High-Level Small-Step Operational Semantics for Software Transactions

Katherine F. Moore
Dan Grossman

The University of Washington

Motivating Our Approach

- Operational Semantics
  - Model key *programming-language* features
    - Functions, memory allocation, …, *transactions*
- High-Level
  - No TM implementation details
  - Programmer's view
  - Most appropriate for language specification
- Small-Step
  - Transactions take many steps to complete
  - Lets us investigate interleavings, parallelism

Outline

- A basic transaction language
  - Used to demonstrate our approach
- Transactions with Internal Parallelism
  - Type system limits spawn actions
- Transactions with Weak Isolation
  - Proof techniques for weak/strong equivalence given static restrictions
  - Ongoing Work: Weak languages with rollback

Technical Details…

- Program State: $a; H; e_1 \,||\, \ldots \,||\, e_n$
  - Every e is a thread
  - H is a heap that maps addresses to mutable contents.
  - a indicates if there is an active transaction
    - ❍ for no, and ● for yes

- To move from one program state to another, evaluate a single thread.

$$a; H; e \rightarrow a'; H'; e'; e_{opt}$$

  - expressions spawn *at most one* other thread in a step

A Basic Transaction Language

- Many rules are unaffected by transactions:
  - Function application

$$\frac{}{a;H;(\lambda x.e)\ v \rightarrow a;H;e\{v/x\};\cdot}$$

- Some rules are specific to transactions:
  - Entering or exiting transaction

$$\frac{}{\circ;H;\text{atomic}\ e \rightarrow \bullet;H;\text{inatomic}(e);\cdot}$$

$$\frac{}{\bullet;H;\text{inatomic}(v);\cdot \rightarrow \circ;H;v;\cdot}$$

A Basic Transaction Language

- These rules prevent heap access in parallel with a transaction:
  - Read:

$$\frac{}{\circ\,;H;read(l) \rightarrow \circ;H;H(l);\cdot}$$

  - Write:

$$\frac{}{\circ\,;H;l := v \rightarrow \circ;H,l \mapsto v;l;\cdot}$$

- But wait! Transactions need a way to read and write the heap also….

A Basic Transaction Language

- Executing inside a transaction allows e to pick any a-bit it wants. Now e can
  - read or write H
  - enter and exit nested transactions.

$$\frac{a; H; e \rightarrow a'; H'; e';\cdot}{\bullet; H; \text{inatomic}(e) \rightarrow \bullet; H'; \text{inatomic}(e');\cdot}$$

- Nothing *else* can read or write H until the transaction completes

A Basic Transaction Language

- Spawning threads inside a transaction causes a dynamic failure:

$$\frac{a;H;e \rightarrow a';H';e';\cdot}{\bullet;H;\mathrm{inatomic}(e) \rightarrow \bullet;H';\mathrm{inatomic}(e');\cdot}$$

$$\frac{}{a;H;spawn(e) \rightarrow a;H;0;e}$$

- Easy to prevent *e* from attempting this using a type system.

Recap…

- Closed nested transactions with strong isolation
- Small-step with no TM details

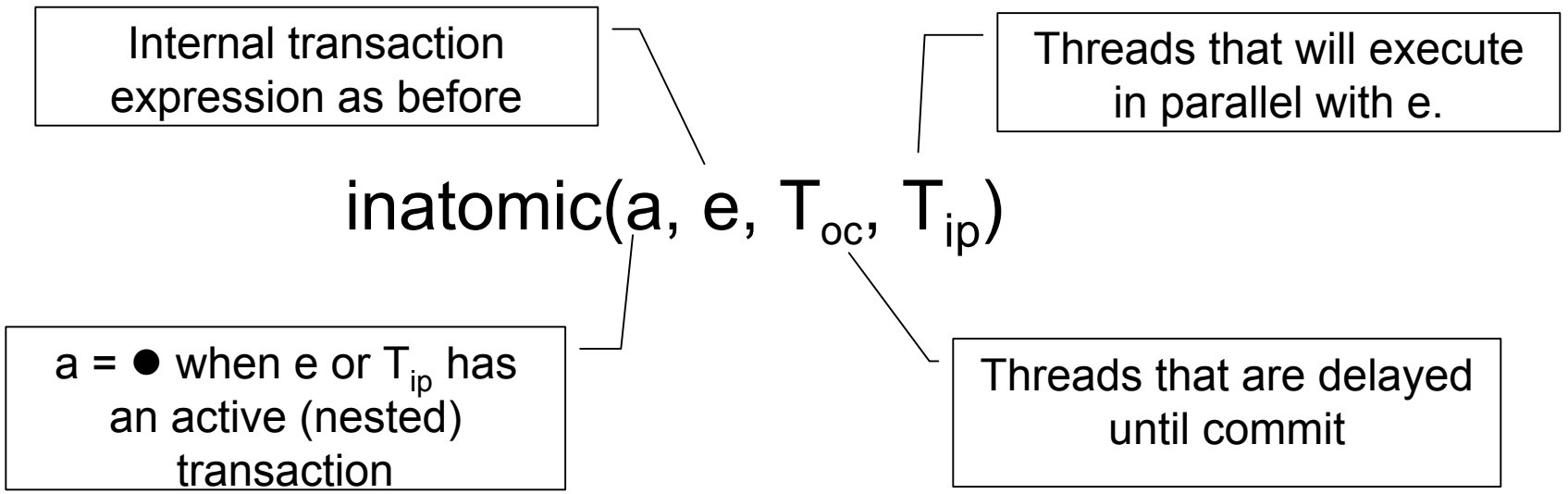
- Some questions we can now ask…
  - Could we allow spawn inside a transaction? What would it mean?
  - Can we formalize weak isolation? How can we show when weak/strong are equivalent?

Internal Parallelism

- We consider three “viable” kinds of spawn:
  - Spawn_{tl} : Top level spawn
    - Never okay in a transaction
    - Just like basic language
  - Spawn_{oc} : On commit spawn
    - Can occur anywhere
    - Delay computation until the containing transaction completes.
  - Spawn_{ip} : Internally parallel spawn
    - Must execute in a transaction
    - Transaction waits for spawned thread to complete before exit
- As before, a type system can prevent dynamic errors due to spawn expressions executing “at the wrong time”.

Internal Parallelism

- New state for transactions:



Outline

- A basic transaction language
  - Used to demonstrate our approach
- Transactions with Internal Parallelism
  - Type system limits spawn actions
- Transactions with Weak Isolation
  - Proof techniques for weak/strong equivalence given static restrictions
  - Ongoing Work: Weak languages with rollback

Weak-Isolation

- Weak isolation relaxes the restrictions on a:
  - Read:

$$\frac{}{a;H;read(l) \rightarrow a;H;H(l);\cdot}$$

  - Write:

$$\frac{}{a;H;l := v \rightarrow a;H,l \mapsto v;l;\cdot}$$

- Intuitively:
  - This language has strictly more behaviors

    ….but only for programs that contain data races between transactions and non-transactions

Formalizing intuition: Equivalence

- Goal:
  - Define a *conservative* subset of programs such that:

$$\circ;\cdot;e \rightarrow^{*}_{strong} a;H;e_1 \,||\, \ldots \,||\, e_n$$

$$\text{iff} \quad \circ;\cdot;e \rightarrow^{*}_{weak} a;H;e_1 \,||\, \ldots \,||\, e_n$$

- How?
  - Disallow the transactional / nontransactional races that cause problems.

Heap Partition

- Strict Partition:
  - Each address is accessed *always* in a transaction, or *never* in a transaction
  - Conservative starting point for the structure of the equivalence proof
  - Defined formally via a type system
    - Shows that evaluation *preserves* the partition

Weak/Strong Equivalence Theorem

If e type-checks, then...

$$\circ;\cdot;e \rightarrow^{*}_{strong} a;H;e_1 \,||\, \ldots \,||\, e_n$$

iff $\circ;\cdot;e \rightarrow^{*}_{weak} a;H;e_1 \,||\, \ldots \,||\, e_n$

- Statement of Theorem is high-level
- Proof requires commuting operations between threads to show transactions are serializable

Weak*er* Transaction Languages

- Original definition of weak isolation is somewhat naïve, for example…
  - What would happen if a transaction aborts?
- Weak with rollback
  - Recently proven equivalent to strong given a heap partition
  - Must show that rollback is correct
- Weak with lazy update
  - Ongoing work

Conclusions

- Defined transactions without exposing programmers to TM details
  - Formalized transactions from programmers' perspective
  - Our *model* has a single transaction at a time
- Defined transactions that allow internal spawn
  - Multiple reasonable semantics
- Defined weak-isolation and strong-isolation
  - Proven they are equivalent under certain conditions
  - Future …. leverage this proof technique for less-conservative restrictions (privatization, read-only, thread-local, …)