Transactional Events for ML

Laura Effinger-Dean, Matthew Kehrt and Dan Grossman

Computer Science & Engineering
University of Washington

ICFP 2008, Victoria, BC
September 22, 2008

Motivation

- Concurrent ML is a well-known, natural programming model
  - Concise, elegant encodings
  - Not powerful enough for some useful protocols!
- Transactional events are a powerful extension to CML.
  - Guarded receive, barriers, and more
  - Originally implemented in Haskell
- We present a design for TE in ML
- Major challenge: mutation within transactional events

Contributions

1. Reasonable semantics for mutation within transactional events
2. Formal operational semantics and proof of correctness
3. Implementation in the OCaml compiler/runtime

Outline

Concurrent ML (Reppy '92)

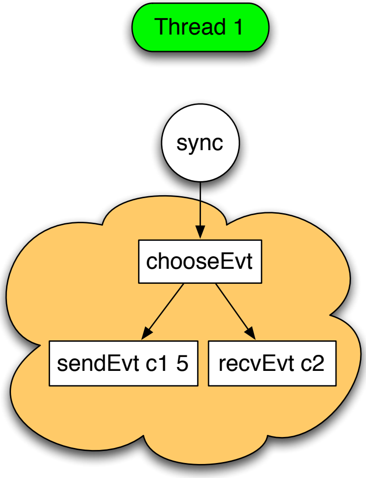
- First-class events describe communications:
  - Use sendEvt and recvEvt to communicate over typed channels
  - chooseEvt combinator describes an event that executes exactly one of two sub-events
- sync actually performs (synchronizes on) an event

Example

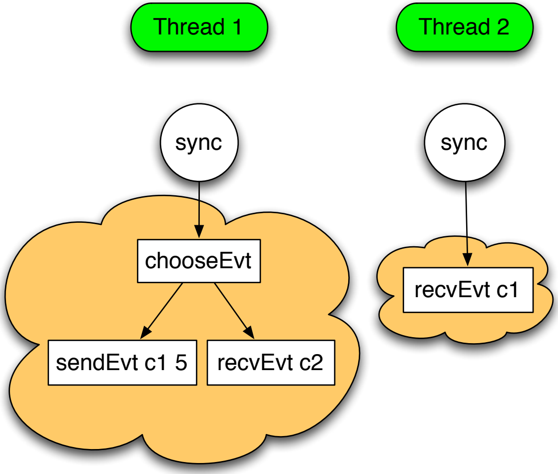
Send on c1 *or* receive on c2:

```
let foo = chooseEvt
            (sendEvt c1 5)
            (recvEvt c2)
let _ = sync foo (* perform the event *)
```

CML: Example

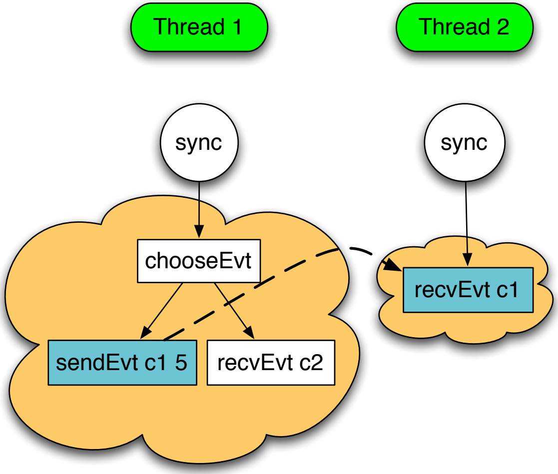


CML: Example



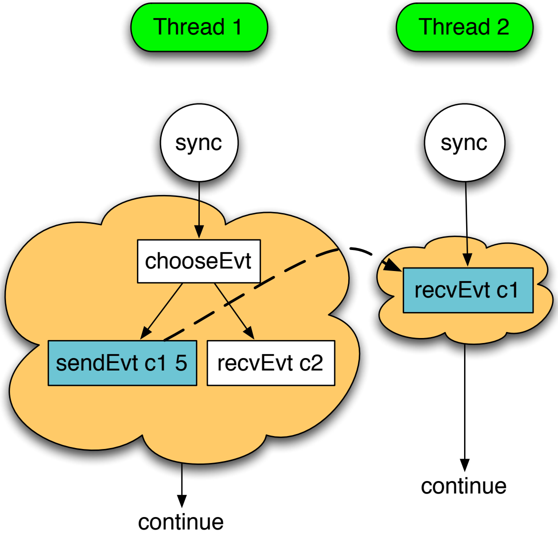
Thread 1
Thread 2
sync
sync
chooseEvt
sendEvt c1 5
recvEvt c2
recvEvt c1

CML: Example



Thread 1
Thread 2
sync
sync
chooseEvt
sendEvt c1 5
recvEvt c2
recvEvt c1

CML: Example



Thread 1
Thread 2
sync
sync
chooseEvt
sendEvt c1 5
recvEvt c2
recvEvt c1
continue
continue

Transactional events

Transactional events (Donnelly and Fluet, ICFP '06) extend CML with a sequencing combinator `thenEvt`.

`thenEvt` type

```
val thenEvt : 'a event ->
  ('a  -> 'b event) -> 'b event
```

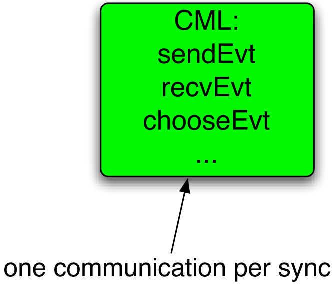
`thenEvt` succeeds when both sub-events succeed:

Example

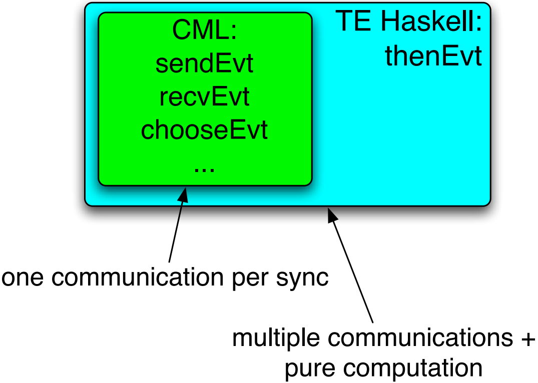
```
let _ = sync (thenEvt (recvEvt c1)
  (fun x -> sendEvt c2 x))
```

Multiple communications per sync.

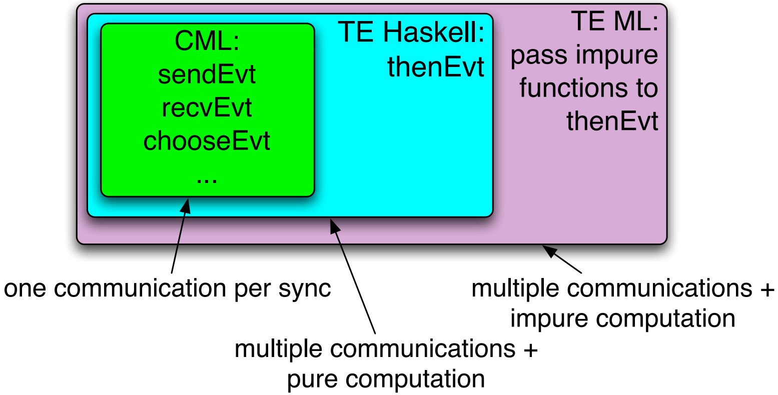
CML, TE Haskell, and TE ML



CML, TE Haskell, and TE ML

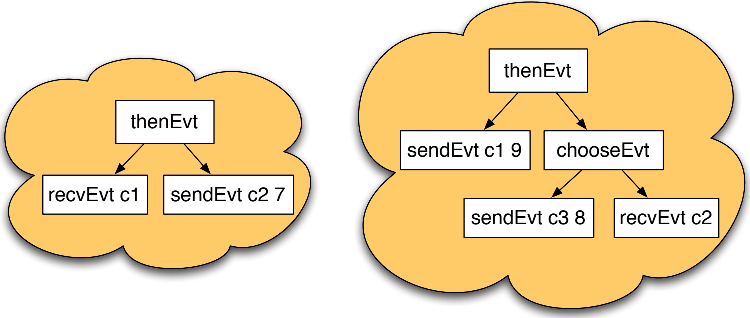


CML, TE Haskell, and TE ML



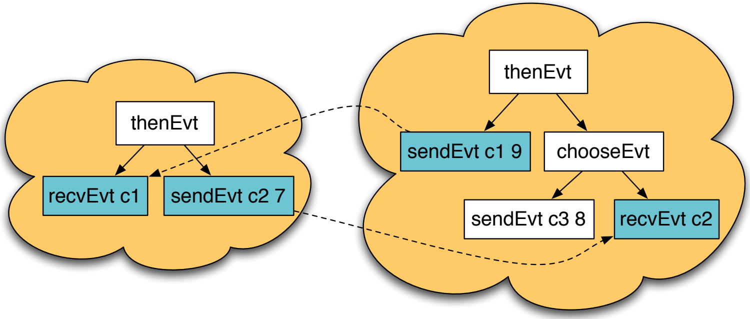
TE: Example

An example using both `thenEvt` and `chooseEvt`.



TE: Example

An example using both `thenEvt` and `chooseEvt`.



TE applications

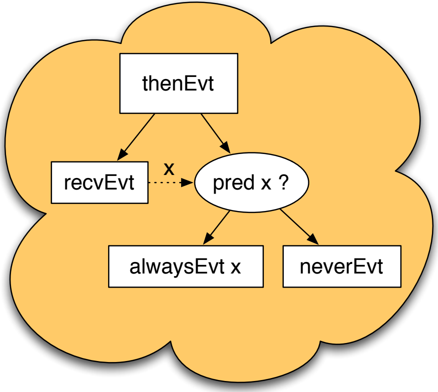
Cleanly express sophisticated communication protocols:

- Group two or more communications as a transaction
- Guarded receive (difficult in CML)
- *n*-way rendezvous (impossible in CML)

Guarded Receive

Example

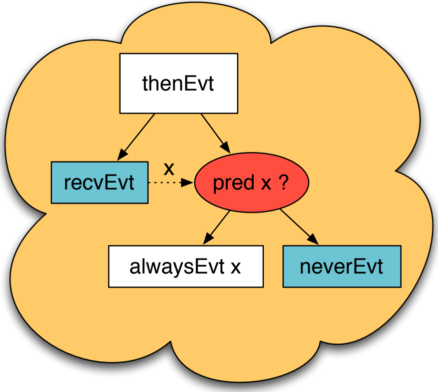
```
let guardedRecv pred c = thenEvt (recvEvt c)
  (fun x -> if pred x then alwaysEvt x else neverEvt)
```



Guarded Receive

Problem

What happens if `pred` modifies the heap, and then returns false?



Mutation in transactional events

- If we naïvely update the heap:
  - Visible effects of unsuccessful events
  - Inconsistent order for heap accesses
- In Haskell, none of these problems arise — any function passed to `thenEvt` is *pure*!
- Can we use TE in an impure language?

TE for ML

The problem

How should we define the semantics of mutation within transactional events?

Outline

Three proposals

We'll consider three alternatives for mutation within `thenEvt`.

1. Disallow mutation within transactions.
2. Model mutable locations using CML-style refserver threads.
3. Group the heap accesses of each thread into atomic "chunks."

Spoiler alert: option 3 is our solution.

Proposal #1

Disallowing mutation

If, at runtime, a transaction attempts to read or write mutable memory, halt the program with an error.

Proposal #1

Disallowing mutation

If, at runtime, a transaction attempts to read or write mutable memory, halt the program with an error.

- Pro: Easy to implement
- Con: Mutation is *unavoidable* in ML
    - Functions with pure interfaces may have hidden side effects
    - e.g., here the call to `fib` fails only if `fib` is memoized:

Example

```
let evenFibonacciGuard = guardedRecvEvt
  (fun x -> fib x % 2 = 0)
```

Proposal #2

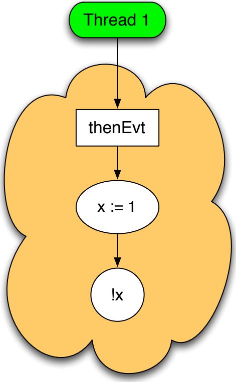
CML-style refservers

Create a new “refserver” thread for each heap location. If a thread tries to read heap location x, instead receive the current value from the refserver for x. If a thread writes to x, translate it to a send.

Proposal #2

CML-style refservers

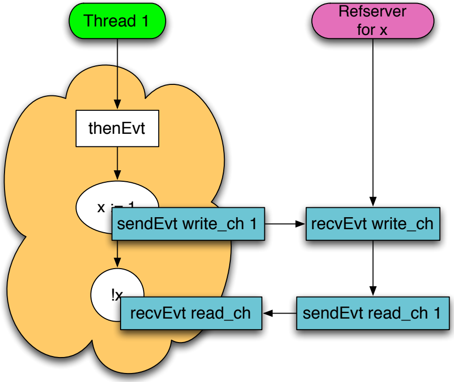
Create a new "refserver" thread for each heap location. If a thread tries to read heap location x, instead receive the current value from the refserver for x. If a thread writes to x, translate it to a send.



Proposal #2

CML-style refservers

Create a new "refserver" thread for each heap location. If a thread tries to read heap location x, instead receive the current value from the refserver for x. If a thread writes to x, translate it to a send.



Proposal #2

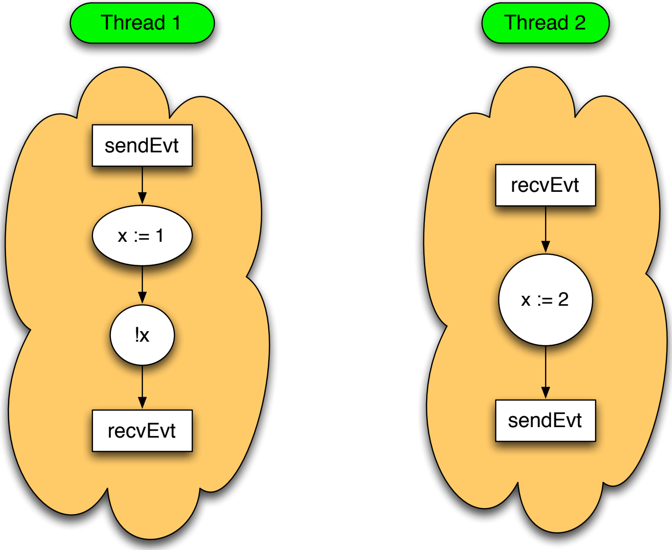
- Pro: Straightforward translation, uses existing infrastructure
- Con: Guarantees too much
  - *Required* to find a successful intereaving if one exists
  - Programs can abuse this guarantee, e.g.: (r starts at 0)

Example

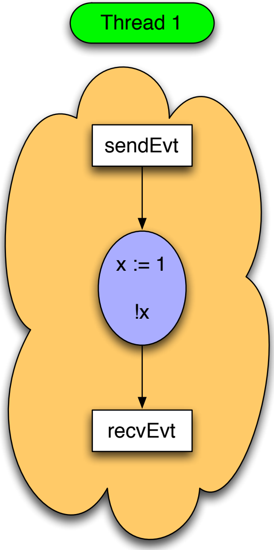
```
Thread 1: thenEvt (sendEvt c 0)
                  (fun _ -> r := 1; r := 0;
                            alwaysEvt ())
Thread 2: thenEvt (recvEvt c)
                  (fun _ -> if !r = 1
                            then alwaysEvt ()
                            else neverEvt)
```

- Con: Too slow — searches all possible interleavings!

Proposal #3: Chunking



Proposal #3: Chunking



Proposal #3

Chunking

A "chunk" is mini-transaction with all of one thread's heap accesses between consecutive communications. In the *chunking* semantics, every heap access executes as part of a chunk.

Chunking is a good compromise:

- Allows mutation
- Weaker guarantees than refservers:
  - Searches fewer possible interleavings
  - Does not break any useful programs we know of
- Much faster than refservers

Outline

Formal semantics

Formal model of chunking semantics:

- High-level, nondeterministic operational semantics
  - Clear definition of which transactions can succeed
- A low-level, (mostly-)deterministic semantics
  - Models the OCaml implementation
- Proof of equivalence between high- and low-level
  - Formally verified in Coq

Implementation

Prototype implementation by modifying OCaml runtime.

- Low-level support for speculatively executing events
- Inside transactional events, reads/writes of mutable data use functional first-class heaps
- Interesting details on nested sync, thread-scheduling, ...

Extensions

- See the paper for nested synchronizations, e.g.:

Example

```
let foo = sync (thenEvt (sendEvt c1 5) (fun _ ->
                  let x = sync (recvEvt c2);
                  sendEvt c3 x))
```

- Future work:
  - Other side effects, e.g. I/O, thread creation, and exceptions
  - OCaml is not parallel — would transactional events work in a parallel or distributed setting?

Conclusions

- Transactional events are an elegant and powerful abstraction for concurrent programming.
- Our work allows TE to be used in impure languages.
- We have presented:
  - A reasonable semantics for mutation and nested synchronization within transactions
  - A formal description of our semantics
  - An implementation of our semantics in the OCaml runtime

Thank you!

Thanks to our reviewers, to everyone who gave feedback on the paper and talk, and to Matthew Fluet for his helpful input on this project.

Questions?

Proof and implementation: http://wasp.cs.washington.edu/tecaml