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

1. Background
2. Mutation Within Transactional Events
3. Formal Semantics and Implementation
4. Conclusion
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:

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

Thread 1

sync

chooseEvt

sendEvt c1 5
recvEvt c2
CML: Example

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

Thread 2
- sync
  - recvEvt c1
CML: Example

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

Thread 2
- sync
- recvEvt c1
CML: Example

Thread 1

sync

chooseEvt

sendEvt c1 5 recvEvt c2

continue

Thread 2

sync

recvEvt c1

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

**thenEvt type**

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

thenEvt succeeds when both sub-events succeed:

**Example**

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

Multiple communications per sync.
CML:
sendEvt
recvEvt
chooseEvt
...

one communication per sync
CML, TE Haskell, and TE ML

CML: sendEvt recvEvt chooseEvt ...

one communication per sync

TE Haskell: thenEvt

multiple communications + pure computation
CML, TE Haskell, and TE ML

CML:
- sendEvt
- recvEvt
- chooseEvt
- ...

TE Haskell:
- thenEvt

TE ML:
- pass impure functions to thenEvt

one communication per sync

multiple communications + impure computation

multiple communications + pure computation
An example using both thenEvt and chooseEvt.
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 \texttt{pred} modifies the heap, and then returns false?
Mutation in transactional events

- If we naively 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?
<table>
<thead>
<tr>
<th>The problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>How should we define the semantics of mutation within transactional events?</td>
</tr>
</tbody>
</table>
Outline

1 Background

2 Mutation Within Transactional Events

3 Formal Semantics and Implementation

4 Conclusion
Three proposals

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

1. Disallow mutation within transactions.
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.

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 #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 \( \text{fib} \) fails only if \( \text{fib} \) is memoized:

Example

```ocaml
define evenFibonacciGuard = guardedRecvEvt (fun x -> fib x % 2 = 0)
```
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.
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.
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**

<table>
<thead>
<tr>
<th>Thread 1: thenEvt (sendEvt c 0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(fun _ -&gt; r := 1; r := 0;</td>
</tr>
<tr>
<td>alwaysEvt ())</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Thread 2: thenEvt (recvEvt c)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(fun _ -&gt; if !r = 1</td>
</tr>
<tr>
<td>then alwaysEvt ()</td>
</tr>
<tr>
<td>else neverEvt)</td>
</tr>
</tbody>
</table>

- **Con**: Too slow — searches all possible interleavings!
Proposal #3: Chunking

```plaintext
x := 1
!x
recvEvt
Thread 1
sendEvt
Thread 2
recvEvt
x := 2
```
Proposal #3: Chunking

Thread 1
- sendEvt
- $x := 1$
- $!x$
- recvEvt

Thread 2
- recvEvt
- $x := 2$
- sendEvt
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

1. Background

2. Mutation Within Transactional Events

3. Formal Semantics and Implementation

4. Conclusion
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
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.:

  ```ocaml
  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