Compiling Concurrency Correctly
Cutting out the Middle Man

Liyang HU and Graham Hutton
{lyh,gmh}@cs.nott.ac.uk

Functional Programming Laboratory
School of Computer Science
University of Nottingham Nottingham, England
United Kingdom of Great Britain and Northern Ireland

Trends in Functional Programming
Komarno, Slovakia
09:30 (Local Time), 2nd June, 2009

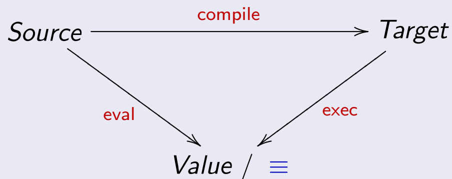
Allergy Advice

This talk may contain. . .

- Compiler Correctness
- Addition and the Natural Numbers
- Non-Determinism
- Concurrency
- and a soupçon of Martin-Löf Type Theory-flavoured Mathematics

What Do You Mean, Compiler Correctness?

Compiler Correctness



What Do You Mean, Compiler Correctness?

Learning the Local Lingo (Intermediate)

- Take a source language, for example $(\mathbb{N}, \oplus)$:

$$\mathsf{E} ::= \mathsf{val}\ \mathbb{N} \mid \mathsf{E} \oplus \mathsf{E}$$

- with an appropriate (big-step) evaluator:

$$\begin{aligned}
&\mathsf{eval} : \mathsf{E} \rightarrow \mathbb{N} \\
&\mathsf{eval}\ (\mathsf{val}\ m) = m \\
&\mathsf{eval}\ (a \oplus b) = \mathsf{eval}\ a + \mathsf{eval}\ b
\end{aligned}$$

What Do You Mean, Compiler Correctness?

Learning the Local Lingo (Advanced)

- Along with a corresponding virtual (stack) machine:

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

- and a suitable (big-step) interpretor:

$$\begin{aligned}
&\mathsf{exec} : \mathsf{List}\ \mathbb{N} \longrightarrow \mathsf{List}\ \mathsf{I} \longrightarrow \mathsf{List}\ \mathbb{N}\\
&\mathsf{exec}\ \sigma\ [] = \sigma\\
&\mathsf{exec}\ \sigma\ (\mathsf{PUSH}\ m :: c) = \mathsf{exec}\ (m :: \sigma)\ c\\
&\mathsf{exec}\ (n :: m :: \sigma)\ (\mathsf{ADD} :: c) = \mathsf{exec}\ (m + n :: \sigma)\ c
\end{aligned}$$

What Do You Mean, Compiler Correctness?

Compiler Correctness

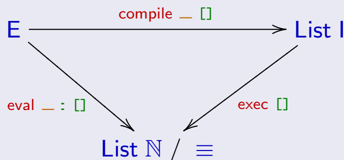
- Given a compiler:

$$\begin{aligned} \text{compile} &: \mathsf{E} \rightarrow \mathsf{List\ I} \rightarrow \mathsf{List\ I} \\ \text{compile}\ (\text{val}\ m)\ \ c &= \text{PUSH}\ m :: c \\ \text{compile}\ (a \oplus b)\ c &= \text{compile}\ a\ (\text{compile}\ b\ (\text{ADD} :: c)) \end{aligned}$$

- Compiler correctness is the statement that:

$$\forall a.\quad \text{eval}\ a :: [] \quad \equiv \quad \text{exec}\ []\ (\text{compile}\ a\ [])$$

  - Alternatively, in diagrammatic form:



  - Proof proceeds more easily if we generalise over c and σ

What Do You Mean, Compiler Correctness?

Compiler Correctness

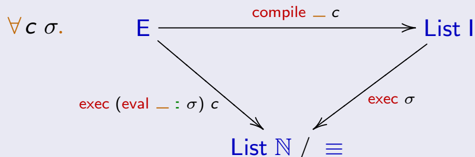
- Given a compiler:

$$\mathsf{compile} : \mathsf{E} \to \mathsf{List\ I} \to \mathsf{List\ I}$$
$$\mathsf{compile}\ (\mathsf{val}\ m)\ \ c = \mathsf{PUSH}\ m :: c$$
$$\mathsf{compile}\ (a \oplus b)\ c = \mathsf{compile}\ a\ (\mathsf{compile}\ b\ (\mathsf{ADD} :: c))$$

- Compiler correctness is the statement that:

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

  - Alternatively, in diagrammatic form:



  - Proof proceeds more easily if we generalise over c and σ

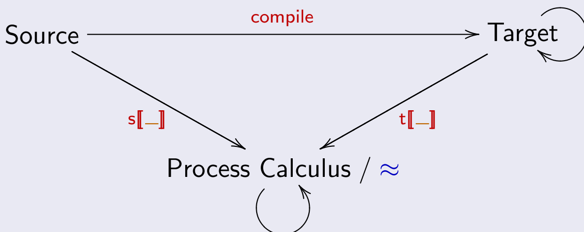
What Do You Mean, Compiler Correctness?

What about. . .

- Small-step or operational semantics?
    - Take the transitive closure of reduction relation?
- Non-determinism and concurrency?
    - Generalise to *Sets* of results?

Existing Approach

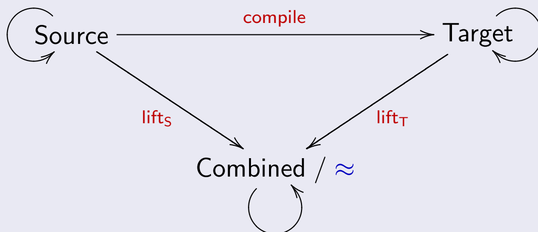
Compiler Correctness for Parallel Languages (Wand, 1995)



- Target language has a binary small-step reduction relation
- Process Calculus has a ternary labelled state transition relation
- Bisimilar (≈) systems $\iff$ indistinguishable by observer
- Compiler correctness $\iff$ $\forall p.\ s[\![p]\!] \approx t[\![\text{compile } p]\!]$
  - Target operational semantics must also be adequate w.r.t. PC

Our Approach

Compiling Concurrency Correctly (HU and Hutton, 2009)



- Labelled transitions for both source and target languages
- Combined semantics inherited directly from source and target
- Compiler correctness $\iff \forall p.\ \mathrm{lift}_S\ p \approx \mathrm{lift}_T\ (\mathrm{compile}\ p)$
  - . . . for some generalisation of compile, lift_S and lift_T

A Simple Language of Naturals and Addition

Expression Syntax, Actions and Labels

$\mathsf{E} ::= \mathsf{val}\ \mathbb{N} \mid \mathsf{E} \oplus \mathsf{E} \quad \mathsf{Action} ::= \boxplus \mid \lightning \mid \square\ \mathbb{N} \quad \mathsf{Label} ::= \tau \mid\ !\,\mathsf{Action}$

Expression Semantics

$$_ \mapsto\!<_>_ \subseteq \mathsf{E} \times \mathsf{Label} \times \mathsf{E}$$

$$\mathsf{val}\ m \oplus \mathsf{val}\ n \mapsto\!<\ !\boxplus\ > \mathsf{val}\ (m + n) \qquad (\mapsto\text{-}\boxplus)$$

$$\mathsf{val}\ m \oplus \mathsf{val}\ n \mapsto\!<\ !\lightning\ > \mathsf{val}\ 0 \qquad (\mapsto\text{-}\lightning)$$

$$\frac{b \mapsto\!<\Lambda>\ b'}{\mathsf{val}\ m \oplus b \mapsto\!<\Lambda>\ \mathsf{val}\ m \oplus b'} \qquad (\mapsto\text{-R})$$

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

A Simple Language of Naturals and Addition

Just Natural Numbers and Addition?

- Sufficient to capture notion of sequencing of computations
  - i.e. left-to-right evaluation semantics
  - Formally, $(\mathbb{N}, +)$ is a monoid — a degenerate form of a monad. . .
- Abstract from unrelated details of a real language
- Focus on the essence of the problem
  - i.e. how to deal with non-determinism

Action Set

- Identify branches in reduction path ($\boxplus$ and $\lightning$)
  - Distinct branches labelled with distinct actions
- Compare final results ($\square\ \mathbb{N}$)

A Virtual Machine for Naturals and Addition

Instruction Set and Machine State

$$\mathsf{I} ::= \mathsf{PUSH}\ \mathbb{N} \mid \mathsf{ADD} \qquad \mathsf{M} ::= \langle \mathsf{List}\ \mathsf{I}\ ,\ \mathsf{List}\ \mathbb{N} \rangle$$

Virtual Machine Operational Semantics

$$_\rightarrowtail\!<_>_ \subseteq \mathsf{M} \times \mathsf{Label} \times \mathsf{M}$$

$$\langle \mathsf{PUSH}\ m :: c\ ,\ \sigma \rangle \rightarrowtail\!< \tau > \langle c\ ,\ m :: \sigma \rangle \qquad (\rightarrowtail\text{-PUSH})$$

$$\langle \mathsf{ADD} :: c\ ,\ n :: m :: \sigma \rangle \rightarrowtail\!< !\boxplus > \langle c\ ,\ m + n :: \sigma \rangle \qquad (\rightarrowtail\text{-ADD})$$

$$\langle \mathsf{ADD} :: c\ ,\ n :: m :: \sigma \rangle \rightarrowtail\!< !\lightning > \langle c\ ,\ 0 :: \sigma \rangle \qquad (\rightarrowtail\text{-ZAP})$$

Compiler Correctness for Concurrent Languages

Compiler

$$\mathsf{compile} : \mathsf{E} \rightarrow \mathsf{List}\ \mathsf{I} \rightarrow \mathsf{List}\ \mathsf{I}$$
$$\mathsf{compile}\ (\mathsf{val}\ m)\ \ c = \mathsf{PUSH}\ m :: c$$
$$\mathsf{compile}\ (a \oplus b)\ c = \mathsf{compile}\ a\ (\mathsf{compile}\ b\ (\mathsf{ADD} :: c))$$

Concurrent Correctness?

- Executing compile a [] '*behaves the same*' as evaluating a
- Both source and target semantics are small-step
  - compile a reduces according to $_\rightarrowtail\!<_>_$
  - a reduces according to $_\mapsto\!<_>_$
- We want to reason...
  - Intensionally, rather than extensionally
  - Locally, rather than globally
- Demands a suitable notion of 'branching equivalence'
  - We chose our action set Action to identify such branching!

Compiler Correctness for Concurrent Languages

Bisimulation

- Simulation: "*anything you can do, I can do (better)*"

  — Irving Berlin, *Annie Get Your Gun* (1946)

- Bisimulation: given a labelled transition relation $_ \rightsquigarrow < _ > _$,

$$x \approx y \iff$$
$$\forall x', \alpha.\ x \rightsquigarrow < \alpha > x' \wedge \exists y'.\ y \rightsquigarrow < \alpha > y' \wedge x' \approx y'$$
$$\wedge \forall y', \alpha.\ y \rightsquigarrow < \alpha > y' \wedge \exists x'.\ x \rightsquigarrow < \alpha > x' \wedge y' \approx x'$$

- Equivalence relation: reflexive, symmetric, transitive

Combined Semantics

Deterministic Compiler Correctness, Revisited

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

Combined Machine and Semantics

$$\mathsf{C} ::= \langle \mathsf{E}\,,\ \mathsf{M} \rangle \mid \langle \mathsf{M} \rangle \mid \langle\,\rangle \qquad _ \twoheadrightarrow\!<_>_ \subseteq \mathsf{C} \times \mathsf{Label} \times \mathsf{C}$$

$$\frac{a\ \mapsto\!<\Lambda>\ b}{\langle a\,,\ t \rangle\ \twoheadrightarrow\!<\Lambda>\ \langle b\,,\ t \rangle} \quad (\twoheadrightarrow\text{-}\mapsto)$$

$$\frac{t\ \rightarrowtail\!<\Lambda>\ u}{\langle t \rangle\ \twoheadrightarrow\!<\Lambda>\ \langle u \rangle} \quad (\twoheadrightarrow\text{-}\rightarrowtail)$$

$$\langle \mathsf{val}\ m\,,\ \langle c\,,\ \sigma \rangle \rangle\ \twoheadrightarrow\!<\tau>\ \langle \langle c\,,\ m :: \sigma \rangle \rangle \quad (\twoheadrightarrow\text{-switch})$$

$$\langle \langle [\,]\,,\ m :: [\,] \rangle \rangle\ \twoheadrightarrow\!<\,!\square\ m>\ \langle\,\rangle \quad (\twoheadrightarrow\text{-done})$$

Compiling Non-Determinism Correctly

Visible Transitions

- Suppress silent τ transitions:

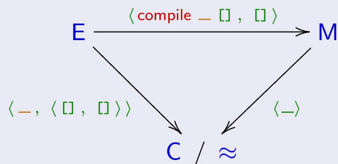
$$\frac{x \twoheadrightarrow\!< \tau >^{\star} x' \quad x' \twoheadrightarrow\!< !\alpha > y' \quad y' \twoheadrightarrow\!< \tau >^{\star} y}{x \Mapsto\!< \alpha > y}$$

Non-Deterministic Compiler Correctness

- Compiler Correctness Theorem:

$$\forall a. \quad \langle a , \langle [] , [] \rangle \rangle \approx \langle \langle \text{compile } a \; [] , [] \rangle \rangle$$

  - Alternatively, in diagrammatic form:



- For proof, see paper, available online, later

Compiling Non-Determinism Correctly

Visible Transitions

- Suppress silent τ transitions:

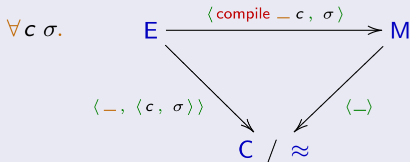
$$\frac{x \twoheadrightarrow\!<\tau>^\star x' \quad x' \twoheadrightarrow\!<!\alpha> y' \quad y' \twoheadrightarrow\!<\tau>^\star y}{x \Mapsto\!<\alpha> y}$$

Non-Deterministic Compiler Correctness

- Compiler Correctness Theorem:

$$\forall a\ c\ \sigma.\ \langle a, \langle c, \sigma \rangle\rangle \approx \langle\langle \mathsf{compile}\ a\ c, \sigma \rangle\rangle$$

  - Alternatively, in diagrammatic form:



- For proof, see paper, available online, later

Compiling Concurrency Correctly

Here's One I Prepared Earlier...

$$E ::= \mathsf{val}\ \mathbb{N} \mid E \oplus E \mid \mathsf{fork}\ E$$

- "Play it again, Sam"
- See paper for details

Conclusion and Future Work

In short. . .

- Write small-step semantics as labelled transition rules
- Generalise deterministic compiler correctness to a small-step scenario
- Much less complex and error-prone than existing technique
- Shown to work with non-trivial example (N. Danielsson, 2009)

Doing Mathematics with Agda

- Agda is. . . *Curry-Howard correspondence in action!*
  - a dependently-typed programming language
  - a proof-assistant based on Martin-Löf Type Theory
- Colouring Convention:
  - relations encoded as types
  - functions
  - rules encoded as constructors
- Invaluable in the *development* of this work

Conclusion and Future Work

It's Not Over ('Til It's Over) — Starship (1987)

- More realistic notions of concurrency, e.g.:
  - Synchronisation
  - Communication
- Interaction with other language features, e.g.:
  - Mutable state
  - Input and output
  - Exceptions and interrupts
- Software transactional memory

Conclusion and Future Work

Thank you for staying awake!

- This slide is intentionally left blank.