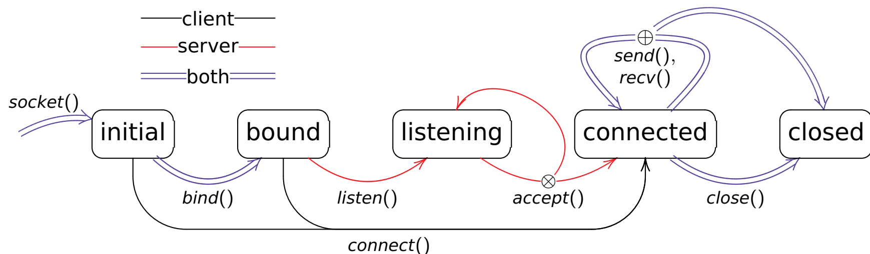
Stateful Contracts for Affine Types

Jesse A. Tov   Riccardo Pucella

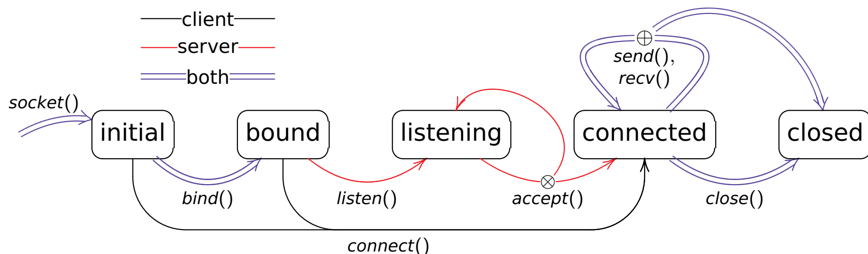
College of Computer and Information Science
Northeastern University

19th European Symposium on Programming
Paphos, Cyprus
23 March 2010

The Berkeley Sockets State Machine



The Berkeley Sockets State Machine



module Socket

```
val connect : α sock → string → int →
                (α initial + α bound) →
                  α connected
                        ⋮
```

Fractional Capabilities

module Region

```
type ρ rgn
type (ρ, α) ref
type (ρ, γ) cap qualifier Affine

val split : (ρ, γ) cap → (ρ, γ/2) cap × (ρ, γ/2) cap
val join  : (ρ, γ/2) cap × (ρ, γ/2) cap → (ρ, γ) cap

val write : (ρ, α) ref × α × (ρ, γ) cap → (ρ, γ) cap
val free  : ρ rgn × (ρ, 1) cap → unit
                                  ⋮
```

Fractional Capabilities

module Region

```
type ρ rgn
type (ρ, α) ref
type (ρ, γ) cap qualifier Affine

val split : (ρ, γ) cap → (ρ, γ/2) cap × (ρ, γ/2) cap
val join  : (ρ, γ/2) cap × (ρ, γ/2) cap → (ρ, γ) cap

val write : (ρ, α) ref × α × (ρ, γ) cap → (ρ, γ) cap
val free  : ρ rgn × (ρ, 1) cap → unit
                              ⋮
```

Session Types

module Session

```
type β channel qualifier Affine
```

Session Types

module Session

```
type β channel qualifier Affine

type α send
type α recv
type α ; β
```

Session Types

module Session

```
type β channel qualifier Affine

type α send
type α recv
type α ; β
```

```
(string recv; int recv; string send; unit) channel
```

Session Types

module Session

```
type β channel qualifier Affine

type α send
type α recv
type α ; β

val send : α → (α send; β) channel → β channel
val recv : (α recv; β) channel → α × β channel
                            ⋮
```

`(string recv; int recv; string send; unit) channel`

Example: Session Types and Threads

```
module Session
  ⋮
```

Example: Session Types and Threads

module Session

⋮

module Thread

```
type thread

val fork : (unit → unit) → thread
```

⋮

Example: Session Types and Threads

module Session

⋮

module Thread

```
type thread

val fork : (unit → unit) → thread
```

⋮

```
Thread.fork
  (fun () → startServer (Session.recv c))
```

Can we do this safely?

Example: Session Types and Threads

module Session

⋮

module Thread

```
let fork (thunk : unit → unit) =
  Prim.fork thunk;
  Prim.fork thunk
```

⋮

```
Thread.fork
  (fun () → startServer (Session.recv c))
```

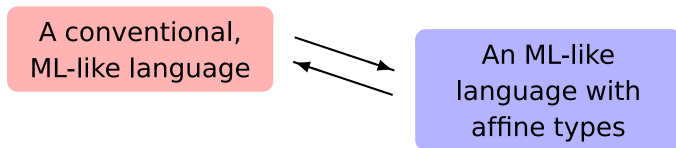
Can we do this safely?

What We've Done

A conventional,
ML-like language

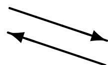
An ML-like
language with
affine types

What We've Done



What We've Done

A conventional, ML-like language

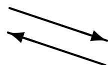


An ML-like language with affine types

Nothing funny over here (no special knowledge of the affine type system)

What We've Done

A conventional, ML-like language

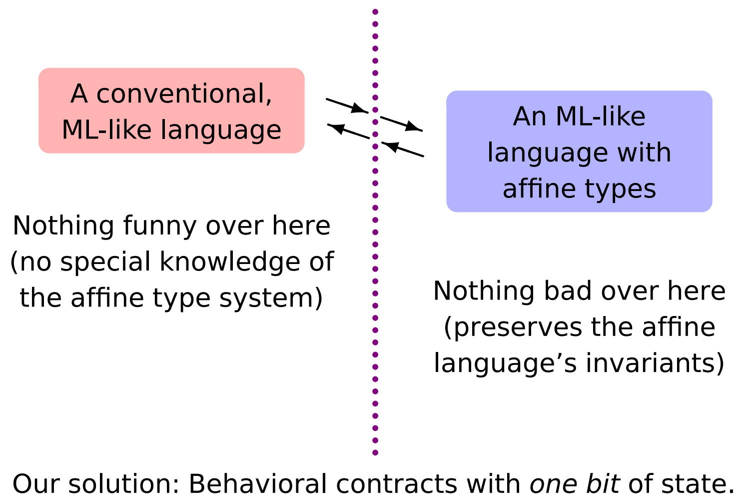


An ML-like language with affine types

Nothing funny over here (no special knowledge of the affine type system)

Nothing bad over here (preserves the affine language's invariants)

What We've Done



Our solution: Behavioral contracts with *one bit* of state.

How to Type the Example

```
fork (fun () → startServer (recv c))
     :
```

How to Type the Example

recv : (α recv; β) chan $\rightarrow$ $\alpha \times \beta$ chan

fork (fun () $\rightarrow$ startServer (recv c))
:

How to Type the Example

```
recv  : (α recv; β) chan → α × β chan
c     : (string recv; int recv; string send; unit) chan
```

```
fork (fun () → startServer (recv c))
     :
```

How to Type the Example

```
recv  : (α recv; β) chan → α × β chan
c     : (string recv; int recv; string send; unit) chan
fork  : (unit → unit) → thread
```

```
fork (fun () → startServer (recv c))
     :
```

How to Type the Example

```
recv  : (α recv; β) chan → α × β chan
c     : (string recv; int recv; string send; unit) chan
fork  : (unit → unit) → thread
```

```
fun () → startServer (recv c)
      : unit ─1→ unit
```

```
fork (fun () → startServer (recv c))
      :
```

How to Type the Example

recv : (α recv; β) chan $\rightarrow$ α $\times$ β chan
c : (string recv; int recv; string send; unit) chan
fork : (unit $\rightarrow$ unit) $\rightarrow$ thread

fun () $\rightarrow$ startServer (recv c)
: unit $\xrightarrow{1}$ unit

fork (fun () $\rightarrow$ startServer (recv c))
: type error!

How to Type the Example

```
recv  : (α recv; β) chan → α × β chan
c     : (string recv; int recv; string send; unit) chan
fork  : (unit → unit) → thread
```

```
fun () → startServer (recv c)
      : unit →¹ unit
```

```
fork (fun () → startServer (recv c))
      :
```

How to Type the Example

```
recv   : (α recv; β) chan → α × β chan
c      : (string recv; int recv; string send; unit) chan
fork   : (unit → unit) → thread

fun () → startServer (recv c)
       : unit ─1→ unit
```

```
interface fork' :> (unit ─1→ unit) → Thread.thread
    = Thread.fork
```

```
fork (fun () → startServer (recv c))
       :
```

How to Type the Example

```
recv  : (α recv; β) chan → α × β chan
c     : (string recv; int recv; string send; unit) chan
fork  : (unit → unit) → thread
```

```
fun () → startServer (recv c)
      : unit →¹ unit
```

```
interface fork' :> (unit →¹ unit) → Thread.thread
    = Thread.fork
```

```
fork' (fun () → startServer (recv c))
      :
```

How to Type the Example

```
recv  : (α recv; β) chan → α × β chan
c     : (string recv; int recv; string send; unit) chan
fork  : (unit → unit) → thread

fun () → startServer (recv c)
      : unit -1-> unit
```

```
interface fork' :> (unit -1-> unit) → Thread.thread
    = Thread.fork
```

```
fork' (fun () → startServer (recv c))
      : Thread.thread
```

Implementation Overview

1. Map types to types
2. Map types to behavioral contracts[1]:
3. Wrap values with contracts at language boundaries

[1] Tobin-Hochstadt and Felleisen, '06 and '08

Implementation Overview

1. Map types to types

$$(\boxed{\tau})^{\mathscr{A}} = \boxed{\tau} \qquad (\boxed{\tau})^{\mathscr{C}} = \boxed{\tau}$$

2. Map types to behavioral contracts[1]:

3. Wrap values with contracts at language boundaries

[1] Tobin-Hochstadt and Felleisen, '06 and '08

Implementation Overview

1. Map types to types

$$(\boxed{\tau})^{\mathscr{A}} = \boxed{\tau} \qquad (\boxed{\tau})^{\mathscr{C}} = \boxed{\tau}$$

2. Map types to behavioral contracts[1]:

$$\mathscr{AC}[\![\boxed{\tau}]\!] = \boxed{\mathsf{e}} \qquad \mathscr{CA}[\![\boxed{\tau}]\!] = \boxed{\mathsf{e}}$$

3. Wrap values with contracts at language boundaries

[1] Tobin-Hochstadt and Felleisen, ’06 and ’08

Implementation Overview

1. Map types to types

$$(\boxed{\tau})^{\mathscr{A}} = \boxed{\tau} \qquad (\boxed{\tau})^{\mathscr{C}} = \boxed{\tau}$$

2. Map types to behavioral contracts[1]:

$$\mathscr{AC}[\![\boxed{\tau}]\!] = \boxed{e} \qquad \mathscr{CA}[\![\boxed{\tau}]\!] = \boxed{e}$$

3. Wrap values with contracts at language boundaries

$$\boxed{\boxed{m} : \tau} \Longrightarrow \boxed{\boxed{\mathscr{AC}[\![\boxed{\tau}]\!](m)}}$$

$$\boxed{\text{let } m : \tau = v} \Longrightarrow \boxed{\text{let } m_{\mathscr{C}} : (\boxed{\tau})^{\mathscr{C}} = \mathscr{CA}[\![\boxed{\tau}]\!](\boxed{m})}$$

[1] Tobin-Hochstadt and Felleisen, '06 and '08

Map Types to Types

$$(\text{int})^{\mathscr{C}} = \text{int}$$

$$(\text{int})^{\mathscr{A}} = \text{int}$$

Map Types to Types

$$(\text{int})^{\mathscr{C}} = \text{int} \qquad (\text{int})^{\mathscr{A}} = \text{int}$$

$$(\tau \times \sigma)^{\mathscr{C}} = (\tau)^{\mathscr{C}} \times (\sigma)^{\mathscr{C}} \qquad (\tau \times \sigma)^{\mathscr{A}} = (\tau)^{\mathscr{A}} \times (\sigma)^{\mathscr{A}}$$

Map Types to Types

$$(\!|\,\text{int}\,|\!)^{\mathscr{C}} = \text{int} \qquad (\!|\,\text{int}\,|\!)^{\mathscr{A}} = \text{int}$$

$$(\!|\,\tau \times \sigma\,|\!)^{\mathscr{C}} = (\!|\,\tau\,|\!)^{\mathscr{C}} \times (\!|\,\sigma\,|\!)^{\mathscr{C}} \qquad (\!|\,\tau \times \sigma\,|\!)^{\mathscr{A}} = (\!|\,\tau\,|\!)^{\mathscr{A}} \times (\!|\,\sigma\,|\!)^{\mathscr{A}}$$

$$(\!|\,\tau \xrightarrow{\infty} \sigma\,|\!)^{\mathscr{C}} = (\!|\,\tau\,|\!)^{\mathscr{C}} \to (\!|\,\sigma\,|\!)^{\mathscr{C}} \qquad (\!|\,\tau \to \sigma\,|\!)^{\mathscr{A}} = (\!|\,\tau\,|\!)^{\mathscr{A}} \xrightarrow{\infty} (\!|\,\sigma\,|\!)^{\mathscr{A}}$$

Map Types to Types

$$(\!|\text{int}|\!)^{\mathscr{C}} = \text{int} \qquad (\!|\text{int}|\!)^{\mathscr{A}} = \text{int}$$

$$(\!|\tau \times \sigma|\!)^{\mathscr{C}} = (\!|\tau|\!)^{\mathscr{C}} \times (\!|\sigma|\!)^{\mathscr{C}} \qquad (\!|\tau \times \sigma|\!)^{\mathscr{A}} = (\!|\tau|\!)^{\mathscr{A}} \times (\!|\sigma|\!)^{\mathscr{A}}$$

$$(\!|\tau \xrightarrow{\infty} \sigma|\!)^{\mathscr{C}} = (\!|\tau|\!)^{\mathscr{C}} \to (\!|\sigma|\!)^{\mathscr{C}} \qquad (\!|\tau \to \sigma|\!)^{\mathscr{A}} = (\!|\tau|\!)^{\mathscr{A}} \xrightarrow{\infty} (\!|\sigma|\!)^{\mathscr{A}}$$

$$(\!|\tau \xrightarrow{1} \sigma|\!)^{\mathscr{C}} = (\!|\tau|\!)^{\mathscr{C}} \to (\!|\sigma|\!)^{\mathscr{C}}$$

Behavioral Contracts[2]

```
type party
type α contract = party × party → α → α
```

[2] Findler and Felleisen ’02

Behavioral Contracts[2]

```
type party
type α contract = party × party → α → α

(* evenContract : int contract *)
let evenContract (neg: party, pos: party) (x: int) =
  if isEven x
    then x
    else blame pos
```

[2] Findler and Felleisen '02

Behavioral Contracts[2]

```
type party
type α contract = party × party → α → α

(* evenContract : int contract *)
let evenContract (neg: party, pos: party) (x: int) =
  if isEven x
    then x
    else blame pos

(* α contract × β contract → (α → β) contract *)
let funContract
      (dom: α contract, codom: β contract)
      (neg: party, pos: party) (f: α → β) =
  fun (x: α) → codom (neg,pos) (f (dom (pos,neg) x))
```

[2] Findler and Felleisen '02

Behavioral Contracts[2]

```
type party
type α contract = party × party → α → α

(* evenContract : int contract *)
let evenContract (neg: party, pos: party) (x: int) =
  if isEven x
    then x
    else blame pos

(* α contract × β contract → (α → β) contract *)
let funContract
      (dom: α contract, codom: β contract)
      (neg: party, pos: party) (f: α → β) =
  fun (x: α) → codom (neg,pos) (f (dom (pos,neg) x))
```

[2] Findler and Felleisen '02

Behavioral Contracts[2]

```
type party
type α contract = party × party → α → α

(* evenContract : int contract *)
let evenContract (neg: party, pos: party) (x: int) =
  if isEven x
    then x
    else blame pos

(* α contract × β contract → (α → β) contract *)
let funContract
      (dom: α contract, codom: β contract)
      (neg: party, pos: party) (f: α → β) =
  fun (x: α) → codom (neg,pos) (f (dom (pos,neg) x))
```

[2] Findler and Felleisen ’02

Behavioral Contracts[2]

```
type party
type α contract = party × party → α → α

(* evenContract : int contract *)
let evenContract (neg: party, pos: party) (x: int) =
  if isEven x
    then x
    else blame pos

(* α contract × β contract → (α → β) contract *)
let funContract
      (dom: α contract, codom: β contract)
      (neg: party, pos: party) (f: α → β) =
  fun (x: α) → codom (neg,pos) (f (dom (pos,neg) x))
```

[2] Findler and Felleisen '02

Behavioral Contracts[2]

```
type party
type α contract = party × party → α → α

(* evenContract : int contract *)
let evenContract (neg: party, pos: party) (x: int) =
  if isEven x
    then x
    else blame pos

(* α contract × β contract → (α → β) contract *)
let funContract
      (dom: α contract, codom: β contract)
      (neg: party, pos: party) (f: α → β) =
  fun (x: α) → codom (neg,pos) (f (dom (pos,neg) x))
```

[2] Findler and Felleisen '02

Behavioral Contracts[2]

```
type party
type α contract = party × party → α → α

(* evenContract : int contract *)
let evenContract (neg: party, pos: party) (x: int) =
  if isEven x
    then x
    else blame pos

(* α contract × β contract → (α → β) contract *)
let funContract
      (dom: α contract, codom: β contract)
      (neg: party, pos: party) (f: α → β) =
  fun (x: α) → codom (neg,pos) (f (dom (pos,neg) x))
```

[2] Findler and Felleisen '02

Stateful Behavioral Contracts

```
(* α contract × β contract → (α → β) contract *)
let funContract
      (dom: α contract, codom: β contract)
      (neg: party, pos: party) (f: α → β) =
  fun (x: α) → codom (neg,pos) (f (dom (pos,neg) x))
```

Stateful Behavioral Contracts

```
(* α contract × β contract → (α → β) contract *)
let funContract
      (dom: α contract, codom: β contract)
      (neg: party, pos: party) (f: α → β) =
  fun (x: α) → codom (neg,pos) (f (dom (pos,neg) x))

(* α contract × β contract → (α → β) contract *)
let affineFunContract
      (dom: α contract, codom: β contract)
      (neg: party, pos: party) (f: α → β) =

    fun (x: α) →



             codom (neg,pos) (f (dom (pos,neg) x))
```

Stateful Behavioral Contracts

```
(* α contract × β contract → (α → β) contract *)
let funContract
      (dom: α contract, codom: β contract)
      (neg: party, pos: party) (f: α → β) =
  fun (x: α) → codom (neg,pos) (f (dom (pos,neg) x))

(* α contract × β contract → (α → β) contract *)
let affineFunContract
      (dom: α contract, codom: β contract)
      (neg: party, pos: party) (f: α → β) =
  let used = ref false in
    fun (x: α) →



             codom (neg,pos) (f (dom (pos,neg) x))
```

Stateful Behavioral Contracts

```
(* α contract × β contract → (α → β) contract *)
let funContract
      (dom: α contract, codom: β contract)
      (neg: party, pos: party) (f: α → β) =
  fun (x: α) → codom (neg,pos) (f (dom (pos,neg) x))

(* α contract × β contract → (α → β) contract *)
let affineFunContract
      (dom: α contract, codom: β contract)
      (neg: party, pos: party) (f: α → β) =
  let used = ref false in
    fun (x: α) →
      if used↑
        then
        else
             codom (neg,pos) (f (dom (pos,neg) x))
```

Stateful Behavioral Contracts

```
(* α contract × β contract → (α → β) contract *)
let funContract
      (dom: α contract, codom: β contract)
      (neg: party, pos: party) (f: α → β) =
  fun (x: α) → codom (neg,pos) (f (dom (pos,neg) x))

(* α contract × β contract → (α → β) contract *)
let affineFunContract
      (dom: α contract, codom: β contract)
      (neg: party, pos: party) (f: α → β) =
  let used = ref false in
    fun (x: α) →
      if used↑
        then blame neg
        else
             codom (neg,pos) (f (dom (pos,neg) x))
```

Stateful Behavioral Contracts

```
(* α contract × β contract → (α → β) contract *)
let funContract
      (dom: α contract, codom: β contract)
      (neg: party, pos: party) (f: α → β) =
  fun (x: α) → codom (neg,pos) (f (dom (pos,neg) x))

(* α contract × β contract → (α → β) contract *)
let affineFunContract
      (dom: α contract, codom: β contract)
      (neg: party, pos: party) (f: α → β) =
  let used = ref false in
    fun (x: α) →
      if used↑
        then blame neg
        else used ← true;
             codom (neg,pos) (f (dom (pos,neg) x))
```

Map Types to Contracts

$$\mathscr{CA}[\![\ \text{int}\ ]\!] = \text{idC}_{\text{int}}$$

Map Types to Contracts

$$\mathscr{CA}[\![\ \text{int}\ ]\!] = \text{idC}_{\text{int}}$$

$$\mathscr{AC}[\![\ \text{int}\ ]\!] = \text{idC}_{\text{int}}$$

Map Types to Contracts

$$\mathscr{CA}[\![\ \text{int}\ ]\!] = \text{idC}_{\text{int}}$$

$$\mathscr{CA}[\![\ \tau \xrightarrow{1} \sigma\ ]\!] = \text{affineFunContract}\ (\mathscr{CA}[\![\ \tau\ ]\!], \mathscr{AC}[\![\ \sigma\ ]\!])$$

$$\mathscr{AC}[\![\ \text{int}\ ]\!] = \text{idC}_{\text{int}}$$

Map Types to Contracts

$$\mathscr{CA}[\![\, \text{int} \,]\!] = \text{idC}_{\text{int}}$$

$$\mathscr{CA}[\![\, \tau \xrightarrow{1} \sigma \,]\!] = \text{affineFunContract}\ (\mathscr{CA}[\![\, \tau \,]\!], \mathscr{AC}[\![\, \sigma \,]\!])$$

$$\mathscr{AC}[\![\, \text{int} \,]\!] = \text{idC}_{\text{int}}$$

$$\mathscr{AC}[\![\, \tau \xrightarrow{1} \sigma \,]\!] = \text{funContract}\ (\mathscr{AC}[\![\, \tau \,]\!], \mathscr{CA}[\![\, \sigma \,]\!])$$

Map Types to Contracts

$$\mathscr{CA}[\![\ \text{int}\ ]\!] = \text{idC}_{\text{int}}$$

$$\mathscr{CA}[\![\ \tau \xrightarrow{1} \sigma\ ]\!] = \text{affineFunContract}\ (\mathscr{CA}[\![\ \tau\ ]\!], \mathscr{AC}[\![\ \sigma\ ]\!])$$

$$\mathscr{CA}[\![\ \tau \xrightarrow{\infty} \sigma\ ]\!] = \text{funContract}\ (\mathscr{CA}[\![\ \tau\ ]\!], \mathscr{AC}[\![\ \sigma\ ]\!])$$

$$\mathscr{AC}[\![\ \text{int}\ ]\!] = \text{idC}_{\text{int}}$$

$$\mathscr{AC}[\![\ \tau \xrightarrow{1} \sigma\ ]\!] = \text{funContract}\ (\mathscr{AC}[\![\ \tau\ ]\!], \mathscr{CA}[\![\ \sigma\ ]\!])$$

$$\mathscr{AC}[\![\ \tau \xrightarrow{\infty} \sigma\ ]\!] = \text{funContract}\ (\mathscr{AC}[\![\ \tau\ ]\!], \mathscr{CA}[\![\ \sigma\ ]\!])$$

Our Prototype

ex39-talk-type-error.aff

```
type prot = string send; int send; string recv; unit




let listen (r: prot rendezvous) =
  let c = accept r in
    Thread.fork (fun () → startServer (recv c))
                              ⋮
```

Our Prototype

ex39-talk-type-error.aff

```
type prot = string send; int send; string recv; unit




let listen (r: prot rendezvous) =
  let c = accept r in
    Thread.fork (fun () → startServer (recv c))
                              ⋮
```

ex39-talk-type-error.aff: Type error (line 42, column 5): Application (operand) got unit $\xrightarrow{1}$ unit where unit $\xrightarrow{\infty}$ unit expected

Our Prototype

ex40-talk-blame-error.aff

```
type prot = string send; int send; string recv; unit
```

```
interface fork' :> (unit -1-> unit) -∞-> thread = badFork
```

$\texttt{interface fork'} :> (\texttt{unit} \xrightarrow{1} \texttt{unit}) \xrightarrow{\infty} \texttt{thread} = \texttt{badFork}$

```
let listen (r: prot rendezvous) =
  let c = accept r in
    fork' (fun () → startServer (recv c))
```

⋮

Our Prototype

ex40-talk-blame-error.aff

```
type prot = string send; int send; string recv; unit
```

interface fork' :> (unit $\xrightarrow{1}$ unit) $\xrightarrow{\infty}$ thread = badFork

```
let listen (r: prot rendezvous) =
  let c = accept r in
    fork' (fun () → startServer (recv c))
```

⋮

ex40-talk-blame-error.aff: Blame ("fork'", "applied one-shot function twice")

Our Prototype

ex41-talk-correct.aff

```
type prot = string send; int send; string recv; unit

interface fork' :> (unit -1-> unit) -∞-> thread = fork

let listen (r: prot rendezvous) =
  let c = accept r in
    fork' (fun () → startServer (recv c))

            ⋮
```

Our Prototype

ex41-talk-correct.aff

```
type prot = string send; int send; string recv; unit

interface fork' :> (unit -1-> unit) -∞-> thread = fork

let listen (r: prot rendezvous) =
  let c = accept r in
    fork' (fun () → startServer (recv c))

                    ⋮
```

```
Hello
```

Formalization: Two Calculi

A conventional, ML-like language

An ML-like language with affine types

Formalization: Two Calculi

A conventional, ML-like language

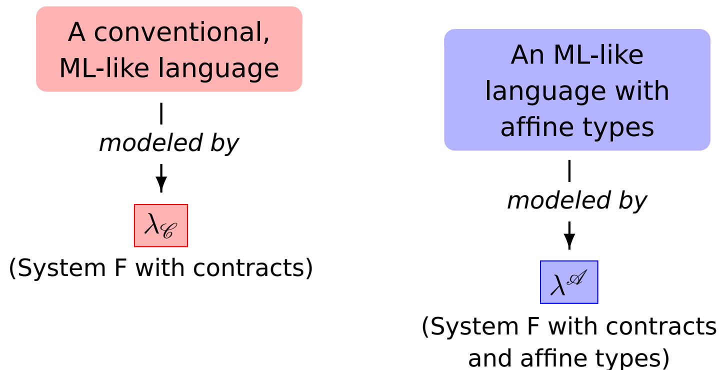
modeled by

$\lambda_{\mathscr{C}}$

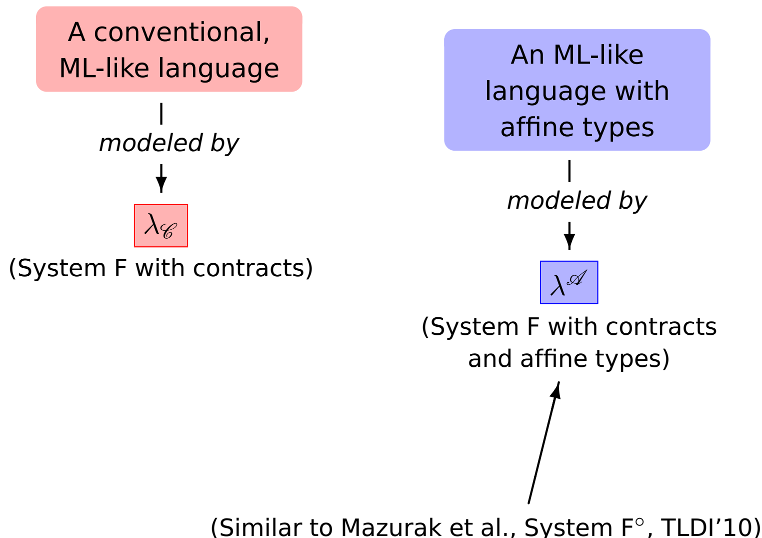
(System F with contracts)

An ML-like language with affine types

Formalization: Two Calculi

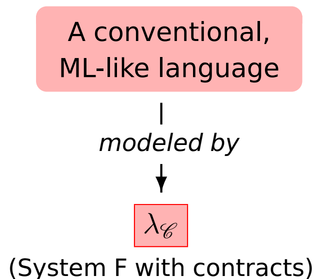


Formalization: Two Calculi

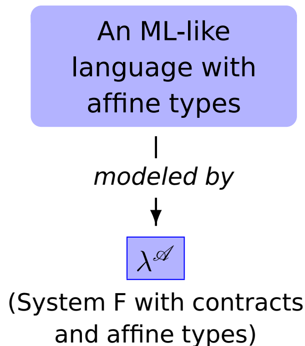


(Similar to Mazurak et al., System F°, TLDI'10)

Formalization: Two Calculi

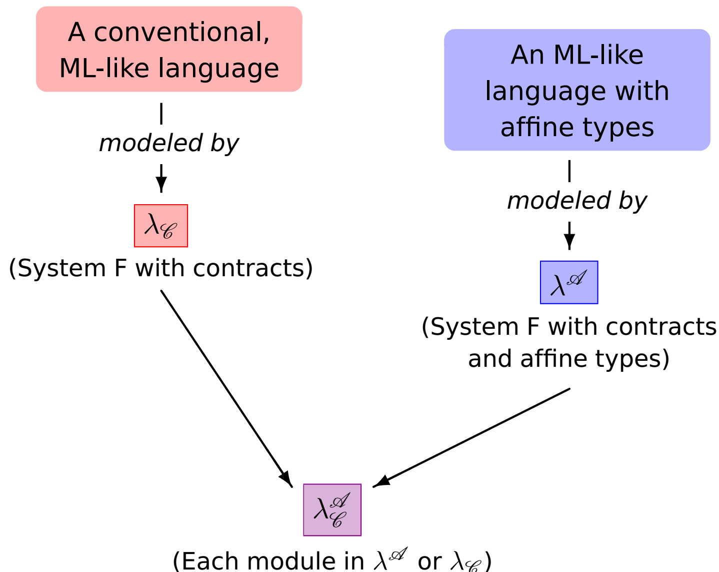


(System F with contracts)



(System F with contracts
and affine types)

Formalization: Two Calculi



Type Soundness

Since $\lambda^{\mathscr{A}}$ and $\lambda_{\mathscr{C}}$ modules invoke each other, this results in mixed terms in the reduction semantics:

$$f^g\ v \underset{M}{\longmapsto} \underset{g\,f}{\mathbf{CA}}^{\sigma}(\lambda x.M)\ v \underset{M}{\longmapsto} \underset{g\,f}{\mathbf{CA}}^{\sigma}((\lambda x.M)\ {}^{\tau}\underset{f\,g}{\mathbf{AC}}(v))$$

A (rather hairy) internal type system gives us preservation.

$$\frac{\Sigma_1, \Sigma_2; \cdot; \cdot \rhd^M_{\mathscr{C}} v : \sigma^w \qquad |\sigma^w| = a}{[\Sigma_1]^\ell, \ell{:}\mathbb{B}, [\Sigma_2]; \Delta; \Gamma \rhd^M_{\mathscr{C}} \underset{g\,f}{\mathbf{CA}}[\ell]^{\sigma^w}(v) : (\sigma^w)^{\mathscr{C}}}$$

Which leads to our type soundness result:

Well-typed programs go wrong only by
blaming an interface or $\lambda_{\mathscr{C}}$ module.

Related Work

Substructural type systems:

- Ahmed et al. '05; Fluet '07 (λ^{URAL})
- Wadler '90, '91, '95 (use types, standard types, let!)
- Fähndrich and DeLine '02; Pottier '07 (focusing and adoption)

Conclusion: Practical Substructural Types

Done Dereliction subtyping, affine abstract types, stateful contracts

Doing Module system

To do Type inference, linear exceptions, . . .

Thank You

Done Dereliction subtyping, affine abstract types, stateful contracts

Doing Module system

To do Type inference, linear exceptions, . . .

Contact us:

- tov@ccs.neu.edu

- http://www.ccs.neu.edu/~tov/pubs/affine-contracts/
  - Color version of paper (easier to read)
  - Long version of paper (harder to read)
  - Prototype with extensive examples