Compiler Correctness for Software Transactional Memory

Liyang HU

Foundations of Programming Group
School of Computer Science and IT
University of Nottingham

FoP Away Day, January 17th, 2007

Why Concurrency?

Limits of Technology

- Speed: 4GHz; plateaued over 2 years ago
- Power: 130W(!) from a die less than 15mm by 15mm
- Size: 65nm in 2006 – about 300 atoms across

Recent Trends

- Dual, even quad cores on a single package
- Multiprocessing has arrived for the *mass market*

Concurrent Programming (Is Hard!)

- Market leader: mutual exclusion
- Difficult to reason with

Example

Race Conditions

```
deposit :: Account → Integer → IO ()
deposit account amount = do
   balance ← read account
   write account (balance + amount)
```

Thread		account Balance
A	balanceA ← read account	initial
B	balanceB ← read account	initial
B	write account (balanceB + amountB)	initial + amountB
A	write account (balanceA + amountA)	initial + amountA

Example

Lack of Compositionality

```
deposit :: Account → Integer → IO ()
deposit account amount = do
    lock account
    balance ← read account
    write account (balance + amount)
    release account
transfer :: Account → Account → Integer → IO ()
transfer from to amount = do
    withdraw from amount
    deposit  to   amount
```

Example

Lack of Compositionality (Solution?)

```
deposit :: Account → Integer → IO ()
deposit account amount = do
   balance ← read account
   write account (balance + amount)
transfer :: Account → Account → Integer → IO ()
transfer from to amount = do
   lock from; lock to
   withdraw from amount
   deposit  to   amount
   release from; release to
```

Example

Deadlock

- Thread A: $\mathit{transfer}\ x\ y\ 100$
- Thread B: $\mathit{transfer}\ y\ x\ 200$

Thread		Account x	Account y	Action
A	*lock* x	free	free	acquites lock on x
B	*lock* y	held by A	free	acquires lock on y
B	*lock* x	held by A	held by B	waits for $x\ldots$
A	*lock* y	held by A	held by B	waits for $y\ldots$

Mutual Exclusion

Pitfalls

- Race conditions
- Priority inversion
- Deadlock
- Locking is often *advisory*

Drawbacks

- Correct code does not compose
- Overly conservative
- Granularity versus scalability

Transactional Model

What are Transactions?

- Arbitrary command sequence as an indivisible unit
- Declarative rather than descriptive
- Optimistic execution

Transactional Solution

```
work = do
    begin
    transfer a b 100
    commit
```

Advantages of Transactions

ACID Properties

- Atomicity: all or nothing
    - Fewer interleavings to consider
- Consistency: ensure invariants
    - System-enforced
- Isolation: no observable intermediate state
    - Guaranteed non-interference
- Durability: persistence through system failure
    - Simplifies error-handling
    - Not applicable for transactional *memory*

Optimistic Execution

Transactional Deposit

```
deposit :: Account → Integer → IO ()
deposit account amount = do
    begin
    balance ← read account
       -- another transaction commits, modifying account
    write account (balance + amount)
    commit    -- fails
```

Failure and Retry

- DBMS tracks transaction *dependencies*
- External writes to *account* after initial *read* unacceptable
- Application can *retry* if aborted (not traditionally automatic)

Hardware Assistance

Atomic Instructions

- E.g. fetch-and-add, test-and-set
- Used to efficiently implement *mutual exclusion*

Avoiding Explicit Synchronisation

- Compare-and-Swap
    - `CAS (a), b, c` – if $(a) \equiv b$ then swap (a) with c
- Load-Linked / Store Conditional
    - Load-linked places watch on memory bus; begins ‘transaction’
    - Access to watched location invalidates transaction
    - Store-conditional returns error code on failure
- Still not quite fully-fledged transactions

More Versatility?

Proposed Extensions

- Multi-word CAS
- Hardware Transactional Memory (Herlihy and Moss, 1993)
- Not available on a processor near you. . .

Software Transactional Memory

- Why wait for hardware? (Shavit and Touitou, 1995)
- Typical STM *libraries* difficult to use
- Language extension in Java (Harris and Fraser, 2003)

STM in Haskell

Composable Memory Transactions (Harris et al., 2005)

- Implemented in Glasgow Haskell Compiler
- Library and runtime system only; no language change

STM Haskell Primitives

```
instance Monad STM where { ... }
newTVar   :: STM (TVar α)
readTVar  :: TVar α → STM α
writeTVar :: TVar α → α → STM ()
retry     :: STM α
orElse    :: STM α → STM α → STM α
atomic    :: STM α → IO α
```

Restricting Side-Effects

IO Actions

```
launchMissiles :: IO ()

atomic $ do
  launchMissiles   -- compile-time error: type mismatch
   . . . retry . . .
```

STM Monad

- Irreversible side-effects prohibited – the IO monad
- Can only read/write TVars
- But any *pure* code is allowed

Alternative Blocking

Try Again

- STM Haskell introduces the *retry* keyword
- Used where programs would block, or signal recoverable error

Composition

- *orElse* combines two transactions: *a ‘orElse‘ b*
- Leftist: tries *a* first, returns if *a* returns
- If *a* calls *retry*, attempt *b*; one or the other succeeds

Code Flexibility

Blocking or Non-Blocking?

```
popBlocking    :: TVar [Integer] → STM Integer
popBlocking ts = do
  s ← readTVar ts
  case s of []       → retry
            (x : xs) → do writeTVar xs; return x
popNonblocking :: TVar [Integer] → STM (Maybe Integer)
popNonblocking ts = liftM Just (popBlocking ts)
                      ‘orElse‘ return Nothing
```

- Similarly turn non-blocking into blocking

Formal Semantics

Transition Rule for *atomic*

$$\frac{m \rightarrow^{*} \overline{n}}{atomic\ m \rightarrow \overline{n}}$$

The Need for a Low-Level Semantics

- Mixed big and small step semantics
  - No concurrent/optimistic execution of transactions
  - Doesn't use *logs*, as mentioned in the implementation
- Informal description of implementation
  - No attempt to relate to formal semantics
- How do we show *any* implementation correct?

Simplification of STM Haskell

Syntax

$$\mathrm{E} ::= \mathbb{Z} \mid \mathrm{E} + \mathrm{E} \mid \textbf{rd}\ \mathrm{Name} \mid \textbf{wr}\ \mathrm{Name}\ \mathrm{E} \mid \textbf{atomic}\ \mathrm{E}$$

Comparison with STM Haskell

STM Haskell		Model
$(\gg=)$	$:: \text{STM}\ \alpha \to (\alpha \to \text{STM}\ \beta) \to \text{STM}\ \beta$	$e + f$
return	$:: \alpha \to \text{STM}\ \alpha$	$m \in \mathbb{Z}$
retry	$:: \text{STM}\ \alpha$	
orElse	$:: \text{STM}\ \alpha \to \text{STM}\ \alpha \to \text{STM}\ \alpha$	
readTVar	$:: \text{TVar}\ \alpha \to \text{STM}\ \alpha$	**rd** v
writeTVar	$:: \text{TVar}\ \alpha \to \alpha \to \text{STM}\ ()$	**wr** $v\ e$
newTVar	$:: \text{STM}\ (\text{TVar}\ \alpha)$	
atomic	$:: \text{STM}\ \alpha \to \text{IO}\ \alpha$	**atomic** e

Small-Step Semantics

$$\frac{\langle e,\ \sigma\rangle \longrightarrow \langle e',\ \sigma'\rangle}{\langle e + f,\ \sigma\rangle \longrightarrow \langle e' + f,\ \sigma'\rangle}\ (\textsc{AddL}) \qquad \frac{\langle f,\ \sigma\rangle \longrightarrow \langle f',\ \sigma'\rangle}{\langle n + f,\ \sigma\rangle \longrightarrow \langle n + f',\ \sigma'\rangle}\ (\textsc{AddR})$$

$$\frac{}{\langle n + m,\ \sigma\rangle \longrightarrow \langle \overline{n + m},\ \sigma\rangle}\ (\textsc{Add}\mathbb{Z}) \qquad \frac{}{\langle \mathbf{rd}\ v,\ \sigma\rangle \longrightarrow \langle \sigma(v),\ \sigma\rangle}\ (\textsc{Read})$$

$$\frac{\langle e,\ \sigma\rangle \longrightarrow \langle e',\ \sigma'\rangle}{\langle \mathbf{wr}\ v\ e,\ \sigma\rangle \longrightarrow \langle \mathbf{wr}\ v\ e',\ \sigma'\rangle}\ (\textsc{WriteE}) \qquad \frac{\langle e,\ \sigma\rangle \longrightarrow^{*} \langle n,\ \sigma'\rangle}{\langle \mathbf{atomic}\ e,\ \sigma\rangle \longrightarrow \langle n,\ \sigma'\rangle}\ (\textsc{Atomic})$$

$$\frac{}{\langle \mathbf{wr}\ v\ \overline{n},\ \sigma\rangle \longrightarrow \langle \sigma(v),\ \sigma[v \mapsto n]\rangle}\ (\textsc{Write}\mathbb{Z})$$

Concurrent Evaluation

Expression Soup

$$\mathrm{P} ::= \mathrm{E} \mid \mathrm{P} \talloblong \mathrm{P}$$

$$\frac{\langle e,\ \sigma\rangle \longrightarrow \langle e',\ \sigma'\rangle}{\langle e,\ \sigma\rangle \twoheadrightarrow \langle e,\ \sigma'\rangle}\ (\textsc{Seq})$$

$$\frac{\langle p,\ \sigma\rangle \twoheadrightarrow \langle p',\ \sigma'\rangle}{\langle p \talloblong q,\ \sigma\rangle \twoheadrightarrow \langle p' \talloblong q,\ \sigma'\rangle}\ (\textsc{ParL})$$

$$\frac{\langle q,\ \sigma\rangle \twoheadrightarrow \langle q',\ \sigma'\rangle}{\langle p \talloblong q,\ \sigma\rangle \twoheadrightarrow \langle p \talloblong q',\ \sigma'\rangle}\ (\textsc{ParR})$$

Example

- $\textbf{rd}\ \texttt{"x"} + \textbf{rd}\ \texttt{"x"} \talloblong \textbf{wr}\ \texttt{"x"}\ 1$ — yields 0, 1 or 2
- $\textbf{atomic}\ (\textbf{rd}\ \texttt{"x"} + \textbf{rd}\ \texttt{"x"}) \talloblong \textbf{wr}\ \texttt{"x"}\ 1$ — yields only 0 or 2

Virtual Machine

Instruction Set

```
Instruction ::= PUSH ℤ     | ADD         -- stack machine
              | LOAD Name  | SWAP Name   -- shared store
              | BEGIN      | COMMIT      -- transactions
```

- Typical stack machine with a shared store
- LOAD and SWAP are transaction-local if one is active
- BEGIN marks the start of a transaction
- COMMIT marks the end; retries on failure

Implementation?

- Easiest: stop-the-world; no interleaving of transactions

Logs and Transaction Frames

Goals

1. Isolate changes to global state
2. Re-run transaction on abort

Transaction Frame

We need to record:

1. for each variable accessed,
    - its original value – to check for conflicting commits; and
    - value of writes to it – subsequent reads return this value
2. the transaction's starting address – to re-run if commit fails
3. and strictly speaking, the stack too. . .

Each frame is a pair

$\langle ip, \; rw\rangle \in \text{TransactionFrame} \equiv \text{Instruction}^* \times (\text{Name} \rightarrow \mathbb{Z} \; \times \mathbb{Z})$

Concurrent Execution

Threads

$$\langle ip,\ sp,\ tp\rangle \in \mathsf{Thread} \equiv \mathsf{Instruction}^* \times \mathbb{Z}^* \times \mathsf{TransactionFrame}^*$$

Thread Soup

$$\begin{aligned}\mathsf{Program} ::= {} & \mathsf{Thread} \\ \mid {} & \mathsf{Program} \parallel \mathsf{Program}\end{aligned}$$

- Rules $(\textsc{Seq})$, $(\textsc{ParL})$ and $(\textsc{ParR})$ will suffice
- Threads execute paired with a shared store

Compiler

E to Instruction*

$$
\begin{array}{ll}
\multicolumn{2}{l}{compE \in \mathsf{E} \rightarrow \mathsf{Instruction}^* \rightarrow \mathsf{Instruction}^*} \\
compE\ \overline{n} & c = \mathsf{PUSH}\ \ n : c \\
compE\ (e + f) & c = compE\ e\ (compE\ f\ (\mathsf{ADD} : c)) \\
compE\ (\mathbf{rd}\ v) & c = \mathsf{LOAD}\ v : c \\
compE\ (\mathbf{wr}\ v\ e) & c = compE\ e\ (\mathsf{SWAP}\ v : c) \\
compE\ (\mathbf{atomic}\ e) & c = \mathsf{BEGIN} : compE\ e\ (\mathsf{COMMIT} : c)
\end{array}
$$

P to Program

$$
\begin{array}{ll}
\multicolumn{2}{l}{compP \in \mathsf{P} \rightarrow \mathsf{Program}} \\
compP\ e & = \langle |compE e[]|,\ |[]|,\ |[]| \rangle \\
compP\ (p \parallel q) & = compP\ p \parallel compP\ q
\end{array}
$$

Correctness

Sequential

$\forall e \in \mathrm{E},\ \sigma \in \mathrm{Name} \to \mathbb{Z},\ n \in \mathbb{Z}.$

$$\begin{array}{ccc} \langle e,\ \sigma\rangle & \longrightarrow^* & \langle \overline{n},\ \sigma'\rangle \\ & \text{iff} & \\ \langle\langle compE\ e\ [\,],\ [\,],\ [\,]\rangle , \sigma\rangle & \longrightarrow^* & \langle\langle [\,],\ [n],\ [\,]\rangle , \sigma'\rangle \end{array}$$

Concurrent

$\forall p \in \mathrm{P},\ \sigma \in \mathrm{Name} \to \mathbb{Z},\ ns \in \mathrm{P}.$

$$\begin{array}{ccc} \langle p,\ \sigma\rangle & \twoheadrightarrow^* & \langle ns,\ \sigma'\rangle \\ & \text{iff} & \\ \langle compP\ p,\ \sigma\rangle & \twoheadrightarrow^* & \langle rs,\ \sigma'\rangle \end{array}$$

- $ns \in \mathrm{P}$ contains only integer expressions of the form $\overline{n}$
- $rs \in$ Program structurally identical to ns but with $\overline{n} \mapsto \langle [\,],\ [n],\ [\,]\rangle$

Model Verification

Implementation

- Small-step semantics, compiler and VM in Haskell
- Can express compiler correctness as following function:

$$\begin{aligned}&\mathit{propCC} :: \mathrm{P} \longrightarrow \mathrm{Bool}\\&\mathit{propCC}\ p = (\mathit{result} \circ \mathit{run})\ (p, \sigma_0) \equiv (\mathit{result} \circ \mathit{run})\ (\mathit{compP}\ p, \sigma_0)\end{aligned}$$

QuickCheck

- Generates random input, attempts to falsify proposition:

```
> quickCheck propCC
OK, passed 100 tests.
>
```

- Inspires confidence that a formal proof is possible. . .

Interference and Serialisability

Questions

- What kind of interference can we allow?
- How do we serialise transactions? When do they ‘happen’?

Interfering Transactions

Thread				TVars	
A	B	C	D	x	y
rd $x \mapsto 0$				0	0
	wr x 1			1	0
		wr y (**rd** $x \mapsto 1$)		1	1
rd $y \mapsto 1$				1	1
			wr x 0	0	1
...				0	1
commit?					

Optimistic Speculation

Answers

- Permitted interference?
    - On initial access, bet on variable's final pre-commit value
    - Allow *any* changes, provided original value restored
    - If so, the transaction commits successfully
- At what point does a transaction take place?
    - Certainly not when the transaction begins
    - Pre-commit, x and y matches what thread A initially read
    - Hence, can collapse down to the successful commit point

Read / Write Reordering

- Reads happen immediately
- Writes buffered until commit time
- Commit behaves almost like MCAS

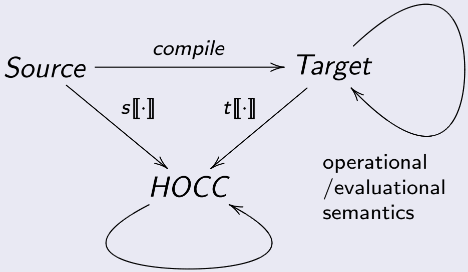
On Equality

Equality Strengths

- Value, or structural
  - Fast for primitive values, bad for lazy thunks
- Pointer
  - Efficient for unevaluated thunks and primitive values
  - Can't replace value by a copy of the same
- Version
  - Considers writes without regard to actual values involved
  - By pairing values with an incrementing version number
  - Or by a watch on the memory location, c.f. LL/SC
- State
  - All changes to shared state undesirable
- World
  - All interleaving undesirable

Existing Methodology

Compiler Correctness for Parallel Languages (Wand, 1995)



- Compiler correct if $s[\![p]\!]$ bisimilar to $t[\![compile\ p]\!]$
- Target operational semantics adequate relative to HOCC

Something Simpler?

Aim and Overview

- Avoid so many layers of translation; too much room for error
- Give source/target languages small-step/operational semantics
- Augment semantics with labelled transition system
- Direct bisimulation between the two semantics

Expressions and Evaluation

Expressions

$$\mathsf{E} ::= \mathbb{Z} \mid \mathsf{E} + \mathsf{E}$$

- Addition supplemented with a $(\textsc{Zap})$ rule
- Simple form of non-determinacy
- Left-biased evaluation

Labelled Transition System

$$\mathsf{Action} ::= \mathbb{Z} + \mathbb{Z} \mid \mathbb{Z} \lightning \mathbb{Z}$$

$$\mathsf{Label} ::= \mathsf{Action} \mid \tau$$

$$\overset{\cdot}{\longrightarrow} \subseteq \mathsf{E} \times \mathsf{Label} \times \mathsf{E}$$

Evaluation

Reduction Rules

$$\frac{}{\overline{n} + \overline{m} \xrightarrow{n+m} \overline{n + m}}\ (\textsc{Add}) \qquad \frac{}{\overline{n} + \overline{m} \xrightarrow{n \lightning m} 0}\ (\textsc{Zap})$$

$$\frac{e \xrightarrow{\alpha} e'}{e + f \xrightarrow{\alpha} e' + f}\ (\textsc{AddL}) \qquad \frac{f \xrightarrow{\alpha} f'}{\overline{n} + f \xrightarrow{\alpha} \overline{n} + f'}\ (\textsc{AddR})$$

Choice of Action

- Differentiate base case reductions in source language
- Two symbols are enough but...
- Conceivably, a broken compiler could keep structure intact
- Include operands to ensure the same values are computed

Compiler

Virtual Machine

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

$$\mathsf{M} = \mathsf{I}^{\star} \times \mathbb{Z}^{\star}$$

$$\overset{\cdot}{\longrightarrow} \subseteq \mathsf{M} \times \mathsf{Label} \times \mathsf{M}$$

Compiler

$$\begin{aligned}
&\mathit{compile} :: \mathsf{E} \rightarrow \mathsf{I}^{\star} \rightarrow \mathsf{I}^{\star} \\
&\mathit{compile}\quad \overline{n}\qquad\quad \imath s = \mathsf{PUSH}\ \ n : \imath s \\
&\mathit{compile}\quad (x + y)\ \imath s = \mathit{compile}\ x\ \imath s' \\
&\quad \textbf{where}\ \imath s' = \mathit{compile}\ y\ (\mathsf{ADD} : \imath s)
\end{aligned}$$

Execution

Virtual Machine Transitions

$$\langle \text{PUSH}\ \ n : \iota s,\ \sigma \rangle \xrightarrow{\tau} \langle \iota s,\ n : \sigma \rangle \qquad (\text{PUSH})$$

$$\langle \text{ADD} : \iota s,\ m : n : \sigma \rangle \xrightarrow{n+m} \langle \iota s,\ n + m : \sigma \rangle \qquad (\text{ADD})$$

$$\langle \text{ADD} : \iota s,\ m : n : \sigma \rangle \xrightarrow{n \lightning m} \langle \iota s,\ 0 : \sigma \rangle \qquad (\text{ZAP})$$

- Similar non-deterministic semantics, c.f. (Add) and (Zap)

Mixed Bisimulation

Motivation

- Can express correctness as $\langle \mathit{compile}\ x\ [\,],\ [\,]\rangle \approx x$
  - At every reduction step, anything LHS can do, RHS can follow
- Proof for something like this: structural induction on e?
- Need to generalise on stack, instruction continuation. . .
- Introduce expression *contexts*, $c[\![\cdot]\!]$?
- Can certainly relate stack and continuation to context
  - But proof turns very messy; this is a simple language!

Combined Machine – Existing Technology!

$$\mathsf{C} \equiv (\mathsf{E} + \mathbf{1}) \times \mathsf{M}$$

$$\overset{\cdot}{\longrightarrow} \subseteq \mathsf{C} \times \mathsf{Label} \times \mathsf{C}$$

Combined Semantics

Transition Rules

$$\frac{x \xrightarrow{\alpha} x'}{\langle x,\ \iota s,\ \sigma \rangle \xrightarrow{\alpha} \langle x',\ \iota s,\ \sigma \rangle} \qquad (\textsc{Eval})$$

$$\frac{}{\langle \overline{n},\ \iota s,\ \sigma \rangle \xrightarrow{\tau} \langle \bullet,\ \iota s,\ n : \sigma \rangle} \qquad (\textsc{Switch})$$

$$\frac{\langle \iota s,\ \sigma \rangle \xrightarrow{\alpha} \langle \iota s',\ \sigma' \rangle}{\langle \bullet,\ \iota s,\ \sigma \rangle \xrightarrow{\alpha} \langle \bullet,\ \iota s',\ \sigma' \rangle} \qquad (\textsc{Exec})$$

Weak Simulation

Definition

A non-empty relation $\mathfrak{R} \subseteq \mathsf{C} \times \mathsf{C}$ is a *weak simulation* iff for all $c \, \mathfrak{R} \, d$,

$$c \xrightarrow{\alpha} c' \quad \text{implies} \quad \exists d'.\, d \xRightarrow{\alpha} d' \wedge c' \, \mathfrak{R} \, d'$$

- There exists a maximal $\mathfrak{R}$: we name it $\succcurlyeq$
- $c \succcurlyeq d$ and $c \preccurlyeq d$ iff $c \approx d$

Lemma (Eliding τ)

If $c \xrightarrow{\tau} c'$ is the only possible transition by c, then:

$$\frac{c \xrightarrow{\tau} c'}{c \preccurlyeq c'} \quad \text{and} \quad \frac{c \xrightarrow{\tau} c'}{c \succcurlyeq c'}, \quad \text{or} \quad \frac{c \xrightarrow{\tau} c'}{c \approx c'}$$

Compiler Correctness

Theorem 1 (Soundness)

$$\langle x,\ \imath s,\ \sigma \rangle \succcurlyeq \langle \bullet,\ \mathit{compile}\ x\ \imath s,\ \sigma \rangle$$

Everything program does permitted by expression semantics

Proof Overview

- In this case, soundness and completeness proofs ares identical
    - Recover separate proofs by replacing $\approx$ with $\preccurlyeq$ or $\succcurlyeq$
- Completeness may not always be possible or even required
- Corollary (Correctness): $\langle x,\ [],\ [] \rangle \approx \langle \bullet,\ \mathit{compile}\ x\ [],\ [] \rangle$
- Selected highlights follow. . .
    - For full details, see my first year transfer dissertation

Compiler Correctness

Theorem 2 (Completeness)

$$\langle x,\ \imath s,\ \sigma\rangle \preccurlyeq \langle \bullet,\ \mathit{compile}\ x\ \imath s,\ \sigma\rangle$$

Program does everything permitted by expression semantics

Proof Overview

- In this case, soundness and completeness proofs ares identical
  - Recover separate proofs by replacing $\approx$ with $\preccurlyeq$ or $\succcurlyeq$
- Completeness may not always be possible or even required
- Corollary (Correctness): $\langle x,\ [],\ []\rangle \approx \langle \bullet,\ \mathit{compile}\ x\ [],\ []\rangle$
- Selected highlights follow. . .
  - For full details, see my first year transfer dissertation

Compiler Correctness

Theorem 3 (Bisimulation)

$$\langle x,\ \iota s,\ \sigma\rangle \approx \langle \bullet,\ \mathit{compile}\ x\ \iota s,\ \sigma\rangle$$

Program is a bisimulation of expression semantics

Proof Overview

- In this case, soundness and completeness proofs ares identical
  - Recover separate proofs by replacing $\approx$ with $\preccurlyeq$ or $\succcurlyeq$
- Completeness may not always be possible or even required
- Corollary (Correctness): $\langle x,\ [\,],\ [\,]\rangle \approx \langle \bullet,\ \mathit{compile}\ x\ [\,],\ [\,]\rangle$
- Selected highlights follow. . .
  - For full details, see my first year transfer dissertation

Theorem 3: $\langle x,\ \mathit{is},\ \sigma\rangle \approx \langle \bullet,\ \mathit{compile}\ x\ \mathit{is},\ \sigma\rangle$

Inductive Case: $x \equiv y + z$

Have induction hypothesis for y:

$$\forall \mathit{is}', \sigma'.\ \langle y,\ \mathit{is}',\ \sigma'\rangle \approx \langle \bullet,\ \mathit{compile}\ y\ \mathit{is}',\ \sigma'\rangle$$

and also for z. Then:

$$\begin{array}{ll} & \langle \bullet,\ \mathit{compile}\ (y+z)\ \mathit{is},\ \sigma\rangle \\ \equiv & \quad \{\ \text{definition of } \mathit{compile}\ \} \\ & \langle \bullet,\ \mathit{compile}\ y\ (\mathit{compile}\ z\ (\mathrm{ADD} : \mathit{is})),\ \sigma\rangle \\ \approx & \quad \{\ \text{induction hypothesis for } y\ \} \\ & \langle y,\ \mathit{compile}\ z\ (\mathrm{ADD} : \mathit{is}),\ \sigma\rangle \\ \approx & \quad \{\ \text{by lemma 4, given induction hypothesis for } z\ \} \\ & \langle y+z,\ \mathit{is},\ \sigma\rangle \end{array}$$

Additional Lemmas

Lemma 4 (Evaluate Left)

Given $\langle \bullet,\ \mathit{compile}\ z\ \mathit{is}',\ \sigma' \rangle \approx \langle z,\ \mathit{is}',\ \sigma' \rangle$,

$$\langle y,\ \mathit{compile}\ z\ (\mathsf{ADD} : \mathit{is}),\ \sigma \rangle \approx \langle y + z,\ \mathit{is},\ \sigma \rangle$$

Proof – case $y \not\equiv \overline{m}$

LHS:

$$\dfrac{y \xrightarrow{\alpha} y'}{\langle y,\ \mathit{compile}\ z\ (\mathsf{ADD}\ \mathit{is}),\ \sigma \rangle}\ (\textsc{Eval})$$

$$\xrightarrow{\alpha} \langle y',\ \mathit{compile}\ z\ (\mathsf{ADD} : \mathit{is}),\ \sigma \rangle$$

RHS:

$$\dfrac{\dfrac{y \xrightarrow{\alpha} y'}{y + z \xrightarrow{\alpha} y' + z}\ (\textsc{AddL})}{\langle y + z,\ \mathit{is},\ \sigma \rangle \xrightarrow{\alpha} \langle y' + z,\ \mathit{is},\ \sigma \rangle}\ (\textsc{Eval})$$

Additional Lemmas

Lemma 5 (Evaluate Right)

$$\langle z,\ \mathsf{ADD} : \imath s,\ \overline{m} : \sigma \rangle \approx \langle \overline{m} + z,\ \imath s,\ \sigma \rangle$$

- *The $z \not\equiv \overline{n}$ case proceeds as lemma 4*

Proof Method

- Uses simple *equational reasoning* and logic
- No need to consider *sets* of machine states / expressions
- Where there is non-determinism, we can chase diagrams
  - Weak bisimulation: traces $\alpha\tau$ and $\tau\alpha$ are equivalent

Chasing Diagrams

Proof of lemma 5 – case $z \equiv \overline{n}$

$$\begin{tikzcd}
 & & \langle \overline{m} + \overline{n},\ \iota s,\ \sigma \rangle \arrow[ddll, bend right, "m+n"', "(\textsc{Add})/(\textsc{Eval})"] \arrow[ddrr, bend left, "m \lightning n", "(\textsc{Zap})/(\textsc{Eval})"'] \arrow[d, phantom, "\approx\ \text{Lemma 5}"] & & \\
 & & \langle \overline{n},\ \mathsf{ADD} : \iota s,\ \overline{m} : \sigma \rangle \arrow[dd, "\tau"', "(\textsc{Switch})"] & & \\
\langle \overline{m+n},\ \iota s,\ \sigma \rangle \arrow[dd, "\tau"', "(\textsc{Switch})"] & & & & \langle 0,\ \iota s,\ \sigma \rangle \arrow[dd, "\tau"', "(\textsc{Switch})"] \\
 & & \langle \bullet,\ \mathsf{ADD} : \iota s,\ \overline{n} : \overline{m} : \sigma \rangle \arrow[dll, "m+n"', "(\textsc{Add})"] \arrow[drr, "m \lightning n", "(\textsc{Zap})"'] & & \\
\langle \bullet,\ \iota s,\ \overline{m+n} : \sigma \rangle & & & & \langle \bullet,\ \iota s,\ 0 : \sigma \rangle
\end{tikzcd}$$

Conclusion

Future Work

- Extension of language with parallelism
- Exceptions and interrupts
- Proof of STM model
- Richer transactional memory constructs?
    - Forking within transactions
    - Compensating transactions
    - Data invariants