Concurrency and Classical Linear F (work in progress!)

Steve Zdancewic
with Karl Mazurak

University of Pennsylvania

6-9-2009

Concurrency and Classical Linear F (work in progress!)

Steve Zdancewic
with Karl Mazurak

University of Pennsylvania

6-9-2009

"Somewhere inside linear logic, there is a programming language struggling to get out." – Phil Wadler, 1993

Motivation: Concurrency

Everyone agrees: it's important.

- Naturally concurrent applications (networking)
- Massively multicore systems (by fiat)

It's also tricky.

- Low-level: non-determinism, race conditions, deadlocks, ...
- High-level: following communication protocols
- Gap between concurrency calculi and familiar programming

How do we bring the familiar typed λ-calculi into the concurrent future while adding as little new baggage as possible?
– This work may or may not have something to say about that!

Motivation: Linearity

Linear logic does away with contraction and weakening.

Intuitionistic linear systems are often used to guarantee a lack of aliasing, (Wadler 1990, Turner & Wadler, Bierman, . . .) e.g. in memory management (Jia & Walker, Fluet, Ahmed & Morrisett, . . .).

Can classical linear types help with concurrency?

- Girard (1987) certainly thought so.
- Abramsky (1990) defined a syntax of "proof expressions" living in a concurrent soup.
  - They don't look much like programs, though.
  - Other sequent-calculus-based formalisms, like the dual calculus (Wadler, 2003 & 2004) and proof nets are similarly non-standard.

This work

Hypothesis: we can do better! (Or at least simpler.)
(Or at least I'll learn something interesting along the way.)

In particular, let's try the familiar System F extended with:

- A very simple kind system for linearity (no ! type)
- Two polymorphic constants for classicality/concurrency
- A simple process model, in the style of the π-calculus, that remains hidden from the programmer

In lieu of a better name, call this language F°.

Also observe that intuitionistic linear types are useful apart from memory management.

Goals / Hopes?

In its full generality, F° should allow communication protocols among parallel threads to be expressed at the type level.

- i.e., senders will always send what receivers are expecting.
    - As in session types (Honda et al., 1998), but built from simple primitives based on linear logic.

- Programmers can also use linearity and polymorphism to enforce their own protocols at abstraction barriers.

- Philosophy: start with "pure" concurrency and communication primitives (e.g. no deadlock, no nondeterminism, no nontermination) and later add these features, potentially guarded by effects types or monads.

My Questions:

- Is this new/interesting?
- Is this approach useful? Can it be turned into/added to a "real" programming language?
- Connections to other approaches? [e.g. parallel graph-rewriting, (typed) pi calculi, join calculus, Erlang, etc.]

Outline

1. Intuitionistic F°
    - Syntax and typing
    - Example: a safe filesystem API
    - Existing results (briefly)
2. Classical F°
    - Basic intuitions
    - Initial extension: one-shot channels
    - Examples: encoded functionality
    - Embedded negation and communication protocols
3. Going further?
    - Additive types
    - Recursive types
    - Non-determinism

Intuitionistic $\mathbb{F}^\circ$

Intuitionistic F°: Syntax

$$\begin{array}{rcll}
\kappa & ::= & \star \mid \circ & \textit{kinds} \\
\tau & ::= & \alpha \mid \tau \xrightarrow{\kappa} \tau \mid \forall \alpha{:}\kappa.\, \tau & \textit{types} \\
\\
e & ::= & x \mid \lambda^{\kappa} x{:}\tau.\, e \mid e\ e & \textit{expressions} \\
 & \mid & \Lambda \alpha{:}\kappa.\, v \mid e\ [\tau] & \\
\\
v & ::= & \lambda^{\kappa} x{:}\tau.\, e \mid \Lambda \alpha{:}\kappa.\, v & \textit{values} \\
\Gamma & ::= & \cdot \mid \Gamma, \alpha{:}\kappa \mid \Gamma, x{:}\tau & \textit{typing contexts}
\end{array}$$

Linearity managed at the kind level (linear $\circ$, non-linear $\star$).

Standard call-by-value operational semantics, but note the value restriction! Necessary for soundness.

Intuitionistic F°: Kinding

$$[\text{K-TVar}]\ \frac{\alpha{:}\kappa \in \Gamma}{\Gamma \vdash \alpha : \kappa} \qquad [\text{K-Sub}]\ \frac{\Gamma \vdash \tau : \star}{\Gamma \vdash \tau : \circ}$$

$$[\text{K-Arrow}]\ \frac{\Gamma \vdash \tau_1 : \kappa_1 \qquad \Gamma \vdash \tau_2 : \kappa_2}{\Gamma \vdash \tau_1 \xrightarrow{\kappa} \tau_2 : \kappa}$$

$$[\text{K-All}]\ \frac{\Gamma, \alpha{:}\kappa \vdash \tau : \kappa' \qquad \alpha \notin \mathrm{dom}(\Gamma)}{\Gamma \vdash \forall \alpha{:}\kappa.\, \tau : \kappa'}$$

Intuitionistic F$^\circ$: Typing

Type abstraction and type application rules are standard.

$$[\text{T-Var}]\ \frac{(\Gamma_1, \Gamma_2)^\star}{\Gamma_1, x{:}\tau, \Gamma_2 \vdash x : \tau}$$

$$[\text{T-Lam}]\ \frac{\Gamma^\kappa \qquad \Gamma \vdash \tau_1 : \kappa_1 \qquad \Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda^\kappa x{:}\tau_1.\, e : \tau_1 \xrightarrow{\kappa} \tau_2}$$

$$[\text{T-App}]\ \frac{\Gamma_1 \vdash e_1 : \tau_1 \xrightarrow{\kappa} \tau_2 \qquad \Gamma_2 \vdash e_2 : \tau_1 \qquad \Gamma_1 \Cup \Gamma_2 = \Gamma}{\Gamma \vdash e_1\ e_2 : \tau_2}$$

$\Gamma_1 \Cup \Gamma_2 = \Gamma$ is context splitting.
Γ^κ ensures that τ can be given kind κ for every $x{:}\tau$ in Γ.

Example: A Safe Filesystem

Most operating systems provide a filesystem interface something like

$$\begin{aligned} \mathsf{UnsafeFH} &: \star \\ \\ \mathsf{unsafeOpen} &: \mathsf{String} \xrightarrow{\star} \mathsf{UnsafeFH} \\ \mathsf{unsafeRead} &: \mathsf{UnsafeFH} \xrightarrow{\star} \mathsf{Char} \\ \mathsf{unsafeWrite} &: \mathsf{Char} \xrightarrow{\star} \mathsf{UnsafeFH} \xrightarrow{\star} \mathsf{Unit} \\ \mathsf{unsafeClose} &: \mathsf{UnsafeFH} \xrightarrow{\star} \mathsf{Unit} \end{aligned}$$

We label these unsafe because filehandles might not be closed, or they might be improperly used after being closed.

Intuitionistic F° can provide a safer alternative.
(But so can other systems.)

Example: A Safe Filesystem

For α of kind $\circ$, take

$$\begin{aligned}\mathsf{FH}(\alpha) \;=\; \{\; & \mathsf{handle} : \alpha, \\ & \mathsf{read} : \alpha \xrightarrow{\star} (\mathsf{Char}, \alpha), \\ & \mathsf{write} : \mathsf{Char} \xrightarrow{\star} \alpha \xrightarrow{\star} \alpha, \\ & \mathsf{close} : \alpha \xrightarrow{\circ} \mathsf{Unit} \\ \}\; & \end{aligned}$$

For simplicity, we use standard System F encodings of tuples, records, existentials, etc..

Example: A Safe Filesystem

Now, define open as

$$
\begin{array}{lcl}
\mathsf{open} & : & \mathsf{String} \xrightarrow{\star} \exists\alpha{:}\circ.\ \mathsf{FH}(\alpha) \\
\mathsf{open} & = & \lambda^\star f{:}\mathsf{String}.\ \mathbf{let}\ handle = \mathsf{unsafeOpen} f\ \mathbf{in} \\
 & & \quad \mathbf{pack}\ \alpha = \mathsf{UnsafeFH}\ \mathbf{in} \\
 & & \quad \{\ \mathsf{handle} = handle, \\
 & & \qquad \mathsf{read} = \lambda^\star h{:}\mathsf{UnsafeFH}.\ (\mathsf{unsafeRead}\ h, h), \\
 & & \qquad \mathsf{write} = \lambda^\star c{:}\mathsf{Char}.\ \lambda^\star h{:}\mathsf{UnsafeFH}. \\
 & & \qquad\qquad\qquad \mathsf{unsafeWrite}\ s\ h; h, \\
 & & \qquad \mathsf{close} = \mathsf{unsafeClose} \\
 & & \quad \} : \mathsf{FH}(\alpha)
\end{array}
$$

Filehandle packages created by open must be closed if the program type is non-linear. Once this happens, further reads and writes are impossible.

Classical $\mathbb{F}^{\circ}$

Classical F°: Motivations

Where to start? Concurrency is a **big** area and classicality is confusing; begin with “simple” continuations.

Recall Felleisen’s **control** operator (Felleisen and Hieb, 1992), a term for double-negation elimination:

$$\begin{aligned} E\big[\textbf{control}\ (\lambda c.\ e)\big] &\longrightarrow (\lambda c.\ e)\ (\lambda x.\ \textbf{abort}\ E\big[x\big]) \\ E\big[\textbf{abort}\ e\big] &\longrightarrow e \end{aligned}$$

Unfortunately, **abort** clearly has no place in a linear system!

Even if **abort**s occur only within **control**s, we could still lose track of linear resources.

Classical $\mathbb{F}^\circ$: Adding Parallelism

Idea: applying the continuation c should return to the original evaluation context E, but the existing context doesn't need to be dropped. Why not finish evaluating it in parallel?

$$\begin{array}{rcl}
E\left[\mathbf{control}\ (\lambda c.\ e)\right] & \longrightarrow & \nu a.\ E\left[\mathbf{control}\ a{\triangleright}\tau\right] \mid (\lambda c.\ e)\ \tau{\triangleright}a \\
E_1\left[\mathbf{control}\ a{\triangleright}\tau\right] \mid E_2\left[\tau{\triangleright}a\ v\right] & \longrightarrow & E_1\left[v\right] \mid E_2\left[\varnothing\right] \\
e \mid \varnothing & \longrightarrow & e
\end{array}$$

Suddenly we have a process calculus, with channels embedded in expressions:

$$\begin{array}{rcll}
P & ::= & e \,\big|\, P \mid P \,\big|\, \nu a^\tau.\ P & \\
e & ::= & \ldots & \\
 & \big| & a{\triangleright}\tau & \textit{a source of } \tau\textit{'s} \\
 & & \tau{\triangleright}a & \textit{a sink for } \tau\textit{'s} \\
 & & \varnothing & \textit{a ``done'' token}
\end{array}$$

Classical F°: Go and Yield

Make our concurrency more useful by splitting **control** into **go** and **yield**, where

$$\begin{array}{rcl} E\left[\mathbf{go}\ (\lambda c.\ e)\right] & \longrightarrow & \nu a.\ E\left[a{\triangleright}\tau\right] \mid (\lambda c.\ e)\ \tau{\triangleright}a \\ E_1\left[\mathbf{yield}\ a{\triangleright}\tau\right] \mid E_2\left[\tau{\triangleright}a\ v\right] & \longrightarrow & E_1\left[v\right] \mid E_2\left[\varnothing\right] \end{array}$$

Not pictured:

- Various congruence rules for **go** and **yield**
- Process evaluation rules (as expected)
- Process equivalence relation $\equiv$ (standard)

For typing, we need an answer type $\perp\!\!\!\!\!\smile$ (the multiplicative bottom type). Not equivalent to $\forall\alpha{:}\circ.\ \alpha$ (the additive bottom type).

Classical F°: Typing

$$[\text{K-Ans}]\ \Gamma \vdash \curlywedgedownarrow : \circ \qquad [\text{T-Done}]\ \frac{\Gamma^\star}{\Gamma \vdash \varnothing : \curlywedgedownarrow}$$

$$[\text{T-Source}]\ \frac{a^\tau \in \Gamma \qquad \Gamma^\star}{\Gamma \vdash a \triangleright \tau : (\tau \overset{\circ}{\to} \curlywedgedownarrow) \overset{\circ}{\to} \curlywedgedownarrow} \qquad [\text{T-Sink}]\ \frac{a^\tau \in \Gamma \qquad \Gamma^\star}{\Gamma \vdash \tau \triangleright a : \tau \overset{\circ}{\to} \curlywedgedownarrow}$$

$$[\text{T-Go}]\ \frac{\Gamma \vdash e : (\tau \overset{\circ}{\to} \curlywedgedownarrow) \overset{\circ}{\to} \curlywedgedownarrow}{\Gamma \vdash \mathbf{go}\ e : (\tau \overset{\circ}{\to} \curlywedgedownarrow) \overset{\circ}{\to} \curlywedgedownarrow} \qquad [\text{T-Yield}]\ \frac{\Gamma \vdash e : (\tau \overset{\circ}{\to} \curlywedgedownarrow) \overset{\circ}{\to} \curlywedgedownarrow}{\Gamma \vdash \mathbf{yield}\ e : \tau}$$

Logical interpretation of **yield** is "double negation elimination".
Logical interpretation for **go**. . . ?

For brevity in what follows: $\triangleright\tau \triangleq (\tau \overset{\circ}{\to} \curlywedgedownarrow) \overset{\circ}{\to} \curlywedgedownarrow$

Example: Forking Background Threads

First, a function to "background" $\bot \xrightarrow{\circ} \bot$ functions:

$$
\begin{array}{lcl}
\mathsf{fork} & : & (\bot \xrightarrow{\circ} \bot) \xrightarrow{\star} \mathsf{Unit} \\
\mathsf{fork} & = & \lambda^{\star} f{:}\bot \xrightarrow{\circ} \bot.\ \mathbf{yield}\ (\mathbf{go}\ \lambda^{\circ} g{:}\mathsf{Unit} \xrightarrow{\circ} \bot.\ f\ (g\ \mathsf{unit}))
\end{array}
$$

Example: Forking Background Threads

First, a function to “background” $\bot \xrightarrow{\circ} \bot$ functions:

$$\begin{array}{lcl} \mathsf{fork} & : & (\bot \xrightarrow{\circ} \bot) \xrightarrow{\star} \mathsf{Unit} \\ \mathsf{fork} & = & \lambda^{\star} f{:}\bot \xrightarrow{\circ} \bot.\ \mathbf{yield}\ (\mathbf{go}\ \lambda^{\circ} g{:}\mathsf{Unit} \xrightarrow{\circ} \bot.\ f\ (g\ \mathsf{unit})) \end{array}$$

Given f of type $\bot \xrightarrow{\circ} \bot$, we have:

$$\begin{array}{lcl} E\left[\mathsf{fork}\ f\right] & \longrightarrow & E\left[\mathbf{yield}\ (\mathbf{go}\ \lambda^{\circ} g{:}\mathsf{Unit} \xrightarrow{\circ} \bot.\ f\ (g\ \mathsf{unit}))\right] \\ & \longrightarrow^{*} & \nu a^{\mathsf{Unit}}.\ (E\left[\mathbf{yield}\ a{\triangleright}\mathsf{Unit}\right] \mid f\ (\mathsf{Unit}{\triangleright}a\ \mathsf{unit})) \\ & \longrightarrow & \nu a^{\mathsf{Unit}}.\ (E\left[\mathsf{unit}\right] \mid f\ \varnothing) \\ & \equiv & E\left[\mathsf{unit}\right] \mid f\ \varnothing \end{array}$$

Example: Asynchronous Forwarding

Use fork to spawn a “forwarder” for a value source:

$$\begin{array}{lcl} \mathsf{fwd} & : & \forall\alpha{:}\circ.\, \triangleright\alpha \xrightarrow{\star} (\alpha \xrightarrow{\circ} \bot) \xrightarrow{\circ} \bot \\ \mathsf{fwd} & = & \Lambda\alpha{:}\circ.\, \lambda^{\star}x{:}\triangleright\alpha.\, \lambda^{\circ}f{:}\alpha \xrightarrow{\circ} \bot.\, \mathbf{yield}\ (\mathbf{go}\ \lambda^{\circ}g{:}\bot \xrightarrow{\circ} \bot. \\ & & \quad \mathsf{fork}\ g; f\ (\mathbf{yield}\ x)) \end{array}$$

Example: Asynchronous Forwarding

Use fork to spawn a "forwarder" for a value source:

$$\begin{array}{lcl} \mathrm{fwd} & : & \forall\alpha{:}\circ.\ \triangleright\alpha \xrightarrow{\star} (\alpha \xrightarrow{\circ} \curlywedge) \xrightarrow{\circ} \curlywedge \\ \mathrm{fwd} & = & \Lambda\alpha{:}\circ.\ \lambda^{\star}x{:}\triangleright\alpha.\ \lambda^{\circ}f{:}\alpha \xrightarrow{\circ} \curlywedge.\ \mathbf{yield}\ (\mathbf{go}\ \lambda^{\circ}g{:}\curlywedge \xrightarrow{\circ} \curlywedge. \\ & & \quad \mathrm{fork}\ g; f\ (\mathbf{yield}\ x)) \end{array}$$

Given x of type $\triangleright\tau$ and f of type $\tau \xrightarrow{\circ} \curlywedge$:

$$\begin{array}{lcl} E\left[\mathrm{fwd}\ [\tau]\ x\ f\right] & \longrightarrow^{*} & E\left[\mathbf{yield}\ (\mathbf{go}\ \lambda^{\circ}g{:}\curlywedge \xrightarrow{\circ} \curlywedge.\ \mathrm{fork}\ g; f\ (\mathbf{yield}\ x))\right] \\ & \longrightarrow^{*} & \nu a^{\curlywedge}.\ (E\left[\mathbf{yield}\ a{\triangleright}\curlywedge\right] \mid \mathrm{fork}\ \curlywedge{\triangleright}a; f\ (\mathbf{yield}\ x)) \\ & \longrightarrow^{*} & \nu a^{\curlywedge}.\ (E\left[\mathbf{yield}\ a{\triangleright}\curlywedge\right] \mid (f\ (\mathbf{yield}\ x) \mid \curlywedge{\triangleright}a\ \varnothing)) \\ & \longrightarrow^{*} & \nu a^{\curlywedge}.\ (E\left[\varnothing\right] \mid f\ (\mathbf{yield}\ x)) \\ & \equiv & E\left[\varnothing\right] \mid f\ (\mathbf{yield}\ x) \end{array}$$

Example: Forgetting Non-linear Sources

We'd also like to discard sources of non-linear types without waiting on them.

$$\begin{aligned} \mathsf{forget} &: \forall\alpha{:}\star.\ \triangleright\alpha \xrightarrow{\star} \mathsf{Unit} \\ \mathsf{forget} &= \Lambda\alpha{:}\star.\ \lambda^{\star}x{:}\triangleright\alpha.\ \textbf{yield}\ (\textbf{go}\ \lambda^{\circ}f{:}\mathsf{Unit} \xrightarrow{\circ} \curlywedge. \\ &\qquad \textbf{let}\ z = f\ \mathsf{unit}\ \textbf{in}\ \textbf{let}\ y = \textbf{yield}\ x\ \textbf{in}\ z) \end{aligned}$$

In other words:

- Perform a trivial exchange at type Unit.
- Use the post-send "clean-up code" to wait for and discard the non-linear data.

Classical F°: What's Missing

This is all very nice, but plenty still needs proving:

- **Soundness:** Do well-typed processes ever go wrong?
- **Confluence:** now that we have a process calculus, is evaluation still completely deterministic?
- **Strong normalization:** as above, can we still guarantee that evaluation always terminates?
- **Annotated semantics:** extend exsisting work to prove that values of non-linear type are lone values, unaccompanied by any other waiting processes.

Classical F$^\circ$: Taking Stock

So what exactly do we have at this point?

- One view: something very much like the handled futures of Niehren et al. (2006) for flexible background computation.
- Alternatively: support for very primitive communication protocols, in which only a single value is passed.

Taking the latter view, how can we generalize further?

The problem: for one-shot protocols, $\triangleright\tau$ is an appropriate type for the main thread's channel endpoint—meaning "receive a τ"—and for the entire spawned thread—$(\tau \overset{\circ}{\to} \perp) \overset{\circ}{\to} \perp$, or "given the ability to consume a τ and exit, do so". This is not true in general, though.

What follows is all very sketchy!

Example: Diffie-Hellman Key Exchange

1. Alice sends Bob her public integer.
2. Bob sends Alice his public integer, and both parties compute a shared secret.
3. Alice sends an encrypted string to Bob.

If Alice's code is a spawned thread, it could have type

$$(\mathsf{Int} \overset{\circ}{\to} \triangleright(\mathsf{Int}, \mathsf{String} \overset{\circ}{\to} \text{⅄})) \overset{\circ}{\to} \text{⅄}$$

After it is spawned, Bob might like a source of type

$$\triangleright(\mathsf{Int}, \mathsf{Int} \overset{\circ}{\to} \triangleright\mathsf{String})$$

(What Bob will get is not quite this, but functionally equivalent.)

Classical F$^\circ$: Protocols and Dualization

First, though, what types (written ρ) make sense as protocols?
The easiest answer:

$$\rho \;::=\; \perp \mid \tau \overset{\circ}{\to} \rho$$

(Note that this already includes $\triangleright\tau$.)

For a protocol type ρ, define its dual $\widetilde{\rho}$ as

$$\begin{aligned} \widetilde{\perp} &= \text{Unit} \\ \widetilde{\tau \overset{\circ}{\to} \rho} &= \triangleright(\tau, \widetilde{\rho}) \end{aligned}$$

(If it weren't for Unit, $\widetilde{\rho}$ would also be a protocol type.)

Now, redefine **go** to mediate between $\rho \overset{\circ}{\to} \perp$ and $\widetilde{\rho}$.

Example: Diffie-Hellman Key Exchange

Returning to the previous example, if we let

$$\rho = \mathsf{Int} \overset{\circ}{\rightarrow} \triangleright(\mathsf{Int}, \mathsf{String} \overset{\circ}{\rightarrow} \curlywedge)$$

where Alice's thread has type $\rho \overset{\circ}{\rightarrow} \curlywedge$, then Bob will receive a source of type

$$\widetilde{\rho} = \triangleright(\mathsf{Int}, \triangleright((\mathsf{Int}, \mathsf{String} \overset{\circ}{\rightarrow} \curlywedge) \overset{\circ}{\rightarrow} \curlywedge, \mathsf{Unit}))$$

Not quite what Bob wanted, but he will get a continuation c of type $(\mathsf{Int}, \mathsf{String} \overset{\circ}{\rightarrow} \curlywedge) \overset{\circ}{\rightarrow} \curlywedge$. He can continue the protocol with

$$\mathbf{go}\ \lambda^{\circ} f{:}\mathsf{String} \overset{\circ}{\rightarrow} \curlywedge.\ c\ (i, f)$$

sending i to Alice and eventually receiving her encrypted string.

Classical F°: Origins of Dualization

In standard classical linear logic, negation (written $P^{\perp}$ is often be defined on all but atomic propositions:

$$\begin{aligned}
(A^{\perp})^{\perp} &\triangleq A \\
\mathbf{1}^{\perp} &\triangleq \perp \\
\perp^{\perp} &\triangleq \mathbf{1} \\
(P \parr Q)^{\perp} &\triangleq P^{\perp} \otimes Q^{\perp} \\
&\vdots
\end{aligned}$$

Additionally, $P \multimap Q \triangleq P^{\perp} \parr Q$

In this light, **go** pushes a top-level negation into a type, guarding this act of classical reasoning with a double negation.

Classical F°: Protocol Expressions

These types look promising, but what inhabits them? (This is a very open question.)

We must provide arguments to new threads of type $\rho \xrightarrow{\circ} ⅄$; in short, we must create communicators of ρ type (previously just sinks or type $\tau \xrightarrow{\circ} ⅄$).

Generalize sinks to $\lambda^{\circ}\_{:}\tau{\triangleright}a.\ e$, where $\tau{\triangleright}a = \lambda^{\circ}\_{:}\tau{\triangleright}a.\ \varnothing$.

Do sources need a similar generalization (to add non-communicated results), or can everything be done on the sink side? How should channel name bindings be handled? Currently unsure.

"Mundane" functions between $\rho \xrightarrow{\circ} ⅄$ and $\widetilde{\rho}$ should also exist; are they needed for congruence rules?

Classical F°: Broader Protocols

The set of ρ-types seems quite limited. How about expanding it?

- Additives (i.e., lazy tuples) should not present a problem.
- Universal types dualize to existential types.
- Type variables are a problem.
  - $\widetilde{\alpha}$ must be added to the language of types.
  - Variables must only be instantiated with other ρ types.

This suggests identifying ρ-types at the kind level. One possibility: a three kind system, where $\circ$ describes only protocol types and a new kind $\bullet$ functions as the necessarily linear "top kind".

Would require more limits on the formation of arrow types.

Going Further

A Simple Extension: Additives

So far our (encoded) linear tuples have required that all components be used.

Another option: lazy tuples which allow exactly one projection. We write these as $\langle \tau_1, \ldots, \tau_n \rangle$, with the obvious rules, including

$$[\text{T-Choice}]\ \frac{\Gamma \vdash e_1{:}\tau_1 \qquad \ldots \qquad \Gamma \vdash e_n{:}\tau_n}{\Gamma \vdash \langle e_1, \ldots, e_n \rangle{:}\langle \tau_1, \ldots, \tau_n \rangle}$$

Note the sharing of resources!

Can encode sums in the obvious way.

Not yet formalized, but seems like a relatively simple extension.

Going Further: Recursion

Rather than **fix**, F° seems well-suited for recursive types, because it could allow for non-terminating protocols.

More grounds for extending to the three kind system; we need recursive type variables to be ρ-types.

Questions:

- Isorecursive or equirecursive types?
- Are we still confluent? (I sure hope so!)

Note that complex recursive protocols can be hidden behind abstract protocols similar to the filesystem example.

Going Further: Non-determinism

All this talk of confluence, but plenty of useful concurrent programs are intentionally non-deterministic.

One possible primitive: an ambiguous choice operator

$$\mathbf{amb} : \triangleright\tau_1 \xrightarrow{\star} \triangleright\tau_2 \xrightarrow{\circ} \langle \tau_1 \xrightarrow{\circ} \triangleright\tau_2 \xrightarrow{\circ} \tau', \triangleright\tau_1 \xrightarrow{\circ} \tau_2 \xrightarrow{\circ} \tau' \rangle \xrightarrow{\circ} \tau'$$

($\langle \tau_1, \tau_2 \rangle$ is the lazy, or additive, or "projection-only" product.)

amb waits for the first of its two source arguments to be ready and uses its third argument accordingly.

Seems in line with the needs of concurrent programs.

Does it preserve strong normalization?

Questions?

Classical F°: Key Evaluation Rules

$$[\text{E-AppSource}]\ a{\triangleright}\tau\ v \longrightarrow v\ (\textbf{yield}\ a{\triangleright}\tau)$$

$$[\text{E-YieldOther}]\ \frac{v \neq a{\triangleright}\tau}{\textbf{yield}\ v \longrightarrow \textbf{yield}\ (\textbf{go}\ v)}$$

$$[\text{EP-Go}]\ \frac{a \text{ not free in } E\big[\textbf{go}\ v\big] \qquad v = b{\triangleright}\tau \ \vee\ v = \tau \xrightarrow{\circ} \curlywedge{\triangleright}b \ \vee\ v = \lambda^{\circ}x{:}\tau \xrightarrow{\circ} \curlywedge.\ e}{E\big[\textbf{go}\ v\big] \longrightarrow \nu a^{\tau}.\ \big(E\big[a{\triangleright}\tau\big] \mid v\ \tau{\triangleright}a\big)}$$

$$[\text{EP-Comm}]\ E_1\big[\textbf{yield}\ a{\triangleright}\tau\big] \mid E_2\big[\tau{\triangleright}a\ v\big] \longrightarrow E_1\big[v\big] \mid E_2\big[\varnothing\big]$$

$$[\text{EP-Done}]\ \varnothing \mid P \longrightarrow P$$

Example: A Safe Filesystem

Polymorphism is helpful; consider the similar

$$\begin{aligned}\mathsf{ROFH}(\alpha) \;=\; \{\, & \mathsf{handle} : \alpha, \\ & \mathsf{read} : \alpha \xrightarrow{\star} (\mathsf{Char}, \alpha), \\ & \mathsf{close} : \alpha \xrightarrow{\circ} \mathsf{Unit} \\ \} & \end{aligned}$$

If we give our library functions types like

$$\mathsf{readLine} : \forall \alpha{:}\circ.\, \mathsf{ROFH}(\alpha) \xrightarrow{\star} (\mathsf{String}, \mathsf{ROFH}(\alpha))$$

we can create a ROFH(α) from a FH(α), pass it to readLine, and reconstruct the FH(α) from the return value.

Why not the ! modality?

Standard linear systems assume linearity by default and allow non-linearity by means of a ! modality:

$$\frac{[\Gamma] \qquad \Gamma \vdash e : \tau}{\Gamma \vdash !e : !\tau}$$

$$\frac{\Gamma_1 \vdash e_1 : !\tau_1 \qquad \Gamma_2, [x{:}\tau_1] \vdash e_2 : \tau_2 \qquad \Gamma_1 \Cup \Gamma_2 = \Gamma}{\Gamma \vdash \mathbf{let}\ !x = e_1\ \mathbf{in}\ e_2 : \tau_2}$$

where $[\Gamma]$ and $[x{:}\tau_1]$ denote non-linearity.

- F° can simulate $!\tau$ with Unit $\overset{\star}{\to} \tau$.
- Kinds make non-linearity more natural.
- F° is aimed at applications where linear and non-linear universes don't overlap as often.

Intuitionistic F°: Annotated Semantics

How are we sure that linearity gets us what we want?

"Linear subterms are never duplicated or discarded"—this is very much a false assumption! Consider:

$$\lambda^\star y{:}\tau.\ (\lambda^\circ x{:}\tau.\ x)\ y$$

This is harmless, but call-by-name isn't so lucky.

For example, (close h) may be discarded without being run!

But call-by-name isn't sound, so we're safe, right. . . ?

(Similar issues with Λs and the value restriction.)

Intuitionistic F°: Annotated Semantics

Verifying that intuition and reality intersect:

- Define alternate semantics where annotated expressions $(e{:}\tau)$ and types $(\tau{:}\kappa)$ are substituted upon application.
- Track introduction and elimination of annotations in reduction.
- Prove that linear annotations "match up".
- Corollary: non-linear values are free from linear annotations on subterms. (This will be important later!)

A reminder that linearity is really about the use of **variables**.

Discussed in as-yet-unpublished paper (Mazurak and Zdancewic, 2010?), along with general encoding of regular protocols.