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
### Race Conditions

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

<table>
<thead>
<tr>
<th>Thread</th>
<th>account Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td><code>balanceA ← read account</code></td>
</tr>
<tr>
<td>B</td>
<td><code>balanceB ← read account</code></td>
</tr>
<tr>
<td>B</td>
<td><code>write account (balanceB + amountB)</code></td>
</tr>
<tr>
<td>A</td>
<td><code>write account (balanceA + amountA)</code></td>
</tr>
</tbody>
</table>
**Example**

**Lack of Compositionality**

\[
\text{deposit} :: \text{Account} \to \text{Integer} \to \text{IO} ()
\]
\[
\text{deposit account amount} = \text{do}
\]
\[
\quad \text{lock account}
\]
\[
\quad \text{balance} \leftarrow \text{read account}
\]
\[
\quad \text{write account} (\text{balance} + \text{amount})
\]
\[
\quad \text{release account}
\]

\[
\text{transfer} :: \text{Account} \to \text{Account} \to \text{Integer} \to \text{IO} ()
\]
\[
\text{transfer from to amount} = \text{do}
\]
\[
\quad \text{withdraw from amount}
\]
\[
\quad \text{deposit to amount}
\]
Example

Lack of Compositionality (Solution?)

\[ \text{deposit} :: \text{Account} \rightarrow \text{Integer} \rightarrow \text{IO} () \]

\[ \text{deposit} \text{ account amount} = \text{do} \]

\[ \text{balance} \leftarrow \text{read account} \]

\[ \text{write account} (\text{balance} + \text{amount}) \]

\[ \text{transfer} :: \text{Account} \rightarrow \text{Account} \rightarrow \text{Integer} \rightarrow \text{IO} () \]

\[ \text{transfer from to amount} = \text{do} \]

\[ \text{lock from; lock to} \]

\[ \text{withdraw from amount} \]

\[ \text{deposit to amount} \]

\[ \text{release from; release to} \]
Example

### Deadlock

- **Thread A**: `transfer x \( y \) 100`  
- **Thread B**: `transfer y x \( 200 \)`

<table>
<thead>
<tr>
<th>Thread</th>
<th>Account ( x )</th>
<th>Account ( y )</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td><code>lock x</code></td>
<td>free</td>
<td>free</td>
</tr>
<tr>
<td>B</td>
<td><code>lock y</code></td>
<td>held by A</td>
<td>free</td>
</tr>
<tr>
<td>B</td>
<td><code>lock x</code></td>
<td>held by A</td>
<td>held by B</td>
</tr>
<tr>
<td>A</td>
<td><code>lock y</code></td>
<td>held by A</td>
<td>held by B</td>
</tr>
</tbody>
</table>
Mutual Exclusion

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

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

\[
deposit :: \text{Account} \rightarrow \text{Integer} \rightarrow \text{IO}() \\
deposit\ \text{account}\ \text{amount} = \text{do} \\
\begin{align*}
&\quad \text{begin} \\
&\quad \quad balance \leftarrow \text{read account} \\
&\quad \quad \quad \text{-- another transaction commits, modifying account} \\
&\quad \quad write\ \text{account}\ (balance + amount) \\
&\quad commit\quad \text{-- fails}
\end{align*}
\]

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) ≡ 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

```haskell
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**

\[
\text{launchMissiles :: IO ()}
\]

\[
\text{atomic $\_do$

\text{launchMissiles} \quad -- \text{compile-time error: type mismatch}

\text{\ldots retry \ldots}$

**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?

\[
\begin{align*}
popBlocking & \quad :: \; \text{TVar} \; [\text{Integer}] \rightarrow \text{STM} \; \text{Integer} \\
popBlocking \; ts & = \text{do} \\
& \quad s \leftarrow \text{readTVar} \; ts \\
\text{case} \; s \; \text{of} \; [] & \quad \rightarrow \; \text{retry} \\
& \quad (x : xs) \rightarrow \text{do} \; \text{writeTVar} \; xs; \; \text{return} \; x \\
popNonblocking & \quad :: \; \text{TVar} \; [\text{Integer}] \rightarrow \text{STM} \; (\text{Maybe} \; \text{Integer}) \\
popNonblocking \; ts & = \text{liftM} \; \text{Just} \; (\text{popBlocking} \; ts) \\
& \quad \text{orElse} \; \text{return} \; \text{Nothing}
\end{align*}
\]

- Similarly turn non-blocking into blocking
Formal Semantics

Transition Rule for $atomic$

$$m \rightarrow^* \bar{n}$$

$atomic \ m \rightarrow \bar{n}$

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

E ::= \mathbb{Z} | E + E | \text{rd} \ Name | \text{wr} \ Name \ E | \text{atomic} \ E

Comparison with STM Haskell

<table>
<thead>
<tr>
<th>STM Haskell</th>
<th>Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\gg) \quad :: \text{STM} \ \alpha \rightarrow (\alpha \rightarrow \text{STM} \ \beta) \rightarrow \text{STM} \ \beta</td>
<td>e + f</td>
</tr>
<tr>
<td>\text{return} \quad :: \alpha \rightarrow \text{STM} \ \alpha</td>
<td>m \in \mathbb{Z}</td>
</tr>
<tr>
<td>\text{retry} \quad :: \text{STM} \ \alpha</td>
<td></td>
</tr>
<tr>
<td>\text{orElse} \quad :: \text{STM} \ \alpha \rightarrow \text{STM} \ \alpha \rightarrow \text{STM} \ \alpha</td>
<td></td>
</tr>
<tr>
<td>\text{readTVar} \quad :: \text{TVar} \ \alpha \rightarrow \text{STM} \ \alpha</td>
<td></td>
</tr>
<tr>
<td>\text{writeTVar} \quad :: \text{TVar} \ \alpha \rightarrow \alpha \rightarrow \text{STM} \ ()</td>
<td></td>
</tr>
<tr>
<td>\text{newTVar} \quad :: \text{STM} \ (\text{TVar} \ \alpha)</td>
<td></td>
</tr>
<tr>
<td>\text{atomic} \quad :: \text{STM} \ \alpha \rightarrow \text{IO} \ \alpha</td>
<td></td>
</tr>
<tr>
<td></td>
<td>atomic \ e</td>
</tr>
</tbody>
</table>
Small-Step Semantics

\[
\langle e, \sigma \rangle \longrightarrow \langle e', \sigma' \rangle
\]
\[
\langle e + f, \sigma \rangle \longrightarrow \langle e' + f, \sigma' \rangle \quad \text{(AddL)}
\]
\[
\langle n + m, \sigma \rangle \longrightarrow \langle n + m, \sigma \rangle \quad \text{(AddZ)}
\]
\[
\langle w r \ n \ e, \sigma \rangle \longrightarrow \langle w r \ n \ e', \sigma' \rangle \quad \text{(WRITEE)}
\]
\[
\langle w r \ v \ n, \sigma \rangle \longrightarrow \langle \sigma(v), \sigma[v \leftarrow n] \rangle \quad \text{(WRITEZ)}
\]
\[
\langle f, \sigma \rangle \longrightarrow \langle f', \sigma' \rangle
\]
\[
\langle n + f, \sigma \rangle \longrightarrow \langle n + f', \sigma' \rangle \quad \text{(AddR)}
\]
\[
\langle r d \ v, \sigma \rangle \longrightarrow \langle \sigma(v), \sigma \rangle \quad \text{(READ)}
\]
\[
\langle e, \sigma \rangle \longrightarrow ^* \langle n, \sigma' \rangle \quad \text{(ATOMIC)}
\]
Concurrent Evaluation

Expression Soup

\[ P ::= E \mid P \parallel P \]

\[ \langle p, \sigma \rangle \longrightarrow \langle p', \sigma' \rangle \]

\[ \langle p \parallel q, \sigma \rangle \longrightarrow \langle p' \parallel q, \sigma' \rangle \]

\[ (\text{SEQ}) \]

\[ (\text{PARL}) \]

\[ \langle e, \sigma \rangle \longrightarrow \langle e', \sigma' \rangle \]  

\[ \langle q, \sigma \rangle \longrightarrow \langle q', \sigma' \rangle \]

\[ \langle p \parallel q, \sigma \rangle \longrightarrow \langle p \parallel q', \sigma' \rangle \]

\[ (\text{PARL}) \]

Example

- \( \text{rd "x" + rd "x" \parallel wr "x" 1} \) — yields 0, 1 or 2
- \( \text{atomic (rd "x" + rd "x") \parallel wr "x" 1} \) — 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

\[ \langle ip, sp, tp \rangle \in \text{Thread} \equiv \text{Instruction}^* \times \mathbb{Z}^* \times \text{TransactionFrame}^* \]

Thread Soup

Program ::= Thread
| Program || Program

- Rules (SEQ), (PARL) and (PARR) will suffice
- Threads execute paired with a shared store
### E to Instruction*

\( \text{compE} \in E \rightarrow \text{Instruction}^* \rightarrow \text{Instruction}^* \)

- \( \text{compE} \ \bar{n} \quad c = \text{PUSH} \ n : c \)
- \( \text{compE} \ (e + f) \quad c = \text{compE} \ e \ (\text{compE} \ f \ (\text{ADD} : c)) \)
- \( \text{compE} \ (\text{rd} \ v) \quad c = \text{LOAD} \ v : c \)
- \( \text{compE} \ (\text{wr} \ v \ e) \quad c = \text{compE} \ e \ (\text{SWAP} \ v : c) \)
- \( \text{compE} \ (\text{atomic} \ e) \quad c = \text{BEGIN} : \text{compE} \ e \ (\text{COMMIT} : c) \)

### P to Program

\( \text{compP} \in P \rightarrow \text{Program} \)

- \( \text{compP} \ e \quad = \langle | \text{compE} e [], [], [], | \rangle \)
- \( \text{compP} \ (p \parallel q) = \text{compP} \ p \parallel \text{compP} \ q \)
Correctness

**Sequential**

\[ \forall e \in E, \sigma \in \text{Name} \rightarrow \mathbb{Z}, \; n \in \mathbb{Z}. \]

\[ \langle e, \sigma \rangle \xrightarrow{*} \langle n, \sigma' \rangle \]

iff

\[ \langle \langle \text{compE} e [[],[],[]], \sigma \rangle \xrightarrow{*} \langle [[], [n], []], \sigma' \rangle \]

**Concurrent**

\[ \forall p \in P, \sigma \in \text{Name} \rightarrow \mathbb{Z}, \; ns \in P. \]

\[ \langle p, \sigma \rangle \xrightarrow{*} \langle ns, \sigma' \rangle \]

iff

\[ \langle \text{compP} p, \sigma \rangle \xrightarrow{*} \langle rs, \sigma' \rangle \]

- \( ns \in P \) contains only integer expressions of the form \( \bar{n} \)
- \( rs \in \text{Program} \) structurally identical to \( ns \) but with \( \bar{n} \mapsto \langle [], [n], [] \rangle \)
Model Verification

Implementation

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

  \[
  \text{propCC} :: \mathbb{P} \rightarrow \text{Bool} \\
  \text{propCC} \ p = (\text{result} \circ \text{run}) (p, \sigma_0) \equiv (\text{result} \circ \text{run}) (\text{compP} \ p, \sigma_0)
  \]

QuickCheck

- Generates random input, attempts to falsify proposition:

  > \text{quickCheck} \ \text{propCC} \\
  \text{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

<table>
<thead>
<tr>
<th>Thread</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>TVars</th>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>rd x ⇝ 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>wr x 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>rd y ⇝ 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>commit?</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
**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 $t[\text{compile } p]$
- 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

\[ E ::= \mathbb{Z} \mid E + E \]

- Addition supplemented with a \((\text{ZAP})\) rule
- Simple form of non-determinacy
- Left-biased evaluation

Labelled Transition System

Action ::= \mathbb{Z} + \mathbb{Z} \mid \mathbb{Z} \downarrow \mathbb{Z}
Label ::= Action \mid \tau
\[ \rightarrow \subseteq E \times Label \times E \]
Evaluation

Reduction Rules

- **(ADD)**
  \[ \overline{n} + m \overset{n+m}{\rightarrow} \overline{n} + m \]

- **(ZAP)**
  \[ \overline{n} + m \overset{n\downarrow m}{\rightarrow} 0 \]

- **(ADDL)**
  \[ e \overset{\alpha}{\rightarrow} e' \]
  \[ e + f \overset{\alpha}{\rightarrow} e' + f \]

- **(ADDR)**
  \[ f \overset{\alpha}{\rightarrow} f' \]
  \[ \overline{n} + f \overset{\alpha}{\rightarrow} \overline{n} + f' \]

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

\[
I ::= \text{PUSH } \mathbb{Z} \mid \text{ADD} \\
M = I^* \times \mathbb{Z}^* \\
\rightarrow \subseteq M \times \text{Label} \times M
\]

### Compiler

\[
\text{compile} :: E \rightarrow I^* \rightarrow I^* \\
\text{compile} \ n \is = \text{PUSH } n : \is \\
\text{compile} \ (x + y) \is = \text{compile} x \is' \\
\text{where } \is' = \text{compile} y \ (\text{ADD} : \is)
\]
Virtual Machine Transitions

\[
\begin{align*}
\langle \text{PUSH} \ n : \iota \sigma \rangle & \xrightarrow{\tau} \langle \iota \sigma, \ n : \sigma \rangle & \text{(PUSH)} \\
\langle \text{ADD} : \iota \sigma, \ m : \ n : \sigma \rangle & \xrightarrow{n+m} \langle \iota \sigma, \ n + m : \sigma \rangle & \text{(ADD)} \\
\langle \text{ADD} : \iota \sigma, \ m : \ n : \sigma \rangle & \xrightarrow{n^+m} \langle \iota \sigma, \ 0 : \sigma \rangle & \text{(ZAP)}
\end{align*}
\]

• Similar non-deterministic semantics, c.f. (\text{ADD}) and (\text{ZAP})
Mixed Bisimulation

Motivation

- Can express correctness as \langle compile \times [], [], \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, $c\llbracket \cdot \rrbracket$?
- Can certainly relate stack and continuation to context
  - But proof turns very messy; this is a simple language!

Combined Machine – Existing Technology!

\[ C \equiv (E + 1) \times M \]
\[ \rightarrow \subseteq C \times \text{Label} \times C \]
Combined Semantics

Transition Rules

\[ x \xrightarrow{\alpha} x' \]

\[ \langle x, is, \sigma \rangle \xrightarrow{\alpha} \langle x', is, \sigma \rangle \quad \text{(EVAL)} \]

\[ \langle n, is, \sigma \rangle \xrightarrow{\tau} \langle \bullet, is, n : \sigma \rangle \quad \text{(SWITCH)} \]

\[ \langle is, \sigma \rangle \xrightarrow{\alpha} \langle is', \sigma' \rangle \quad \text{(EXEC)} \]

\[ \langle \bullet, is, \sigma \rangle \xrightarrow{\alpha} \langle \bullet, is', \sigma' \rangle \]
**Definition**

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

$$c \xrightarrow{\alpha} c' \implies \exists d'. d \xrightarrow{\alpha} d' \land c' \mathcal{R} d'$$

- There exists a maximal $\mathcal{R}$: we name it $\succeq$
- $c \succeq d$ and $c \preceq d$ iff $c \simeq d$

**Lemma (Eliding $\tau$)**

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

- $c \xrightarrow{\tau} c' \implies c \preceq c'$
- $c \xrightarrow{\tau} c' \implies c \succeq c'$, or $c \xrightarrow{\tau} c' \implies c \simeq c'$
Compiler Correctness

**Theorem 1 (Soundness)**

\[ \langle x, is, \sigma \rangle \trianglerighteq \langle \bullet, \text{compile } x \ is, \sigma \rangle \]

Everything program does permitted by expression semantics

**Proof Overview**

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

Theorem 2 (Completeness)

\[ \langle x, \text{is}, \sigma \rangle \preceq \langle \bullet, \text{compile } x \text{ is}, \sigma \rangle \]

Program does everything permitted by expression semantics

Proof Overview

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

Theorem 3 (Bisimulation)

\[ \langle x, is, \sigma \rangle \approx \langle \bullet, \text{compile } x is, \sigma \rangle \]

*Program is a bisimulation of expression semantics*

Proof Overview

- In this case, soundness and completeness proofs are identical
  - Recover separate proofs by replacing \( \approx \) with \( \preceq \) or \( \succeq \)
- Completeness may not always be possible or even required
- Corollary (Correctness): \( \langle x, [], [] \rangle \approx \langle \bullet, \text{compile } x [], [] \rangle \)
- Selected highlights follow...
  - For full details, see my first year transfer dissertation
Theorem 3: \( \langle x, s, \sigma \rangle \approx \langle \bullet, \text{compile } x \ s, \sigma \rangle \)

**Inductive Case:** \( x \equiv y + z \)

Have induction hypothesis for \( y \):

\[
\forall s', \sigma'. \langle y, s', \sigma' \rangle \approx \langle \bullet, \text{compile } y \ s', \sigma' \rangle
\]

and also for \( z \). Then:

\[
\langle \bullet, \text{compile } (y + z) \ s, \sigma \rangle
\]

\[
\equiv \quad \{ \text{definition of compile } \}
\]

\[
\langle \bullet, \text{compile } y (\text{compile } z (\text{ADD} : s)), \sigma \rangle
\]

\[
\approx \quad \{ \text{induction hypothesis for } y \}
\]

\[
\langle y, \text{compile } z (\text{ADD} : s), \sigma \rangle
\]

\[
\approx \quad \{ \text{by lemma 4, given induction hypothesis for } z \}
\]

\[
\langle y + z, s, \sigma \rangle
\]
Lemma 4 (Evaluate Left)

Given \( \langle \bullet, \text{compile } z\ is', \ \sigma' \rangle \approx \langle z, is', \ \sigma' \rangle \),

\[ \langle y, \text{compile } z \ (\text{ADD : } is), \ \sigma \rangle \approx \langle y + z, is, \ \sigma \rangle \]

Proof – case \( y \neq m \)

LHS:

\[ y \xrightarrow{\alpha} y' \]

\[ \langle y, \text{compile } z \ (\text{ADD : } is), \ \sigma \rangle \xrightarrow{\alpha} \langle y', \text{compile } z \ (\text{ADD : } is), \ \sigma \rangle \] (EVAL)

RHS:

\[ y + z \xrightarrow{\alpha} y' + z \] (ADDL)

\[ \langle y + z, is, \ \sigma \rangle \xrightarrow{\alpha} \langle y' + z, is, \ \sigma \rangle \] (EVAL)
Lemma 5 (Evaluate Right)

\[ \langle z, \text{ADD} : is, \overline{m} : \sigma \rangle \approx \langle \overline{m} + z, is, \sigma \rangle \]

- The \( z \neq \overline{n} \) 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
Proof of lemma 5 – case $z \equiv \bar{n}$
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