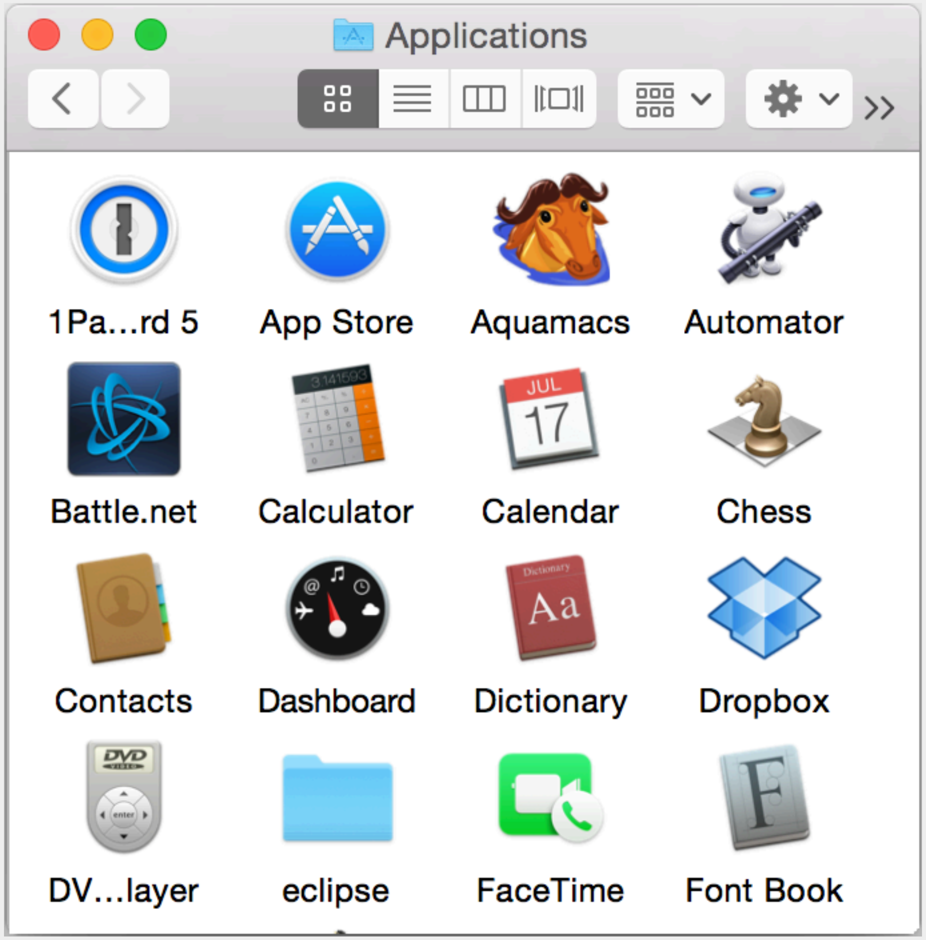
Curry-Howard for GUIs:
classical linear linear temporal logic
(work in progress!)

Steve Zdancewic
Jennifer Paykin
Neel Krishnaswami

WG 2.8 2015

How do we *think* about GUIs?



- an array of buttons
- each button *waits* for a click
- each button has a different effect (e.g. starts a different app)
- logically, each button is an *independent* process

A Callback-driven GUI API

```
button.onClick : (ClickData -> IO ()) -> IO ()

-- add the callback to the appropriate handler
button.onClick callback =
   handlers[click] := callback::(!handlers[click])
```

- pass to the button a function to be invoked on a click event
- callback stored in a per-widget collection
- button.onClick has a continuation type

⇒ classical logic?

Temporal Behavior

- GUI widgets *wait* for events
- handling of an event yields a natural notion of clock "tick"
  - process all of the callbacks for one event

- some resources are available only "now"
  - e.g. the data associated with the current event

- some resources are available "always" (at any point in the future)
  - e.g. a callback associated with a widget

- some resources are available only "eventually"
  - e.g. the data from some future event

From Callbacks to Eventually

```
button.onClick   : (ClickData -> IO ()) -> IO ()
                 : □(ClickData -> IO ()) -> IO ()
                 : (□¬ClickData) -> IO ()
                 : ¬□¬ClickData
                 : ◇ClickData
```

- callback creation ~ *eventually modality* of temporal logic
  - also called the "possibility" modality
- classical logic (should) yield a CPS-based implementation

- Question: Can we make anything out of this observation?

Type Structure

- Ordinary Types A useable "now"
- Always Types $\square A$ useable at *any* (future) time
- Eventually Types $\Diamond A$ useable at *some* (future) time

  - See [Pfenning & Davies] for modal logic

Always Modality

- The type $\Box A$ is "always A" or "necessarily A".
- Box is a comonad.

$$\boxed{\Delta; \Gamma \vdash A}$$

$$\frac{\Delta; \cdot \vdash A}{\Delta; \Gamma \vdash \Box A} \qquad \frac{\Delta; \Gamma \vdash \Box A \quad \Delta, A; \Gamma \vdash B}{\Delta; \Gamma \vdash B}$$

$$\frac{A \in \Delta}{\Delta; \Gamma \vdash A}$$

Always Modality

- The type $\square A$ is "always A" or "necessarily A".
- Box is a comonad.

$$\boxed{\Delta; \Gamma \vdash A}$$

$$\frac{\Delta;\ \cdot \vdash e : A}{\Delta; \Gamma \vdash \text{box } e : \square A} \qquad \frac{\Delta; \Gamma \vdash e_1 : \square A \qquad \Delta, a{:}A; \Gamma \vdash e_2 : B}{\Delta; \Gamma \vdash \text{let box } a = e_1 \text{ in } e_2 : B}$$

$$\frac{a{:}A \in \Delta}{\Delta; \Gamma \vdash a : A}$$

Eventually Modality

- The type $\Diamond A$ is "eventually A" or "possibly A".
- Diamond is a monad.

$$\frac{\Delta;\ \Gamma \vdash A}{\Delta;\ \Gamma \vdash \Diamond A}$$

$$\frac{\Delta; \Gamma \vdash \Diamond A \quad \Delta; A \vdash \Diamond B}{\Delta; \Gamma \vdash \Diamond B}$$

Eventually Modality

- The type $\Diamond A$ is "eventually A" or "possibly A".
- Diamond is a monad.

$$\frac{\Delta;\ \Gamma \vdash e : A}{\Delta;\ \Gamma \vdash \text{future } e : \Diamond A}$$

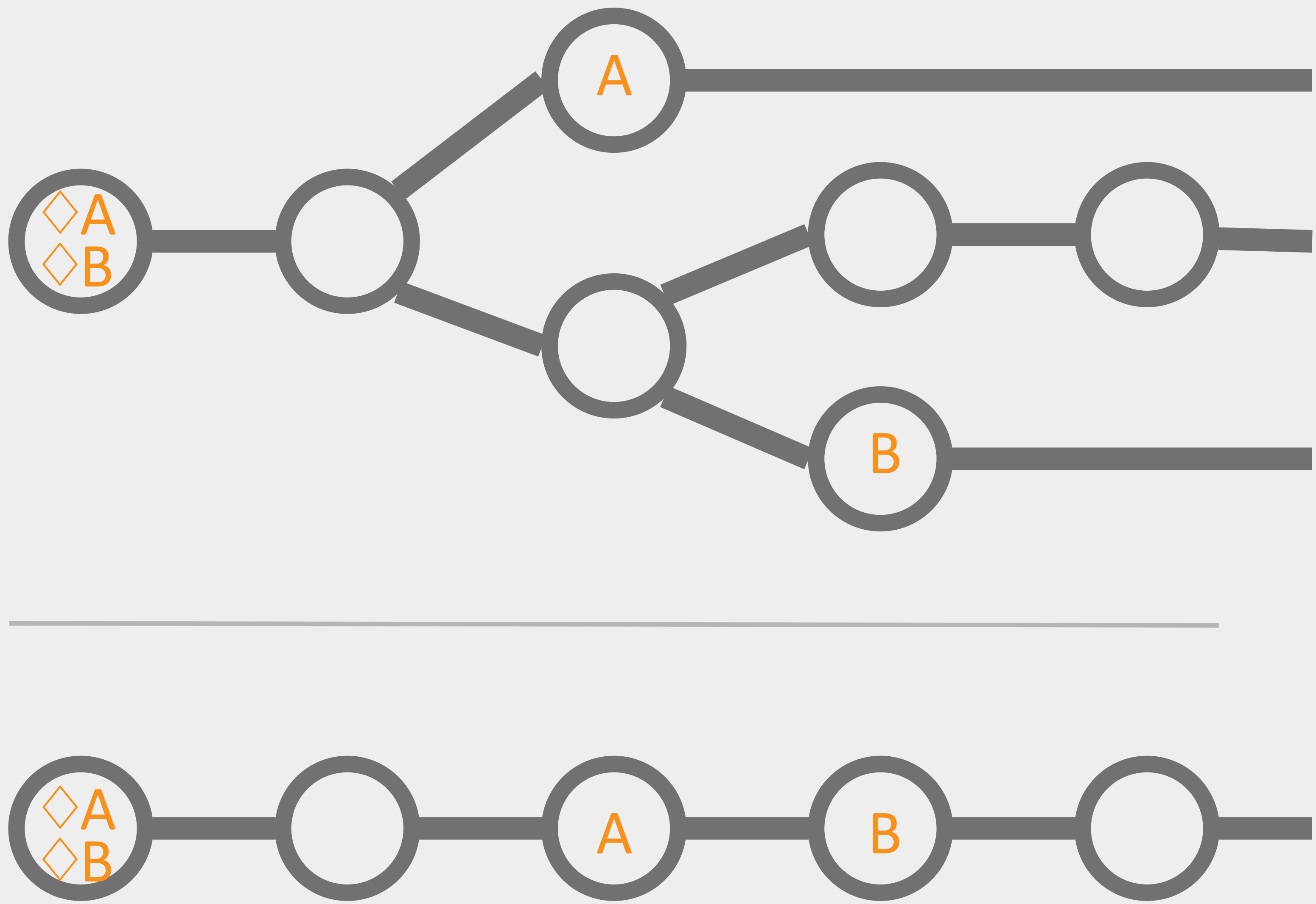
$$\frac{\Delta; \Gamma \vdash e_1 : \Diamond A \qquad \Delta; x{:}A \vdash e_2 : \Diamond B}{\Delta; \Gamma \vdash \text{wait } x = e_1 \text{ in } e_2 : \Diamond B}$$

Linear Temporal Logic

- The type $\Diamond A$ means "eventually A".
  - Would like to think of this as an "A event"
  - Built-in primitives could provide other sources of $\Diamond A$
- But… not enough structure to order them
  - In a GUI, we often think of the *sequence* of events

total order: for any A, B. $A \leq B$ or $B \leq A$

Branching vs Linear Time



Linear Temporal Logic

- Encode the ordering as this rule:

$$\frac{\Delta;\, \Gamma_1 \vdash \Diamond B \qquad \Delta;\, A, \Diamond B \vdash \Diamond C \qquad \Delta;\, \Gamma_2 \vdash \Diamond A \qquad \Delta;\, \Diamond A, B \vdash \Diamond C}{\Delta;\, \Gamma_1, \Gamma_2 \vdash \Diamond C}$$

- Call this operation “select”:
    - Wait for whichever event fires first, choose a continuation based on the outcome
    - The second operation will still eventually happen

Linear Temporal Logic

- Encode the ordering as this rule:

$$\frac{\Delta; \Gamma \vdash e_1 : \Diamond B \qquad \Delta; a{:}A, b{:}\Diamond B \vdash c_1 : \Diamond C \qquad \Delta; \Gamma \vdash e_2 : \Diamond A \qquad \Delta; a{:}\Diamond A, b{:}B \vdash c_2 : \Diamond C}{\Delta; \Gamma \vdash \texttt{select } e_1 \mid e_2 \texttt{ as } \mid a, b \to c_1 \mid a, b \to c_2 : \Diamond C}$$

- Call this operation "select":
  - Wait for whichever event fires first, choose a continuation based on the outcome
  - The second operation will still eventually happen

Linear Time, Logically

- lets us say that any two events can be ordered:

$$\Diamond A \longrightarrow \Diamond B \longrightarrow \Diamond(A \times \Diamond B) + \Diamond(\Diamond A \times B)$$

- also permits synchronization on "eventually always" propositions:

$$\text{sync:} \quad \Diamond\Box A \longrightarrow \Diamond\Box B \longrightarrow \Diamond\Box(A \times B)$$

Classical *Linear* Linear Temporal Logic

- Ordinary Types A useable "now"
- Always Types $\Box A$ useable at *any* (future) time
- Eventually Types $\Diamond A$ useable at *some* (future) time

- Classical Linear Logic ⇒ Concurrent Programming
  - See e.g. [Wadler] [Pfenning]
  - π-calculus notation

- Benefits: similar to Rust's affine types
  - separation of resources
  - race prevention

Safety

```
button.onClick : (ClickData -> Safe) -> Safe
```

- here `IO ()` is the answer type.
- this is too permissive; we don't want *all* terms of type `IO ()`
- … only those commands that preserve the event loop invariants
    - Idea: for GUIs replace IO () with Safe, a refinement that permits only "good" computations
    - show that safety is preserved when composing richer types

What is Safety?

A widget contains:

- Some first-order data (color, height, text, etc.)
- A collection of event handlers
- So a heap can be formalized as:

$$\begin{array}{llll} \text{Data heap} & h & ::= & \cdot \mid h, h \mid l : d \\ \text{Queue} & q & \in & \text{Loc} \to \mathcal{M}^{\text{fin}}(\text{Val}) \\ \text{Store} & \sigma & \in & \text{Data} \times \text{Queue} \end{array}$$

- Key problem: event handlers are *higher-order state*

Safety, Semantically

$$
\begin{aligned}
\mathsf{Ok} &= \left\{ (\sigma, t, \sigma') \;\middle|\; \begin{array}{l} \forall \phi \# h.\ \exists \pi \in \mathsf{Perm}. \\ \qquad \langle \sigma \cdot \phi; t \rangle \Downarrow \left\langle \pi(\sigma') \cdot \phi; () \right\rangle \end{array} \right\} \\
\mathsf{Safe}_n &= \left\{ \overbrace{(h, q)}^{\sigma} \;\middle|\; \begin{array}{l} \forall l \in \mathsf{Loc}, e \in \mathsf{Event}. \\ \qquad \mathsf{Safe}^*_n((h, [q|l : \varnothing]), e, q(l)) \end{array} \right\} \\
\mathsf{Safe} &= \bigcap_n \mathsf{Safe}_n
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{Safe}^*_0(\sigma, e, ks) &= \top \\
\mathsf{Safe}^*_{n+1}(\sigma, e, \epsilon) &= \top \\
\mathsf{Safe}^*_{n+1}(\sigma, e, k \cdot ks) &= \begin{array}{l} \exists \sigma' \in \mathsf{Safe}_n. \\ \qquad \mathsf{Ok}(\sigma, k\ e, \sigma') \wedge \\ \qquad \mathsf{Safe}^*_n(\sigma', e, ks) \end{array}
\end{aligned}
$$

- Safe = heaps maintaining safety on callbacks

Separation Algebra of Safe Heaps

$$
\begin{array}{lcl}
h \# h' & \triangleq & \mathrm{dom}(h) \cap \mathrm{dom}(h') = \emptyset \\
(h, q) \# (h', q') & \triangleq & h \# h' \\
\\
h \cdot h & = & \begin{cases} h, h' & \text{if } h \# h \\ \bot & \text{otherwise} \end{cases} \\
q \cdot q' & = & \lambda l.\ q(l) \cup q'(l) \\
\\
\epsilon & = & (\cdot, []) \\
(h, q) \cdot (h', q') & = & \begin{cases} (h \cdot h', q \cdot q') & \text{if } h \# h \\ \bot & \text{otherwise} \end{cases}
\end{array}
$$

Compilation Strategy

Classical Logic	::	Classical Linear Logic
$\Downarrow$		$\Downarrow$
Intuitionistic Logic	::	Tensorial Logic

- Translation is double-negation (i.e. CPS translation)
- Mellies' *tensorial logic*

Realizability Semantics of Continuations

$\text{Type} \quad = \quad \{\langle \sigma; v\rangle \mid \sigma \in \text{Safe}\}$

1

$A * B$

0

$A_1 + A_2$

$\neg A$

$\Box A$

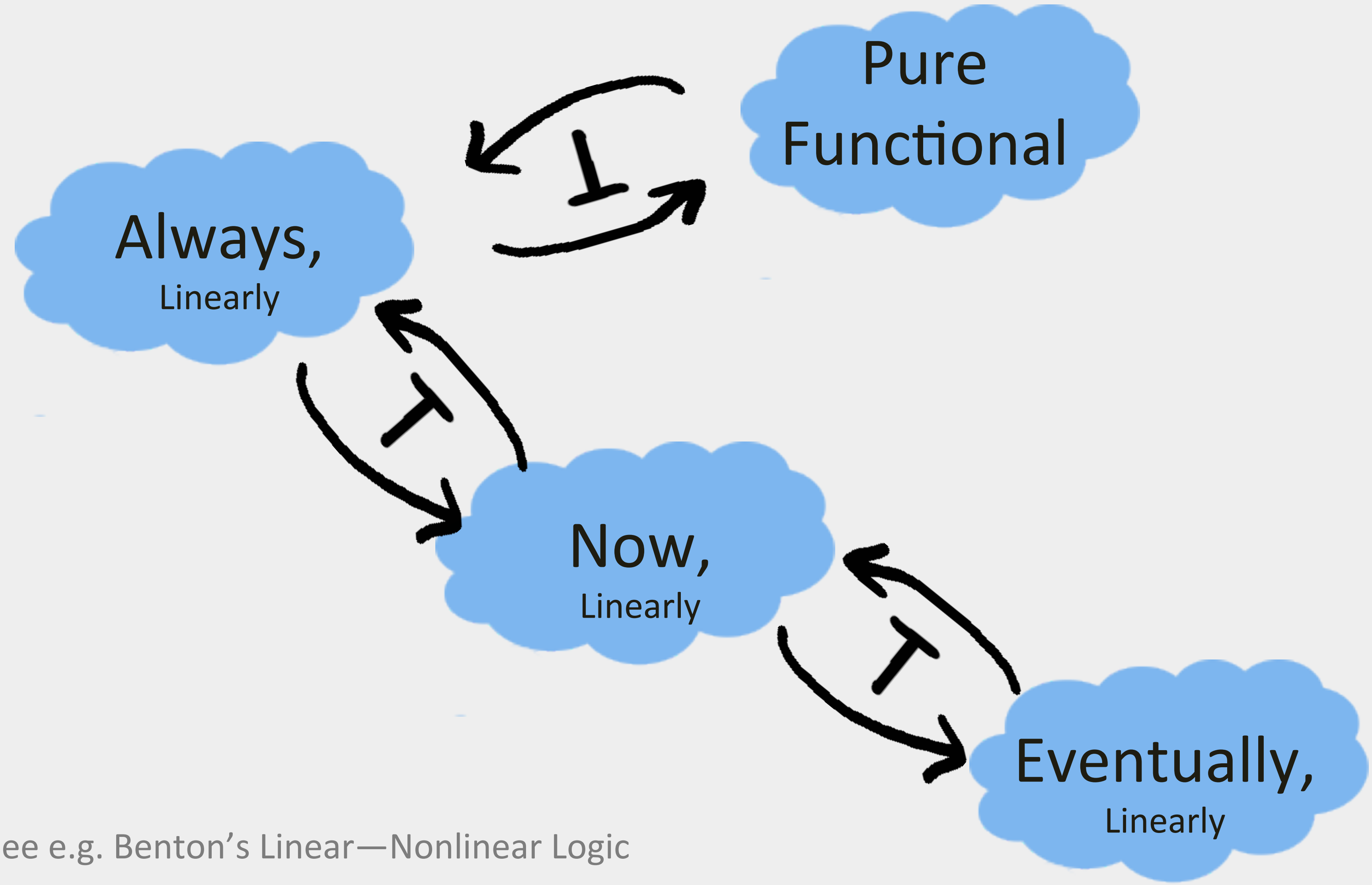
Double Negation (CPS)

$$
\begin{aligned}
[\![0]\!] &= 0 \\
[\![A \oplus B]\!] &= [\![A]\!] + [\![B]\!] \\[1em]
[\![I]\!] &= 1 \\
[\![A \otimes B]\!] &= [\![A]\!] * [\![B]\!] \\[1em]
[\![\top]\!] &= \neg 0 \\
[\![A \mathbin{\&} B]\!] &= \neg(\neg [\![A]\!] + \neg [\![B]\!]) \\[1em]
[\![\bot]\!] &= \neg 1 \\
[\![A \parr B]\!] &= \neg(\neg [\![A]\!] * \neg [\![B]\!]) \\[1em]
[\![\Box A]\!] &= \Box [\![A]\!] \\
[\![\Diamond A]\!] &= \neg \Box \neg [\![A]\!]
\end{aligned}
$$

Status

- Still nailing down the semantics
  - Interaction between linearity & temporal logic
  - Proofs that the safety invariants compose
- Playing around with syntax
  - Sequent formulations of the type system
  - Pi-calculus? Mu calculus?
- No implementation (yet!)
- Jennifer: thinking about "composition of logical features"
  - Combining semantics

Features as Computational "Worlds"



See e.g. Benton's Linear—Nonlinear Logic

"Adjoint functors arise everywhere..."

Type System	Type	Term	
Logic	Proposition	Proof	
Category	Object	Morphism	Adjunction

Questions

- Connection to Functional Reactive Programming?
- Behavior/Signal vs. Event
  - $\Box A \sim T \longrightarrow A$ where T is the domain of Time
  - $\Diamond A \sim T \times A$

- Connection to Concurrent ML?
  - first-class synchronization primitives?

Interactive Programs

- event loop *waits* for events
- programs register *callbacks* with the event handler
- event loop invokes the callbacks for each event
- GUI Programs are(?):
  - higher-order
  - concurrent
  - imperative
  - CPS

```
-- event loop
while (true) {
  let event = get_event();
  for (f in handlers[event]) {
    f(event.data);
  }
}

-- handlers
handlers[key] =
   [fun d -> …; fun d -> …;]

handlers[click] =
   [fun d -> …; fun d -> …;]

handlers[mouseMove] =
   [fun d -> …;]
```

Linear Type Structure

○ Linear Types	A	useable exactly once "now"
○ Always Types	□A	useable once at any (future) time
○ Eventually Types	◇A	useable once at some (future) time
○ Persistent Types	!A	unrestricted uses at any time

- box is a comonad
- diamond is a monad
- sequent calculus:

$$\frac{\vdash \diamond\Delta, A}{\vdash \diamond\Delta, \square A} \qquad \frac{\vdash \Delta, A}{\vdash \Delta, \diamond A}$$

Safety

```
button.onClick   : (ClickData -> IO ()) -> IO ()
                 : (ClickData -> Safe) -> Safe
                 : □(ClickData -> Safe) -> Safe
                 : □¬ClickData -> Safe
                 : ¬□¬ClickData
                 : ◇ClickData
```

- callback creation ~ eventually modality of temporal logic
- linearity lets us characterize independence
- classical logic (should) yield a CPS-based implementation

Facts about CPS Translation

$$
\begin{aligned}
[\![1]\!] &= (1 \longrightarrow a) \longrightarrow a \\
[\![A \times B]\!] &= (A \longrightarrow B \longrightarrow a) \longrightarrow a \\
[\![A \longrightarrow B]\!] &= A \longrightarrow (B \longrightarrow a) \longrightarrow a
\end{aligned}
$$

- CPS is a double negation translation of the types:

  $$\neg A = A \longrightarrow a$$

- the "answer type" is a
- CPS translation is *parametric* in the answer type [Friedman 76]