Chapter 1

Compiling Concurrency Correctly: Cutting Out the Middle Man

Liyang HU[1], Graham Hutton[1]
Category: Research

Abstract The standard approach [23] to proving compiler correctness for concurrent languages requires the use of multiple translations into an intermediate process calculus. We present a simpler approach that avoids the need for such an intermediate language, using a new method that allows us to directly establish a bisimulation between the source and target languages. We illustrate the technique on two small languages, using the Agda system to present and formally verify our compiler correctness proofs.

1.1 INTRODUCTION

The standard approach [23] 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:

$$\begin{tikzcd}
\text{Source} \arrow[rr, "\mathsf{compile}"] \arrow[dr, "\mathsf{s}[\![-]\!]"'] & & \text{Target} \arrow[loop right] \arrow[dl, "\mathsf{t}[\![-]\!]"] \\
& \text{Process Calculus} \;/ \approx \arrow[loop below] &
\end{tikzcd}$$

[1] Functional Programming Laboratory, School of Computer Science,
University of Nottingham, NG8 1BB, United Kingdom; {lyh,gmh}@cs.nott.ac.uk

That is, for some given compiler from a source language to a target language, we define separate denotational semantics $\mathsf{s}\llbracket\_\rrbracket$ and $\mathsf{t}\llbracket\_\rrbracket$ 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 if $\mathsf{s}\llbracket p\rrbracket$ and $\mathsf{t}\llbracket\mathsf{compile}\ p\rrbracket$ 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 language is defined by a map $\mathsf{s}\llbracket\_\rrbracket$ 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 endomorphic arrows in the above diagram—with a semantic function $\mathsf{t}\llbracket\_\rrbracket$ mapping the former to the latter. Thus we additionally need to show that the operational semantics of the target language is *adequate* with respect to that of the process calculus via the translation $\mathsf{t}\llbracket\_\rrbracket$.

In this paper, we present a simpler approach that avoids the need for an intermediate process calculus. Our contributions include:

- A *combined semantics* which allows us to establish a direct bisimulation between the source and target languages.
- Illustration of our method on two small example languages, along with their respective virtual machines and compilers.
- A compiler correctness proof for both example languages, using the Agda system for our formal reasoning.

After a brief overview of related work (§1.2), we begin by describing a small nondeterministic language with its associated virtual machine and compiler (§1.3), before introducing and justifying our 'combined semantics' (§1.4). We then state our compiler correctness theorem and outline its proof (§1.5). Sections 1.6 and 1.7 scale the technique to a more general language with concurrency. Finally we conclude with some reflections on Agda and outline directions for future work.

This paper is aimed at the reader with functional programming background and an interest in the semantics or implementation of concurrency. We make use of Agda [17, 20]—a dependently typed programming language and interactive proof assistant—as a vehicle for expressing many of the ideas in this paper, as well as a formal means of checking our proofs. Knowledge of Agda is not a requirement and we will introduce any concepts when necessary. We originally used standard Agda colouring conventions throughout to aid readability; while not reproduced here in print, a colour version of our paper is downloadable from `http://liyang.hu/#pub-cccctc`. The full Agda proofs may be found at the same location.

1.2 RELATED WORK

The formal notion of 'compiler correctness' dates back to 1967, when McCarthy and Painter [13] 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 Dave [3] for a detailed bibliography.) However it is only recently, by making essential use of a formal theorem prover, that it has become possible to verify a realistic compiler in full. In particular, Leroy [12] has produced a certified compiler for a C-like core language in the Coq [21] 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 80s, Wand and Sullivan [24] 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 and Sullivan then adapt this paradigm to the concurrent setting [23], which is further elaborated by Gladstein [5, 4].

Our work in this paper follows on from Hutton and Wright [11], 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 Hutton and Wright [11], the addition of effects and concurrency requires an intensional notion of comparing intermediate actions via a suitable notion of bisimulation. The purpose of this paper is to explore this idea, while retaining the approach of directly relating the source and target without the need for an intermediate language.

1.3 A NON-DETERMINISTIC LANGUAGE

In order to focus on the essence of this problem, we abstract from the details of a real language and consider a simple expression language consisting of integers and addition [9, 10, 11]. This minimal language suffices for explaining our basic ideas. We give it its usual 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 for the language complete the definition. We present and justify a novel technique for proving compiler correctness in the presence of non-determinism.

1.3.1 Expression Syntax and Semantics

Let us begin by defining the syntax of our expression language:

$$\mathsf{Expr} ::= \mathbb{N} \mid \mathsf{Expr} \oplus \mathsf{Expr}$$

While seemingly simplistic, this language nevertheless contains the essential computational aspects we desire. In particular, the notion of a monad [16, 22] has been widely used within the literature as a basis for computation. In our language, the monoid $(\mathbb{N}, 0, \oplus)$ may be seen as a degenerate monad. This simplification allows us to avoid orthogonal issues, namely binding and substitution. We maintain the key aspect of sequencing by giving our language a left-to-right evaluation order.

We can express the above definition as an Agda datatype:

```
data Expr : Set where
  val  : ℕ    → Expr
  _⊕_ : Expr → Expr → Expr
```

The syntax is reminiscent of ‘generalised algebraic datatypes’ (GADTs) in Haskell, with the exception of Expr : Set above, where Set denotes the built-in ‘type of types’. The underscores either side of $\oplus$ denote argument positions for operators.

We define the operational semantics of expressions in the usual fashion as a transition system, extended with labels to indicate the nature of each transition:

$$\begin{array}{rcl} \mathsf{Action} & ::= & \boxplus \mid \lightning \mid \square\, \mathbb{N} \\ \mathsf{Label} & ::= & \tau \mid\, !\,\mathsf{Action} \\ \_\mapsto<\_>\_ & \subseteq & \mathsf{Expr} \times \mathsf{Label} \times \mathsf{Expr} \end{array}$$

Each transition either emits (denoted by ‘!’) one of $\boxplus$, $\lightning$ or $\square$ (read as ‘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 silent transitions may be identified by simple pattern-matching in our Agda proofs. The corresponding translation of Action and Label is straightforward as usual:

```
data Action : Set where
  ⊞ : Action
  ↯ : Action
  □ : ℕ → Action
data Label : Set where
  τ : Label
  ! : Action → Label
```

The transition rules are presented in the usual natural deduction style. From here on, we shall use m and n for natural numbers, a and b for expressions, α for actions and Λ for labels. Let us first consider the two base rules:

$$\frac{}{m \oplus n \mapsto< !\boxplus > m + n} \qquad (\textsc{Add})$$

$$\frac{}{m \oplus n \mapsto< !\lightning > 0} \qquad (\textsc{Zap})$$

That is, when evaluating the expression $m \oplus n$, one of two things can happen: either the two numbers are summed as usual, or they are ‘zapped’ to zero; each transition is labelled accordingly. The addition of the (ZAP) rule introduces a simple form of non-determinism, as a first step towards moving from a sequential, deterministic language, to a concurrent, non-deterministic language. The remaining pair of rules ensure a left-biased reduction order, as discussed previously:

$$\frac{b \mapsto< \Lambda > b'}{m \oplus b \mapsto< \Lambda > m \oplus b'} \qquad \text{(ADDR)}$$

$$\frac{a \mapsto< \Lambda > a'}{a \oplus b \mapsto< \Lambda > a' \oplus b} \qquad \text{(ADDL)}$$

In Agda, we can encode the _↦<_>_ relation directly as a datatype, where each transition rule gives rise to a constructor. As Agda is a dependently typed language, we may parametrise datatype definitions by values as well as types. In this instance, _↦<_>_ is indexed by a pair of expressions, along with a label:

```
data _↦<_>_ : Expr → Label → Expr → Set where
  ↦-⊞ :                   val m ⊕ val n ↦< !⊞ > val (m + n)
  ↦-↯ :                   val m ⊕ val n ↦< !↯ > val 0
  ↦-R : b ↦< Λ > b′ → val m ⊕ b      ↦<  Λ > val m ⊕ b′
  ↦-L : a ↦< Λ > a′ →     a ⊕ b      ↦<  Λ >     a′ ⊕ b
```

Under the above encoding, an expression of type $a \mapsto< \Lambda > b$ may be viewed as a witness for a single-step transition from a to b, labelled with Λ. Because Agda is a total language, there is no $\perp$ or 'bottom' value, so any type-correct expression really is a witness (or proof) of the corresponding type (or proposition).

Choice of Action Set

An unanswered question so far is: 'how was the set of actions chosen?' As we shall see later in §1.4, we wish to distinguish between different choices in the reduction path a given expression can take. In this instance, we need to know which of the (ADD) or (ZAP) rules were applied, hence the use of distinct actions ⊞ and ↯ respectively. Later in §1.4.1 we also wish to distinguish between different final results for an expression, which are revealed using the □ action.

1.3.2 Compiler, Virtual Machine and its Semantics

The virtual machine for our language has a simple stack-based design, with only two instructions, defined as follows:

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

A program comprises a list of such instructions. The compiler for our expression language is as shown below. In order to make our proofs more straightforward, we take a code continuation as an additional argument [8], which corresponds to writing the compiler in a accumulator-passing style:

```
Program : Set
Program = List Instruction

compile : Expr → Program → Program
compile (val m) c = PUSH m  :: c
compile (a ⊕ b) c = compile a (compile b (ADD :: c))
```

To execute c : Program, we pair it with a Stack, implemented here as List ℕ. This is precisely how we represent any given state t : Machine of the virtual machine:

```
Stack : Set
Stack = List ℕ

data Machine : Set where
  ⟨_,_⟩ : Program → Stack → Machine
```

Finally, we can specify the operational semantics of the virtual machine through the _↣<_>_ relation:

```
data _↣<_>_ : Machine → Label → Machine → Set where
  ↣-PUSH : ⟨PUSH m :: c,          σ⟩ ↣<  τ > ⟨c,       m :: σ⟩
  ↣-ADD  : ⟨ADD      :: c, n :: m :: σ⟩ ↣< !⊞ > ⟨c, m + n :: σ⟩
  ↣-ZAP  : ⟨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 ⊞ or ↯—in correspondence with the ↣-⊞ and ↣-↯ rules.

1.4 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:

$$\begin{tikzcd}
\text{Source} \arrow[rr, "\mathsf{compile}"] \arrow[dr, "\mathsf{eval}"'] & & \text{Target} \arrow[dl, "\mathsf{exec}"] \\
& \text{Result} &
\end{tikzcd}$$

With a deterministic language and virtual machine—such as our Zap language without the two 'zap' rules—it is natural to use a high-level denotational or big-step semantics for the expression language, which we can realise as an interpreter eval : Expr → ℕ. In turn, the low-level operational or small-step semantics for the virtual machine can be realised as a function exec : Stack → Program →

Stack, that takes an initial stack along with a list of instructions and returns the final stack. Compiler correctness is the statement that:

$$\forall c\,\sigma\,a.\ \mathsf{exec}\ \sigma\ (\mathsf{compile}\ a\ c) \equiv \mathsf{exec}\ (\mathsf{eval}\ a :: \sigma)\ c \tag{\textsc{Det}}$$

or equivalently as the following commuting diagram:

$$\begin{tikzcd}
\forall c\,\sigma. & \mathsf{Expr} \arrow[rr, "\mathsf{compile}\ \_\ c"] \arrow[dr, "\mathsf{exec}\ (\mathsf{eval}\ \_ :: \sigma)\ c"'] & & \mathsf{Program} \arrow[dl, "\mathsf{exec}\ \sigma"] \\
& & \mathsf{Stack} \,/\, \equiv &
\end{tikzcd}$$

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 atop 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 [11], 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 [15], which we shall make concrete in §1.4.2. As we shall see, the local reasoning afforded by this approach also leads to simpler and more natural proofs.

1.4.1 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 a detour [23] via an intermediate process calculus and allows us to directly establish a bisimulation between the source and target languages. Our approach to non-deterministic compiler correctness—making use of such a combined machine—is illustrated below:

$$\begin{tikzcd}
\text{Source} \arrow[loop left] \arrow[rr, "\mathsf{compile}"] \arrow[dr, "\mathsf{lift}_{\mathsf{S}}"'] & & \text{Target} \arrow[loop right] \arrow[dl, "\mathsf{lift}_{\mathsf{T}}"] \\
& \text{Combined} \,/\, \approx \arrow[loop below] &
\end{tikzcd}$$

In the case of our Zap language, a combined machine x : Combined has three distinct phases of execution,

```
data Combined : Set where
  ⟨_,_⟩ : Expr → Machine → Combined
  ⟨_⟩   : Machine → Combined
  ⟨ ⟩   : Combined
```

whose semantics is defined by the following transition relation:

```
data _↠<_>_ : Combined → Label → Combined → Set where
  ↠-↦      : a ↦< Λ > b → ⟨a ,   t⟩ ↠<  Λ   > ⟨b , t ⟩
  ↠-↣      : t ↣< Λ > u → ⟨      t⟩ ↠<  Λ   > ⟨    u⟩
  ↠-switch :  ⟨val m , ⟨c   , σ      ⟩⟩ ↠<  τ   > ⟨⟨c , m ∷ σ⟩⟩
  ↠-done   :  ⟨          ⟨[ ] , m ∷ [ ]⟩⟩ ↠< !□ m > ⟨ ⟩
```

The first constructor ⟨_,_⟩ of Combined pairs an expression with a virtual machine continuation. In this initial phase, a combined machine $\langle a , \langle c , \sigma \rangle \rangle$ can be understood as the small-step analogue of the right side of the (DET) 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 ⟨_⟩ constructor. This is the small-step analogue of pushing the result of eval a onto σ, again following the right side of (DET), namely exec (eval a ∷ σ) c.

The second Combined constructor ⟨_⟩ 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 exec σ c, which matches the left side of (DET), and also the right side after the evaluation of the embedded expression has completed.

Lastly, the ↠-done rule reveals the computed result using the □ action, and terminates with the empty ⟨ ⟩ state. This construction allows us to distinguish between final result values using only the basic notion of bisimulation.

1.4.2 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’ _⤇<_>_ in terms of our existing _↠<_>_ relation, where only Actions are exposed:

$$\frac{x \twoheadrightarrow\!<\!\tau\!>^{\star} x' \quad x' \twoheadrightarrow\!<\, !\,\alpha\, >\, y' \quad y' \twoheadrightarrow\!<\!\tau\!>^{\star} y}{x \Mapsto\!<\, \alpha\, >\, y}$$

We define _↠<τ>⋆_ to be the reflexive, transitive closure of _↠< τ >_. The above relation is implemented by the following datatype:

```
data _⤇<_>_ : Combined → Action → Combined → Set where
  ⤇ : x ↠<τ>⋆ x′ → x′ ↠< !α > y′ → y′ ↠<τ>⋆ y → x ⤇< α > y
```

The 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:

$$\begin{aligned} x \approx y \quad \Longleftrightarrow \quad & \forall\, x', \alpha.\ x \Mapsto< \alpha > x' \implies \exists\, y'.\ y \Mapsto< \alpha > y' \wedge x' \approx y' \\ \wedge & \forall\, y', \alpha.\ y \Mapsto< \alpha > y' \implies \exists\, x'.\ x \Mapsto< \alpha > x' \wedge y' \approx x' \end{aligned}$$

This corresponds to the following Agda definition, in which we encode conjunctions (∧) as products (×):

```
data _≈_ : Combined → Combined → Set where
  bisim : ∀{x y} →
    (∀{x′ α} → x ⤇< α > x′ → ∃ λ y′ → y ⤇< α > y′ × x′ ≈ y′) →
    (∀{y′ α} → y ⤇< α > y′ → ∃ λ x′ → x ⤇< α > x′ × y′ ≈ x′) →
    x ≈ y
```

It is straightforward to show that ≈ is an equivalence relation:

```
≈-reflexive  : x ≈ x
≈-symmetric  : x ≈ y → y ≈ x
≈-transitive : x ≈ y → y ≈ z → x ≈ z
```

1.5 COMPILER CORRECTNESS FOR ZAP

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 a followed by c is weakly bisimilar to the reduction of the expression a followed by the machine continuation $\langle c\,, \sigma \rangle$,

```
correctness : ⟨⟨compile a c , σ⟩⟩ ≈ ⟨a , ⟨c , σ⟩⟩
```

or equivalently as the following commuting diagram:

$$\begin{tikzcd}
\forall c\ \sigma.\ \mathsf{Expr} \arrow[rr, "\mathsf{compile}\ \_\ c"] \arrow[dr, "\langle \_ ,\, \langle c\,, \sigma \rangle \rangle"'] & & \mathsf{Program} \arrow[dl, "\langle \langle \_ ,\, \sigma \rangle \rangle"] \\
& \mathsf{Combined}\ /\ \approx &
\end{tikzcd}$$

In particular, instantiating c and σ to empty lists results in the corollary that, for any arbitrary expression a, $\langle\langle \mathsf{compile}\ a\ [\,]\,, [\,]\rangle\rangle \approx \langle a\,, \langle [\,]\,, [\,]\rangle\rangle$ holds.

We can prove correctness by structural induction on the expression a; the equational reasoning combinators defined in the Agda standard library [2] allow us to present the proof in a simple calculational style. For example, the inductive case is given below:

```
correctness {a ⊕ b} {c} {σ} =
  begin
    ⟨⟨compile (a ⊕ b) c , σ⟩⟩
  ≡⟨ byDefinition ⟩
    ⟨⟨compile a (compile b (ADD :: c)) , σ⟩⟩
  ≈⟨ correctness ⟩
    ⟨a , ⟨compile b (ADD :: c) , σ⟩⟩
  ≈⟨ eval-left ⟩
    ⟨a ⊕ b , ⟨c , σ⟩⟩
  □
```

This case makes use of an eval-left lemma, which in turn uses eval-right, which finally depends on ADD≈m⊕n. These three lemmas are stated as follows:

eval-left	: ⟨ a , ⟨compile b (ADD :: c) , σ⟩⟩	≈	⟨ $a \oplus b$,	⟨c , σ⟩⟩
eval-right	: ⟨ b , ⟨ADD :: c , m :: σ⟩⟩	≈	⟨val $m \oplus b$,	⟨c , σ⟩⟩
ADD≈m⊕n	: ⟨val n , ⟨ADD :: c , m :: σ⟩⟩	≈	⟨val m ⊕ val n ,	⟨c , σ⟩⟩

The eval-left lemma asserts that evaluating the left argument of ⊕, then executing the compiled code corresponding to the right argument followed by an ADD instruction, is bisimilar to evaluating the original ⊕ expression. The eval-right lemma is the analogue of eval-left for the right argument of ⊕. The proofs for these two lemmas proceed by induction on the size of an expression.

$$\begin{tikzcd}
 & & \langle \text{val } m \oplus \text{val } n , \langle c , \sigma \rangle\rangle \arrow[lldd, bend right, "!\boxplus"', "\twoheadrightarrow\text{-}\mapsto\ \mapsto\text{-}\boxplus"] \arrow[rrdd, bend left, "!↯", "\twoheadrightarrow\text{-}\mapsto\ \mapsto\text{-}↯"'] \arrow[d, phantom, "\wr\!\wr\ \text{ADD}\approx\text{m}\oplus\text{n}"] & & \\
 & & \langle \text{val } n , \langle \text{ADD} :: c , m :: \sigma \rangle\rangle \arrow[dd, "\tau\ \ \wr\!\wr\ \text{elide-}\tau"', "\twoheadrightarrow\text{-switch}"] & & \\
\langle \text{val } (m+n) , \langle c , \sigma \rangle\rangle \arrow[dd, "\wr\!\wr\ \text{elide-}\tau\ \ \tau"', "\twoheadrightarrow\text{-switch}"] & & & & \langle \text{val } 0 , \langle c , \sigma \rangle\rangle \arrow[dd, "\tau"', "\twoheadrightarrow\text{-switch}\ \ \wr\!\wr\ \text{elide-}\tau"] \\
 & & \langle\langle \text{ADD} :: c , n :: m :: \sigma \rangle\rangle \arrow[lld, "!\boxplus"', "\twoheadrightarrow\text{-}\rightarrowtail\ \rightarrowtail\text{-ADD}"] \arrow[rrd, "!↯", "\twoheadrightarrow\text{-}\rightarrowtail\ \rightarrowtail\text{-ZAP}"'] & & \\
\langle\langle c , m+n :: \sigma \rangle\rangle & & & & \langle\langle c , 0 :: \sigma \rangle\rangle
\end{tikzcd}$$

FIGURE 1.1. Proof sketch for the ADD≈m⊕n lemma.

The non-determinism in our language comes into play only when considering the ADD≈m⊕n lemma. A sketch of its proof may be recovered by chasing the arrows in Figure 1.1, in which nodes represent machine states, while arrows (decorated with their respective labels and witnesses) correspond to transitions. Suppose we take as a starting point either side of the ADD≈m⊕n bisimilarity, given by the nodes $\langle \mathsf{val}\ n , \langle \mathsf{ADD} :: c , m :: \sigma \rangle \rangle$ and $\langle \mathsf{val}\ m \oplus \mathsf{val}\ n , \langle c , \sigma \rangle \rangle$ at the top of the figure. Whether we take the left branch (corresponding to the two rules for addition) or the right (corresponding to the two ‘zap’ rules), the other state can always follow, emitting the same action. The resulting states are bisimilar by the elide-τ lemma, defined below.

The elide-τ *Lemma*

A key lemma used throughout our correctness proofs states that if there exists a silent transition between two states x and y, then x and y are bisimilar:

```
elide-τ : x ↠< τ > y → x ≈ y
```

In one direction of $x \approx y$, 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 other direction, the proof relies on the fact that wherever there is a choice in the reduction of any given state, each possible transition is identified with a distinct non-silent label. Hence a witness to $x \twoheadrightarrow< \tau > y$ implies that this must be the *unique* transition that can be made from x. From this, it is then immediate that y can match any visible transition made by x.

1.6 AN EXPLICITLY CONCURRENT LANGUAGE

In the previous section, we have described our methodology for tackling the question of compiler correctness for a simple non-deterministic language. In this section, we shall demonstrate that our technique scales to a more explicit form of non-determinism, namely the dynamic creation of concurrently executing threads.

1.6.1 Expression and Virtual Machine Syntax

As with the Zap language, we base our language on that of natural numbers and addition. We build on this by adding a fork primitive, which introduces a simple and familiar approach to explicit concurrency:

```
data Expr : Set where
  val   : ℕ → Expr
  _⊕_  : Expr → Expr → Expr
  fork  : Expr → Expr
```

An expression fork a will begin evaluation of a in a new thread, immediately returning val 0, in a manner reminiscent of Haskell’s forkIO [19] primitive. Similarly, we extend the virtual machine with a FORK instruction, which spawns the sequence of instructions in a new thread:

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

The compiler remains unchanged, except an extra case for fork:

```
Program : Set
Program = List Instruction

Stack : Set
Stack = List ℕ

compile : Expr → Program → Program
compile (val m) c = PUSH m :: c
compile (a ⊕ b) c = compile a (compile b (ADD :: c))
compile (fork a) c = FORK (compile a [ ]) :: c
```

As before, we represent each virtual machine Machine thread by a pair Program × Stack: a sequence of instructions, together with a stack of natural numbers.

1.6.2 Actions and Semantics

We extend the set of actions by ^{+}a, to indicate the spawning of a new thread a, and the action … to indicate preemption of the foreground thread:

```
data Action (l : Set) : Set where
  τ    : Action l
  ⊞    : Action l
  +_   : l → Action l
  □_   : ℕ → Action l
  …_   : Action l → Action l
```

Due to the addition of explicit concurrency in this Fork language, we no longer require the 'zap' action or its associated semantics. The definition of Action is parametrised by either expressions or virtual machines.

With the Zap language, a single τ label sufficed, because the semantics did not diverge at the 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 a given point. We had previously mandated that distinct choices in the reduction path must be labelled with distinct actions: in this case, we folded the τ symbol into the definition of Action, such that both τ and … τ are considered to be silent, yet they remain distinct.

This choice does complicate matters somewhat: previously, we could syntactically match a Label with τ to determine if a transition was silent; in the same way, we know *a priori* that Action-labelled transitions cannot be silent. Here we use a more elaborate scheme:

```
data Silent {l : Set} : Action l → Set where
  is-τ   : Silent τ
  is-… : ∀ {α} → Silent α → Silent (… α)
```

This definition of the Silent indexed datatype encodes a predicate on actions: Silent α is inhabited precisely when α is silent.

It remains for us to define the semantics for expressions,

```
data _↦<_>_ : Expr → Action Expr → Expr → Set where
  ↦-⊞ :                      val m ⊕ val n ↦< ⊞ >   val (m + n)
  ↦-R  : b ↦< Λ > b′ → val m ⊕ b      ↦< Λ >   val m ⊕ b′
  ↦-L  : a ↦< Λ > a′ →       a ⊕ b      ↦< Λ >       a′ ⊕ b
  ↦-⁺ :                              fork a     ↦< ⁺ a > val 0
```

and virtual machines,

```
data _↣<_>_ : Machine → Action Machine → Machine → Set where
  ↣-PUSH : ⟨PUSH m :: c,          σ⟩ ↣< τ >             ⟨c,       m :: σ⟩
  ↣-ADD   : ⟨ADD       :: c, n :: m :: σ⟩ ↣< ⊞ >             ⟨c, m + n :: σ⟩
  ↣-FORK : ⟨FORK c′ :: c,          σ⟩ ↣< ⁺ ⟨c′, [ ]⟩ > ⟨c,        0 :: σ⟩
```

labelled with the above action set. In both cases, the $^{+}$ action carries the forked child as a parameter.

1.7 CONCURRENT COMPILER CORRECTNESS

1.7.1 Combined Machines, Thread Soups and Semantics

Our definition of a combined machine remains unchanged from the Zap language, with the constructors ⟨_, _⟩, ⟨_⟩ and ⟨ ⟩ corresponding to the three phases of execution. We model a 'thread soup' [18] as a List of combined machines, and define a transition relation _↠<_>_ on said thread soups:

```
Soup : Set
Soup = List Combined

data _↠<_>_ : Soup → Action ⊤ → Soup → Set where
  ↠-↦          : a ↦< α > b → ⟨a, t⟩ :: s ↠< sip_E α > ⟨b, t ⟩ :: soupCon_E  α s
  ↠-↣          : t ↣< α > u →      ⟨ t⟩ :: s ↠< sip_M α >    ⟨ u ⟩ :: soupCon_M α s
  ↠-preempt : r ↠< α > s →         x  :: r ↠< … α >          x   :: s
  ↠-switch   :               ⟨val m, ⟨c, σ⟩⟩ :: s ↠< τ > ⟨⟨c, m :: σ⟩⟩  :: s
  ↠-done      :               ⟨⟨ [ ] , m :: [ ]⟩⟩ :: s ↠< □ m >           ⟨ ⟩ :: s
```

As in the Zap language, the ↠-↦ and ↠-↣ rules inherit the transitions of expressions and virtual machines, along with ↠-switch and ↠-done for housekeeping. The $\mathsf{sip}_{\mathsf{E}}$ function lifts an Action Expr to an Action Combined; the $\mathsf{soupCon}_{\mathsf{E}}$ helper prepends the forked expression to the soup in the case of a $^{+}$ action and otherwise leaves the soup unchanged. Corresponding helpers $\mathsf{sip}_{\mathsf{M}}$ and $\mathsf{soupCon}_{\mathsf{M}}$ act on virtual machines instead. Finally, we allow thread interleaving via the ↠-preempt rule.

1.7.2 Fork Compiler Correctness Proof

For our Fork language, we take the same definition of weak bisimilarity as that of the Zap language, but parametrised on an updated visible transition relation:

```
data _⤇<_>_ : Combined → Action Combined → Combined → Set where
  ⤇ : x ↠<τ>⋆ x′ → x′ ↠< α > y′ → y′ ↠<τ>⋆ y → ¬ Silent α →
    x ⤇< α > y
```

Previously we were able to enforce syntactically that α is a visible action; now, we must include ¬ Silent α as an additional side-condition. Conversely, as there are multiple actions which we consider to be silent, we must define the binary silent transition relation _↠<τ>_ as a dependent triple comprising an action α, a witness of Silent α, along with the α-transition itself:

```
_↠<τ>_ : Soup → Soup → Set
r ↠<τ> s = ∃ λ α → Silent α × r ↠< α > s
```

As before, we take _↠<τ>⋆_ to be the reflexive, transitive closure of _↠<τ>_.

The previous correctness theorem is then further generalised over s : Soup:

```
correctness : ⟨a , ⟨c , σ⟩⟩ :: s ≈ ⟨⟨compile a c , σ⟩⟩ :: s
```

We prove correctness by first showing that the _↠<_>_ transition relation is well-founded and then proceeding by induction on said transition. Details of the completed proof in Agda may be found on the authors' websites[2].

1.8 CONCLUSION AND FURTHER WORK

In this article we have presented a new approach to compiler correctness for concurrent languages which avoids the need for an intermediate process calculus as used in previous work [23]. In particular, by generalising the usual deterministic compiler correctness statement to a non-deterministic setting, we are able to establish a direct bisimulation between the source and target languages. In retrospect, our generalisation is both natural and straightforward, but surprisingly this approach appears not to have been considered in the literature.

The use of Agda has been a key aspect of our work. As one would expect, the use of a formal tool ensures that our definitions and proofs are free from ambiguities and omissions, and provides a mechanical guarantee of the correctness of our results. In our experience, Agda has also proved invaluable in the *development* of our definitions, theorems and proofs. While it is a matter of personal preference, some find 'proof scripts' à la Coq [21] difficult to understand offline, whereas the direct manipulation of proof terms in Agda seems both more natural and less fragile with respect to changes. The direct application of the Curry–Howard correspondence blurs the distinction between producing proofs and programming, which allows us to take advantage of our intuitions from both activities. At present, Agda's most prominent downside is its immaturity relative to systems that come with well-developed libraries and tools such as Coq, although the Agda standard library [2] is evolving at a rapid pace.

There are a number of possible directions for future work. First of all, it is important to consider how our approach scales from the minimal languages

[2] http://liyang.hu/#pub-cccctc

considered in this paper to more realistic notions of concurrency that include synchronisation and communication, as well as other language features such as name binding, mutable state, input and output, exceptions and interrupts [11]. Indeed, Danielsson has successfully implemented [1] the main result of Hutton and Wright [11] in Agda, making use of a 'combined machine'. The mechanised proof—using the combined semantics—is significantly shorter than the original pen-and-paper proof, which suggests our approach can scale to more elaborate scenarios.

In a similar vein, we are particularly keen to develop a formally verified implementation of a compiler for software transactional memory [7, 6]. Finally, it would also be interesting to attempt to encode the correctness of a compiler for a concurrent language directly in its type, following the lead of McKinna and Wright [14], rather than as a separate theorem.

ACKNOWLEDGEMENTS

We would like to thank the four anonymous referees for their constructive comments and suggestions, which have significantly improved this paper. We are also grateful to the members of the Functional Programming Laboratory in Nottingham for useful discussions and feedback.

REFERENCES

[1] N. A. Danielsson. Personal communication. Available from `http://cs.nott.ac.uk/~nad/listings/Interrupts/`, January 2009.

[2] N. A. Danielsson. The Agda Standard Library. Available from `http://cs.nott.ac.uk/~nad/listings/lib/`, May 2009.

[3] M. A. Dave. Compiler Verification: A Bibliography. *ACM SIGSOFT Software Engineering Notes*, 28(6):2–2, November 2003.

[4] D. S. Gladstein. *Compiler Correctness for Concurrent Languages*. PhD Thesis, Northeastern University, Massachusetts, December 1994.

[5] D. S. Gladstein and M. Wand. Compiler Correctness for Concurrent Languages. In *Proceedings of Coordination*, volume 1061 of *Lecture Notes in Computer Science*. Springer, April 1996.

[6] T. Harris, S. Marlow, S. Peyton Jones, and M. P. Herlihy. Composable Memory Transactions. In *Proceedings of Principles and Practice of Parallel Programming*, June 2005.

[7] L. HU and G. Hutton. Towards a Verified Implementation of Software Transactional Memory. In *Proceedings of Trends in Functional Programming*, May 2008.

[8] G. Hutton. *Programming in Haskell*. Cambridge University Press, January 2007.

[9] G. Hutton and J. 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.

[10] G. Hutton and J. Wright. Calculating an Exceptional Machine. In *Proceedings of Trends in Functional Programming*, volume 5, February 2006.

[11] G. Hutton and J. Wright. What is the Meaning of These Constant Interruptions? *Journal of Functional Programming*, 17(6):777–792, November 2007.

[12] X. 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.

[13] J. McCarthy and J. Painter. Correctness of a Compiler for Arithmetic Expressions. In *Proceedings of Symposia in Applied Mathematics*, volume 19. AMS, 1967.

[14] J. McKinna and J. Wright. A Type-Correct, Stack-Safe, Provably Correct Expression Compiler in EPIGRAM. To appear in the Journal of Functional Programming, 2009.

[15] R. Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, 1989.

[16] E. Moggi. Computational Lambda-Calculus and Monads. In *Proceedings of Logic in Computer Science*, pages 14–23. IEEE Computer Society Press, June 1989.

[17] U. Norell. *Towards a Practical Programming Language Based on Dependent Type Theory*. PhD thesis, Chalmers University of Technology, September 2007.

[18] S. 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.

[19] S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Proceedings of Principles of Programming Languages*, pages 295–308, 1996.

[20] The Agda Team. The Agda Wiki. Available from `http://wiki.portal.chalmers.se/agda/`, May 2009.

[21] The Coq Development Team. The Coq Proof Assistant. Available from `http://coq.inria.fr/`, June 2008.

[22] P. Wadler. Comprehending Monads. *Mathematical Structures in Computer Science*, 2:461–493, 1992.

[23] M. Wand. Compiler Correctness for Parallel Languages. In *Proceedings of Functional Programming Languages and Computer Architecture*, pages 120–134, June 1995.

[24] M. Wand and G. 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.