# A Monadic Account of First-Class Synchronous Events

Matthew Fluet

January 17, 2006

# Introduction

"A value of type `IO a` is an 'action' that, when performed, may do some input/output, before delivering a value of type `a`."

[Peyton Jones, *Tackling the Awkward Squad*]

"The type $\tau$ `event` is the type of a synchronous operation that returns a value of type $\tau$ when it is synchronized upon."

[Reppy, *Concurrent Programming in ML*]

# Introduction

"A value of type `IO a` is an 'action' that, when performed, may do some input/output, before delivering a value of type `a`."

[Peyton Jones, *Tackling the Awkward Squad*]

essence of *monadic encapsulation of I/O*

"The type $\tau$ `event` is the type of a synchronous operation that returns a value of type $\tau$ when it is synchronized upon."

[Reppy, *Concurrent Programming in ML*]

essence of *first-class synchronous events*

# Introduction

Does the `event` type constructor of Concurrent ML form a *monad*?

# Introduction

Does the `event` type constructor of Concurrent ML form a *monad*?

Since *non-deterministic choice* is a fundamental event combinator, does the `event` type constructor of Concurrent ML form a *monad-with-plus*?

# Introduction

Does the `event` type constructor of Concurrent ML form a *monad*?

Since *non-deterministic choice* is a fundamental event combinator, does the `event` type constructor of Concurrent ML form a *monad-with-plus*?

If the `event` type constructor *does not* form a monad-with-plus, then what goes wrong? And how can it be "fixed" retaining the *spirit* of CML?

# Outline

- Introduction
- Background
  - Concurrent ML
  - Monads and Monad Laws
- Monadic CML?
- Concurrent Functional Pidgin
  - Syntax and Semantics
  - Encoding CML
  - Expressiveness
  - Encoding STM Haskell
- Future Work and Conclusions

# Concurrent ML

Concurrent ML = Standard ML + concurrency primitives

# Concurrent ML

Concurrent ML = Standard ML + concurrency primitives

Key ideas:

# Concurrent ML

Concurrent ML = Standard ML + concurrency primitives

Key ideas: *describe* synchronous operations with first-class *events*

# Concurrent ML

Concurrent ML = Standard ML + concurrency primitives

Key ideas: *describe* synchronous operations with first-class *events*

*combine* simple *events* into complex *events*

# Concurrent ML

Concurrent ML = Standard ML + concurrency primitives

Key ideas:
- *describe* synchronous operations with first-class *events*
- *combine* simple *events* into complex *events*
- *perform* synchronous operations by synchronizing on *events*

# Concurrent ML: Interface

# Concurrent ML: Interface

```
val  spawn      :  (unit -> unit) -> thread_id
val  channel    :  unit -> 'a chan
```

# Concurrent ML: Interface

```
val  spawn      :  (unit -> unit) -> thread_id
val  channel    :  unit -> 'a chan

val  recvEvt    :  'a chan -> 'a event
val  sendEvt    :  'a chan * 'a -> unit event
```

# Concurrent ML: Interface

```
val  spawn      :  (unit -> unit) -> thread_id
val  channel    :  unit -> 'a chan

val  recvEvt    :  'a chan -> 'a event
val  sendEvt    :  'a chan * 'a -> unit event

val  wrap       :  'a event * ('a -> 'b) -> 'b event
```

# Concurrent ML: Interface

```
val  spawn      :  (unit -> unit) -> thread_id
val  channel    :  unit -> 'a chan

val  recvEvt    :  'a chan -> 'a event
val  sendEvt    :  'a chan * 'a -> unit event

val  wrap       :  'a event * ('a -> 'b) -> 'b event
val  guard      :  (unit -> 'a event) -> 'a event
```

# Concurrent ML: Interface

```
val  spawn      :  (unit -> unit) -> thread_id
val  channel    :  unit -> 'a chan

val  recvEvt    :  'a chan -> 'a event
val  sendEvt    :  'a chan * 'a -> unit event

val  wrap       :  'a event * ('a -> 'b) -> 'b event
val  guard      :  (unit -> 'a event) -> 'a event
val  choose     :  'a event list -> 'b event
```

# Concurrent ML: Interface

```
val  spawn      :  (unit -> unit) -> thread_id
val  channel    :  unit -> 'a chan

val  recvEvt    :  'a chan -> 'a event
val  sendEvt    :  'a chan * 'a -> unit event

val  wrap       :  'a event * ('a -> 'b) -> 'b event
val  guard      :  (unit -> 'a event) -> 'a event
val  choose     :  'a event list -> 'b event

val  sync       :  'a event -> 'a
```

# Concurrent ML: Interface

```
val  spawn      :  (unit -> unit) -> thread_id
val  channel    :  unit -> 'a chan

val  recvEvt    :  'a chan -> 'a event
val  sendEvt    :  'a chan * 'a -> unit event

val  wrap       :  'a event * ('a -> 'b) -> 'b event
val  guard      :  (unit -> 'a event) -> 'a event
val  choose     :  'a event list -> 'b event

val  sync       :  'a event -> 'a

val  never      :  'a event
val  alwaysEvt  :  'a -> 'a event
val  withNack   :  (unit event -> 'a event) -> 'a event
```

# Concurrent ML: Discussion

implement a *protocol*

$$c_1 \; ; \; c_2 \; ; \; \cdots \; ; \; c_{n-1} \; ; \; c_n$$

# Concurrent ML: Discussion

implement a *protocol*

$$c_1 \; ; \; c_2 \; ; \; \cdots \; ; \; c_{n-1} \; ; \; c_n$$

designate a *commit point*: $c_i$

$$c_i$$

# Concurrent ML: Discussion

implement a *protocol*

$$c_1 \; ; \; c_2 \; ; \; \cdots \; ; \; c_{n-1} \; ; \; c_n$$

designate a *commit point*: $c_i$

post-synchronous communications

added with `wrap`

$$c_i \overbrace{\; ; \; c_{i+1} \; ; \; \cdots \; ; \; c_n}$$

# Concurrent ML: Discussion

implement a *protocol*

$$c_1 \; ; \; c_2 \; ; \; \cdots \; ; \; c_{n-1} \; ; \; c_n$$

designate a *commit point*: $c_i$

post-synchronous communications
added with `wrap`

$$\underbrace{c_1 \; ; \; \cdots \; ; \; c_{i-1} \; ;}_{\text{pre-synchronous communications added with } \texttt{guard}} \; c_i \; \overbrace{; \; c_{i+1} \; ; \; \cdots \; ; \; c_n}^{\text{post-synchronous communications added with } \texttt{wrap}}$$

# Monads

# Monads

- Embed impure computational effects in a pure language (Haskell)
- Distinguish between
  - (pure) values of type $\tau$
  - (impure) computations yielding values of type $\tau$,
    which are themselves values of type $\mathsf{M}\ \tau$,
    where $\mathsf{M}$ abstracts a notion of computation and computational effect
- Laws restrict $\mathsf{M}$ to “well-behaved” computations

# Signature and Laws for *Monad*

# Signature and Laws for *Monad*

$$\begin{array}{rcl} \mathsf{M} & :: & \star \to \star \\ \mathsf{munit} & :: & \alpha \to \mathsf{M}\ \alpha \\ \mathsf{mbind} & :: & \mathsf{M}\ \alpha \to (\alpha \to \mathsf{M}\ \beta) \to \mathsf{M}\ \beta \end{array}$$

# Signature and Laws for *Monad*

$$\begin{array}{rcl} \mathsf{M} & :: & \star \to \star \\ \mathsf{munit} & :: & \alpha \to \mathsf{M}\ \alpha \\ \mathsf{mbind} & :: & \mathsf{M}\ \alpha \to (\alpha \to \mathsf{M}\ \beta) \to \mathsf{M}\ \beta \end{array}$$

$$\begin{array}{lrcl} \textit{MBIND\_LUNIT} & \mathsf{mbind}\ (\mathsf{munit}\ a)\ f & \cong & f\ a \\ \textit{MBIND\_RUNIT} & \mathsf{mbind}\ m\ \mathsf{munit} & \cong & m \\ \textit{MBIND\_ASSOC} & \mathsf{mbind}\ (\mathsf{mbind}\ m\ f)\ g & \cong & \mathsf{mbind}\ m\ (\lambda x.\ \mathsf{mbind}\ (f\ x)\ g) \end{array}$$

# Signature and Laws for *Monad*

$$
\begin{array}{rcl}
\mathsf{M} & :: & \star \to \star \\
\mathsf{munit} & :: & \alpha \to \mathsf{M}\ \alpha \\
\mathsf{mbind} & :: & \mathsf{M}\ \alpha \to (\alpha \to \mathsf{M}\ \beta) \to \mathsf{M}\ \beta \\
\mathsf{mmap} & :: & (\alpha \to \beta) \to (\mathsf{M}\ \alpha \to \mathsf{M}\ \beta) \\
\mathsf{mjoin} & :: & \mathsf{M}\ (\mathsf{M}\ \alpha) \to \mathsf{M}\ \alpha
\end{array}
$$

$$
\begin{array}{lrcl}
\textit{MBIND\_LUNIT} & \mathsf{mbind}\ (\mathsf{munit}\ a)\ f & \cong & f\ a \\
\textit{MBIND\_RUNIT} & \mathsf{mbind}\ m\ \mathsf{munit} & \cong & m \\
\textit{MBIND\_ASSOC} & \mathsf{mbind}\ (\mathsf{mbind}\ m\ f)\ g & \cong & \mathsf{mbind}\ m\ (\lambda x.\ \mathsf{mbind}\ (f\ x)\ g) \\
 & & & \\
\textit{MMAP\_DEF} & \mathsf{mmap}\ f\ m & \cong & \mathsf{mbind}\ m\ (\lambda x.\ \mathsf{munit}\ (f\ x)) \\
\textit{MJOIN\_DEF} & \mathsf{mjoin}\ m & \cong & \mathsf{mbind}\ m\ \mathsf{id}
\end{array}
$$

# Signature and Laws for *Monad*

$$
\begin{array}{rcl}
\mathsf{M} & :: & \star \to \star \\
\mathsf{munit} & :: & \alpha \to \mathsf{M}\ \alpha \\
\mathsf{mbind} & :: & \mathsf{M}\ \alpha \to (\alpha \to \mathsf{M}\ \beta) \to \mathsf{M}\ \beta \\
\mathsf{mmap} & :: & (\alpha \to \beta) \to (\mathsf{M}\ \alpha \to \mathsf{M}\ \beta) \\
\mathsf{mjoin} & :: & \mathsf{M}\ (\mathsf{M}\ \alpha) \to \mathsf{M}\ \alpha
\end{array}
$$

$$
\begin{array}{lrcl}
\textit{MBIND\_LUNIT} & \mathsf{mbind}\ (\mathsf{munit}\ a)\ f & \cong & f\ a \\
\textit{MBIND\_RUNIT} & \mathsf{mbind}\ m\ \mathsf{munit} & \cong & m \\
\textit{MBIND\_ASSOC} & \mathsf{mbind}\ (\mathsf{mbind}\ m\ f)\ g & \cong & \mathsf{mbind}\ m\ (\lambda x.\ \mathsf{mbind}\ (f\ x)\ g)
\end{array}
$$

$$
\begin{array}{lrcl}
\textit{MMAP\_DEF} & \mathsf{mmap}\ f\ m & \cong & \mathsf{mbind}\ m\ (\lambda x.\ \mathsf{munit}\ (f\ x)) \\
\textit{MJOIN\_DEF} & \mathsf{mjoin}\ m & \cong & \mathsf{mbind}\ m\ \mathsf{id} \\
\textit{MBIND\_DEF} & \mathsf{mbind}\ m\ f & \cong & \mathsf{mjoin}\ (\mathsf{mmap}\ f\ m)
\end{array}
$$

# Signature and Laws for *MonadPlus*

$$\begin{aligned} \mathsf{mzero} &:: \mathsf{M}\ \alpha \\ \mathsf{mplus} &:: \mathsf{M}\ \alpha \to \mathsf{M}\ \alpha \to \mathsf{M}\ \alpha \end{aligned}$$

# Signature and Laws for *MonadPlus*

$$\begin{aligned} \mathsf{mzero} &:: \mathsf{M}\ \alpha \\ \mathsf{mplus} &:: \mathsf{M}\ \alpha \to \mathsf{M}\ \alpha \to \mathsf{M}\ \alpha \end{aligned}$$

| | | | |
|---|---|---|---|
| *MPLUS_LUNIT* | $\mathsf{mplus}\ \mathsf{mzero}\ m$ | $\cong$ | $m$ |
| *MPLUS_RUNIT* | $\mathsf{mplus}\ m\ \mathsf{mzero}$ | $\cong$ | $m$ |
| *MPLUS_ASSOC* | $\mathsf{mplus}\ m_1\ (\mathsf{mplus}\ m_2\ m_3)$ | $\cong$ | $\mathsf{mplus}\ (\mathsf{mplus}\ m_1\ m_2)\ m_3$ |
| *MPLUS_COMMUT* | $\mathsf{mplus}\ m_1\ m_2$ | $\cong$ | $\mathsf{mplus}\ m_2\ m_1$ |

# Signature and Laws for *MonadPlus*

$$\begin{aligned} \mathsf{mzero} &:: \mathsf{M}\ \alpha \\ \mathsf{mplus} &:: \mathsf{M}\ \alpha \to \mathsf{M}\ \alpha \to \mathsf{M}\ \alpha \end{aligned}$$

| | | | |
|---|---|---|---|
| *MPLUS_LUNIT* | $\mathsf{mplus}\ \mathsf{mzero}\ m$ | $\cong$ | $m$ |
| *MPLUS_RUNIT* | $\mathsf{mplus}\ m\ \mathsf{mzero}$ | $\cong$ | $m$ |
| *MPLUS_ASSOC* | $\mathsf{mplus}\ m_1\ (\mathsf{mplus}\ m_2\ m_3)$ | $\cong$ | $\mathsf{mplus}\ (\mathsf{mplus}\ m_1\ m_2)\ m_3$ |
| *MPLUS_COMMUT* | $\mathsf{mplus}\ m_1\ m_2$ | $\cong$ | $\mathsf{mplus}\ m_2\ m_1$ |
| *MBIND_LZERO* | $\mathsf{mbind}\ \mathsf{mzero}\ f$ | $\cong$ | $\mathsf{mzero}$ |
| *MBIND_RZERO* | $\mathsf{mbind}\ m\ (\lambda x.\ \mathsf{mzero})$ | $\cong$ | $\mathsf{mzero}$ |

# Signature and Laws for *MonadPlus*

$$\begin{aligned} \mathsf{mzero} &:: \mathsf{M}\ \alpha \\ \mathsf{mplus} &:: \mathsf{M}\ \alpha \to \mathsf{M}\ \alpha \to \mathsf{M}\ \alpha \end{aligned}$$

| | | | |
|---|---|---|---|
| *MPLUS_LUNIT* | $\mathsf{mplus}\ \mathsf{mzero}\ m$ | $\cong$ | $m$ |
| *MPLUS_RUNIT* | $\mathsf{mplus}\ m\ \mathsf{mzero}$ | $\cong$ | $m$ |
| *MPLUS_ASSOC* | $\mathsf{mplus}\ m_1\ (\mathsf{mplus}\ m_2\ m_3)$ | $\cong$ | $\mathsf{mplus}\ (\mathsf{mplus}\ m_1\ m_2)\ m_3$ |
| *MPLUS_COMMUT* | $\mathsf{mplus}\ m_1\ m_2$ | $\cong$ | $\mathsf{mplus}\ m_2\ m_1$ |
| *MBIND_LZERO* | $\mathsf{mbind}\ \mathsf{mzero}\ f$ | $\cong$ | $\mathsf{mzero}$ |
| *MBIND_RZERO* | $\mathsf{mbind}\ m\ (\lambda x.\ \mathsf{mzero})$ | $\cong$ | $\mathsf{mzero}$ |

*LDIST*
$$\mathsf{mbind}\ (\mathsf{mplus}\ m_1\ m_2)\ f \cong \mathsf{mplus}\ (\mathsf{mbind}\ m_1\ f)\ (\mathsf{mbind}\ m_2\ f)$$

*RDIST*
$$\mathsf{mbind}\ m\ (\lambda x.\ \mathsf{mplus}\ (f\ x)\ (g\ x)) \cong \mathsf{mplus}\ (\mathsf{mbind}\ m\ f)\ (\mathsf{mbind}\ m\ f)$$

# Monadic CML? (First Attempt)

Does the `event` type constructor of Concurrent ML form a *monad-with-plus*?

# Monadic CML? (First Attempt)

Does the `event` type constructor of Concurrent ML form a *monad-with-plus*?

| MonadPlus | CML |
|---|---|
| M | `event` |
| munit | |
| mbind | |
| mmap | |
| mjoin | |
| mzero | |
| mplus | |

# Monadic CML? (First Attempt)

Does the `event` type constructor of Concurrent ML form a *monad-with-plus*?

| MonadPlus | CML |
|---|---|
| M | `event` |
| munit | `alwaysEvt` |
| mbind | |
| mmap | |
| mjoin | |
| mzero | |
| mplus | |

# Monadic CML? (First Attempt)

Does the `event` type constructor of Concurrent ML form a *monad-with-plus*?

| MonadPlus | CML |
|---|---|
| M | `event` |
| munit | `alwaysEvt` |
| mbind | |
| mmap | `fn f => fn ev => wrap (ev, f)` |
| mjoin | |
| mzero | |
| mplus | |

# Monadic CML? (First Attempt)

Does the `event` type constructor of Concurrent ML form a *monad-with-plus*?

| MonadPlus | CML |
|---|---|
| M | `event` |
| munit | `alwaysEvt` |
| mbind | |
| mmap | `fn f => fn ev => wrap (ev, f)` |
| mjoin | |
| mzero | `never` |
| mplus | `fn ev1 => fn ev2 => choose [ev1, ev2]` |

# Monadic CML? (First Attempt)

Does the `event` type constructor of Concurrent ML form a *monad-with-plus*?

| MonadPlus | CML |
|---|---|
| M | `event` |
| munit | `alwaysEvt` |
| mbind | |
| mmap | `fn f => fn ev => wrap (ev, f)` |
| mjoin | `sync` |
| mzero | `never` |
| mplus | `fn ev1 => fn ev2 => choose [ev1, ev2]` |

# Monadic CML? (First Attempt)

Does the `event` type constructor of Concurrent ML form a *monad-with-plus*?

| MonadPlus | CML |
|---|---|
| M | `event` |
| munit | `alwaysEvt` |
| mbind | `fn ev => fn f => sync (wrap (ev, f))` |
| mmap | `fn f => fn ev => wrap (ev, f)` |
| mjoin | `sync` |
| mzero | `never` |
| mplus | `fn ev1 => fn ev2 => choose [ev1, ev2]` |

# Monadic CML? (First Attempt)

Does the `event` type constructor of Concurrent ML form a *monad-with-plus*?

| MonadPlus | CML |
|---|---|
| M | `event` |
| munit | `alwaysEvt` |
| mbind | `fn ev => fn f => sync (wrap (ev, f))` |
| mmap | `fn f => fn ev => wrap (ev, f)` |
| mjoin | `sync` |
| mzero | `never` |
| mplus | `fn ev1 => fn ev2 => choose [ev1, ev2]` |

*MBIND_RUNIT*

`alwaysEvt (sync ev)` $\not\cong$ `ev`

# Monadic CML? (First Attempt)

Does the `event` type constructor of Concurrent ML form a *monad-with-plus*?

| MonadPlus | CML |
|---|---|
| M | `event` |
| munit | `alwaysEvt` |
| mbind | `fn ev => fn f => sync (wrap (ev, f))` |
| mmap | `fn f => fn ev => wrap (ev, f)` |
| mjoin | `sync` |
| mzero | `never` |
| mplus | `fn ev1 => fn ev2 => choose [ev1, ev2]` |

*MBIND_RUNIT*

`alwaysEvt (sync ev)` $\not\cong$ `ev`

Synchronizes *too early*

# Monadic CML? (Second Attempt)

Does the `event` type constructor of Concurrent ML form a *monad-with-plus*?

| MonadPlus | CML |
|---|---|
| M | `event` |
| munit | `alwaysEvt` |
| mbind | |
| mmap | `fn f => fn ev => wrap (ev, f)` |
| mjoin | |
| mzero | `never` |
| mplus | `fn ev1 => fn ev2 => choose [ev1, ev2]` |

# Monadic CML? (Second Attempt)

Does the `event` type constructor of Concurrent ML form a *monad-with-plus*?

| MonadPlus | CML |
|---|---|
| M | `event` |
| munit | `alwaysEvt` |
| mbind | |
| mmap | `fn f => fn ev => wrap (ev, f)` |
| mjoin | `fn ev => wrap (ev, sync)` |
| mzero | `never` |
| mplus | `fn ev1 => fn ev2 => choose [ev1, ev2]` |

# Monadic CML? (Second Attempt)

Does the `event` type constructor of Concurrent ML form a *monad-with-plus*?

| MonadPlus | CML |
|---|---|
| M | `event` |
| munit | `alwaysEvt` |
| mbind | `fn ev => fn f => wrap (wrap (ev, f), sync)` |
| mmap | `fn f => fn ev => wrap (ev, f)` |
| mjoin | `fn ev => wrap (ev, sync)` |
| mzero | `never` |
| mplus | `fn ev1 => fn ev2 => choose [ev1, ev2]` |

# Monadic CML? (Second Attempt)

Does the `event` type constructor of Concurrent ML form a *monad-with-plus*?

| MonadPlus | CML |
|---|---|
| M | `event` |
| munit | `alwaysEvt` |
| mbind | `fn ev => fn f => wrap (wrap (ev, f), sync)` |
| mmap | `fn f => fn ev => wrap (ev, f)` |
| mjoin | `fn ev => wrap (ev, sync)` |
| mzero | `never` |
| mplus | `fn ev1 => fn ev2 => choose [ev1, ev2]` |

*MBIND_RZERO*

`wrap (wrap (ev, fn x => never), sync)` $\not\cong$ `never`

# Monadic CML? (Second Attempt)

Does the `event` type constructor of Concurrent ML form a *monad-with-plus*?

| MonadPlus | CML |
|---|---|
| M | `event` |
| munit | `alwaysEvt` |
| mbind | `fn ev => fn f => wrap (wrap (ev, f), sync)` |
| mmap | `fn f => fn ev => wrap (ev, f)` |
| mjoin | `fn ev => wrap (ev, sync)` |
| mzero | `never` |
| mplus | `fn ev1 => fn ev2 => choose [ev1, ev2]` |

*MBIND_RZERO*

`wrap (wrap (ev, fn x => never), sync)` $\not\cong$ `never`

Synchronizes *too late*

# Concurrent Functional Pidgin (CFP)

# Concurrent Functional Pidgin (CFP)

Concurrent Functional Pidgin $=$

Concurrent ML (first-class synchronous events, forming a *monad-with-plus*)

$+$ Concurrent Haskell (first-class I/O actions, forming a *monad*)

# Concurrent Functional Pidgin (CFP)

Concurrent Functional Pidgin $=$

Concurrent ML (first-class synchronous events, forming a *monad-with-plus*)

$+$ Concurrent Haskell (first-class I/O actions, forming a *monad*)

| | | | |
|---|---|---|---|
| *Types* | $\tau$ | $::=$ | $\mathsf{b}$ — base type constants (int, char, etc.) |
| | | $\mid$ | $\alpha$ — type variables |
| | | $\mid$ | $\tau_2 \rightarrow \tau_2$ — function type |
| | | $\mid$ | $\tau_1 \times \cdots \times \tau_n$ — tuple type |
| | | $\mid$ | $\mathsf{Chan}\ \tau$ — channel type |
| | | $\mid$ | $\mathsf{Evt}\ \tau$ — event type (monadic) |
| | | $\mid$ | $\mathsf{IO}\ \tau$ — IO type (monadic) |

# CFP: Syntax

| | | | | |
|---|---|---|---|---|
| *Expressions* | $e$ | $::=$ | $x$ | variables |
| | | $\mid$ | $\mathfrak{b}$ | base constants (chars, ints, etc.) |
| | | $\mid$ | let $x = e_1$ in $e_2$ | let binding |
| | | $\mid$ | $\lambda x.\, e$ | function abstraction |
| | | $\mid$ | $\mu f.x.\, e$ | recursive function abstraction |
| | | $\mid$ | $e_1\ e_2$ | function application |
| | | $\mid$ | $(e_1, \ldots, e_n)$ | tuples |
| | | $\mid$ | $\pi_i\ e$ | tuple projection |
| | | $\mid$ | $\mathfrak{p}$ | primitives (monadic) |

# CFP: Syntax

| | | | | |
|---|---|---|---|---|
| *Expressions* | $e$ | $::=$ | $x$ | variables |
| | | $\mid$ | $\mathfrak{b}$ | base constants (chars, ints, etc.) |
| | | $\mid$ | let $x = e_1$ in $e_2$ | let binding |
| | | $\mid$ | $\lambda x.\, e$ | function abstraction |
| | | $\mid$ | $\mu f.x.\, e$ | recursive function abstraction |
| | | $\mid$ | $e_1\ e_2$ | function application |
| | | $\mid$ | $(e_1, \ldots, e_n)$ | tuples |
| | | $\mid$ | $\pi_i\ e$ | tuple projection |
| | | $\mid$ | $\mathfrak{p}$ | primitives (monadic) |

Pure Functional Core Language

# CFP: Evt Primitives

$$
\begin{array}{rcll}
\mathsf{alwaysEvt} & :: & \alpha \to \mathsf{Evt}\ \alpha & \text{(munit for Evt)} \\
\mathsf{thenEvt} & :: & \mathsf{Evt}\ \alpha \to (\alpha \to \mathsf{Evt}\ \beta) \to \mathsf{Evt}\ \beta & \text{(mbind for Evt)} \\
\mathsf{neverEvt} & :: & \mathsf{Evt}\ \alpha & \text{(mzero for Evt)} \\
\mathsf{chooseEvt} & :: & \mathsf{Evt}\ \alpha \to \mathsf{Evt}\ \alpha \to \mathsf{Evt}\ \alpha & \text{(mplus for Evt)} \\
\mathsf{channel} & :: & \mathsf{Evt}\ (\mathsf{Chan}\ \alpha) & \\
\mathsf{recvEvt} & :: & \mathsf{Chan}\ \alpha \to \mathsf{Evt}\ \alpha & \\
\mathsf{sendEvt} & :: & \mathsf{Chan}\ \alpha \to \alpha \to \mathsf{Evt}\ () &
\end{array}
$$

# CFP: Evt Primitives

$$
\begin{array}{rcll}
\mathsf{alwaysEvt} & :: & \alpha \to \mathsf{Evt}\ \alpha & \text{(munit for Evt)} \\
\mathsf{thenEvt} & :: & \mathsf{Evt}\ \alpha \to (\alpha \to \mathsf{Evt}\ \beta) \to \mathsf{Evt}\ \beta & \text{(mbind for Evt)} \\
\mathsf{neverEvt} & :: & \mathsf{Evt}\ \alpha & \text{(mzero for Evt)} \\
\mathsf{chooseEvt} & :: & \mathsf{Evt}\ \alpha \to \mathsf{Evt}\ \alpha \to \mathsf{Evt}\ \alpha & \text{(mplus for Evt)} \\
\mathsf{channel} & :: & \mathsf{Evt}\ (\mathsf{Chan}\ \alpha) & \\
\mathsf{recvEvt} & :: & \mathsf{Chan}\ \alpha \to \mathsf{Evt}\ \alpha & \\
\mathsf{sendEvt} & :: & \mathsf{Chan}\ \alpha \to \alpha \to \mathsf{Evt}\ () &
\end{array}
$$

# CFP: IO Primitives

$$
\begin{array}{rcll}
\mathsf{unitIO} & :: & \alpha \to \mathsf{IO}\ \alpha & \text{(munit for IO)} \\
\mathsf{bindEvt} & :: & \mathsf{IO}\ \alpha \to (\alpha \to \mathsf{IO}\ \beta) \to \mathsf{IO}\ \beta & \text{(mbind for IO)} \\
\mathsf{getChar} & :: & \mathsf{IO\ Char} & \\
\mathsf{putChar} & :: & \mathsf{Char} \to \mathsf{IO}\ () & \\
\mathsf{spawn} & :: & \mathsf{IO}\ () \to \mathsf{IO\ ThreadID} & \\
\mathsf{getTId} & :: & \mathsf{IO\ ThreadID} & \\
\mathsf{sync} & :: & \mathsf{Evt}\ \alpha \to \mathsf{IO}\ \alpha &
\end{array}
$$

# CFP: IO Primitives

| | | | |
|---|---|---|---|
| unitIO | :: | $\alpha \to \mathsf{IO}\ \alpha$ | (munit for IO) |
| bindEvt | :: | $\mathsf{IO}\ \alpha \to (\alpha \to \mathsf{IO}\ \beta) \to \mathsf{IO}\ \beta$ | (mbind for IO) |
| getChar | :: | IO Char | |
| putChar | :: | Char → IO () | |
| spawn | :: | IO () → IO ThreadID | |
| getTId | :: | IO ThreadID | |
| sync | :: | Evt $\alpha$ → IO $\alpha$ | |

# CFP Evaluation

Three levels of evaluation:

# CFP Evaluation

Three levels of evaluation:

- Sequential Evaluation of pure terms

$$e \hookrightarrow e'$$

# CFP Evaluation

Three levels of evaluation:

- Sequential Evaluation of pure terms

$$e \hookrightarrow e'$$

- Synchronous Evaluation of synchronizing events (Evt monad)

$$(e_1, \ldots, e_k) \rightsquigarrow_k (e'_1, \ldots, e'_k)$$

# CFP Evaluation

Three levels of evaluation:

- Sequential Evaluation of pure terms

$$e \hookrightarrow e'$$

- Synchronous Evaluation of synchronizing events (Evt monad)

$$(e_1, \ldots, e_k) \rightsquigarrow_k (e'_1, \ldots, e'_k)$$

- Concurrent Evaluation of concurrent threads (IO monad)

$$\mathcal{T} \xrightarrow{a} \mathcal{T}' \qquad \mathcal{T}; \mathcal{S} \xrightarrow{a} \mathcal{T}'; \mathcal{S}'$$

# Synchronous Evaluation

*Evaluation Contexts* $\quad M^{Evt} \quad ::= \quad [] \mid \mathsf{thenEvt}\ M_1^{Evt}\ v_2$

# Synchronous Evaluation

*Evaluation Contexts* $M^{Evt} ::= [] \mid \mathsf{thenEvt}\ M_1^{Evt}\ v_2$

$$\textsc{EvtEval}\ \frac{e \hookrightarrow e'}{(M^{Evt}[e]) \rightsquigarrow_1 (M^{Evt}[e'])}$$

# Synchronous Evaluation

*Evaluation Contexts* $M^{Evt} ::= [] \mid \mathsf{thenEvt}\ M_1^{Evt}\ v_2$

$$\textsc{EvtEval}\ \frac{e \hookrightarrow e'}{(M^{Evt}[e]) \rightsquigarrow_1 (M^{Evt}[e'])}$$

$$\textsc{EvtBind}\ \frac{}{(M^{Evt}[\mathsf{thenEvt}\ (\mathsf{alwaysEvt}\ v_1)\ v_2]) \rightsquigarrow_1 (M^{Evt}[v_2\ v_1])}$$

# Synchronous Evaluation

*Evaluation Contexts* $M^{Evt} ::= [] \mid \mathsf{thenEvt}\ M_1^{Evt}\ v_2$

EvtEval

$$\frac{e \hookrightarrow e'}{(M^{Evt}[e]) \rightsquigarrow_1 (M^{Evt}[e'])}$$

EvtBind

$$\frac{}{(M^{Evt}[\mathsf{thenEvt}\ (\mathsf{alwaysEvt}\ v_1)\ v_2]) \rightsquigarrow_1 (M^{Evt}[v_2\ v_1])}$$

EvtPlus1

$$\frac{}{(M^{Evt}[\mathsf{chooseEvt}\ v_1\ v_2]) \rightsquigarrow_1 (M^{Evt}[v_1])}$$

EvtPlus2

$$\frac{}{(M^{Evt}[\mathsf{chooseEvt}\ v_1\ v_2]) \rightsquigarrow_1 (M^{Evt}[v_2])}$$

# Synchronous Evaluation

# Synchronous Evaluation

EvtSendRecv

$$\frac{}{(M_1^{Evt}[\mathsf{sendEvt}\ \kappa\ v], M_2^{Evt}[\mathsf{recvEvt}\ \kappa]) \rightsquigarrow_2 (M_1^{Evt}[\mathsf{alwaysEvt}\ ()], M_2^{Evt}[\mathsf{alwaysEvt}\ v])}$$

Channel

$$\frac{\kappa' \text{ fresh}}{(M^{Evt}[\mathsf{channel}]) \rightsquigarrow_1 (M^{Evt}[\mathsf{alwaysEvt}\ \kappa'])}$$

# Synchronous Evaluation

EvtSendRecv

$$\frac{}{(M_1^{Evt}[\mathsf{sendEvt}\ \kappa\ v], M_2^{Evt}[\mathsf{recvEvt}\ \kappa]) \rightsquigarrow_2 (M_1^{Evt}[\mathsf{alwaysEvt}\ ()], M_2^{Evt}[\mathsf{alwaysEvt}\ v])}$$

Channel

$$\frac{\kappa' \text{ fresh}}{(M^{Evt}[\mathsf{channel}]) \rightsquigarrow_1 (M^{Evt}[\mathsf{alwaysEvt}\ \kappa'])}$$

Permutation

$$\frac{p \in \mathrm{Perm}_k \qquad (e_{p(1)}, \ldots, e_{p(k)}) \rightsquigarrow_k (e'_1, \ldots, e'_k)}{(e_1, \ldots, e_k) \rightsquigarrow_k (e'_{\overline{p}(1)}, \ldots, e'_{\overline{p}(k)})}$$

Subset

$$\frac{1 \leq j \leq k \qquad (e_1, \ldots, e_j) \rightsquigarrow_j (e'_1, \ldots, e'_j)}{(e_1, \ldots, e_j, e_{j+1} \ldots, e_k) \rightsquigarrow_k (e'_1, \ldots, e'_j, e_{j+1} \ldots, e_k)}$$

# Synchronous Evaluation

What are the terminal configurations for $\rightsquigarrow_k$?

# Synchronous Evaluation

What are the terminal configurations for $\rightsquigarrow_k$?

$$(e_1, \ldots, e_k) \rightsquigarrow_k^* \text{???}$$

# Synchronous Evaluation

What are the terminal configurations for $\rightsquigarrow_k$?

$$(e_1, \ldots, e_k) \rightsquigarrow_k^* \text{???}$$

- "good" terminal configurations :
  - $(\mathsf{alwaysEvt}\ v_1, \ldots, \mathsf{alwaysEvt}\ v_n)$
- "bad" terminal configurations :
  - $(\ldots, \mathsf{neverEvt}, \ldots)$
  - $(\ldots, \mathsf{recvEvt}\ \kappa, \ldots)$ (unmatched)
  - $(\ldots, \mathsf{sendEvt}\ \kappa\ v, \ldots)$ (unmateched)

# Concurrent Evaluation

| | | | |
|---|---|---|---|
| *Thread IDs* | $\theta$ | $\in$ | $TId$ |
| *Concurrent Threads* | $T$ | $::=$ | $\langle \theta, e \rangle$ |
| *Concurrent Thread Groups* | $\mathcal{T}$ | $::=$ | $\{T, \ldots\}$ |

*Actions* $a ::= ?c \mid !c \mid \epsilon$

# Concurrent Evaluation

*Evaluation Contexts* $M^{IO} ::= [] \mid \mathsf{bindIO}\ M_1^{IO}\ v_2$

IOEVAL

$$\frac{e \hookrightarrow e'}{\mathcal{T} \uplus \{\langle \theta, M^{IO}[e] \rangle\} \xrightarrow{\epsilon} \mathcal{T} \uplus \{\langle \theta, M^{IO}[e'] \rangle\}}$$

IOBIND

$$\frac{}{\mathcal{T} \uplus \{\langle \theta, M^{IO}[\mathsf{bindIO}\ (\mathsf{unitIO}\ v_1)\ v_2] \rangle\} \xrightarrow{\epsilon} \mathcal{T} \uplus \{\langle \theta, M^{IO}[v_2\ v_1] \rangle\}}$$

# Concurrent Evaluation

IOGetChar

$$\frac{}{\mathcal{T} \uplus \{\langle \theta, M^{IO}[\mathsf{getChar}]\rangle\} \xrightarrow{?c} \mathcal{T} \uplus \{\langle \theta, M^{IO}[\mathsf{unitIO}\ c]\rangle\}}$$

IOPutChar

$$\frac{}{\mathcal{T} \uplus \{\langle \theta, M^{IO}[\mathsf{putChar}\ c]\rangle\} \xrightarrow{!c} \mathcal{T} \uplus \{\langle \theta, M^{IO}[\mathsf{unitIO}\ ()]\rangle\}}$$

# Concurrent Evaluation

IOGetChar

$$\mathcal{T} \uplus \{\langle \theta, M^{IO}[\mathsf{getChar}]\rangle\} \xrightarrow{?c} \mathcal{T} \uplus \{\langle \theta, M^{IO}[\mathsf{unitIO}\ c]\rangle\}$$

IOPutChar

$$\mathcal{T} \uplus \{\langle \theta, M^{IO}[\mathsf{putChar}\ c]\rangle\} \xrightarrow{!c} \mathcal{T} \uplus \{\langle \theta, M^{IO}[\mathsf{unitIO}\ ()]\rangle\}$$

Spawn

$$\frac{\theta' \text{ fresh}}{\mathcal{T} \uplus \{\langle \theta, M^{IO}[\mathsf{spawn}\ v]\rangle\} \xrightarrow{\epsilon} \mathcal{T} \uplus \{\langle \theta, M^{IO}[\mathsf{unitIO}\ \theta']\rangle, \langle \theta', v\rangle\}}$$

GetTId

$$\mathcal{T} \uplus \{\langle \theta, M^{IO}[\mathsf{getTId}]\rangle\} \xrightarrow{\epsilon} \mathcal{T} \uplus \{\langle \theta, M^{IO}[\mathsf{unitIO}\ \theta]\rangle\}$$

# Concurrent Evaluation

SYNC

$$\frac{}{\mathcal{T} \uplus \{\langle \theta_1, M_1^{IO}[\mathsf{sync}\ v_1]\rangle, \ldots, \langle \theta_k, M_k^{IO}[\mathsf{sync}\ v_k]\rangle\} \rightarrow}$$

# Concurrent Evaluation

$$\textsc{Sync}\ \frac{(v_1, \ldots, v_k) \rightsquigarrow_k}{\mathcal{T} \uplus \{\langle \theta_1, M_1^{IO}[\mathsf{sync}\ v_1]\rangle, \ldots, \langle \theta_k, M_k^{IO}[\mathsf{sync}\ v_k]\rangle\} \rightarrow}$$

# Concurrent Evaluation

$$\textsc{Sync}\quad \frac{(v_1, \ldots, v_k) \rightsquigarrow_k^* (\mathsf{alwaysEvt}\ v_1', \ldots, \mathsf{alwaysEvt}\ v_k')}{\mathcal{T} \uplus \{\langle \theta_1, M_1^{IO}[\mathsf{sync}\ v_1]\rangle, \ldots, \langle \theta_k, M_k^{IO}[\mathsf{sync}\ v_k]\rangle\} \rightarrow}$$

# Concurrent Evaluation

$$\textsc{Sync}\quad \frac{(v_1, \ldots, v_k) \rightsquigarrow_k^* (\mathsf{alwaysEvt}\ v_1', \ldots, \mathsf{alwaysEvt}\ v_k')}{\begin{array}{l}\mathcal{T} \uplus \{\langle \theta_1, M_1^{IO}[\mathsf{sync}\ v_1]\rangle, \ldots, \langle \theta_k, M_k^{IO}[\mathsf{sync}\ v_k]\rangle\} \longrightarrow \\ \quad \mathcal{T} \uplus \{\langle \theta_1, M_1^{IO}[\mathsf{unitIO}\ v_1']\rangle, \ldots, \langle \theta_k, M_k^{IO}[\mathsf{unitIO}\ v_k']\rangle\}\end{array}}$$

# Concurrent Evaluation

$$\textsc{Sync}\quad \frac{(v_1, \ldots, v_k) \rightsquigarrow_k^* (\mathsf{alwaysEvt}\ v_1', \ldots, \mathsf{alwaysEvt}\ v_k')}{\mathcal{T} \uplus \{\langle \theta_1, M_1^{IO}[\mathsf{sync}\ v_1]\rangle, \ldots, \langle \theta_k, M_k^{IO}[\mathsf{sync}\ v_k]\rangle\} \xrightarrow{\epsilon} \mathcal{T} \uplus \{\langle \theta_1, M_1^{IO}[\mathsf{unitIO}\ v_1']\rangle, \ldots, \langle \theta_k, M_k^{IO}[\mathsf{unitIO}\ v_k']\rangle\}}$$

# Concurrent Evaluation (Angelic)

$$\textsc{Sync}\ \frac{(v_1, \ldots, v_k) \rightsquigarrow_k^* (\mathsf{alwaysEvt}\ v_1', \ldots, \mathsf{alwaysEvt}\ v_k')}{\mathcal{T} \uplus \{\langle \theta_1, M_1^{IO}[\mathsf{sync}\ v_1]\rangle, \ldots, \langle \theta_k, M_k^{IO}[\mathsf{sync}\ v_k]\rangle\} \xrightarrow{\epsilon} \mathcal{T} \uplus \{\langle \theta_1, M_1^{IO}[\mathsf{unitIO}\ v_1']\rangle, \ldots, \langle \theta_k, M_k^{IO}[\mathsf{unitIO}\ v_k']\rangle\}}$$

# Concurrent Evaluation

# Concurrent Evaluation

SYNC

$$\frac{}{\mathcal{T} \uplus \{\langle \theta_1, M_1^{IO}[\mathsf{sync}\ v_1]\rangle, \ldots, \langle \theta_k, M_k^{IO}[\mathsf{sync}\ v_k]\rangle\}} \quad \xrightarrow{\epsilon}$$

# Concurrent Evaluation

SYNC

$$\mathcal{T} \uplus \{\langle \theta_1, M_1^{IO}[\mathsf{sync}\ v_1]\rangle, \ldots, \langle \theta_k, M_k^{IO}[\mathsf{sync}\ v_k]\rangle\}; \mathcal{S} \xrightarrow{\epsilon}$$

# Concurrent Evaluation

SYNC

$$\frac{\sigma' \text{ fresh}}{\mathcal{T} \uplus \{\langle \theta_1, M_1^{IO}[\mathsf{sync}\ v_1]\rangle, \ldots, \langle \theta_k, M_k^{IO}[\mathsf{sync}\ v_k]\rangle\}; \mathcal{S} \xrightarrow{\epsilon} \mathcal{T}; \mathcal{S} \uplus \{\langle \sigma', \langle\langle \theta_1, M_1^{IO}, v_1, v_1\rangle, \ldots \langle \theta_k, M_k^{IO}, v_k, v_k\rangle\rangle\rangle\}}$$

# Concurrent Evaluation

SyncInit

$$\frac{\sigma' \text{ fresh}}{\mathcal{T} \uplus \{\langle \theta_1, M_1^{IO}[\mathsf{sync}\ v_1]\rangle, \ldots, \langle \theta_k, M_k^{IO}[\mathsf{sync}\ v_k]\rangle\}; \mathcal{S} \xrightarrow{\epsilon} \mathcal{T}; \mathcal{S} \uplus \{\langle \sigma', \langle\langle \theta_1, M_1^{IO}, v_1, v_1\rangle, \ldots \langle \theta_k, M_k^{IO}, v_k, v_k\rangle\rangle\rangle\}}$$

# Concurrent Evaluation

SyncInit

$$\frac{\sigma' \text{ fresh}}{\mathcal{T} \uplus \{\langle \theta_1, M_1^{IO}[\mathsf{sync}\ v_1]\rangle, \ldots, \langle \theta_k, M_k^{IO}[\mathsf{sync}\ v_k]\rangle\}; \mathcal{S} \xrightarrow{\epsilon} \mathcal{T}; \mathcal{S} \uplus \{\langle \sigma', \langle\langle \theta_1, M_1^{IO}, v_1, v_1\rangle, \ldots \langle \theta_k, M_k^{IO}, v_k, v_k\rangle\rangle\rangle\}}$$

SyncStep

$$\frac{(e_1', \ldots, e_k') \rightsquigarrow_k (e_1'', \ldots, e_k'')}{\mathcal{T}; \mathcal{S} \uplus \{\langle \sigma, \langle\langle \theta_1, M_1^{IO}, v_1, e_1'\rangle, \ldots \langle \theta_k, M_k^{IO}, v_k, e_k'\rangle\rangle\rangle\} \xrightarrow{\epsilon} \mathcal{T}; \mathcal{S} \uplus \{\langle \sigma, \langle\langle \theta_1, M_1^{IO}, v_1, e_1''\rangle, \ldots \langle \theta_k, M_k^{IO}, v_k, e_k''\rangle\rangle\rangle\}}$$

# Concurrent Evaluation

SyncInit

$$\frac{\sigma' \text{ fresh}}{\begin{array}{l}\mathcal{T} \uplus \{\langle \theta_1, M_1^{IO}[\mathsf{sync}\ v_1]\rangle, \ldots, \langle \theta_k, M_k^{IO}[\mathsf{sync}\ v_k]\rangle\}; \mathcal{S} \xrightarrow{\epsilon} \\ \quad \mathcal{T}; \mathcal{S} \uplus \{\langle \sigma', \langle\langle \theta_1, M_1^{IO}, v_1, v_1\rangle, \ldots \langle \theta_k, M_k^{IO}, v_k, v_k\rangle\rangle\rangle\}\end{array}}$$

SyncStep

$$\frac{(e_1', \ldots, e_k') \rightsquigarrow_k (e_1'', \ldots, e_k'')}{\begin{array}{l}\mathcal{T}; \mathcal{S} \uplus \{\langle \sigma, \langle\langle \theta_1, M_1^{IO}, v_1, e_1'\rangle, \ldots \langle \theta_k, M_k^{IO}, v_k, e_k'\rangle\rangle\rangle\} \xrightarrow{\epsilon} \\ \quad \mathcal{T}; \mathcal{S} \uplus \{\langle \sigma, \langle\langle \theta_1, M_1^{IO}, v_1, e_1''\rangle, \ldots \langle \theta_k, M_k^{IO}, v_k, e_k''\rangle\rangle\rangle\}\end{array}}$$

SyncCommit

$$\frac{}{\begin{array}{l}\mathcal{T}; \mathcal{S} \uplus \{\langle \sigma, \langle\langle \theta_1, M_1^{IO}, v_1, \mathsf{alwaysEvt}\ v_1'\rangle, \ldots \langle \theta_k, M_k^{IO}, v_k, \mathsf{alwaysEvt}\ v_k'\rangle\rangle\rangle\} \xrightarrow{\epsilon} \\ \quad \mathcal{T} \uplus \{\langle \theta_1, M_1^{IO}[\mathsf{unitIO}\ v_1']\rangle, \ldots, \langle \theta_k, M_k^{IO}[\mathsf{unitIO}\ v_k']\rangle\}; \mathcal{S}\end{array}}$$

# Concurrent Evaluation

SyncInit

$$\frac{\sigma' \text{ fresh}}{\mathcal{T} \uplus \{\langle \theta_1, M_1^{IO}[\mathsf{sync}\ v_1]\rangle, \ldots, \langle \theta_k, M_k^{IO}[\mathsf{sync}\ v_k]\rangle\}; \mathcal{S} \xrightarrow{\epsilon} \mathcal{T}; \mathcal{S} \uplus \{\langle \sigma', \langle\langle \theta_1, M_1^{IO}, v_1, v_1\rangle, \ldots \langle \theta_k, M_k^{IO}, v_k, v_k\rangle\rangle\rangle\}}$$

SyncStep

$$\frac{(e_1', \ldots, e_k') \rightsquigarrow_k (e_1'', \ldots, e_k'')}{\mathcal{T}; \mathcal{S} \uplus \{\langle \sigma, \langle\langle \theta_1, M_1^{IO}, v_1, e_1'\rangle, \ldots \langle \theta_k, M_k^{IO}, v_k, e_k'\rangle\rangle\rangle\} \xrightarrow{\epsilon} \mathcal{T}; \mathcal{S} \uplus \{\langle \sigma, \langle\langle \theta_1, M_1^{IO}, v_1, e_1''\rangle, \ldots \langle \theta_k, M_k^{IO}, v_k, e_k''\rangle\rangle\rangle\}}$$

SyncCommit

$$\frac{}{\mathcal{T}; \mathcal{S} \uplus \{\langle \sigma, \langle\langle \theta_1, M_1^{IO}, v_1, \mathsf{alwaysEvt}\ v_1'\rangle, \ldots \langle \theta_k, M_k^{IO}, v_k, \mathsf{alwaysEvt}\ v_k'\rangle\rangle\rangle\} \xrightarrow{\epsilon} \mathcal{T} \uplus \{\langle \theta_1, M_1^{IO}[\mathsf{unitIO}\ v_1']\rangle, \ldots, \langle \theta_k, M_k^{IO}[\mathsf{unitIO}\ v_k']\rangle\}; \mathcal{S}}$$

SyncAbort

$$\frac{}{\mathcal{T}; \mathcal{S} \uplus \{\langle \sigma, \langle\langle \theta_1, M_1^{IO}, v_1, e_1'\rangle, \ldots \langle \theta_k, M_k^{IO}, v_k, e_k'\rangle\rangle\rangle\} \xrightarrow{\epsilon} \mathcal{T} \uplus \{\langle \theta_1, M_1^{IO}[\mathsf{sync}\ v_1]\rangle, \ldots, \langle \theta_k, M_k^{IO}[\mathsf{sync}\ v_k]\rangle\}; \mathcal{S}}$$

# Concurrent Evaluation (Djinnish)

SyncInit

$$\frac{\sigma' \text{ fresh}}{\begin{array}{l}\mathcal{T} \uplus \{\langle \theta_1, M_1^{IO}[\mathsf{sync}\ v_1]\rangle, \ldots, \langle \theta_k, M_k^{IO}[\mathsf{sync}\ v_k]\rangle\}; \mathcal{S} \xrightarrow{\epsilon} \\ \quad \mathcal{T}; \mathcal{S} \uplus \{\langle \sigma', \langle\langle \theta_1, M_1^{IO}, v_1, v_1\rangle, \ldots \langle \theta_k, M_k^{IO}, v_k, v_k\rangle\rangle\rangle\}\end{array}}$$

SyncStep

$$\frac{(e_1', \ldots, e_k') \rightsquigarrow_k (e_1'', \ldots, e_k'')}{\begin{array}{l}\mathcal{T}; \mathcal{S} \uplus \{\langle \sigma, \langle\langle \theta_1, M_1^{IO}, v_1, e_1'\rangle, \ldots \langle \theta_k, M_k^{IO}, v_k, e_k'\rangle\rangle\rangle\} \xrightarrow{\epsilon} \\ \quad \mathcal{T}; \mathcal{S} \uplus \{\langle \sigma, \langle\langle \theta_1, M_1^{IO}, v_1, e_1''\rangle, \ldots \langle \theta_k, M_k^{IO}, v_k, e_k''\rangle\rangle\rangle\}\end{array}}$$

SyncCommit

$$\frac{}{\begin{array}{l}\mathcal{T}; \mathcal{S} \uplus \{\langle \sigma, \langle\langle \theta_1, M_1^{IO}, v_1, \mathsf{alwaysEvt}\ v_1'\rangle, \ldots \langle \theta_k, M_k^{IO}, v_k, \mathsf{alwaysEvt}\ v_k'\rangle\rangle\rangle\} \xrightarrow{\epsilon} \\ \quad \mathcal{T} \uplus \{\langle \theta_1, M_1^{IO}[\mathsf{unitIO}\ v_1']\rangle, \ldots, \langle \theta_k, M_k^{IO}[\mathsf{unitIO}\ v_k']\rangle\}; \mathcal{S}\end{array}}$$

SyncAbort

$$\frac{}{\begin{array}{l}\mathcal{T}; \mathcal{S} \uplus \{\langle \sigma, \langle\langle \theta_1, M_1^{IO}, v_1, e_1'\rangle, \ldots \langle \theta_k, M_k^{IO}, v_k, e_k'\rangle\rangle\rangle\} \xrightarrow{\epsilon} \\ \quad \mathcal{T} \uplus \{\langle \theta_1, M_1^{IO}[\mathsf{sync}\ v_1]\rangle, \ldots, \langle \theta_k, M_k^{IO}[\mathsf{sync}\ v_k]\rangle\}; \mathcal{S}\end{array}}$$

# Concurrent Evaluation

**Theorem 1**

*$\mathcal{T} \xrightarrow{a_1;\cdots;a_n}{}^{*} \mathcal{T}'$ in the angelic semantics if and only if*

*$\mathcal{T};\{\} \xrightarrow{a'_1;\cdots;a'_m}{}^{*} \mathcal{T}';\{\}$ in the djinnish semantics,*

*where $a_1;\cdots;a_n$ equals $a'_1;\cdots;a'_m$ (modulo the insertion and deletion of $\epsilon$ actions).*

# Concurrent Functional Pidgin (Key Ideas)

*all-or-nothing* property of event synchronization

# Concurrent Functional Pidgin (Key Ideas)

*all-or-nothing* property of event synchronization

- non-deterministic search with backtracking
- transaction with abort

# Concurrent Functional Pidgin (Key Ideas)

*all-or-nothing* property of event synchronization

- non-deterministic search with backtracking
- transaction with abort

no need to designate a *commit point*

$$c_1 \ ; \ c_2 \ ; \ \cdots \ ; \ c_{n-1} \ ; \ c_n$$

# Concurrent Functional Pidgin (Key Ideas)

*all-or-nothing* property of event synchronization

- non-deterministic search with backtracking
- transaction with abort

no need to designate a *commit point*

$$c_1 \; ; \; c_2 \; ; \; \cdots \; ; \; c_{n-1} \; ; \; c_n$$

irrevocable (i.e. observable) actions cannot occur during synchronization

# Encoding Simple CML

# Encoding Simple CML

$$\begin{aligned} \mathsf{CMLEvt} &:: \star \to \star \\ \mathsf{CMLEvt}\ \alpha &= \mathsf{IO}\ (\mathsf{Evt}\ (\mathsf{IO}\ \alpha)) \end{aligned}$$

# Encoding Simple CML

$$\begin{aligned} \mathsf{CMLEvt} &:: \star \to \star \\ \mathsf{CMLEvt}\ \alpha &= \mathsf{IO}\ (\mathsf{Evt}\ (\mathsf{IO}\ \alpha)) \end{aligned}$$

$$\begin{aligned} \mathsf{lift} &:: \mathsf{Evt}\ \alpha \to \mathsf{CMLEvt}\ \alpha \\ \mathsf{lift}\ e &= \mathsf{unitIO}\ (\mathsf{mapEvt}\ e\ \mathsf{unitIO}) \end{aligned}$$

# Encoding Simple CML

$$
\begin{aligned}
\mathsf{CMLEvt} &:: \star \to \star \\
\mathsf{CMLEvt}\ \alpha &= \mathsf{IO}\ (\mathsf{Evt}\ (\mathsf{IO}\ \alpha))
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{lift} &:: \mathsf{Evt}\ \alpha \to \mathsf{CMLEvt}\ \alpha \\
\mathsf{lift}\ e &= \mathsf{unitIO}\ (\mathsf{mapEvt}\ e\ \mathsf{unitIO})
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{recvCMLEvt} &:: \mathsf{Chan}\ \alpha \to \mathsf{CMLEvt}\ \alpha \\
\mathsf{recvCMLEvt}\ c &= \mathsf{lift}\ (\mathsf{recvEvt}\ c) \\
\mathsf{sendCMLEvt} &:: \mathsf{Chan}\ \alpha \to \alpha \to \mathsf{CMLEvt}\ () \\
\mathsf{sendCMLEvt}\ c\ x &= \mathsf{lift}\ (\mathsf{sendEvt}\ c\ x)
\end{aligned}
$$

# Encoding Simple CML

$$
\begin{aligned}
\mathsf{CMLEvt} &:: \star \to \star \\
\mathsf{CMLEvt}\ \alpha &= \mathsf{IO}\ (\mathsf{Evt}\ (\mathsf{IO}\ \alpha))
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{lift} &:: \mathsf{Evt}\ \alpha \to \mathsf{CMLEvt}\ \alpha \\
\mathsf{lift}\ e &= \mathsf{unitIO}\ (\mathsf{mapEvt}\ e\ \mathsf{unitIO})
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{recvCMLEvt} &:: \mathsf{Chan}\ \alpha \to \mathsf{CMLEvt}\ \alpha \\
\mathsf{recvCMLEvt}\ c &= \mathsf{lift}\ (\mathsf{recvEvt}\ c) \\
\mathsf{sendCMLEvt} &:: \mathsf{Chan}\ \alpha \to \alpha \to \mathsf{CMLEvt}\ () \\
\mathsf{sendCMLEvt}\ c\ x &= \mathsf{lift}\ (\mathsf{sendEvt}\ c\ x)
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{alwaysCMLEvt} &:: \alpha \to \mathsf{CMLEvt}\ \alpha \\
\mathsf{alwaysCMLEvt}\ x &= \mathsf{lift}\ (\mathsf{alwaysEvt}\ x) \\
\mathsf{neverCMLEvt} &:: \mathsf{CMLEvt}\ \alpha \\
\mathsf{neverCMLEvt} &= \mathsf{lift}\ \mathsf{neverEvt}
\end{aligned}
$$

# Encoding Simple CML

$$
\begin{array}{rcl}
\mathsf{wrapCMLEvt} & :: & \mathsf{CMLEvt}\ \alpha \to (\alpha \to \mathsf{IO}\ \beta) \to \mathsf{CMLEvt}\ \beta \\
\mathsf{wrapCMLEvt}\ iei\ f & = & \mathsf{mapIO}\ (\lambda ei.\ \mathsf{mapEvt}\ (\lambda io.\ \mathsf{bindIO}\ io\ f)\ ei)\ iei \\
 & & \\
\mathsf{guardCMLEvt} & :: & \mathsf{IO}\ (\mathsf{CMLEvt}\ \alpha) \to \mathsf{CMLEvt}\ \alpha \\
\mathsf{guardCMLEvt}\ iiei & = & \mathsf{joinIO}\ iiei \\
 & & \\
\mathsf{chooseCMLEvt} & :: & \mathsf{CMLEvt}\ \alpha \to \mathsf{CMLEvt}\ \alpha \to \mathsf{CMLEvt}\ \alpha \\
\mathsf{chooseCMLEvt}\ iei_1\ iei_2 & = & \mathsf{bindIO}\ iei_1\ (\lambda ei_1. \\
 & & \mathsf{bindIO}\ iei_2\ (\lambda ei_2. \\
 & & \mathsf{unitIO}\ (\mathsf{chooseEvt}\ ei_1\ ei_2))) \\
 & & \\
\mathsf{syncCML} & :: & \mathsf{CMLEvt}\ \alpha \to \mathsf{IO}\ \alpha \\
\mathsf{syncCML}\ iei & = & \mathsf{bindIO}\ iei\ (\lambda ei.\ \mathsf{joinIO}\ (\mathsf{sync}\ ei))
\end{array}
$$

# Encoding Full CML

$$
\begin{array}{rcl}
\mathsf{SVar} & :: & \star \\
\mathsf{SVar} & = & \mathsf{Chan}\ () \\
 & & \\
\mathsf{svar} & :: & \mathsf{Evt}\ \mathsf{SVar} \\
\mathsf{svar} & = & \mathsf{channel} \\
 & & \\
\mathsf{setSVar} & :: & \mathsf{SVar} \rightarrow \mathsf{IO}\ () \\
\mathsf{setSVar}\ s & = & \mathsf{let\ loop}\ s = \mathsf{bindIO}\ (\mathsf{sync}\ (\mathsf{sendEvt}\ s\ ()))\ (\lambda().\ \\
 & & \qquad\qquad\qquad\qquad (\mathsf{loop}\ s)\ )\ \mathsf{in} \\
 & & \mathsf{spawn}\ (\mathsf{loop}\ s) \\
 & & \\
\mathsf{getSVarEvt} & :: & \mathsf{SVar} \rightarrow \mathsf{Evt}\ () \\
\mathsf{getSVarEvt}\ s & = & \mathsf{recvEvt}\ s
\end{array}
$$

# Encoding Full CML

# Encoding Full CML

$$\begin{aligned} \mathsf{CMLEvt} &:: \star \to \star \\ \mathsf{CMLEvt}\ \alpha &= \mathsf{IO}\ (\mathsf{List\ SVar} \times \mathsf{Evt}\ (\mathsf{List\ SVar} \times \mathsf{IO}\ \alpha)) \end{aligned}$$

# Encoding Full CML

$$\begin{aligned} \mathsf{CMLEvt} &:: \star \to \star \\ \mathsf{CMLEvt}\ \alpha &= \mathsf{IO}\ (\mathsf{List\ SVar} \times \mathsf{Evt}\ (\mathsf{List\ SVar} \times \mathsf{IO}\ \alpha)) \end{aligned}$$

$$\begin{aligned} \mathsf{lift} &:: \mathsf{Evt}\ \alpha \to \mathsf{CMLEvt}\ \alpha \\ \mathsf{lift}\ e &= \mathsf{unitIO}\ (\mathsf{nil}, \mathsf{mapEvt}\ e\ (\lambda x.\ (\mathsf{nil}, \mathsf{unitIO}\ x))) \end{aligned}$$

# Encoding Full CML

$$
\begin{aligned}
\mathsf{withNackCMLEvt} &:: (\mathsf{CMLEvt}\ () \to \mathsf{IO}\ (\mathsf{CMLEvt}\ \alpha)) \to \mathsf{CMLEvt}\ \alpha \\
\mathsf{withNackCMLEvt}\ f &= \mathsf{bindIO}\ (\mathsf{sync\ svar})\ (\lambda s. \\
&\qquad \mathsf{mapIO}\ (\lambda(ss, ei).\ (\mathsf{cons}\ s\ ss, ei)) \\
&\qquad\qquad (\mathsf{joinIO}\ (f\ (\mathsf{lift}\ (\mathsf{getSVarEvt}\ s)))))
\end{aligned}
$$

# Encoding Full CML

$$
\begin{aligned}
\mathsf{withNackCMLEvt} &:: (\mathsf{CMLEvt}\ () \to \mathsf{IO}\ (\mathsf{CMLEvt}\ \alpha)) \to \mathsf{CMLEvt}\ \alpha \\
\mathsf{withNackCMLEvt}\ f &= \mathsf{bindIO}\ (\mathsf{sync\ svar})\ (\lambda s. \\
&\qquad \mathsf{mapIO}\ (\lambda(ss, ei).\ (\mathsf{cons}\ s\ ss, ei)) \\
&\qquad\qquad (\mathsf{joinIO}\ (f\ (\mathsf{lift}\ (\mathsf{getSVarEvt}\ s)))))
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{chooseCMLEvt} &:: \mathsf{CMLEvt}\ \alpha \to \mathsf{CMLEvt}\ \alpha \to \mathsf{CMLEvt}\ \alpha \\
\mathsf{chooseCMLEvt}\ iei_1\ iei_2 &= \\
&\quad \mathsf{bindIO}\ iei_1\ (\lambda(ss_1, ei_1). \\
&\quad \mathsf{bindIO}\ iei_2\ (\lambda(ss_2, ei_2). \\
&\quad \mathsf{unitIO}\ (\mathsf{append}\ ss_1\ ss_2, \\
&\qquad \mathsf{chooseEvt}\ (\mathsf{mapEvt}\ (\lambda(ss, i).\ (\mathsf{append}\ ss_2\ ss, i))\ ei_1) \\
&\qquad\qquad (\mathsf{mapEvt}\ (\lambda(ss, i).\ (\mathsf{append}\ ss_1\ ss, i))\ ei_2))))
\end{aligned}
$$

# Encoding Full CML

$$
\begin{aligned}
\mathsf{syncCML} &:: \mathsf{CMLEvt}\ \alpha \to \mathsf{IO}\ \alpha \\
\mathsf{syncCML}\ iei &= \mathsf{bindIO}\ iei\ (\lambda(\_, ei). \\
&\quad \mathsf{bindIO}\ (\mathsf{sync}\ ei)\ (\lambda(ss, i). \\
&\quad \mathsf{fold}\ (\lambda(s, i).\ \mathsf{bindIO}\ (\mathsf{setSVar}\ s)\ (\lambda\_.\ i))\ i\ ss))
\end{aligned}
$$

# Expressiveness

**Theorem 2 (CML Expressivity)**

*Given the standard CML event combinators and an $n$-way rendezvous base-event constructor, one* cannot *implement an $(n+1)$-way rendezvous operation abstractly (i.e., as an event value).*

# Expressiveness

**Theorem 2 (CML Expressivity)**

*Given the standard CML event combinators and an $n$-way rendezvous base-event constructor, one* cannot *implement an $(n+1)$-way rendezvous operation abstractly (i.e., as an event value).*

```
signature TRICHAN =
   sig
      type 'a trichan
      val trichan :  unit -> 'a trichan
      val swapEvt :  'a trichan * 'a -> ('a * 'a) event
   end
```

# Expressiveness

**Theorem 3 (CFP Expressivity)**

*Given the standard CFP event combinators and an $n$-way rendezvous base-event constructor, one* can *implement an $(n+1)$-way rendezvous operation abstractly (i.e., as an event value).*

# Expressiveness

**Theorem 3 (CFP Expressivity)**

*Given the standard CFP event combinators and an $n$-way rendezvous base-event constructor, one* can *implement an $(n+1)$-way rendezvous operation abstractly (i.e., as an event value).*

$$\begin{aligned} \mathsf{Trichan} &:: \star \to \star \\ \mathsf{Trichan}\ \alpha &= \mathsf{Chan}\ (\mathsf{Chan}\ \alpha \times \mathsf{Chan}\ (\alpha \times \alpha)) \\ \\ \mathsf{trichan} &:: \mathsf{Evt}\ (\mathsf{Trichan}\ \alpha) \\ \mathsf{trichan} &= \mathsf{channel} \end{aligned}$$

# Expressiveness

$$
\begin{array}{rcl}
\mathsf{swapEvt} & :: & \mathsf{Trichan}\ \alpha \to \alpha \to \mathsf{Evt}\ (\alpha \times \alpha) \\
\mathsf{swapEvt}\ t\ x & = & \mathsf{let\ leader} = \mathsf{thenEvt\ newChan}\ (\lambda c^1_{in}. \\
 & & \quad \mathsf{thenEvt\ newChan}\ (\lambda c^1_{out}. \\
 & & \quad \mathsf{thenEvt\ newChan}\ (\lambda c^2_{in}. \\
 & & \quad \mathsf{thenEvt\ newChan}\ (\lambda c^2_{out}. \\
 & & \quad \mathsf{thenEvt}\ (\mathsf{sendEvt}\ t\ (c^1_{in}, c^1_{out}))\ .\ (\lambda().\ \\
 & & \quad \mathsf{thenEvt}\ (\mathsf{sendEvt}\ t\ (c^2_{in}, c^2_{out}))\ .\ (\lambda().\ \\
 & & \quad \mathsf{thenEvt}\ (\mathsf{recvEvt}\ c^1_{in})\ (\lambda x^1. \\
 & & \quad \mathsf{thenEvt}\ (\mathsf{recvEvt}\ c^2_{in})\ (\lambda x^2. \\
 & & \quad \mathsf{thenEvt}\ (\mathsf{sendEvt}\ c^1_{out}\ (x^2, x))\ (\lambda(). \\
 & & \quad \mathsf{thenEvt}\ (\mathsf{sendEvt}\ c^2_{out}\ (x, x^1))\ (\lambda(). \\
 & & \quad \mathsf{alwaysEvt}\ (x^1, x^2)))))))))) \ \mathsf{in} \\
 & & \mathsf{let\ client} = \mathsf{thenEvt}\ (\mathsf{recvEvt}\ t, \lambda(c_{in}, c_{out}). \\
 & & \quad \mathsf{thenEvt}\ (\mathsf{sendEvt}\ c_{in}\ x, \lambda(). \\
 & & \quad \mathsf{recvEvt}\ c_{out})) \ \mathsf{in} \\
 & & \mathsf{choosEvt\ leader\ client}
\end{array}
$$

# Encoding STM Haskell

# Encoding STM Haskell

$$\begin{aligned} \mathsf{STM} &:: \star \to \star \\ \mathsf{STM}\ \alpha &= \mathsf{ThreadID} \to \mathsf{Evt}\ \alpha \end{aligned}$$

$$\begin{aligned} \mathsf{atomic} &:: \mathsf{STM}\ \alpha \to \mathsf{IO}\ \alpha \\ \mathsf{atomic}\ s &= \mathsf{bindIO}\ \mathsf{getTId}\ (\lambda t.\ \mathsf{sync}\ (s\ t)) \end{aligned}$$

# Encoding STM Haskell

$$
\begin{aligned}
\mathsf{STM} &:: \star \to \star \\
\mathsf{STM}\ \alpha &= \mathsf{ThreadID} \to \mathsf{Evt}\ \alpha
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{atomic} &:: \mathsf{STM}\ \alpha \to \mathsf{IO}\ \alpha \\
\mathsf{atomic}\ s &= \mathsf{bindIO}\ \mathsf{getTId}\ (\lambda t.\ \mathsf{sync}\ (s\ t))
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{TVar} &:: \star \to \star \\
\mathsf{TVar}\ \alpha &= \mathsf{Chan\ ThreadId} \times \mathsf{Chan}\ \alpha \times \mathsf{Chan}\ \alpha
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{readTVar} &:: \mathsf{TVar}\ \alpha \to \mathsf{STM}\ \alpha \\
\mathsf{readTVar}\ (tc, rd, wr) &= \lambda t.\ \mathsf{thenEvt}\ (\mathsf{sendEvt}\ (tc, t))\ (\lambda().\ \\
&\qquad (\mathsf{recvEvt}\ rd)\ )
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{writeTVar} &:: \mathsf{TVar}\ \alpha \to \alpha \to \mathsf{STM}\ () \\
\mathsf{writeTVar}\ (tc, rd, wr)\ x &= \lambda t.\ \mathsf{thenEvt}\ (\mathsf{sendEvt}\ (tc, t))\ (\lambda().\ \\
&\qquad (\mathsf{sendEvt}\ wr\ x)\ )
\end{aligned}
$$

# Encoding STM Haskell

$$
\begin{array}{rcl}
\mathsf{tvar} & :: & \alpha \to \mathsf{IO}\ (\mathsf{TVar}\ \alpha) \\
\mathsf{tvar}\ x & = & \mathsf{let\ serve} :: \mathsf{TVar}\ \alpha \to \alpha \to \mathsf{Evt}\ (\alpha \times \mathsf{ThreadID}) \\
 & & \quad \mathsf{serve}\ (tc, rd, wr)\ x = \cdots \\
 & & \\
 & & \quad \mathsf{loopEvt} :: \mathsf{TVar}\ \alpha \to \alpha \to \mathsf{ThreadID} \to \mathsf{Evt}\ \alpha \\
 & & \quad \mathsf{loopEvt}\ (tc, rd, wr)\ x\ t = \cdots \\
 & & \\
 & & \quad \mathsf{loopIO} :: \mathsf{TVar}\ \alpha \to \alpha \to \mathsf{IO}\ () \\
 & & \quad \mathsf{loopIO}\ (tc, rd, wr)\ x = \cdots\ \mathsf{in} \\
 & & \\
 & & \mathsf{bindIO}\ (\mathsf{sync\ channel})\ (\lambda tc. \\
 & & \mathsf{bindIO}\ (\mathsf{sync\ channel})\ (\lambda rd. \\
 & & \mathsf{bindIO}\ (\mathsf{sync\ channel})\ (\lambda wr. \\
 & & \mathsf{bindIO}\ (\mathsf{spawn}\ (\mathsf{loopIO}\ (tc, rd, wr)\ x))\ (\lambda\_. \\
 & & \mathsf{returnIO}\ (tc, rd, wr)
\end{array}
$$

# Future Work and Conclusions

- Does there exist an efficient implementation?
- What new idioms and applications can be expressed?

# Future Work and Conclusions

- Does there exist an efficient implementation?
- What new idioms and applications can be expressed?

- An interesting point in the design space