proof—totalling approximately 60 wrapped lines of code—follows a straightforward method, namely using $\twoheadrightarrow_{\tau^\star}$-dissect and $\twoheadrightarrow_{\not\simeq\tau}$-dissect to tease apart the thread soup, then putting the pieces back together in the right way, depending on what we find inside.

8.5.5 Extracting the Active Thread

A number of the previous results were concerned with transitions from thread soups of the form $r^l \mathbin{+\!\!+} x :: r^r$, where x makes an initial silent transition. This final lemma shows that every silent transition $r \twoheadrightarrow_\tau r'$ is in fact of this form. In other words, we can extract from $r \twoheadrightarrow_\tau r'$ the active thread x and a witness of its transition $x \twoheadrightarrow_{\tau_1} x'$, along with evidence that other threads in r remain unchanged:

$$\begin{aligned}
&\twoheadrightarrow_\tau\text{-extract} \;:\; \forall\, \{r\; r'\} \to r \twoheadrightarrow_\tau r' \to \exists_2\, \lambda\; r^l\; r^r \to \exists_2\, \lambda\; x\; x' \to \\
&\quad r \equiv r^l \mathbin{+\!\!+} x :: r^r \times r' \equiv r^l \mathbin{+\!\!+} x' :: r^r \times x \twoheadrightarrow_{\tau_1} x'
\end{aligned}$$

The proof proceeds simply by induction on the structure of $r \twoheadrightarrow\tau\ r'$.

8.6 The elide-τ Lemma

Given the arsenal assembled in the previous section, the proof of the elide-τ lemma is relatively straightforward:

```
elide-τ : _↠τ_ ⇒ _≈_
elide-τ {r} {s} r↠τs = ♯ r≼s r↠τs & ♯ s≼r where
  s≼r : s ≼ r
  s≼r s′ (⊩-↠ s↠⋆s₀ s₀↠s₁ s₁↠⋆s′)
    = s′ ∧ ⊩-↠ (r↠τs ◁ s↠⋆s₀) s₀↠s₁ s₁↠⋆s′ ∧ ≈′-refl
```

For s≼r the proof is trivial: whatever s does, r can always match it by first making the given $r\twoheadrightarrow\tau s$ transition, after which it can follow s exactly.

In the other direction, we begin by extracting (§8.5.5) the active thread x from $r\twoheadrightarrow\tau s$. This refines r to $r^l +\!\!+ x :: r^r$, which allows us to dissect (§8.5.4) $r\mapsto r'$ using x as the pivot:

```
  r≼s : ∀ {r s} → r ↠τ s → r ≼ s
  r≼s r↠τs r′ r⊩r′
    with →τ-extract r↠τs
  r≼s r↠τs r′ r⊩r′
    | rˡ ∧ rʳ ∧ x ∧ x′ ∧ ≡.refl ∧ ≡.refl ∧ x↠τ₁x′
      with ⊩-dissect rˡ rʳ x↠τ₁x′ r⊩r′
```

In the instance where $r\mapsto r'$ happens to already include the $x\twoheadrightarrow\tau_1 x'$ transition, the proof is trivial:

```
  r≼s r↠τs r′ r⊩r′
```

$$
\begin{array}{l}
\quad \mid r^l \ \wedge \ r^r \ \wedge \ x \ \wedge \ x' \ \wedge \ \equiv\text{.refl} \ \wedge \ \equiv\text{.refl} \ \wedge \ x \twoheadrightarrow_{\tau_1} x' \\
\qquad \mid \mathsf{in_L} \ s \mapsto r' \ = \ r' \ \wedge \ s \mapsto r' \ \wedge \ \approx'\text{-refl}
\end{array}
$$

Here, $\mapsto$-dissect provides the witness $s \mapsto r'$ showing that s can transition to r' too, with $r' \approx' r'$ given by reflexivity of $\_\approx'\_$.

Otherwise $r \mapsto r'$ has yet to make the $x \twoheadrightarrow_{\tau_1} x'$ transition. Two alternatives arise, as the non-silent transition could have been on either side of x. Without loss of generalisation suppose this is on the left, in which case $\mapsto$-dissect refines r' to $r^{l\prime} +\!\!+ x :: r^{r\prime}$, and delivers witnesses of $r^l \mapsto\!\!< \ \alpha \ >\!\! r^{l\prime}$ and $r^r \twoheadrightarrow\!\tau^\star \ r^{r\prime}$:

$$
\begin{array}{l}
\mathsf{r}\preccurlyeq\mathsf{s} \ r \twoheadrightarrow_\tau s \ \_ \ r \mapsto r' \\
\quad \mid r^l \ \wedge \ r^r \ \wedge \ x \ \wedge \ x' \ \wedge \ \equiv\text{.refl} \ \wedge \ \equiv\text{.refl} \ \wedge \ x \twoheadrightarrow_{\tau_1} x' \\
\qquad \mid \mathsf{in_R} \ (r^{l\prime} \ \wedge \ r^{r\prime} \ \wedge \ \equiv\text{.refl} \\
\qquad\quad \wedge \ \mathsf{in_L} \ (\mapsto\!\text{-}\!\twoheadrightarrow \ r^l \twoheadrightarrow\!\tau^\star r^l_0 \ r^l_0 \twoheadrightarrow r^l_1 \ r^l_1 \twoheadrightarrow\!\tau^\star r^{l\prime} \ \wedge \ r^r \twoheadrightarrow\!\tau^\star r^{r\prime})) \\
\qquad \textbf{with} \ \twoheadrightarrow\!\not\simeq\tau\text{-append} \ (x' :: r^r) \ r^l_0 \twoheadrightarrow r^l_1 \\
\mathsf{r}\preccurlyeq\mathsf{s} \ r \twoheadrightarrow_\tau s \ \_ \ r \mapsto r' \\
\qquad \mid r^l \ \wedge \ r^r \ \wedge \ x \ \wedge \ x' \ \wedge \ \equiv\text{.refl} \ \wedge \ \equiv\text{.refl} \ \wedge \ x \twoheadrightarrow_{\tau_1} x' \\
\qquad \mid \mathsf{in_R} \ (r^{l\prime} \ \wedge \ r^{r\prime} \ \wedge \ \equiv\text{.refl} \\
\qquad\quad \wedge \ \mathsf{in_L} \ (\mapsto\!\text{-}\!\twoheadrightarrow \ r^l \twoheadrightarrow\!\tau^\star r^l_0 \ r^l_0 \twoheadrightarrow r^l_1 \ r^l_1 \twoheadrightarrow\!\tau^\star r^{l\prime} \ \wedge \ r^r \twoheadrightarrow\!\tau^\star r^{r\prime})) \\
\qquad \mid r^l_0 x' r^r \twoheadrightarrow r^l_1 x' r^r \ \wedge \ [\![r^l_0 \twoheadrightarrow r^l_1]\!] \equiv [\![r^l_0 x' r^r \twoheadrightarrow r^l_1 x' r^r]\!] \\
\qquad\quad \textbf{rewrite} \ [\![r^l_0 \twoheadrightarrow r^l_1]\!] \equiv [\![r^l_0 x' r^r \twoheadrightarrow r^l_1 x' r^r]\!] \\
\qquad = \ r^{l\prime} +\!\!+ x' :: r^{r\prime} \\
\qquad \wedge \ \mapsto\!\text{-}\!\twoheadrightarrow \ (\twoheadrightarrow\!\tau^\star\text{-}+\!\!+ \ r^l \twoheadrightarrow\!\tau^\star r^l_0 \ (\varepsilon \ \{x \ = \ x' :: r^r\})) \ r^l_0 x' r^r \twoheadrightarrow r^l_1 x' r^r \\
\qquad\quad (\twoheadrightarrow\!\tau^\star\text{-}+\!\!+ \ r^l_1 \twoheadrightarrow\!\tau^\star r^{l\prime} \ (\varepsilon \ \{x \ = \ x' :: r^r\}) \ \lhd\!\rhd \\
\qquad\qquad \twoheadrightarrow\!\tau^\star\text{-}+\!\!+ \ (\varepsilon \ \{x \ = \ r^{l\prime}\}) \ (\twoheadrightarrow\!\tau^\star\text{-}+\!\!+ \ (\varepsilon \ \{x \ = \ x' :: []\}) \ r^r \twoheadrightarrow\!\tau^\star r^{r\prime})) \\
\qquad \wedge \ \approx'\text{-}\approx \ (\text{elide-}\tau \ (\twoheadrightarrow\!\tau\text{-prepend} \ r^{l\prime} \ (x \twoheadrightarrow_{\tau_1} x' \ r^{r\prime})))
\end{array}
$$

Note that the earlier $\twoheadrightarrow\!\tau$-extract had established that s is in fact equal to $r^l +\!\!+ x' :: r^r$. Therefore, we can construct a visible transition from s,

$$r^l \mathbin{+\!\!+} x' :: r^r \mathrel{\mapsto\!\!<} \alpha \mathrel{>} r^{l\prime} \mathbin{+\!\!+} x' :: r^{r\prime}$$

by reconstituting the aforementioned $r^l \mathrel{\mapsto\!\!<} \alpha \mathrel{>} r^{l\prime}$ and $r^r \mathrel{\twoheadrightarrow\tau^\star} r^{r\prime}$. The final bisimilarity component of the proof is obtained by coinduction on:

$$r^{l\prime} \mathbin{+\!\!+} x :: r^{r\prime} \mathrel{\twoheadrightarrow\tau} r^{l\prime} \mathbin{+\!\!+} x' :: r^{r\prime}$$

For the case where the non-silent transition is to the right of x, the proof follows the same approach.

Using the transitivity and reflexivity of $\_\approx\_$ we can generalise elide-τ to silent transition sequences, as well as a symmetric variant:

```
elide-τ* : _↠τ*_ ⇒ _≈_
elide-τ* = Star.fold _≈_ ≈-trans ≈-refl ∘ Star.map elide-τ

elide-τ*′ : _↠τ*_ ⇒ flip _≈_
elide-τ*′ = ≈-sym ∘ elide-τ*
```

8.7 Soup Concatenation Preserves Bisimilarity

With the introduction of explicit concurrency in this Fork language, another important lemma used in our compiler correctness proof concerns the result of combining two pairs of bisimilar thread soups. That is, given $r^l \approx s^l$ and $r^r \approx s^r$, concatenating the soups pairwise results in a pair of bisimilar soups, $r^l \mathbin{+\!\!+} r^r \approx s^l \mathbin{+\!\!+} s^r$.

Intuitively, one can appeal to the following reasoning to see why this is true; without loss of generality, we need only consider $r^l \mathbin{+\!\!+} r^r \preccurlyeq s^l \mathbin{+\!\!+} s^r$ as the other direction can be obtained by symmetry. We must show that whatever visible transition $r^l \mathbin{+\!\!+} r^r$ makes, $s^l \mathbin{+\!\!+} s^r$ is able to follow with the same action. If non-silent transition is due to r^l, then the s^l half of $s^l \mathbin{+\!\!+} s^r$ can match it by $r^l \approx s^l$, and vice versa for r^r. Any silent transitions can be bridged using the elide-$\tau^\star$ lemma.

Let us now formalise the above argument. We are given r' and the three transition sequences $r \twoheadrightarrow \tau^\star r_0$, $r_0 \twoheadrightarrow \not\approx_\tau r_1$ and $r_1 \twoheadrightarrow \tau^\star r'$ comprising $r \mapsto\!\!< \alpha >\!\!\mapsto r'$. By r, we actually mean $r^l \mathbin{+\!\!+} r^r$; therefore we can use $\twoheadrightarrow\tau^\star$-split to partition $r \twoheadrightarrow \tau^\star r_0$ into two sequences $r^l \twoheadrightarrow \tau^\star r_0^l$ and $r^r \twoheadrightarrow \tau^\star r_0^r$, which refines r_0 to $r_0^l \mathbin{+\!\!+} r_0^r$:

```
≼-cong₂ : _++_ Preserves₂ _≈_ ⟶ _≈_ ⟶ _≼_
≼-cong₂ {r^l} {s^l} {r^r} {s^r} r^l≈s^l r^r≈s^r r' (⊨-↠ r↠τ⋆r₀ r₀↠≉τr₁ r₁↠τ⋆r')
  with →τ⋆-split r^l r^r r↠τ⋆r₀
≼-cong₂ {r^l} {s^l} {r^r} {s^r} r^l≈s^l r^r≈s^r r' (⊨-↠ r↠τ⋆r₀ r₀↠≉τr₁ r₁↠τ⋆r')
  | r₀^l ∧ r₀^r ∧ ≡.refl ∧ r^l↠τ⋆r₀^l ∧ r^r↠τ⋆r₀^r
  with →≉τ-split r₀^l r₀^r r₀↠≉τr₁
```

Carrying on in the same vein, we use $\twoheadrightarrow\not\approx_\tau$-split on $r_0 \twoheadrightarrow \not\approx_\tau r_1$ to locate which side of $r_0^l \mathbin{+\!\!+} r_0^r$ the non-silent transition comes from. The proofs for both cases are symmetrical, so let us consider just the left instance: $\twoheadrightarrow\not\approx_\tau$-split returns a witness $r_0^l \twoheadrightarrow \not\approx_\tau r_1^l$, and refines r_1 to $r_1^l \mathbin{+\!\!+} r_0^r$:

```
≼-cong₂ {r^l} {s^l} {r^r} {s^r} r^l≈s^l r^r≈s^r r' (⊨-↠ r↠τ⋆r₀ r₀↠≉τr₁ r₁↠τ⋆r')
  | r₀^l ∧ r₀^r ∧ ≡.refl ∧ r^l↠τ⋆r₀^l ∧ r^r↠τ⋆r₀^r
  | inₗ (r₁^l ∧ ≡.refl ∧ r₀^l↠≉τr₁^l ∧ ⟦r₀^l↠τr₁^l⟧≡⟦r₀↠τr₁⟧)
    with →τ⋆-split r₁^l r₀^r r₁↠τ⋆r'
≼-cong₂ {r^l} {s^l} {r^r} {s^r} r^l≈s^l r^r≈s^r _ (⊨-↠ r↠τ⋆r₀ r₀↠≉τr₁ r₁↠τ⋆r')
  | r₀^l ∧ r₀^r ∧ ≡.refl ∧ r^l↠τ⋆r₀^l ∧ r^r↠τ⋆r₀^r
  | inₗ (r₁^l ∧ ≡.refl ∧ r₀^l↠≉τr₁^l ∧ ⟦r₀^l↠τr₁^l⟧≡⟦r₀↠τr₁⟧)
    | r^l' ∧ r^r' ∧ ≡.refl ∧ r₁^l↠τ⋆r^l' ∧ r₀^r↠τ⋆r^r'
    with ≈→≼ r^l≈s^l r^l' (⊨-↠ r^l↠τ⋆r₀^l r₀^l↠≉τr₁^l r₁^l↠τ⋆r^l')
```

Partitioning $r_1 \twoheadrightarrow \tau^\star r'$ along the two sides of r_1 then gives us the witnesses $r_1^l \twoheadrightarrow \tau^\star r^{l\prime}$ and $r_0^r \twoheadrightarrow \tau^\star r^{r\prime}$; the r' argument is refined to $r^{l\prime} \mathbin{+\!\!+} r^{r\prime}$. The transitions $r^l \twoheadrightarrow \tau^\star r_0^l$, $r_0^l \twoheadrightarrow \not\approx_\tau r_1^l$ and $r_1^l \twoheadrightarrow \tau^\star r^{l\prime}$ are just what we need to build a witness of $r^l \mapsto\!\!< \alpha >\!\!\mapsto r^{l\prime}$.

When this is passed to the $r^l \preccurlyeq s^l$ component of $r^l \approx s^l$, we receive back a matching transition $s^l \mapsto\!\!< \alpha >\!\!\mapsto s^{l\prime}$ and a proof $r^{l\prime} \approx' s^{l\prime}$:

```
≼-cong₂ {r^l} {s^l} {r^r} {s^r} r^l≈s^l r^r≈s^r _ (⊨-↠ r↠τ*r₀ r₀↠≉τr₁ r₁↠τ*r′)
  | r₀^l ∧ r₀^r ∧ ≡.refl ∧ r^l↠τ*r₀^l ∧ r^r↠τ*r₀^r
  | inₗ (r₁^l ∧ ≡.refl ∧ r₀^l↠≉τr₁^l ∧ [[r₀^l↠τr₁^l]]≡[[r₀↠τr₁]])
    | r^l′ ∧ r^r′ ∧ ≡.refl ∧ r₁^l↠τ*r^l′ ∧ r₀^r↠τ*r^r′
    | s^l′ ∧ s^l⊨s^l′ ∧ r^l′≈′s^l′ rewrite [[r₀^l↠τr₁^l]]≡[[r₀↠τr₁]]
      = s^l′ ++ s^r ∧ ⊨-append s^r s^l⊨s^l′
      ∧ ≈′-cong₂ r^l′≈′s^l′ (≈′-≈ (≈-trans
        (elide-τ*′ (r^r↠τ*r₀^r ◅◅ r₀^r↠τ*r^r′)) r^r≈s^r))
```

Finally, constructing a witness of $s^l +\!\!+ s^r \mapsto\!\!< \alpha >\!\!\mapsto s^{l\prime} +\!\!+ s^r$ from the above pieces satisfies one half of the goal; the other half of $r^{l\prime} +\!\!+ r^{r\prime} \approx s^{l\prime} +\!\!+ s^r$ is obtained coinductively, making use of elide-$\tau^{\star\prime}$ and transitivity of $\_\approx\_$ in the process.

The full result that soup contatenation preserves bisimilarity simply requires two symmetric invocations of ≼-cong₂:

```
≈-cong₂ : _++_ Preserves₂ _≈_ ⟶ _≈_ ⟶ _≈_
≈-cong₂ r^l≈s^l r^r≈s^r
  = ♯ ≼-cong₂          r^l≈s^l          r^r≈s^r
  & ♯ ≼-cong₂ (≈-sym r^l≈s^l) (≈-sym r^r≈s^r)
```

This makes $\_\approx\_$ a congruence relation on the monoid (List Combined, $+\!\!+$, []).

We can now fulfill our obligation (§8.4) of providing an interpretor for the syntax of $\_\approx'\_$, as follows:

```
≈′→≈ : _≈′_ ⇒ _≈_
≈′→≈ (≈′-≈ r≈s)          = r≈s
≈′→≈ (≈′-sym r≈′s)       = ≈-sym (≈′→≈ r≈′s)
```

```
≈′→≈ (≈′-trans r≈′s s≈′t)       = ≈-trans (≈′→≈ r≈′s) (≈′→≈ s≈′t)
≈′→≈ (≈′-cong₂ r^l≈′s^l r^r≈′s^r) = ≈-cong₂ (≈′→≈ r^l≈′s^l) (≈′→≈ r^r≈′s^r)
```

8.8 Compiler Correctness

The compiler correctness property for our Fork language is essentially the same as that of the Zap language, but on singleton thread soups rather than combined machines:

```
correctness : ∀ e c σ → ⟨ e , ⟨ c , σ ⟩ ⟩ :: [] ≈ ⟨ ⟨ compile e c , σ ⟩ ⟩ :: []
```

There is no need to generalise over an arbitrary thread soup, since the ≈-cong$_2$ lemma of §8.7 allows us to concatenate as many pairs of bisimilar thread soups as is required.

The proof comprises of two parts, each showing one direction of the bisimulation. Proceeding by case analysis on the visible transition, the fork≼FORK part first shows that fork e cannot make a non-silent transition:

```
correctness (fork e) c σ = ♯ fork≼FORK & ♯ FORK≼fork where
  fork≼FORK : ⟨ fork e , ⟨ c , σ ⟩ ⟩ :: [] ≼ ⟨ ⟨ FORK (compile e []) :: c , σ ⟩ ⟩ :: []
  fork≼FORK s′ (⊨-↠ ((._ ∧ () ∧ ↠-↦ ↦-fork) ◁ _) _ _)
  fork≼FORK s′ (⊨-↠ ((._ ∧ α≃τ ∧ ↠-preempt ()) ◁ _) _ _)
  fork≼FORK s′ (⊨-↠ ε (._ ∧ α≄τ ∧ ↠-preempt ()) s₀↠τ⋆s′)
```

The two ↠-preempt clauses correspond to the fact that the empty soup [] cannot make any transitions at all. In the case of a non-silent ↦-fork transition, the expression side transitions to,

```
⟨ # 0 , ⟨ c , σ ⟩ ⟩ :: ⟨ e , ⟨ [] , [] ⟩ ⟩ :: []
```

while the virtual machine follows by the ↣-FORK rule:

```
  fork≼FORK s′ (⊨-↠ ε (._ ∧ α≄τ ∧ ↠-↦ ↦-fork) s₀↠τ⋆s′)
    = ⟨ ⟨ c , 0 :: σ ⟩ ⟩ :: ⟨ ⟨ compile e [] , [] ⟩ ⟩ :: []
```

```
∧ ⤇-↠ ε (⁺ _ ∧ (λ ()) ∧ ↠-↣ ↣-FORK) ε
∧ ≈′-≈
  (begin
    s′
  ≈⟨ elide-τ* s₀↠τ*s′ ⁻¹⟩
    ⟨ # 0 , ⟨ c , σ ⟩ ⟩ :: ⟨ e , ⟨ [] , [] ⟩ ⟩ :: []
  ≈⟨ ≈-cong₂ (elide-τ (↠τ-switch {[]})) (correctness e [] []) ⟩
    ⟨ ⟨ c , 0 :: σ ⟩ ⟩ :: ⟨ ⟨ compile e [] , [] ⟩ ⟩ :: []
  □)
```

The two reducts of the original threads are bisimilar by the elide-τ lemma, while bisimilarity of the two spawned threads is obtained from the induction hypothesis on e. Finally, we use the $\approx$-cong$_2$ lemma to combine these results, to give the overall bisimilarity of the two thread soups.

In the opposite direction of FORK$\preccurlyeq$fork, the proof follows the same steps, with the first clause showing that the FORK instruction cannot be silent:

```
FORK≼fork : ⟨ ⟨ FORK (compile e []) :: c , σ ⟩ ⟩ :: [] ≼ ⟨ ⟨ fork e , ⟨ c , σ ⟩ ⟩ :: []
FORK≼fork s′ (⤇-↠ ((._ ∧ () ∧ ↠-↣ ↣-FORK) ◁ _) _ _)
FORK≼fork s′ (⤇-↠ ((._ ∧ α≃τ ∧ ↠-preempt ()) ◁ _) _ _)
FORK≼fork s′ (⤇-↠ ε (._ ∧ α≄τ ∧ ↠-preempt ()) s₀↠τ*s′)
```

When the virtual machine makes a transition by the $\rightarrowtail$-FORK rule, the expression follows with $\mapsto$-fork,

```
FORK≼fork s′ (⤇-↠ ε (._ ∧ α≄τ ∧ ↠-↣ ↣-FORK) s₀↠τ*s′)
  = ⟨ # 0 , ⟨ c , σ ⟩ ⟩ :: ⟨ e , ⟨ [] , [] ⟩ ⟩ :: []
  ∧ ⤇-↠ ε (⁺ _ ∧ (λ ()) ∧ ↠-↦ ↦-fork) ε
  ∧ ≈′-≈
    (begin
```

```
  s′
≈⟨ elide-τ⋆′ s₀↠τ⋆s′ ⟩
  ⟨ ⟨ c , 0 :: σ ⟩ ⟩ :: ⟨ ⟨ compile e [] , [] ⟩ ⟩ :: []
≈⟨ ≈-cong₂ (elide-τ (↠τ-switch {[]})) (correctness e [] []) ⁻¹⟩
  ⟨ # 0 , ⟨ c , σ ⟩ ⟩ :: ⟨ e , ⟨ [] , [] ⟩ ⟩ :: []
□)
```

and we obtain the bisimilarity of the two pairs of threads via the ≈-cong$_2$ lemma as before. The remaining clauses of correctness deal with # m and $a \oplus b$, following the same steps as that for the Zap language, modulo cosmetic changes to account for the thread soup. This completes the compiler correctness proof for the Fork language.

8.9 Conclusion

We have demonstrated that our previously introduced technique of showing bisimilarity between combined machines does indeed scale to the explicitly concurrent Fork language, modelled as a simple 'thread soup' of combined machines. The elide-τ lemma was updated for this context using our arsenal of thread soup lemmas, while the result that soup concatenation preserves bisimilarity meant that we could phrase our compiler correctness statement on singleton thread soups. As a result, we were able to reuse most of the correctness proof for the Zap language, with only the fork e case requiring further attention.

Chapter 9

Transaction Correctness

The previous chapter scaled our proof technique to a language with explicit concurrency. In this chapter, we now consider a language with transactions. In order to reconcile the stop-the-world and log-based semantics, we make two simplifications to our approach. First of all, we replace concurrency with arbitrary interference by an external agent, and secondly we replace the compiler and virtual machine with a direct log-based transactional semantics for the source language.

9.1 The Atomic Language

9.1.1 Syntax

For the Atomic language, we migrate to a two-level syntax in a similar manner to the Tran and Proc types of chapter 5. On the transaction level, we extend our base language of natural numbers and addition with the read and write keywords for manipulating transactional variables,

```
data Expression′ : Set where
  #_   : (m : ℕ) → Expression′
  _⊕_  : (a b : Expression′) → Expression′
```

```
read  : (v : Variable) → Expression′
write : (v : Variable) (e : Expression′) → Expression′
```

while the 'IO' level is extended with an atomic keyword that runs a transaction of type Expression′[1]:

```
data Expression : Set where
  #_ : (m : ℕ) → Expression
  _⊕_ : (a b : Expression) → Expression
  atomic : (e : Expression′) → Expression
```

Note that we have not included the fork construct that spawns additional threads as we did in chapters 5 and 8:

```
fork : (e : Expression) → Expression
```

The reason for this is that the presence of fork turned out to significantly complicate the formal reasoning process, so we investigated a simpler approach. First we replace concurrency with a 'mutate' rule that can change the heap at any time during a transaction, which simulates the worst possible concurrent environment, in a similar manner to the worst-case interrupt rule of [HW07]. Secondly we replace the compiler and virtual machine with a direct log-based transactional semantics for the source language, which makes the proof more manageable.

9.1.2 Heaps and Variables

Recall that previously in chapter 5, we modelled the heap as a total map from a fixed set of variable names to their values, initialised to zero. In Agda, we can realise this using the indexed Vec type (§6.1.3) from the standard library. As our proof is to be

[1]Throughout this chapter, I adopt the convention of using a ′ to identify types that are associated with the transaction level of the Atomic language.

independent of the heap size—rather than parametrising the entire proof by it—we simply postulate a number |Heap|,

```
postulate |Heap| : ℕ
```

with the Heap type defined as follows:

```
Heap : Set
Heap = Vec ℕ |Heap|
```

Correspondingly, a Variable is just a synonym for the finite set (§6.1.3) with |Heap| distinct elements:

```
Variable : Set
Variable = Fin |Heap|
```

9.1.3 Stop-the-World Semantics

Now we have enough machinery to describe the high-level stop-the-world semantics for the Atomic language. Due to the two-level stratification of the language, this involves two separate transitions for Expression′ and Expression. The first of these is _↦′_, defined on the transaction level between pairs of Heaps and Expression′s. We begin with the familiar rules for left-biased addition:

```
data _↦′_ : Rel (Heap × Expression′) where
  ↦′-⊕ℕ : ∀ {h m n} →
    h , # m ⊕ # n ↦′ h , # (m + n)
  ↦′-⊕L : ∀ {h h′ a a′} b →
    (a↦a′ : h , a ↦′ h′ , a′) →
    h , a ⊕ b ↦′ h′ , a′ ⊕ b
  ↦′-⊕R : ∀ {h h′ b b′} m →
```

```
  (b↦b′ : h ,          b ↦′ h′ ,          b′) →
          h , # m ⊕ b ↦′ h′ , # m ⊕ b′

↦′-read : ∀ h v →
  h , read v ↦′ h , # h [ v ]

↦′-write_ℕ : ∀ {h v m} →
  h , write v (# m) ↦′ h [ v ]:= m , # m

↦′-write_E : ∀ {h e h′ e′ v} →
  (e↦e′ : h ,          e ↦′ h′ ,          e′) →
          h , write v e ↦′ h′ , write v e′
```

Here the ↦′-read and ↦′-write$_{\mathbb{N}}$ rules refer directly to the heap, while ↦′-write$_{\mathsf{E}}$ effects the reduction of the sub-expression argument to write. We write _↦⋆_ for the reflexive, transitive closure of _↦′_, defined using the Star type:

```
_↦⋆_ : Rel (Heap × Expression′)
_↦⋆_ = Star _↦′_
```

On the 'IO' level, transitions are labelled with a choice of actions,

```
data Action : Set where
  τ ⊞ ☢ : Action
```

where τ is the silent action, ⊞ corresponds to the addition operation, and ☢ indicates the successful completion of a transaction. These simple observable actions make it possible to define a notion of bisimilarity for the stop-the-world and log-based semantics, where there need not be a one-to-one correspondence of transition rules on each side.

The labelled transition is defined as follows, with the first three rules corresponding to the familiar left-biased addition:

```
data _▷_↦_ : Action → Rel (Heap × Expression) where
  ↦-⊕ℕ : ∀ {h m n} →
    ⊞ ▷ h , # m ⊕ # n ↦ h , # (m + n)
  ↦-⊕R : ∀ {α h h′ b b′} m →
    (b↦b′ : α ▷ h ,      b ↦ h′ ,      b′) →
            α ▷ h , # m ⊕ b ↦ h′ , # m ⊕ b′
  ↦-⊕L : ∀ {α h h′ a a′} b →
    (a↦a′ : α ▷ h , a      ↦ h′ , a′) →
            α ▷ h , a ⊕ b  ↦ h′ , a′ ⊕ b
```

The ↦-atomic rule implements a stop-the-world semantics for atomic blocks by taking a reduction sequence $e{\mapsto}^{\star}m$ on the transaction level, and encapsulating it in a single step:

```
  ↦-atomic : ∀ {h e h′ m} →
    (e↦⋆m : h , e ↦⋆ h′ , # m) →
      ☢ ▷ h , atomic e ↦ h′ , # m
  ↦-mutate : ∀ h′ {h e} →
    τ ▷ h , atomic e ↦ h′ , atomic e
```

Since there are no explicit threads in the Atomic language, we introduce a silent ↦-mutate rule to allow the heap to change at any time, which reflects the idea that a concurrent thread may modify the heap while the current thread is running. The above rule implements the worst possible case in which the heap h can be replaced by a completely different heap h'. For simplicity, note that ↦-mutate is limited to contexts where the expression is of the form atomic e, as this is the only construct that interacts with the heap. We shall later examine how a corresponding rule in the log-based semantics allows the heap to mutate during a transaction.

9.1.4 Transaction Logs and Consistency

Before we give the log-based semantics for *atomic* blocks, let us first define what transaction logs are. Recall from chapter 5 that we modelled them as partial maps from variables to numbers, where an entry for a variable exists only when it has been read from or written to. We take a similar approach to Heaps, using vectors of Maybe $\mathbb{N}$[2] initialised to ∘:

```
data Logs : Set where
  constructor _&_
  field
    ρ ω : Vec (Maybe ℕ) |Heap|

∅ : Logs
∅ = Vec.replicate ∘ & Vec.replicate ∘
```

The ρ and ω fields of a Logs record correspond to the read and write logs of chapter 5, and are used in an identical manner to keep track of variables during a running transaction. Let us quickly review the rules for log-based writes and reads in the the context of the current chapter.

Writing to a transaction variable is the most straightforward of the two operations, and is implemented by the following Write function that returns a new pair of logs with the entry for v in ω updated to the new value m.

```
Write : Logs → Variable → ℕ → Logs
Write (ρ & ω) v m = ρ & ω [ v ]:= • m
```

The Read function on the other hand takes a heap, a pair of logs and a variable as arguments, and returns a potentially modified read log along with the in-transaction value of the variable:

[2]For aesthetic reasons I have renamed nothing and just of Maybe to ∘ and • respectively.

```
Read : Heap → Logs → Variable → Logs × ℕ
Read h (ρ & ω) v with ω [ v ]
... | • m = ρ & ω , m
... | ∘ with ρ [ v ]
...   | • m = ρ & ω , m
...   | ∘ = ρ [ v ]:= • m & ω , m where m = h [ v ]
```

If a variable has been written to according to ω, we immediately return the new value. Otherwise we consult the read log ρ: if a previous value for v exists, we return that. In both cases the transaction logs remain unchanged. Only when no cached value for v exists—that is, when we are reading a variable for the first time—do we update the read log ρ with the value of v from the current heap. Note that if a variable is written to before it is read, the corresponding read log entry will never be filled.

On reaching the end of a transaction we either commit or roll back, depending on whether the values of the variables gleaned from the heap during the transaction are consistent with their corresponding values at the end. That is, all values recorded in the read log must match those currently in the heap for corresponding variables. The following predicate allows us to state this succinctly:

```
Consistent : Heap → Logs → Set
Consistent h (ρ & _) = ∀ v m → ρ [ v ] ≡ • m → h [ v ] ≡ m
```

A read log ρ is consistent with the heap h precisely when all non-empty entries in ρ have the same values as the corresponding entries in h. Note that a transactional variable that is written to before being read from will not have a corresponding entry in ρ; this is acceptable since its original value in the heap could not possibly have influenced the behaviour of the transaction. Naturally the empty log $\varnothing$ is consistent with any heap:

```
∅-Consistent : ∀ {h} → Consistent h ∅
```

```
∅-Consistent v m rewrite Vec.lookup∘replicate v (∘ : Maybe ℕ) = λ ()
```

The above uses the Vec.lookup∘replicate function to obtain a proof that the entry for v in the newly-initialised read log is $\circ$, in which case we can use an absurd lambda to eliminate the $\circ \equiv \bullet\ m$ argument.

The Dec P type corresponds to the decidability of some proposition P. It has two constructors yes and no, carrying the appropriate evidence in either case:

```
data Dec (P : Set) : Set where
  yes : (p  :  P) → Dec P
  no  : (¬p : ¬ P) → Dec P
```

Thus an element of Dec P is strictly more informative than a boolean value. Using this, we can give a decision procedure for whether a given heap and read log are indeed consistent. This is implemented below as Consistent?, via the dec helper that decides consistency for one particular variable:

```
Consistent? : (h : Heap) (l : Logs) → Dec (Consistent h l)
Consistent? h (ρ & ω) = Dec.map′ Vec.Pointwise.app Vec.Pointwise.ext
                                 (Vec.Pointwise.decidable dec h ρ) where
  dec : (hᵥ : ℕ) (ρᵥ : Maybe ℕ) → Dec (∀ m → ρᵥ ≡ • m → hᵥ ≡ m)
  dec hᵥ ∘ = yes (λ m ())
  dec hᵥ (• n) with hᵥ ≟ℕ n
  ... | yes hᵥ≡n rewrite hᵥ≡n = yes (λ m → •-inj)
  ... | no hᵥ≢n = no (λ p → hᵥ≢n (p n ≡.refl))
```

The library fuctions Dec.map′ and Vec.Pointwise.decidable are used to generalise the pointwise decision procedure over all variables.

Finally when a transaction is ready to commit, we can use the Update function to commit the contents of the write log to the heap:

```
Update-lookup : Heap → Logs → Variable → ℕ
Update-lookup h (ρ & ω) v = maybe id (h [ v ]) (ω [ v ])

Update : Heap → Logs → Heap
Update h l = Vec.tabulate (Update-lookup h l)
```

This is implemented using the library function Vec.tabulate that takes a function that gives the new value for each index or variable. We have factored out Update-lookup in order to leverage existing proofs in the standard library.

9.1.5 Log-Based Semantics

The log-base semantics makes transitions between pairs of Logs and Expression′ rather than operating directly on a Heap. We can still read from the heap, but it is never modified by the following rules. The first three rules corresponding to left-biased addition should look familiar:

```
data _⊢_⇀′_ (h : Heap) : Rel (Logs × Expression′) where
  ⇀′-⊕ℕ : ∀ {l m n} →
    h ⊢ l , # m ⊕ # n ⇀′ l , # (m + n)
  ⇀′-⊕R : ∀ {l b l′ b′} m →
    (b⇀b′ : h ⊢ l ,      b ⇀′ l′ ,      b′) →
            h ⊢ l , # m ⊕ b ⇀′ l′ , # m ⊕ b′
  ⇀′-⊕L : ∀ {l a l′ a′} b →
    (a⇀a′ : h ⊢ l , a    ⇀′ l′ , a′) →
            h ⊢ l , a ⊕ b ⇀′ l′ , a′ ⊕ b
```

The ⇀′-read rule reduces a read v expression to the value of v using the Read function defined in the previous section, potentially also resulting in a new log:

```
  ⇀′-read : ∀ l v → let l′m = Read h l v in
    h ⊢ l , read v ⇀′ fst l′m , # snd l′m
```

```
  ↣′-write𝕟 : ∀ {l v m} →
    h ⊢ l , write v (# m) ↣′ Write l v m , # m
  ↣′-writeE : ∀ {l e l′ e′ v} →
    (e↣e′ : h ⊢ l , e ↣′ l′ , e′) →
    h ⊢ l , write v e ↣′ l′ , write v e′
```

The ↣′-write$_{\mathbb{N}}$ rule updates the write log via the Write helper when the expression argument to write is just a number, while ↣′-write$_{\mathsf{E}}$ effects the reduction of e in the same manner as the stop-the-world semantics.

We write _⊢_↣′*_ for the reflexive, transitive closure of _⊢_↣′_ under the same heap, again defined using the Star type:

```
_⊢_↣′*_ : Heap → Rel (Logs × Expression′)
_⊢_↣′*_ h = Star (_⊢_↣′_ h)
```

For the 'IO' level of this log-based semantics, we define a transition _▷_↣_ between triples of heaps, transaction states and expressions, labelled with the same Actions we used earlier. During a running transaction, the state comprises of the original expression and the transaction Logs; otherwise it is empty:

```
TState : Set
TState = Maybe (Expression′ × Logs)
```

The rules for addition are identical to those of _▷_↦_:

```
data _▷_↣_ : Action → Rel (Heap × TState × Expression) where
  ↣-⊕𝕟 : ∀ {h m n} →
    ⊞ ▷ h , ∘ , # m ⊕ # n ↣ h , ∘ , # (m + n)
  ↣-⊕R : ∀ {α h t b h′ t′ b′} m →
    (b↣b′ : α ▷ h , t , b ↣ h′ , t′ , b′) →
    α ▷ h , t , # m ⊕ b ↣ h′ , t′ , # m ⊕ b′
```

```
⤇-⊕L : ∀ {α h t a h′ t′ a′} b →
  (a⤇a′ : α ▷ h , t , a      ⤇ h′ , t′ , a′) →
          α ▷ h , t , a ⊕ b ⤇ h′ , t′ , a′ ⊕ b
```

Next we move on to the transaction rules: when the expression to be reduced is of the form atomic e and we have yet to enter the transaction, the ⤇-begin rule sets up the restart expression to e and initialises the transaction logs to $\varnothing$:

```
⤇-begin : ∀ {h e} →
  τ ▷ h , ○ , atomic e ⤇ h , • (e , ∅) , atomic e
```

The second ⤇-step rule allows us to make a single $\_\vdash\_\rightarrowtail'\_$ transition on the transaction level; note that the heap h does not change:

```
⤇-step : ∀ {h r l e l′ e′} →
  (e⤇e′ : h ⊢ l , e ⤇′ l′ , e′) →
  τ ▷ h , • (r , l) , atomic e ⤇ h , • (r , l′) , atomic e′
```

While the Atomic language does not contain explicit parallelism, we can model interference using a ⤇-mutate rule that changes to an arbitrary heap h' at any time during a transaction:

```
⤇-mutate : ∀ h′ {h t e} →
  τ ▷ h , • t , atomic e ⤇ h′ , • t , atomic e
```

Finally we come to the ⤇-abort and ⤇-commit rules, one of which applies when the transactional expression has reduced down to a number:

```
⤇-abort : ∀ {h r l m} (¬cons : ¬ Consistent h l) →
  τ ▷ h , • (r , l) , atomic (# m) ⤇ h , • (r , ∅) , atomic r
⤇-commit : ∀ {h r l m} (cons : Consistent h l) →
  ☢ ▷ h , • (r , l) , atomic (# m) ⤇ Update h l , ○ , # m
```

Both rules carry proof of the consistency or otherwise of the log l with respect to h. While this is not technically necessary and we could make do with a single rule—as consistency is decidable—having two rules labelled with distinct *Actions* makes later proofs easier to work with.

In any case if we do have consistency, we commit the transaction by applying the write log to the heap using the Update function, setting the transaction state to $\circ$, and reducing atomic (# m) to # m. Otherwise the $\rightarrowtail$-abort rule applies, and we silently restart the transaction by resetting the transaction state and the expression.

9.2 Combined Semantics and Bisimilarity

9.2.1 Combined Semantics

In a similar spirit to the combined machines of previous chapters, let us define a Combined type that allows us to select which of our two semantics to use to interpret some given expression:

```
data Combined : Set where
  ↦:  : Combined
  ⇀:_ : (t : TState) → Combined
```

The ↦: constructor indicates that the associated expression should be interpreted according to the stop-the-world semantics, while ⇀:_ implies the log-based semantics. The latter is also used to carry around the transaction state. How this works can be seen in the definition of _▷_⇴_ below:

```
data _▷_⇴_ (α : Action) : Rel (Heap × Combined × Expression) where
  ⇴-↦ : ∀ {h e h′ e′} →
    (e↦e′ : α ▷ h ,      e ↦ h′ ,      e′) →
            α ▷ h , ↦: , e ⇴ h′ , ↦: , e′
```

```
  ↠-↣ : ∀ {h t e h′ t′ e′} →
    (e↣e′ : α ▷ h , t , e ↣ h′ , t′ , e′) →
            α ▷ h , ↣: t , e ↠ h′ , ↣: t′ , e′
```

This combined transition is essentially a disjoint union of $\_\triangleright\_\mapsto\_$ and $\_\triangleright\_\rightarrowtail\_$. We will write $\_\twoheadrightarrow^{\star}_{\tau}\_$ for an arbitrary sequence of silent τ transitions:

```
_↠⋆τ_ : Rel (Heap × Combined × Expression)
_↠⋆τ_ = Star (_▷_↠_ τ)
```

Finally, let us define a visible transition as a sequence of silent τ transitions followed by a single non-τ transition:

```
data _▷_⤇_ (α : Action) (x x″ : Heap × Combined × Expression) : Set where
  constructor ⤇:
  field
    {h′} : Heap
    {c′} : Combined
    {e′} : Expression
    α≢τ  : α ≢ τ
    e↠⋆τe′ : x ↠⋆τ h′ , c′ , e′
    e′↠e″ : α ▷ h′ , c′ , e′ ↠ x″
```

The visible transition above is basically the same idea as that of the Fork and Zap languages of the previous two chapters, but without a second sequence of τ-transitions after the main $e' \twoheadrightarrow e''$ transition.

9.2.2 Bisimilarity of Semantics

The definition of bisimilarity differs in some details compared with the previous two chapters. Since we are just comparing two different semantics for the same expression, we define one half of bisimilarity as follows:

```
_⊢_≼_ : Heap × Expression → Rel Combined
h , e ⊢ x ≼ y = ∀ {h′ x′ e′ α} →
  (x⇛x′ : α ▷ h , x , e ⇛ h′ , x′ , e′) →
  ∃ λ y′ → α ▷ h , y , e ⇛ h′ , y′ , e′ × (h′ , e′ ⊢ x′ ≈ y′)
```

That is given a heap h and expression e, whenever it can make a visible transition to h' and e' under the semantics represented by x, an equivalent transition to the same h' and e' exists under y, such that x' and y' are also bisimilar for h' and e'.

Bisimilarity is then defined as a pair of coinductive _⊢_≼_ relations:

```
data _⊢_≈_ (he : Heap × Expression) (x y : Combined) : Set where
  constructor _&_
  field
    ≈→≼ : ∞ (he ⊢ x ≼ y)
    ≈→≽ : ∞ (he ⊢ y ≼ x)
```

Utilising the same proofs as before, we can show that _⊢_≈_ is reflexive, symmetric, and transitive on Combined, given a heap and an expression:

```
≼-refl : {he : Heap × Expression} → Reflexive (_⊢_≼_ he)
≼-refl x⇛x′ = _ , x⇛x′ , ≈-refl

≈-refl : {he : Heap × Expression} → Reflexive (_⊢_≈_ he)
≈-refl = ♯ ≼-refl & ♯ ≼-refl

≈-sym : {he : Heap × Expression} → Symmetric (_⊢_≈_ he)
≈-sym (x≼y & y≼x) = y≼x & x≼y

≼-trans : {he : Heap × Expression} → Transitive (_⊢_≼_ he)
≼-trans x≼y y≼z x⇛x′ with x≼y x⇛x′
... | y′ , y⇛y′ , x′≈y′ with y≼z y⇛y′
...   | z′ , z⇛z′ , y′≈z′ = z′ , z⇛z′ , ≈-trans x′≈y′ y′≈z′
```

```
≈-trans : {he : Heap × Expression} → Transitive (_⊢_≈_ he)
≈-trans (x≼y & y≼x) (y≼z & z≼y) =
  ♯ ≼-trans (♭ x≼y) (♭ y≼z) & ♯ ≼-trans (♭ z≼y) (♭ y≼x)
```

In this chapter, we managed to avoid the use of an embedded language (the _≈′_ relation of previous chapters) to convince Agda that our coinductive definitions are properly constructor-guarded, and hence productive. That said, we will need to manually inline several uses of ≈-sym in later proofs, as the guardedness checker cannot see through function definitions yet at the time of writing.

9.2.3 Definition of Correctness

Having accumulated enough machinery, we can now give a definition of correctness of the log-based semantics, which is simply the following:

```
correct : ∀ h e → h , e ⊢ ↦: ≈ ⟼: ∘
```

That is for any heap h and expression e, the stop-the-world semantics (as represented by ↦:) and the log-based semantics with an empty transaction state (proxied by ⟼: ∘) are bisimilar up to visible transitions.

9.3 Reasoning Transactionally

In this section, we will cover some useful lemmas concerning heaps and transaction logs that are used to show that the stop-the-world and log-based transaction semantics coincide.

9.3.1 Consistency-Preserving Transitions

First of all, recall that when the log-based semantics needs to read a variable v and it is not present in either of the read and write logs, we update the read log with the

value of v from the heap. The following lemma shows that this operation preserves log consistency:

```
Read-Consistent : ∀ {h} l v → Consistent h l →
  Consistent h (Logs.ρ l [ v ]:= • (h [ v ]) & Logs.ω l)
Read-Consistent {h} (ρ & ω) v cons v′ m with v′ ≟Fin v
... | yes v′≡v rewrite v′≡v | Vec.lookup∘update v ρ (• (h [ v ])) = •-inj
... | no v′≢v rewrite Vec.lookup∘update′ v′≢v ρ (• (h [ v ])) = cons v′ m
```

We have η-expanded Read-Consistent with a second variable v' and m taken by the resulting Consistent type, and need to show that $\rho\,[\,v'\,] \equiv \bullet\, m \to h\,[\,v'\,] \equiv m$.

There are two cases to consider, depending on whether v' coincides with the variable v whose read log entry is being updated. If they are indeed the same, we can use Vec.lookup∘update to show that the updated read log entry is $\bullet\, h_v$, in which case the goal of $\bullet\, h\,[\,v'\,] \equiv \bullet\, m \to h\,[\,v'\,] \equiv m$ can be satisfied by injectivity of $\bullet$. When v and v' correspond to different variables, Vec.lookup∘update′ gives us a proof that the read log entry for v' remains unchanged, and the *cons* argument suffices.

Using the above result, we can demonstrate that any transaction transition under the log-based semantics preserves consistency:

```
⇀′-Consistent : ∀ {h l e l′ e′} → Consistent h l →
  h ⊢ l , e ⇀′ l′ , e′ → Consistent h l′
⇀′-Consistent cons ⇀′-⊕ℕ = cons
⇀′-Consistent cons (⇀′-⊕R m b⇀b′) = ⇀′-Consistent cons b⇀b′
⇀′-Consistent cons (⇀′-⊕L b a⇀a′) = ⇀′-Consistent cons a⇀a′
⇀′-Consistent cons (⇀′-writeE e⇀e′) = ⇀′-Consistent cons e⇀e′
⇀′-Consistent cons ⇀′-writeℕ = cons
⇀′-Consistent cons (⇀′-read l v) with Logs.ω l [ v ]
... | • m = cons
```

```
... | ∘ with Logs.ρ l [ v ]
...   | • m = cons
...   | ∘ = Read-Consistent l v cons
```

The proof proceeds by induction on the structure of the reduction rules, making use of the Read-Consistent lemma when the read log changes. Naturally, we can extend the above to an arbitrary _⊢_↣′*_ sequence by folding over it:

```
↣′*-Consistent : ∀ {h l e l′ e′} → Consistent h l →
  h ⊢ l , e ↣′* l′ , e′ → Consistent h l′
↣′*-Consistent {h} {l} {e} =
  Star.gfoldl fst (const (Consistent h)) ↣′-Consistent {i = l , e}
```

In the opposite direction, we can show a pair of similar but slightly more general consistency-preservation lemmas. This extra generality in fact turns out to be crucial to our later proofs. The Read-Consistent′ lemma shares an analogous structure to that of Read-Consistent, but requires an extra argument showing that the pre-transition read log entry for v is empty:

```
Read-Consistent′ : ∀ {h n} l v → Logs.ρ l [ v ] ≡ ∘ →
  Consistent h (Logs.ρ l [ v ]:= • n & Logs.ω l) → Consistent h l
Read-Consistent′ {h} {n} (ρ & ω) v ρv≡∘ cons′ v′ m with v′ ≟Fin v
... | yes v′≡v rewrite v′≡v | ρv≡∘ = λ ()
... | no v′≢v rewrite ≡.sym (Vec.lookup∘update′ v′≢v ρ (• n)) = cons′ v′ m
```

As before, there are two alternatives: when v' coincides with the variable v whose read log entry is being updated, we use the $\rho_v{\equiv}\circ$ argument to rewrite the goal to $\circ \equiv \bullet\ m \to h\ [\ v\ ] \equiv m$, which is then discharged with an absurd λ. This is essentially making use of the fact that that each read log entry is only ever updated once, from $\circ$ to $\bullet$. When v' differs, the $cons'$ argument suffices.

In the yes case of Read-Consistent, we required that the post-transition read log entry for v be $\bullet\ (h\ [\ v\ ])$. Since the corresponding case here is absurd, this is no longer necessary, and the proof can be generalised to any $\bullet\ n$. This means that the heap h under which the logs and expression make their transition need not be the same as the heap h' with which l and l' are consistent in the following lemma:

```
⇀↽′-Consistent′ : ∀ {h h′ l e l′ e′} → Consistent h′ l′ →
  h ⊢ l , e ⇀↽′ l′ , e′ → Consistent h′ l
⇀↽′-Consistent′ cons′ ⇀↽′-⊕ℕ = cons′
⇀↽′-Consistent′ cons′ (⇀↽′-⊕R m b⇀↽b′) = ⇀↽′-Consistent′ cons′ b⇀↽b′
⇀↽′-Consistent′ cons′ (⇀↽′-⊕L b a⇀↽a′) = ⇀↽′-Consistent′ cons′ a⇀↽a′
⇀↽′-Consistent′ cons′ (⇀↽′-writeE e⇀↽e′) = ⇀↽′-Consistent′ cons′ e⇀↽e′
⇀↽′-Consistent′ cons′ ⇀↽′-writeℕ = cons′
⇀↽′-Consistent′ cons′ (⇀↽′-read (ρ & ω) v) with ω [ v ]
... | • m = cons′
... | ○ with ρ [ v ] | ≡.inspect (_[_] ρ) v
...   | • m | _ = cons′
...   | ○   | ⟦ ρᵥ≡○ ⟧ = Read-Consistent′ (ρ & ω) v ρᵥ≡○ cons′
```

This follows an identical structure to ⇀↽′-Consistent, with the only difference being the use of the ≡.inspect idiom to obtain a proof of $\rho\ [\ v\ ] \equiv \circ$.

9.3.2 Heaps and Logs Equivalence

Recall that a pair of read and write logs is used to give an local view of the heap during a running transaction. For our correctness proof, it will be convenient to define a predicate stating that the view of the heap during the transaction—that is, h overlaid with read and write logs—is equivalent to another heap h' that is accessed directly using the stop-the-world semantics:

```
Equivalent : Heap → Logs → Heap → Set
Equivalent h l h′ = snd ∘ Read h l ≗ _[_] h′
```

We write $f \mathrel{\mathring{=}} g$ to mean pointwise equality of f and g, and is a synonym for $\forall\ x \to f\ x \equiv g\ x$. In other words, Read $h\ l\ v$ gives the same value as h' [v] for all variables.

On commencing a transaction, the logs are initialised to $\varnothing$ by the $\rightarrowtail$-begin rule, while the heaps according to both semantics have yet to diverge. The following definition shows that every heap h is equivalent to itself overlaid with empty logs:

```
∅-Equivalent : ∀ {h} → Equivalent h ∅ h
∅-Equivalent v rewrite Vec.lookup∘replicate v (∘ : Maybe ℕ)
  | Vec.lookup∘replicate v (∘ : Maybe ℕ) = ≡.refl
```

The two rewrites correspond to showing that the write and read logs are always empty, using the Vec.lookup∘replicate lemma to obtain proofs of Vec.replicate ∘ [v] $\equiv$ ∘, so that the value returned by Read reduces to just h [v]. The goal is then trivially satisfied by reflexivity.

In a similar manner to Read-Consistent, the operation of updating the read log for a variable v when it is first read preserves heap-log equivalence.

```
Read-Equivalent : ∀ {h l h′ v} → Logs.ρ l [ v ] ≡ ∘ →
  Equivalent h l h′ → Equivalent h (Logs.ρ l [ v ]:= • (h [ v ]) & Logs.ω l) h′
Read-Equivalent {h} {ρ & ω} {h′} {v} ρᵥ≡∘ equiv v′ with equiv v′
... | equiv-v′ with ω [ v′ ]
...   | • m = equiv-v′
...   | ∘ with v′ ≟Fin v
```

We start by binding the application $equiv\ v'$ to $equiv\text{-}v'$, which starts off with a type of snd (Read $h\ l\ v'$) $\equiv$ h' [v']. This is so that the Read function in its type can

be refined as we perform case analyses on the write and read log entries for v'. Since the write log does not change, the types of both the goal and *equiv*-v' reduces to $m \equiv h'\,[\,v'\,]$ when $\omega\,[\,v'\,]$ is $\bullet\ m$. Otherwise we must consider whether v' refers to the same variable as v whose read log entry is being updated:

```
...     | yes v′≡v rewrite v′≡v | ρᵥ≡∘
          | Vec.lookup∘update v ρ (• (h [ v ])) = equiv-v′
```

If v' is indeed the variable being updated, we can use the $\rho_v{\equiv}\circ$ argument to refine the type of *equiv*-v' to $h\,[\,v\,] \equiv h'\,[\,v\,]$, and a final Vec.lookup∘update rewrites the goal to the same type. Otherwise, we use the Vec.lookup∘update′ lemma to show that $\rho\,[\,v'\,]$ is unaffected by the update:

```
...     | no v′≢v rewrite Vec.lookup∘update′ v′≢v ρ (• (h [ v ])) with ρ [ v′ ]
...       | • m = equiv-v′
...       | ∘ = equiv-v′
```

In the two alternatives above, the types of the goals and *equiv*-v' reduce to $m \equiv h'\,[\,v'\,]$ and $h\,[\,v'\,] \equiv h'\,[\,v'\,]$, corresponding to the cases where v' was already cached in the read log, and when it is read for the first time respectively.

Unlike the Consistent property which only involves the read log, Equivalent also depends on the write log (indirectly via Read). Therefore we must demonstrate that write log updates preserve some notion of heap-log equivalence. We proceed by applying *equiv* to v', and checking whether v' and v are the same variable:

```
Write-Equivalent : ∀ {h l h′ v m} →
  Equivalent h l h′ → Equivalent h (Write l v m) (h′ [ v ]:= m)
Write-Equivalent {h} {ρ & ω} {h′} {v} {m} equiv v′ with equiv v′ | v′ ≟Fin v
... | equiv-v′ | yes v′≡v rewrite v′≡v | Vec.lookup∘update v ω (• m)
      | Vec.lookup∘update v h′ m = ≡.refl
```

In the yes case, we use Vec.lookup∘update to first show that the value returned by Read h (Write l v m) v is in fact m, which corresponds to the left-hand side of the _≡_ goal. The next clause rewrites the right-hand side from (h' [v]:= m) [v] to the same m, and ≡.refl completes this half of the proof.

For the no half where v' is not the variable being written to, the write log entry ω [v'] and the value of h' [v'] are not updated, which is taken care of by the two Vec.lookup∘update′ rewrites. Thus the existing *equiv-v′* suffices to complete the proof, although we do have to inspect the appropriate log entries to verify that *equiv-v′* and the goal have the correct types in all cases:

```
... | equiv-v′ | no v′≢v rewrite Vec.lookup∘update′ v′≢v ω (• m)
      | Vec.lookup∘update′ v′≢v h′ m with ω [ v′ ]
...     | • n = equiv-v′
...     | ○ with ρ [ v′ ]
...       | • n = equiv-v′
...       | ○ = equiv-v′
```

9.3.3 Post-Commit Heap Equality

When a transaction completes successfully, we proceed to update the unmodified heap with the contents of the write log, using the Update function defined at the end of §9.1.4. Given an h' that is equivalent to some heap h overlaid with logs l and that h and l are mutually consistent, we can proceed to show that updating h with the contents of the write log results in a heap identical to one that is modified in-place by the stop-the-world semantics:

```
Commit : ∀ {h l h′} →
  Consistent h l → Equivalent h l h′ → Update h l ≡ h′
Commit {h} {l} {h′} cons equiv =
```

```
  Equivalence.to Vec.Pointwise-≡ ⟨$⟩ Vec.Pointwise.ext hω≗h′ where
  hω≗h′ : ∀ v → Update h l [ v ] ≡ h′ [ v ]
  hω≗h′ v rewrite Vec.lookup∘tabulate (Update-lookup h l) v
      with Logs.ω l [ v ] | equiv v
  ... | • m | equiv-v = equiv-v
```

The main hω≗h′ part of the proof shows pointwise equality of Update h l and h', by considering the entry for v in the write and read logs. When the write log contains $\bullet$ m, the corresponding entry of h would be updated with m; not coincidentally *equiv-v* has been refined to a proof of $m \equiv h'$ [v]. Otherwise the write log contains a $\circ$, and the goal type reduces to h [v] $\equiv h'$ [v]:

```
  ... | ∘     | equiv-v with Logs.ρ l [ v ] | ≡.inspect (_[_] (Logs.ρ l)) v
  ...     | • m | ⟦ ρᵥ≡m ⟧ = ≡.trans (cons v m ρᵥ≡m) equiv-v
  ...     | ∘     | _ = equiv-v
```

We then proceed to inspect the read log: if it contains $\bullet$ m then *equiv-v* refines to a proof of $m \equiv h'$ [v], so we use *cons* to show that h [v] is also equal to m, and transitivity completes the proof. In the last case where both log entries are empty, the Read on the left-hand side of the type of *equiv-v* becomes simply h [v], and so completes the proof. Finally we use the proof of pointwise/definitional equivalence for Vec from the Agda standard library to convert hω≗h′ to a proof of definitional equality.

9.4 Transaction Correctness

During a transaction, the high-level stop-the-world semantics manipulates the heap directly, while the log-based semantics accumulates its reads and writes in a transaction log, eventually committing it to the heap. In this section we show that for any

transaction sequence under one semantics, there exists a matching sequence under the other semantics.

9.4.1 Completeness of Log-Based Transactions

We shall tackle the completeness part of transactional correctness first, as it is the simpler of the two directions. Let us begin by defining a function that extracts the $\_\mapsto^\star\_$ sequence from a visible transition starting from atomic e:

```
↦-extract : ∀ {α h e h″ c″ e″} →
  α ▷ h , ↦: , atomic e ⊨⇛ h″ , c″ , e″ →
  ∃₂ λ h₀ m → α , c″ , e″ ≡ ☢ , ↦: , # m ×
  h₀ , e ↦⋆ h″ , # m
↦-extract (⊨⇛: α≢τ ε (↠-↦ (↦-mutate h₁))) = ⊥-elim (α≢τ ≡.refl)
↦-extract (⊨⇛: α≢τ ε (↠-↦ (↦-atomic e↦′⋆m))) =
  _ , _ , ≡.refl , e↦′⋆m
↦-extract (⊨⇛: α≢τ (↠-↦ (↦-mutate h₁) ◁ e↠⋆τe′) e′↠e″)
    with ↦-extract (⊨⇛: α≢τ e↠⋆τe′ e′↠e″)
... | h₀ , m , ≡.refl , e↦′⋆m =
      h₀ , m , ≡.refl , e↦′⋆m
```

Under the stop-the-world semantics, the only non-silent transition atomic e can make is ↦-atomic, which carries the $e{\mapsto'}^\star m$ we desire. Note that the silent sequence preceding it may contain some number of ↦-mutate rules which we simply discard here, hence the heap at the start of the transaction may not necessarily be the same as that of the visible transition. We also give a proof that will allows us to refine α, c'' and e''.

Next, we show that for each of the transaction rules under the stop-the-world semantics, there is an corresponding rule that preserves the equivalence of the heap

and log of the log-based semantics with the in-place modified heap of the stop-the-world semantics:

```
↦′→⇀′ : ∀ {l h₀ h e h′ e′} →
          Equivalent h₀ l h  → h , e ↦′ h′ , e′ →
  ∃ λ l′ → Equivalent h₀ l′ h′ × h₀ ⊢ l , e ⇀′ l′ , e′
↦′→⇀′ equiv ↦′-⊕ℕ = _ , equiv , ⇀′-⊕ℕ
↦′→⇀′ equiv (↦′-⊕R m b↦b′) = Σ.map₃ (⇀′-⊕R m) (↦′→⇀′ equiv b↦b′)
↦′→⇀′ equiv (↦′-⊕L b a↦a′) = Σ.map₃ (⇀′-⊕L b) (↦′→⇀′ equiv a↦a′)
↦′→⇀′ equiv (↦′-writeE e↦e′) = Σ.map₃ ⇀′-writeE (↦′→⇀′ equiv e↦e′)
```

For ↦′-⊕ℕ, this is simply ⇀′-⊕ℕ; the corresponding rules for ↦′-⊕R, ↦′-⊕L and ↦′-writeE are similarly named, and we can simply map them over a recursive call to fulfil the goal.

The ↦′-writeℕ rule that modifies the heap directly has a counterpart ⇀′-writeℕ that updates the write log, and we use the Write-Equivalent lemma given previously to show that equivalence is maintained:

```
↦′→⇀′ equiv ↦′-writeℕ = _ , Write-Equivalent equiv , ⇀′-writeℕ
↦′→⇀′ {l} equiv (↦′-read h₀ v) with equiv v | ⇀′-read l v
... | equiv-v | ⇀′-read-l-v with Logs.ω l [ v ]
...   | • m rewrite equiv-v = _ , equiv , ⇀′-read-l-v
...   | ∘ with Logs.ρ l [ v ] | ≡.inspect (_[_] (Logs.ρ l)) v
...     | • m | _ rewrite equiv-v = _ , equiv , ⇀′-read-l-v
...     | ∘   | ⟦ ρᵥ≡∘ ⟧ rewrite ≡.sym equiv-v = _
          , Read-Equivalent ρᵥ≡∘ equiv , ⇀′-read-l-v
```

The matching rule for ↦′-read is of course ⇀′-read in all cases, although we need to pre-apply it with l and v and inspect the write and read logs to allow its type and that of the goal to refine so that they coincide. In the last alternative when both both

logs are empty, the Read-Equivalent lemma lets us show that heap-log equivalence still holds with the updated read log.

Lastly we generalise the above to transition sequences of any length, proceeding in the usual manner by applying $\mapsto'{\to}{\rightarrowtail}'$ to each transition from left to right:

```
↦'*→⇀'* : ∀ {h₀ l h e h' e'} →
                Equivalent h₀ l h  → h , e ↦* h' , e' →
  ∃ λ l' → Equivalent h₀ l' h' × h₀ ⊢ l , e ⇀'* l' , e'
↦'*→⇀'* equiv ε = _ , equiv , ε
↦'*→⇀'* equiv (e↦e' ◁ e'↦*e'') with ↦'→⇀' equiv e↦e'
... | l' , equiv' , e⇀e' with ↦'*→⇀'* equiv' e'↦*e''
...   | l'' , equiv'' , e'⇀*e'' = l'' , equiv'' , e⇀e' ◁ e'⇀*e''
```

Using the combination of $\mapsto$-extract and $\mapsto'^{*}{\to}{\rightarrowtail}'^{*}$, we can construct a transition sequence under the log-based semantics given a visible transition under the stop-the-world semantics, such that the heap and final log of the former is equivalent to the final heap of the latter.

9.4.2 Soundness of Log-Based Transactions

The soundness part of transactional correctness involves showing that the stop-the-world semantics can match the log-based semantics when the latter successfully commits. This is more difficult as the heap can be changed at any point during a log-based transaction by the $\rightarrowtail$-mutate rule. Let us begin by defining a variation on $\_\vdash\_\rightarrowtail'\_$ that encapsulates the heap used for each transition:

```
H⊢_⇀'_ : Rel (Logs × Expression')
H⊢ x ⇀' x' = Σ Heap (λ h → h ⊢ x ⇀' x')

H⊢_⇀'*_ : Rel (Logs × Expression')
H⊢_⇀'*_ = Star H⊢_⇀'_
```

We write $\mathsf{H}\vdash \_ \rightarrowtail'^{\star} \_$ for its reflexive, transitive closure. Every step of this transition potentially uses a different heap, in contrast to the $\_\vdash \_ \rightarrowtail'^{\star} \_$ relation where the entire sequence runs with the same heap.

Next we define a function that discards the steps from any aborted transactions, leaving only the final sequence of transitions leading up to a $\rightarrowtail$-commit, along with the heaps used at each step:

```
⇀⇁-extract′ : ∀ {h α r l e h′ c′ e′ h″ c″ e″} →
  H⊢ ∅ , r ⇀⇁′⋆ l , e →
  α ≢ τ → h , ⇀⇁: • (r , l) , atomic e →⋆τ h′ , c′ , e′ →
  α ▷ h′ , c′ , e′ ↠ h″ , c″ , e″ →
  ∃₂ λ l′ m → α , h″ , c″ , e″ ≡ ☢ , Update h′ l′ , ⇀⇁: ○ , # m ×
  Consistent h′ l′ × H⊢ ∅ , r ⇀⇁′⋆ l′ , # m
⇀⇁-extract′ r⇀⇁′⋆e α≢τ ε (↠-⇀⇁ (⇀⇁-step e⇀⇁e′)) = ⊥-elim (α≢τ ≡.refl)
⇀⇁-extract′ r⇀⇁′⋆e α≢τ ε (↠-⇀⇁ (⇀⇁-mutate h′)) = ⊥-elim (α≢τ ≡.refl)
⇀⇁-extract′ r⇀⇁′⋆e α≢τ ε (↠-⇀⇁ (⇀⇁-abort ¬cons)) = ⊥-elim (α≢τ ≡.refl)
```

The first argument to $\rightarrowtail$-extract′ accumulates the sequence of transactions steps starting from the initial r, while the next three correspond to the three fields of a visible transition. The three clauses above eliminate the cases where a silent transition appears in the non-silent position.

If no further transaction steps remain, the only possible rule is $\rightarrowtail$-commit, in which case we return the accumulated sequence $r \rightarrowtail'^{\star} e$ and the proof of consistency carried by $\rightarrowtail$-commit, along with an equality showing the values of α, h'', c'' and e'':

```
⇀⇁-extract′ r⇀⇁′⋆e α≢τ ε (↠-⇀⇁ (⇀⇁-commit cons)) =
  _ , _ , ≡.refl , cons , r⇀⇁′⋆e
```

When the transaction makes a single step, we append it to the end of the accumulator, taking a copy of the heap used for that step:

```
⟼-extract′ {h} r⟼′⋆e α≢τ (↠-⟼ (⟼-step e⟼e′) ◁ e′↠⋆τe″) e″↠e‴ =
  ⟼-extract′ (r⟼′⋆e ◁⋈ (h , e⟼e′) ◁ ε) α≢τ e′↠⋆τe″ e″↠e‴
```

Should we encounter a ⟼-mutate rule, we simply move on to the next step, albeit with a different heap:

```
⟼-extract′ r⟼′⋆e α≢τ (↠-⟼ (⟼-mutate h′) ◁ e′↠⋆τe″) e″↠e‴ =
  ⟼-extract′ r⟼′⋆e α≢τ e′↠⋆τe″ e″↠e‴
```

Lastly if e has reduced completely to a number, but the read log was not consistent with the heap at that point, the transaction aborts and retries. In this case, we reset the accumulator to ε and carry on with the rest of the sequence:

```
⟼-extract′ r⟼′⋆e α≢τ (↠-⟼ (⟼-abort ¬cons) ◁ e′↠⋆τe″) e″↠e‴ =
  ⟼-extract′ ε α≢τ e′↠⋆τe″ e″↠e‴
```

We can write a wrapper for the above that takes a visible transition, and strips off the initial ⟼-begin rule before invoking ⟼-extract′ itself:

```
⟼-extract : ∀ {α h r h″ c″ e″} →
  α ▷ h , ⟼: ∘ , atomic r ⊨⇛ h″ , c″ , e″ →
  ∃₃ λ h′ l′ m → α , h″ , c″ , e″ ≡ ☢ , Update h′ l′ , ⟼: ∘ , # m ×
  Consistent h′ l′ × H⊢ ∅ , r ⟼′⋆ l′ , # m
⟼-extract (⊨⇛: α≢τ ε (↠-⟼ ⟼-begin)) = ⊥-elim (α≢τ ≡.refl)
⟼-extract (⊨⇛: α≢τ (↠-⟼ ⟼-begin ◁ e↠⋆τe′) e′↠e″) =
  _ , ⟼-extract′ ε α≢τ e↠⋆τe′ e′↠e″
```

The next lemma says that we can swap the heap used for any $\_⊢\_⟼′\_$ transition, as long as the target heap is consistent with the original post-transition log $l′$:

```
⟼′-swap : ∀ {h h′ l e l′ e′} → Consistent h′ l′ →
  h ⊢ l , e ⟼′ l′ , e′ → h′ ⊢ l , e ⟼′ l′ , e′
```

```
↣′-swap cons′ ↣′-⊕ℕ = ↣′-⊕ℕ
↣′-swap cons′ (↣′-⊕R m b↣b′) = ↣′-⊕R m (↣′-swap cons′ b↣b′)
↣′-swap cons′ (↣′-⊕L b a↣a′) = ↣′-⊕L b (↣′-swap cons′ a↣a′)
↣′-swap cons′ (↣′-writeE e↣e′) = ↣′-writeE (↣′-swap cons′ e↣e′)
↣′-swap cons′ ↣′-writeℕ = ↣′-writeℕ
```

The first few cases are trivial since they either don't interact with the heap, or the proof burden can be deferred to a recursive call. The last ↣′-read case however does require our attention:

```
↣′-swap {h} {h′} cons′ (↣′-read l v) with ↣′-read {h′} l v
... | ↣′-read-l-v with Logs.ω l [ v ]
...   | • m = ↣′-read-l-v
...   | ∘ with Logs.ρ l [ v ]
...     | • m = ↣′-read-l-v
...     | ∘ rewrite cons′ v (h [ v ])
            (Vec.lookup∘update v (Logs.ρ l) (• (h [ v ]))) = ↣′-read-l-v
```

As long as one of the log entries for v is not empty, the transaction does not interact with the heap, so ↣′-read-l-v by itself suffices. Otherwise by the time we see that both entries are empty, Logs.ρ l' has been refined to Logs.ρ l [v]:= • (h [v]), so the type of $cons'$ is now:

```
cons′ : ∀ v′ m → (Logs.ρ l [ v ]:= • (h [ v ])) [ v′ ] ≡ • m → h′ [ v′ ] ≡ m
```

Instantiating v' to v and m to h [v] and invoking the Vec.lookup∘update lemma leads to a witness of h' [v] ≡ h [v], which we use in a rewrite clause to show that ↣′-read under h' does indeed result in the same l' and e' as it did under h, completing the proof of ↣′-swap.

Of course, we can generalise ↣′-swap to H⊢_↣′⋆_ sequences of any length:

```
↣′⋆-swap : ∀ {h′ l e l″ e″} → Consistent h′ l″ →
  H⊢ l , e ↣′⋆ l″ , e″ → h′ ⊢ l , e ↣′⋆ l″ , e″
↣′⋆-swap {h′} cons″ = snd ∘ Star.gfold id C⊢_↣′⋆_ trans (cons″ , ε) where
  C⊢_↣′⋆_ : Logs × Expression′ → Logs × Expression′ → Set
  C⊢ (l , e) ↣′⋆ (l′ , e′) = Consistent h′ l × h′ ⊢ l , e ↣′⋆ l′ , e′
  trans : ∀ {x x′ x″} → H⊢ x ↣′ x′ → C⊢ x′ ↣′⋆ x″ → C⊢ x ↣′⋆ x″
  trans (h , e↣e′) (cons′ , e′↣⋆e″) =
    ↣′-Consistent′ cons′ e↣e′ , ↣′-swap cons′ e↣e′ ◁ e′↣⋆e″
```

The auxiliary C⊢_↣′⋆_ relation pairs _⊢_↣′⋆_ with a proof of the consistency of h' with the read logs at the start of the sequence, while trans corresponds to the transitivity of a one-step H⊢_↣′_ and C⊢_↣′⋆_. The proof of ↣′⋆-swap results from folding trans over the H⊢_↣′⋆_ argument, using a final snd to discard the consistency part of the C⊢_↣′⋆_ pair.

What we have shown with ↣′⋆-swap is that provided the read log is consistent with the heap just before the commit, then regardless of what different heaps the transaction had originally used, re-running the transaction with just the pre-commit heap—without any intervening heap mutations—delivers the same result.

It remains for us to show that we can construct an equivalent transition under the stop-the-world semantics, given one that uses the same pre-commit heap. We start by taking a single log-based transition step and returning its corresponding stop-the-world rule, while showing that heap-log equivalence is preserved:

```
↣′→↦′ : ∀ {h l e l′ e′ h₀} →
          Equivalent h₀ l h → h₀ ⊢ l , e ↣′ l′ , e′ →
  ∃ λ h′ → Equivalent h₀ l′ h′ × h , e ↦′ h′ , e′
↣′→↦′ equiv ↣′-⊕ℕ = _ , equiv , ↦′-⊕ℕ
↣′→↦′ equiv (↣′-⊕R m b↣b′) = Σ.map₃ (↦′-⊕R m) (↣′→↦′ equiv b↣b′)
```

```
↣′→↦′ equiv (↣′-⊕L b a↣a′) = Σ.map₃ (↦′-⊕L b) (↣′→↦′ equiv a↣a′)
↣′→↦′ equiv (↣′-write_E e↣e′) = Σ.map₃ ↦′-write_E (↣′→↦′ equiv e↣e′)
↣′→↦′ equiv ↣′-write_ℕ = _ , Write-Equivalent equiv , ↦′-write_ℕ
↣′→↦′ {h} equiv (↣′-read l v) with equiv v | ↦′-read h v
... | equiv-v | ↣′-read-h-v with Logs.ω l [ v ]
...   | • m rewrite equiv-v = _ , equiv , ↣′-read-h-v
...   | ○ with Logs.ρ l [ v ] | ≡.inspect (_[_] (Logs.ρ l)) v
...     | • m | _ rewrite equiv-v = _ , equiv , ↣′-read-h-v
...     | ○ | ⟦ ρᵥ≡○ ⟧ rewrite ≡.sym equiv-v = _
          , Read-Equivalent ρᵥ≡○ equiv , ↣′-read-h-v
```

The above definition has an identical structure to that of ↦′→↣′ from the previous section, using the same Write-Equivalent and Read-Equivalent lemmas for the ↣′-write$_\mathbb{N}$ and ↣′-read cases respectively, so we will let the code speak for itself.

Finally we extend ↣′→↦′ to handle any _⊢_↣′*_ sequence, in the same manner as ↦′*→↣′*:

```
↣′*→↦′* : ∀ {h l e l′ e′ h₀} →
            Equivalent h₀ l h → h₀ ⊢ l , e ↣′* l′ , e′ →
  ∃ λ h′ → Equivalent h₀ l′ h′ × h , e ↦* h′ , e′
↣′*→↦′* equiv ε = _ , equiv , ε
↣′*→↦′* equiv (e↣e′ ◁ e′↣*e″) with ↣′→↦′ equiv e↣e′
... | h′ , equiv′ , e↦e′ with ↣′*→↦′* equiv′ e′↣*e″
...   | h″ , equiv″ , e′↦*e″ = h″ , equiv″ , e↦e′ ◁ e′↦*e″
```

To summarise, given a visible transition in which the log-based semantics commits a transaction, we can use ↣-extract to obtain the final successful sequence of _⊢_↣′_ transitions leading up to the commit, along with the heaps used at each step. The ↣′*-swap lemma then lets us swap the different heaps for the pre-commit heap, while

`⇀↦′*→↦′*` maps each log-based transition to their corresponding stop-the-world ones, allowing us to construct an equivalent transaction under the stop-the-world semantics.

9.5 Bisimilarity of Semantics

The proof that the stop-the-world semantics for our Atomic language is bisimilar to the log-based semantics proceeds for the most part by corecursion on the applicable transition rules, as well as structural recursion in the case of `a ⊕ b : Expressions` when either `a` or `b` can make further transitions. We will show each of the cases individually, then assemble the pieces to give the full correctness property.

We begin by showing that bisimilarity holds for numbers, where no further transitions are possible:

```
correct-# : ∀ {h m} → h , # m ⊢ ↦: ≈ ⇀↦: ∘
correct-# = ♯ (⊥-elim ∘ #↛) & ♯ (⊥-elim ∘ #↛)
```

The proof makes use of a trivial `#↛` lemma showing that no visible transitions are possible from expressions of the form `# m`, under either semantics.

9.5.1 Addition

For the first non-trivial case, we define `correct-⊕ℕ` which handles expressions of the form `# m ⊕ # n`. In this case, the only applicable rules are `↦-⊕ℕ` and `⇀↦-⊕ℕ`. We show each direction of bisimilarity separately:

```
correct-⊕ℕ : ∀ {h m n} → h , # m ⊕ # n ⊢ ↦: ≈ ⇀↦: ∘
correct-⊕ℕ {h} {m} {n} = ♯ ↦≼⇀↦ & ♯ ⇀↦≼↦ where
  ↦≼⇀↦ : h , # m ⊕ # n ⊢ ↦: ≼ ⇀↦: ∘
  ↦≼⇀↦ (⊨⇒: α≢τ ε (↠-↦ ↦-⊕ℕ)) =
    _ , ⊨⇒: α≢τ ε (↠-⇀↦ ⇀↦-⊕ℕ) , correct-#
```

```
↦≼↣ (⊨: α≢τ ε (↠-↦ (↦-⊕R _ b↦b′))) = ⊥-elim (#↛ b↦b′)
↦≼↣ (⊨: α≢τ ε (↠-↦ (↦-⊕L _ a↦a′))) = ⊥-elim (#↛ a↦a′)
↦≼↣ (⊨: α≢τ (↠-↦ (↦-⊕R _ b↦b′) ◁ _) _) = ⊥-elim (#↛ b↦b′)
↦≼↣ (⊨: α≢τ (↠-↦ (↦-⊕L _ a↦a′) ◁ _) _) = ⊥-elim (#↛ a↦a′)
```

To show that the log-based semantics can simulate the stop-the-world semantics we inspect the visible transition that `# m ⊕ # n` makes under the latter. As hinted above, the only applicable transition is ↦-⊕ℕ, for which we use ↣-⊕ℕ to show that the log-based semantics can follow. The resulting expression of `# (m + n)` is then bisimilar by the correct-# lemma. The remaining clauses amount to showing that further transitions by `# m` or `# n` alone are impossible.

The proof for the opposite direction proceeds in exactly the same way:

```
↣≼↦ : h , # m ⊕ # n ⊢ ↣: ∘ ≼ ↦:
↣≼↦ (⊨: α≢τ ε (↠-↣ ↣-⊕ℕ)) =
  _ , ⊨: α≢τ ε (↠-↦ ↦-⊕ℕ) , ≈-sym correct-#
↣≼↦ (⊨: α≢τ ε (↠-↣ (↣-⊕R _ b↣b′))) = ⊥-elim (#↛ b↣b′)
↣≼↦ (⊨: α≢τ ε (↠-↣ (↣-⊕L _ a↣a′))) = ⊥-elim (#↛ a↣a′)
↣≼↦ (⊨: α≢τ (↠-↣ (↣-⊕R _ b↣b′) ◁ _) _) = ⊥-elim (#↛ b↣b′)
↣≼↦ (⊨: α≢τ (↠-↣ (↣-⊕L _ a↣a′) ◁ _) _) = ⊥-elim (#↛ a↣a′)
```

9.5.2 Right Evaluation

Given an induction hypothesis of `h , b ⊢ ↦: ≈ ↣: ∘`, we can show that the two semantics are bisimilar for expressions of the form `# m ⊕ b`:

```
correct-⊕R : ∀ {h m b} → h , b ⊢ ↦: ≈ ↣: ∘ → h , # m ⊕ b ⊢ ↦: ≈ ↣: ∘
correct-⊕R {h} {m} {b} b⊢↦≈↣ = ♯ ↦≼↣ & ♯ ↣≼↦ where
```

For the completeness direction, we first use a ↠*/↦-⊕R helper that peels off any ↦-⊕R rules in the visible transition starting from `# m ⊕ b`. This is not always

possible: when b is already a number $\#\ n$, the full expression cannot make any transitions under $\mapsto$-$\oplus$R, so it returns a witness $b{\equiv}n$ that allows us to defer the rest of the proof to one half of the correct-$\oplus\mathbb{N}$ lemma given earlier:

```
↦≼↣↦ : h , # m ⊕ b ⊢ ↦: ≼ ↣: ○
↦≼↣↦ (⊨↦: α≢τ e↠⋆τe′ e′↠e″) with ↠⋆/↦-⊕R α≢τ e↠⋆τe′ e′↠e″
... | inl (n , b≡n , ≡.refl , ≡.refl , ≡.refl) rewrite b≡n =
      ♭ (≈→≼ correct-⊕ℕ) (⊨↦: α≢τ ε (↠-↦ ↦-⊕ℕ))
```

Otherwise b must make some visible transition under $\mapsto$-$\oplus$R, and $\twoheadrightarrow^\star/\mapsto$-$\oplus$R returns $b{\twoheadrightarrow}^\star_\tau b'$: h , $\mapsto:$, b $\twoheadrightarrow^\star_\tau$ h' , $\mapsto:$, b' as well as $b'{\twoheadrightarrow}b''$: $\alpha \triangleright h'$, $\mapsto:$, b' $\twoheadrightarrow$ h'' , $\mapsto:$, b'', essentially constituting a visible transition made by just b itself. The latter transition is labelled with the same α as the original $e'{\twoheadrightarrow}e''$, which in turn has been refined to h' , $\mapsto:$, $\#\ m \oplus b'$ $\twoheadrightarrow^\star_\tau$ h'' , $\mapsto:$, $\#\ m \oplus b''$ by the two equality proofs returned from $\twoheadrightarrow^\star/\mapsto$-$\oplus$R:

```
... | inr (h′ , b′ , h″ , b″ , ≡.refl , ≡.refl , b↠⋆τb′ , b′↠b″)
      with ♭ (≈→≼ b⊢↦≈↣) (⊨↦: α≢τ b↠⋆τb′ b′↠b″)
...   | c″ , b⊨↦b″ , b″⊢↦≈↣ with ⊨↦○↣-⊕R m b⊨↦b″
...     | c″≡↣ , m⊕b⊨↦m⊕b″ rewrite c″≡↣ =
          _ , m⊕b⊨↦m⊕b″ , correct-⊕R b″⊢↦≈↣
```

Next we invoke one half of the induction hypothesis $b{\vdash}{\mapsto}{\approx}{\rightarrowtail}$ with the aforementioned stop-the-world visible transition of $\mapsto:$ $\alpha{\not\equiv}\tau$ $b{\twoheadrightarrow}^\star_\tau b'$ $b'{\twoheadrightarrow}b''$, which returns an equivalent log-based visible transition $b{\mapsto}b''$: $\alpha \triangleright h$, $\rightarrowtail:$ $\circ$, $b \mapsto h''$, $\rightarrowtail:$ $\circ$, b''. Another lemma $\mapsto\circ\rightarrowtail$-$\oplus$R then replaces the $\mapsto$-$\oplus$R rules peeled off earlier with their corresponding $\rightarrowtail$-$\oplus$R rules, and a corecursive call to correct-$\oplus$R completes this part of the proof.

The definitions of $\twoheadrightarrow^\star/\mapsto$-$\oplus$R and $\mapsto\circ\rightarrowtail$-$\oplus$R are straightforward but rather tedious, and they can be found in the full source code on the author's website.

The soundness direction operates in exactly the same fashion, so we shall be brief with the similar details, and focus on the differences:

```
↣≼↦ : h , # m ⊕ b ⊢ ↣: ∘ ≼ ↦:
↣≼↦ (⊨⇒: α≢τ e↠⋆τ e′ e′↠e″) with ↠⋆/↣-⊕R α≢τ e↠⋆τ e′ e′↠e″
... | inl (n , b≡n , ≡.refl , ≡.refl , ≡.refl) rewrite b≡n =
      ♭ (≈→≽ correct-⊕ℕ) (⊨⇒: α≢τ ε (↠-↣ ↣-⊕ℕ))
... | inr (h′ , t′ , b′ , h″ , b″ , ≡.refl , ≡.refl , b↠⋆τb′ , b′↠b″)
      with ♭ (≈→≽ b⊢↦≈↣) (⊨⇒: α≢τ b↠⋆τb′ b′↠b″)
...   | c″ , b⊨⇒b″ , b″⊢↣≈↦ with ⊨⇒∘↦-⊕R m b⊨⇒b″
...     | c″≡↦ , m⊕b⊨⇒m⊕b″ rewrite c″≡↦ =
          _ , m⊕b⊨⇒m⊕b″ , ≈→≽ ↦≈↣ & ≈→≼ ↦≈↣ where
  ↦≈↣ = correct-⊕R (≈-sym b″⊢↣≈↦)
```

A ↠⋆/↣-⊕R helper first attempts to peel off any ↣-⊕R from $e \twoheadrightarrow^\star_\tau e'$ and $e' \twoheadrightarrow e''$; we invoke correct-⊕ℕ should this not be possible. Otherwise the induction hypothesis gives us a stop-the-world visible transition $b \models\!\Rightarrow b''$, and we can use ⊨⇒∘↦-⊕R to turn this back into $m \oplus b \models\!\Rightarrow m \oplus b''$. To show that the semantics are bisimilar for h'' , # m ⊕ b'', one might be tempted to write ≈-sym (correct-⊕R (≈-sym $b'' \vdash \rightarrowtail \approx \mapsto$)). However Agda's termination/productivity checker requires all corecursive calls be guarded by constructors, and cannot see that the function ≈-sym preserves productivity. We get around this issue by inlining the outer ≈-sym, since record projections—namely ≈→≼ and ≈→≽—are seen to be productivity-preserving.

9.5.3 Left Evaluation

The correct-⊕L lemma handles cases where the expression on the left of a _⊕_ can make further visible transitions. It requires suitable induction hypotheses on a and b; in particular that for b must be generalised over any heap:

```
correct-⊕L : ∀ {h a b} → h , a ⊢ ↦: ≈ ⟼: ○ →
  (∀ h′ → h′ , b ⊢ ↦: ≈ ⟼: ○) → h , a ⊕ b ⊢ ↦: ≈ ⟼: ○
correct-⊕L {h} {a} {b} a⊢↦≈⟼ ∀b⊢↦≈⟼ = ♯↦≼⟼ & ♯⟼≼↦ where
```

The $\twoheadrightarrow^\star/\mapsto$-$\oplus$L lemma then lets us distinguish between the case when a is just a number # m, and the case where a makes a visible transition. In the former case, we pass the proof obligation on to the correct-$\oplus$R lemma, instantiating $\forall b \vdash \mapsto \approx \longmapsto$ with the current heap:

```
  ↦≼⟼ : h , a ⊕ b ⊢ ↦: ≼ ⟼: ○
  ↦≼⟼ (⊨⇒: α≢τ e→↠⋆τe′ e′↠e″) with →↠⋆/↦-⊕L α≢τ e→↠⋆τe′ e′↠e″
  ... | inl (m , a≡m) rewrite a≡m =
        ♭ (≈→≼ (correct-⊕R (∀b⊢↦≈⟼ h))) (⊨⇒: α≢τ e→↠⋆τe′ e′↠e″)
  ... | inr (h′ , a′ , h″ , a″ , ≡.refl , ≡.refl , a→↠⋆τa′ , a′↠a″)
        with ♭ (≈→≼ a⊢↦≈⟼) (⊨⇒: α≢τ a→↠⋆τa′ a′↠a″)
  ...   | c″ , a⊨⇒a″ , a″⊢↦≈⟼ with ⊨⇒○⟼-⊕L b a⊨⇒a″
  ...     | c″≡⟼ , a⊕b⊨⇒a″⊕b rewrite c″≡⟼ =
            _ , a⊕b⊨⇒a″⊕b , correct-⊕L a″⊢↦≈⟼ ∀b⊢↦≈⟼
```

Otherwise we have a visible transition $\Rightarrow: \alpha \not\equiv \tau\ a \twoheadrightarrow^\star_\tau a'\ a' \twoheadrightarrow a''$, and the first inductive hypothesis allows us to obtain a corresponding visible transition $a \Rightarrow a''$ under the log-based semantics. Next we replace the b on the right hand side of $\_\oplus\_$ using the $\Rightarrow\circ\longmapsto$-$\oplus$L lemma to obtain the transition $a \oplus b \Rightarrow a'' \oplus b$. Since b has not made any transitions, a corecursive call with $a'' \vdash \mapsto \approx \longmapsto$ and the original $\forall b \vdash \mapsto \approx \longmapsto$ completes the proof.

We proceed with the soundness half of correct-$\oplus$L in exactly the same way as that of correct-$\oplus$R:

```
  ⟼≼↦ : h , a ⊕ b ⊢ ⟼: ○ ≼ ↦:
  ⟼≼↦ (⊨⇒: α≢τ e→↠⋆τe′ e′↠e″) with →↠⋆/⟼-⊕L α≢τ e→↠⋆τe′ e′↠e″
```

```
... | inl (m , a≡m) rewrite a≡m =
      ♭ (≈→≽ (correct-⊕R (∀b⊢↦≈↣ h))) (⊨: α≢τ e→↠⋆τ e′ e′→↠e″)
... | inr (h′ , t′ , a′ , h″ , a″ , ≡.refl , ≡.refl , a→↠⋆τ a′ , a′→↠a″)
      with ♭ (≈→≽ a⊢↦≈↣) (⊨: α≢τ a→↠⋆τ a′ a′→↠a″)
...   | c″ , a⊨⇛a″ , a″⊢↣≈↦ with ⊨⇛∘↦-⊕L b a⊨⇛a″
...     | c″≡↦ , a⊕b⊨⇛a″⊕b rewrite c″≡↦ =
          _ , a⊕b⊨⇛a″⊕b , ≈→≽ ↦≈↣ & ≈→≼ ↦≈↣ where
  ↦≈↣ = correct-⊕L (≈-sym a″⊢↣≈↦) ∀b⊢↦≈↣
```

Observe how correct-⊕L shares the same overall structure as correct-⊕R.

9.5.4 Transactions

Finally we arrive at the correctness proof for atomic expressions, where we need to show that transactions run under the stop-the-world semantics coincide with those using our log-based semantics:

```
correct-atomic : ∀ {h e} → h , atomic e ⊢ ↦: ≈ ↣: ∘
correct-atomic {h} {e} = ♯ ↦≼↣ & ♯ ↣≼↦ where
```

In the completeness direction, we show that the log-based semantics can follow the stop-the-world one by simply running the entire transaction uninterrupted at the same point as the ↦-atomic rule. First the ↦-extract helper from §9.4.1 picks out the $e{\mapsto}'^{\star}m$ sequence:

```
↦≼↣ : h , atomic e ⊢ ↦: ≼ ↣: ∘
↦≼↣ e⊨⇛e″ with ↦-extract e⊨⇛e″
... | h₀ , m , ≡.refl , e↦′⋆m
      with ↦′⋆→↣′⋆ ∅-Equivalent e↦′⋆m
...   | l′ , equiv′ , e↣′⋆m with ↣′⋆-Consistent ∅-Consistent e↣′⋆m
```

```
...     | cons′ rewrite ≡.sym (Commit cons′ equiv′) =
            _ , e⇛m , correct-# where
```

The $\mapsto'^{\star}\to\rightarrowtail'^{\star}$ function from the same section then translates each transition of $e{\mapsto'^{\star}}m$ to its log-based equivalent, as well as constructing a proof $equiv'$ of the equivalence of h_0 and l' with the heap h' at the end of the $e{\mapsto'^{\star}}m$ sequence. By the $\rightarrowtail'^{\star}$-Consistent lemma, we show that the consistency of h_0 and the logs is preserved along the $e{\rightarrowtail'^{\star}}m$ sequence, culminating in a witness $cons'$ of Consistent h_0 l' h'. Finally, a rewrite clause using the Commit lemma convinces Agda that Update h_0 l' and h' are definitionally equal. Since running a transaction results in just a number # m, correct-# suffices to show that both semantics are bisimilar in this case.

It remains for us to construct the visible transition e⇛m that uses the log-based semantics. Should the heap be changed just before the stop-the-world semantics runs the transaction, we need a corresponding $\rightarrowtail$-mutate rule in the log-based transition sequence. The mutate? helper checks whether this is necessary:

```
mutate? : h , ↣: • (e , ∅) , atomic e →⋆τ h₀ , ↣: • (e , ∅) , atomic e
mutate? with h ≟Heap h₀
... | yes h≡h₀ rewrite h≡h₀ = ε
... | no h≢h₀ = ↠-↣ (↣-mutate h₀) ◁ ε
```

Next, the auxiliary definition e↠⋆τm lifts each transition of $e{\rightarrowtail'^{\star}}m$ up to the $\_\triangleright\_\twoheadrightarrow\_$ level using the $\rightarrowtail$-step rule, prepending a $\rightarrowtail$-mutate rules when necessary:

```
e↠⋆τm : h , ↣: • (e , ∅) , atomic e →⋆τ h₀ , ↣: • (e , l′) , atomic (# m)
e↠⋆τm = mutate? ◁◁ Star.gmap _ (↠-↣ ∘ ↣-step) e↣′⋆m

e⇛m : ☢ ▷ h , ↣: ∘ , atomic e ⇛ Update h₀ l′ , ↣: ∘ , # m
e⇛m = ⇛: (λ ()) (↠-↣ ↣-begin ◁ e↠⋆τm) (↠-↣ (↣-commit cons′))
```

Lastly we add a $\rightarrowtail$-begin to beginning of the visible transition to initialise the transaction state, followed by e↠⋆τm as the main body of the transaction. A final non-silent

⤚→-commit carrying the *cons′* witness produces the desired visible transition.

The proof of soundness relies on us having shown that for every transaction that commits successfully, re-running the entire transaction without any interference at the point of ⤚→-commit computes the same result. We can then use this uninterrupted transaction sequence to derive a corresponding visible transition under the stop-the-world semantics.

We start with a similar ⤚→-extract lemma defined in §9.4.2 that returns a sequence $e \rightarrowtail'^{\star} m$: H⊢ ∅ , e ⤚→′⋆ l' , # m where each transition potentially uses a different heap, as well as the *cons′* proof carried by the final ⤚→-commit:

```
⤚→≼↦ : h , atomic e ⊢ ⤚→: ∘ ≼ ↦:
⤚→≼↦ e⇛e″ with ⤚→-extract e⇛e″
... | h₀ , l′ , m , ≡.refl , cons′ , e⤚→′⋆m
      with ⤚→′⋆→↦′⋆ ∅-Equivalent (⤚→′⋆-swap cons′ e⤚→′⋆m)
...   | h′ , equiv′ , e↦′⋆m rewrite ≡.sym (Commit cons′ equiv′) =
        _ , e⇛m , ≈-sym correct-# where
```

There is one additional step involved: we must use the ⤚→′⋆-swap lemma to replace the different heaps used throughout $e \rightarrowtail'^{\star} m$ with h_0—to give a witness of h_0 ⊢ ∅ , e ⤚→′⋆ l' , # m—before we can use ⤚→′⋆→↦′⋆ to convert this to the sequence $e \mapsto'^{\star} m$: h_0 , e ↦⋆ h' , # m. This is necessary because the ⤚→′⋆→↦′⋆ lemma requires its input log-based transitions to be under the same heap in order to show equivalence preservation.

The result of both transaction is the same expression # m, and we use the symmetry of the earlier correct-# lemma to provide evidence of bisimilarity. All that is left is to wrap up the stop-the-world transactional transition sequence $e \mapsto'^{\star} m$ as the visible transition e⇛m. We define a mutate? sequence to correspond to any necessary pre-transaction ↦-mutate rules,

```
mutate? : h , ↦: , atomic e →»⋆τ h₀ , ↦: , atomic e
mutate? with h ≟?Heap h₀
... | yes h≡h₀ rewrite h≡h₀ = ε
... | no h≢h₀ = →»-↦ (↦-mutate h₀) ◁ ε
e⇛m : ☢ ▷ h , ↦: , atomic e ⇛ Update h₀ l′ , ↦: , # m
e⇛m = ⇛: (λ ()) mutate? (→»-↦ (↦-atomic e↦′⋆m))
```

which simply slots in as the silent transitions before the final ↦-atomic, to give the desired visible transition e⇛m. This completes the bisimilarity proof for transactions.

9.5.5 Putting It Together

Having shown for each individual case that our stop-the-world and log-based semantics are indeed bisimilar, all that remains for us is to combine them to give the proof of bisimilarity for any Expression:

```
correct : ∀ h e → h , e ⊢ ↦: ≈ ⇀: ∘
correct h (# m) = correct-#
correct h (a ⊕ b) = correct-⊕L (correct h a) (λ h′ → correct h′ b)
correct h (atomic e) = correct-atomic
```

In the $a \oplus b$ case, observe how correct-⊕L is supplied with the induction hypothesis on a, but that for b is abstracted over an arbitrary heap h'.

9.6 Conclusion

In this chapter we considered a language with transactions, and a worst-case 'mutate' rule in place of explicitly modelling interference by concurrent threads. We gave a fully formalised proof that our stop-the-world and log-based semantics for transactions do indeed coincide, without resorting to any postulated lemmas: the completeness part

simply ran the log-based transaction uninterrupted at the same time as the stop-the-world $\mapsto$-atomic rule. For soundness we used a key $\rightarrowtail'^{\star}$-swap lemma showing that regardless of any heap interference during the log-based transaction, as long as the pre-commit heap h' and logs l' were consistent, replaying the transaction under h' gives the same results. This underlines the importance of Consistent h' l' as the correct criteria for a log-based transaction to commit.

Chapter 10

Conclusion

To conclude, these final pages will comprise an overview of this thesis and an account of how it came to be, followed by a summary of its contributions and some directions for further work.

10.1 Retrospection

The quest for higher-level abstractions to manage the complexities of concurrent programming has been an especially apt topic in recent years, due to reasons outlined in the introductory chapter. With respect to software transactional memory (Chapter 2), I was fortunate enough to be in the right places at the right times to have attended two of Tim Harris's talks on the topic: the first in Cambridge during my undergraduate years, on the JVM-based implementation; and a second time at Imperial College during my MSc course in the early part of 2005, on the composability of STM Haskell.

My work for this thesis began in 2006 under the guidance of Graham Hutton, with an initial goal of reasoning about concurrent programs, in particular those written using STM. To this end, we opted for a simple formal language based on Hutton's Razor, extended with a minimal set of transactional primitives, described in Chapter

§5. While this language—following the reference stop-the-world semantics given by Harris et al. [HMPJH05]—had a simple implementation, it was not immediately clear how STM Haskell dealt with conflicting transactions internally, consequently drawing our attention towards the correctness of the low-level concurrent implementation.

To better understand the implementation issues behind software transactional memory, we began building a stack-based virtual machine and a compiler for our minimal language, of which the final version is given in §5.2. Using Haskell as a metalanguage, it was a straightforward task to transcribe the syntax and semantics of our model as an executable program. Combined with the use of QuickCheck and the Haskell Program Coverage toolkit (Chapter 4), this allowed us to take a 'rapid prototyping' approach to the design of the virtual machine. Most notably, we were able to clarify the appropriate conditions needed for a transaction to commit successfully (§5.2.3), and to realise that writing to an as-yet unread variable within a transaction does not imply a dependency on the current state of the heap.

Of course, regardless of how many times we run QuickCheck on our STM model, overwhelming evidence does not constitute a proof of compiler correctness for our interleaved semantics and log-based implementation of transactional memory. Due to the influence of numerous type-theorists at Nottingham, I had become interested in dependently-typed programming (Chapter 6) and dually, the application of intuitionistic type theory as a framework for conducting formal proofs. Therefore, the goal of a mechanised compiler correctness proof for our model in a dependently-typed language/proof-assistant seemed a natural choice.

Whereas compiler correctness theorems in a deterministic setting (Chapter 3) are concerned only with the final results, with the introduction of non-determinism (of which concurrency is one form) we can no longer afford to ignore *how* results are computed, in addition to what is being computed. Bisimilarity as a notion of behavioural equivalence is a standard tool in the field of process calculi, and

Wand [Wan95, WS95, GW96] et al. were the first to use it to tackle concurrent compiler correctness over a decade ago. Their work relied on giving denotational semantics to both source and target languages in an underlying process calculus, and showing that compilation preserved bisimilarity of denotations. In contrast, we defined our language and virtual machine in an operational manner, and sought a simpler and more direct approach.

Thus, the idea of the combined machine was born, detailed in Chapter 7. A key realisation was that certain kinds transitions preserve bisimilarity, giving rise to the elide-τ lemma. Having tested the waters with the non-deterministic Zap language, Chapter 8 then demonstrates that our approach can indeed scale to handle concurrency, at least for that of the Fork language. As well as updating the elide-τ lemma for explicit concurrency, we also showed that combining bisimilar pairs of thread soups preserves bisimilarity.

Finally, Chapter 9 considers a log-based implementation of transactions. In order to simplify the proof of its equivalence with a stop-the-world semantics, we replaced concurrency with a 'mutate' rule that simulates the worst possible concurrent environment, and directly defined a log-based transaction semantics on the source language. The final correctness proof makes essential use of a notion of equivalence between heaps and logs, and confirms our earlier hypothesis that consistency of the read log with the heap is a sufficient condition for a transaction to commit.

10.2 Summary of Contributions

This thesis addresses the familiar question of compiler correctness in a current context, with a particular emphasis on the implementation of software transactional memory. We have identified a simplified subset of STM Haskell that is amenable to formal reasoning, which has a stop-the-world transactional semantics, together with

a concurrent virtual machine for this language, using the notion of transaction logs. A compiler linking this simplified language to its virtual machine then allowed us to formulate a concrete statement of compiler correctness. We were able to implement the above semantics in a fairly direct manner using the high-level vocabulary provided by Haskell, enabling us to empirically test compiler correctness with the help of QuickCheck and HPC.

Working towards a formal proof of the above hypothesis, we stripped down to a minimal language with trivial non-determinism, and moving to a labelled transition system. The core idea of a combined machine and semantics then allowed us to establish a direct bisimulation between this language and its virtual machine. This technique was put into practice using the Agda proof assistant, giving a machine-checked compiler correctness proof for a language with a simple notion of non-determinism. We then extended the above proof and our approach in the direction of the initially identified subset of STM Haskell, in an incremental manner: first introducing explicit concurrency in the form of a fork primitive, before finally tackling a language with an atomic construct in a simplified setting, resulting in a formal proof of the equivalence of the high-level stop-the-world semantics with a low-level log-based approach.

10.3 Directions for Further Research

Our original aim was to formally verify the compiler correctness result from Chapter 5, whereas in Chapter 9 we verified this result with some simplifications, in particular the removal of the fork construct. While this establishes what we feel to be the essence of the result, we believe our original aim is still achieveable with further work, since our worst-case 'mutate' rule subsumes any interference by other threads in a concurrent setting.

Our simplified model of STM Haskell focuses on the essence of implementing

transactions, and consequently omits many of the facilities expected of a realistic language. Namely, the lack of primitive recursion or even name binding limits the computational power of our model in a very tangible sense. We could be tackle this using a lightweight approach, by borrowing said facilities from the metalanguage and defining the high-level semantics as a functional specification. For example, Gordon's thesis [Gor92] presents such a specification of teletype IO for a subset of Haskell in terms of a low-level metalanguage, while Swierstra [Swi08] advocates the use of Agda as the metalanguage, due to its enforcement of totality.

Given the above as a basis, a machine-verified formalisation of the omitted parts of the STM Haskell specification—in particular retry/orElse and the interaction with exceptions—becomes a much more tractable proposition. Open questions include: How will these additions affect the design of the corresponding virtual machine? Can we maintain the simplicity of our combined machine approach? Is the outline of our reasoning for transactions still valid in this richer language? Our current virtual machine immediately retries failed transactions, rather than waiting until some relevant transactional variable has changed. How can our virtual machine more faithfully model the implementation of STM in GHC?

Going further, we could extend the set of side-effects that can be safely rolled back by the transactional machinery. One widely asserted advantage of STM Haskell over other STM implementations is that its type system restricts transactions—aside from modifying TVars—to pure computations, guaranteeing that rollback is always possible. During my initial work on the model of STM Haskell, the notion of running multiple nested transactions concurrently arose quite naturally, when considering their rôle in the implementation of retry/orElse. While the forkIO primitive is considered impure, forking a nested transaction need not be, as its side-effects can only escape as part of that of its enclosing transaction. It could be interesting to flesh out the precise behaviour of such a forkSTM :: STM () → STM ThreadId primitive, and

to evaluate its utility for concurrent programming in the real world.

Bibliography

[ATS09] Ali-Reza Adl-Tabatabai and Tatiana Shpeisman. *Draft Specification of Transactional Language Constructs for C++*. Intel, August 2009.

[AVWW96] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in ERLANG*. Prentice Hall, second edition edition, 1996.

[Bar84] Hendridk Pieter Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. Elsevier, 1984.

[CGE08] David Cunningham, Khilan Gudka, and Susan Eisenbach. Keep Off the Grass: Locking the Right Path for Atomicity. In *Compiler Construction 2008*, April 2008.

[CH00] Koen Claessen and John Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *ICFP Proceedings*, 2000.

[Chu36] Alonzo Church. An Unsolvable Problem of Elementary Number Theory. *American Journal of Mathematics*, 58(2):345–363, April 1936.

[CPN98] David G. Clarke, John M. Potter, and James Noble. Ownership Types for Flexible Alias Protection. In *Proceedings of the 13th Conference on Object-Oriented Programming, Systems, Languages, and Applications*

(OOPSLA-98), volume 33:10 of *ACM SIGPLAN Notices*, pages 48–64, New York, October 1998. ACM Press.

[Dan10a] Nils Anders Danielsson. Beating the Productivity Checker Using Embedded Languages. In *Workshop on Partiality and Recursion in Interactive Theorem Provers*, July 2010.

[Dan10b] Nils Anders Danielsson. The Agda Standard Library. Available from `http://cs.nott.ac.uk/~nad/listings/lib/`, September 2010.

[Dat95] Christopher J Date. *An Introduction to Database Systems*. Addison-Wesley, 6th edition, 1995.

[Dav03] Maulik A Dave. Compiler Verification: A Bibliography. *ACM SIGSOFT Software Engineering Notes*, 28(6):2–2, November 2003.

[Dij65] Edsger Wybe Dijkstra. Cooperating Sequential Processes. Lecture notes, Technological University, Eindhoven, The Netherlands, September 1965.

[Enn05] Robert Ennals. Software Transactional Memory Should Not be Obstruction-Free. Submitted to SCOOL 2005, 2005.

[FFL05] Cormac Flanagan, Stephen N Freund, and Marina Lifshin. Type Inference for Atomicity. In *Proceedings of Types in Language Design and Implementation*, January 2005.

[Fra03] Keir Fraser. *Practical Lock-Freedom*. PhD thesis, University of Cambridge, September 2003.

[GBD+94] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine—A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994.

[Gla94] David S Gladstein. *Compiler Correctness for Concurrent Languages*. PhD Thesis, Northeastern University, Massachusetts, December 1994.

[GLS99] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI—Portable Parallel Programming with the Message Passing Interface*. MIT Press, second edition edition, 1999.

[Goe06] Brian Goetz. Optimistic Thread Concurrency. Whitepaper, Azul Systems, Inc., January 2006.

[Gor92] Andrew Donald Gordon. *Functional Programming and Input/Output*. PhD Thesis, University of Cambridge, 1992.

[GR07] Andy Gill and Colin Runciman. Haskell Program Coverage. In *Haskell Workshop Proceedings*, September 2007.

[Gra81] Jim Gray. The Transaction Concept: Virtues and Limitations. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 1981.

[Gro07] Dan Grossman. The Transactional Memory / Garbage Collection Analogy. In *OOPSLA Proceedings*, pages 695–706, October 2007.

[GW96] David S Gladstein and Mitchell Wand. Compiler Correctness for Concurrent Languages. In *Proceedings of Coordination*, volume 1061 of *Lecture Notes in Computer Science*. Springer, April 1996.

[HF03] Tim Harris and Keir Fraser. Language Support for Lightweight Transactions. In *OOPSLA Proceedings*, October 2003.

[HH08] Liyang HU and Graham Hutton. Towards a Verified Implementation of Software Transactional Memory. In *Proceedings of Trends in Functional Programming*, May 2008.

[HH09] Liyang HU and Graham Hutton. Compiling Concurrency Correctly: Cutting Out the Middle Man. In *Trends in Functional Programming*, 2009.

[HLM06] Maurice P Herlihy, Victor Luchangco, and Mark Moir. A Flexible Framework for Implementing Software Transactional Memory. In *Proceedings of OOPSLA*, 2006.

[HM93] Maurice P Herlihy and J Eliot B Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *International Symposium on Computer Architecture Proceedings*, 1993.

[HMPJH05] Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice P Herlihy. Composable Memory Transactions. In *Proceedings of Principles and Practice of Parallel Programming*, June 2005.

[Hoa02] Charles Antony Richard Hoare. Towards a Theory of Parallel Programming *(first published 1971)*. In *The Origin of Concurrent Programming: from Semaphores to Remote Procedure Calls*, pages 231–244. Springer-Verlag, 2002.

[HPJ06] Tim Harris and Simon Peyton Jones. Transactional Memory with Data Invariants. In *TRANSACT Proceedings*, March 2006.

[Hut07] Graham Hutton. *Programming in Haskell*. Cambridge University Press, January 2007.

[HW04] Graham Hutton and Joel Wright. Compiling Exceptions Correctly. In *Proceedings of International Conference on Mathematics of Program Construction*, number 3125 in Lecture Notes in Computer Science. Springer, July 2004.

[HW06] Graham Hutton and Joel Wright. Calculating an Exceptional Machine. In *Proceedings of Trends in Functional Programming*, volume 5, February 2006.

[HW07] Graham Hutton and Joel Wright. What is the Meaning of These Constant Interruptions? *Journal of Functional Programming*, 17(6):777–792, November 2007.

[JHB87] Eric H Jensen, Gary W Hagensen, and Jeffrey M Broughton. A New Approach to Exclusive Data Access in Shared Memory Multiprocessors. Technical Report UCRL-97663, Lawrence Livermore National Laboratory, November 1987.

[KLS90] Henry F Korth, Eliezer Levy, and Abraham Silberschatz. A Formal Approach to Recovery by Compensating Transactions. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 1990.

[KSF10] Guy Korland, Nir Shavit, and Pascal Felber. Noninvasive Concurrency with Java STM. In *Programmability Issues for Multi-Core Computers (MULTIPROG)*, January 2010.

[Lam74] Leslie Lamport. A New Solution of Dijkstra's Concurrent Programming Problem. *Communications of the ACM*, 17(8):453–455, August 1974.

[Ler06] Xavier Leroy. Formal Certification of a Compiler Back-End, or: Programming a Compiler with a Proof Assistant. In *Proceedings of Principles of Programming Languages*, volume 33, pages 42–54, 2006.

[LS08] Anand Lal Shimpi. Intel's Atom Architecture: The Journey Begins. Retrieved from `http://www.anandtech.com/printarticle.aspx?i=3276`, page 14., April 2008.

[M+08] Conor McBride et al. The Epigram System. Available from `http://www.e-pig.org/`, 2008.

[Mar10] Simon Marlow, editor. *Haskell 2010 Language Report*. Online, 2010.

[McB02] Conor McBride. Faking It (Simulating Depdent Types in Haskell). *Journal of Functional Programming*, 12(4–5):375–392, July 2002.

[McB05] Conor McBride. Epigram: Practical Programming with Dependent Types. In *Lecture Notes in Computer Science*. Springer-Verlag, 2005.

[McB07] Conor McBride. $\mathsf{R}^\star$ is the new $[\alpha]$. Talk given at Fun in the Afternoon, York., November 2007.

[Mil89] Robin Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, 1989.

[ML80] Per Martin-Löf. Intuitionistic Type Theory. Lecture notes, 1980.

[ML98] Saunders Mac Lane. *Categories for the Working Mathematician*. Springer, second edition edition, September 1998.

[Mog89] Eugenio Moggi. Computational Lambda-Calculus and Monads. In *Proceedings of Logic in Computer Science*, pages 14–23. IEEE Computer Society Press, June 1989.

[Moo65] Gordon Moore. Moore's Law. Retrieved from `http://www.intel.com/technology/mooreslaw/`, 1965.

[MP67] John McCarthy and James Painter. Correctness of a Compiler for Arithmetic Expressions. In *Proceedings of Symposia in Applied Mathematics*, volume 19. AMS, 1967.

[Nodir0o0] Cyprien Noël. Extensible Software Transactional Memory. In *Proceedings of the Conference on Computer Science and Software Engineering*, 2010.

[Nor07] Ulf Norell. *Towards a Practical Programming Language Based on Dependent Type Theory*. PhD thesis, Chalmers University of Technology, September 2007.

[NPS90] Bengt Nordström, Kent Petersson, and Jan M Smith. *Programming in Martin-Löf's Type Theory*. Oxford University Press, 1990.

[PJ01] Simon Peyton Jones. Tackling the Awkward Squad: Monadic Input/Output, Concurrency, Exceptions, and Foreign-Language Calls in Haskell. In *Engineering Theories of Software Construction*, pages 47–96. IOS Press, 2001.

[PJ03a] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

[PJ03b] Simon Peyton Jones. Wearing the Hair Shirt: A Retrospective on Haskell. (Invited talk.). In *Principles of Programming Languages*, 2003.

[PJGF96] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *Proceedings of Principles of Programming Languages*, pages 295–308, 1996.

[PJWW04] Simon Peyton Jones, George Washburn, and Stephanie Weirich. Wobbly Types: Type Inference for Generalised Algebraic Data Types. Technical Report MS-CIS-05-26, University of Pennsylvania, 2004.

[Sab98] Amr Sabry. What is a Purely Functional Programming Language? *Journal of Functional Programming*, 8:1–22, 1998.

[San09] Davide Sangiorgi. On the Origins of Bisimulation and Coinduction. *ACM Transactions on Programming Languages and Systems*, 31(4), May 2009.

[SJ07] Don Stewart and Spencer Janssen. XMonad: A Tiling Window Manager. In *Haskell Workshop*, September 2007.

[ST97] Nir Shavit and Dan Touitou. Software Transactional Memory. *Distributed Computing*, 10(2):99–116, February 1997.

[Sta10] Richard Matthew Stallman. GNU Emacs. Available from `http://www.gnu.org/software/emacs/`, 2010.

[Swi08] Wouter S Swierstra. *A Functional Specification of Effects*. PhD thesis, University of Nottingham, November 2008.

[Tan05] Audrey Tang. Pugs: an implementation of Perl 6 in Haskell, February 2005.

[The08] The Coq Development Team. The Coq Proof Assistant. Available from `http://coq.inria.fr/`, June 2008.

[The10] The Agda Team. The Agda Wiki. Available from `http://wiki.portal.chalmers.se/agda/`, September 2010.

[THLPJ98] Philip W Trinder, Kevin Hammond, Hans-Wolfgang Loidl, and Simon Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1):23–60, January 1998.

[Wad92] Philip Wadler. Comprehending Monads. *Mathematical Structures in Computer Science*, 2:461–493, 1992.

[Wan95] Mitchell Wand. Compiler Correctness for Parallel Languages. In *Proceedings of Functional Programming Languages and Computer Architecture*, pages 120–134, June 1995.

[WB89] Philip Wadler and Stephen Blott. How to Make Ad-Hoc Polymorphism Less Ad-Hoc. In *Proceedings of Principles of Programming Languages*, January 1989.

[WS95] Mitchell Wand and Gregory T Sullivan. A Little Goes a Long Way: A Simple Tool to Support Denotational Compiler-Correctness Proofs. Technical Report NU-CCS-95-19, Northeastern University College of Computer Science, October 1995.