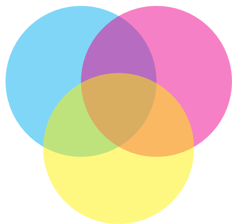
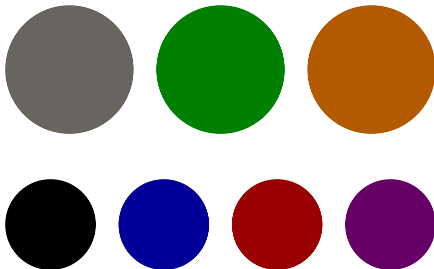
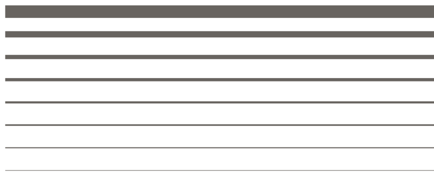




Because propositions in such a logic may no longer be freely copied or ignored, this suggests understanding propositions in substructural logics as representing resources rather than truth.

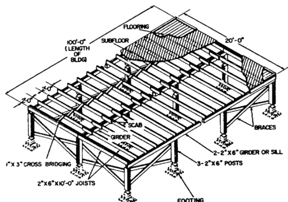


Practical Programming with Substructural Types

Jesse A. Tov

Northeastern University

February 10, 2012



```
int udp_sendmsg(struct sock *sk, struct msghdr *msg, ...)
{
  ⋮
  if (unlikely(sk→pending)) {
    /* Socket is already corked while preparing it */
    /* ... which is an evident application bug. –ANK */
    LIMIT_NETDEBUG(KERN_DEBUG "udp cork app bug 2");
    return -EINVAL;
  }
  lock_sock(sk);
  ret = ip_append_data(sk, msg→msg_iov, ulen, ...);
  ⋮
  release_sock(sk);
  return ret;
}
```

```
int udp_sendmsg(struct sock *sk, struct msghdr *msg, ...)
{
  ⋮
  if (unlikely(sk→pending)) {
    /* Socket is already corked while preparing it */
    /* ... which is an evident application bug. –ANK */
    LIMIT_NETDEBUG(KERN_DEBUG "udp cork app bug 2");
    return -EINVAL;
  }
  lock_sock(sk);
  ret = ip_append_data(sk, msg→msg_iov, ulen, ...);
  ⋮
  release_sock(sk);
  return ret;
}
```

```
int udp_sendmsg(struct sock *sk, struct msghdr *msg, ...)
{
  ⋮
  if (unlikely(sk→pending)) {
    /* Socket is already corked while preparing it */
    /* ... which is an evident application bug. –ANK */
    LIMIT_NETDEBUG(KERN_DEBUG “udp cork app bug 2”);
    return -EINVAL;
  }
  lock_sock(sk);
  ret = ip_append_data(sk, msg→msg_iov, ulen, ...);
  ⋮
  release_sock(sk);
  return ret;
}
```

```
int udp_sendmsg(struct sock *sk, struct msghdr *msg, ...)
{
  ⋮
  if (unlikely(sk→pending)) {
    /* Socket is already corked while preparing it */
    /* ... which is an evident application bug. –ANK */
    LIMIT_NETDEBUG(KERN_DEBUG "udp cork app bug 2");
    return -EINVAL;
  }
  lock_sock(sk);
  ret = ip_append_data(sk, msg→msg_iov, ulen, ...);
  ⋮
  release_sock(sk);
  return ret;
}
```

```
int udp_sendmsg(struct sock *sk, struct msghdr *msg, ...)
{
  ⋮
  if (unlikely(sk→pending)) {
    /* Socket is already corked while preparing it */
    /* ... which is an evident application bug. –ANK */
    LIMIT_NETDEBUG(KERN_DEBUG "udp cork app bug 2");
    return -EINVAL;
  }
  lock_sock(sk);
  ret = ip_append_data(sk, msg→msg_iov, ulen, ...);
  ⋮
  release_sock(sk);
  return ret;
}
```

```
int udp_sendmsg(struct sock *sk, struct msghdr *msg, ...)
{
  ⋮
  if (unlikely(sk→pending)) {
    /* Socket is already corked while preparing it */
    /* ... which is an evident application bug. –ANK */
    LIMIT_NETDEBUG(KERN_DEBUG “udp cork app bug 2”);
    return -EINVAL;
  }
  lock_sock(sk);
  ret = ip_append_data(sk, msg→msg_iov, ulen, ...);
  ⋮
  release_sock(sk);
  return ret;
}
```

```
int udp_sendmsg(struct sock *sk, struct msghdr *msg, ...)
{
  ⋮
  if (unlikely(sk→pending)) {
    /* Socket is already corked while preparing it */
    /* ... which is an evident application bug. –ANK */
    LIMIT_NETDEBUG(KERN_DEBUG "udp cork app bug 2");
    return -EINVAL;
  }
  lock_sock(sk);
  ret = ip_append_data(sk, msg→msg_iov, ulen, ...);
  ⋮
  release_sock(sk);
  return ret;
}
```

```
int udp_sendmsg(struct sock *sk, struct msghdr *msg, ...)
{
  ⋮
  lock_sock(sk);
  if (unlikely(sk→pending)) {
    /* Socket is already corked while preparing it */
    /* ... which is an evident application bug. –ANK */
    LIMIT_NETDEBUG(KERN_DEBUG "udp cork app bug 2");
    return -EINVAL;
  }
  ret = ip_append_data(sk, msg→msg_iov, ulen, ...);
  ⋮
  release_sock(sk);
  return ret;
}
```

```
int udp_sendmsg(struct sock *sk, struct msghdr *msg, ...)
{
  ⋮
  lock_sock(sk);
  if (unlikely(sk→pending)) {
    /* Socket is already corked while preparing it */
    /* ... which is an evident application bug. –ANK */
    release_sock(sk);
    LIMIT_NETDEBUG(KERN_DEBUG "udp cork app bug 2");
    return -EINVAL;
  }
  ret = ip_append_data(sk, msg→msg_iov, ulen, ...);
  ⋮
  release_sock(sk);
  return ret;
}
```

```
int udp_sendmsg(struct sock *sk, struct msghdr *msg, ...)
{
  ⋮
  lock_sock(sk);
  if (unlikely(sk→pending)) {
    /* Socket is already corked while preparing it */
    /* ... which is an evident application bug. –ANK */
    release_sock(sk);
    LIMIT_NETDEBUG(KERN_DEBUG "udp cork app bug 2");
    return -EINVAL;
  }
  ret = ip_append_data(sk, msg→msg_iov, ulen, ...);
  ⋮
  release_sock(sk);
  return ret;
}
```

Problem: Advisory locking

```
int udp_sendmsg(struct sock *sk, struct msghdr *msg, ...)
{
  ⋮
  lock_sock(sk);
  if (unlikely(sk→pending)) {
    /* Socket is already corked while preparing it */
    /* ... which is an evident application bug. –ANK */
    release_sock(sk);
    LIMIT_NETDEBUG(KERN_DEBUG "udp cork app bug 2");
    return -EINVAL;
  }
  ret = ip_append_data(sk, msg→msg_iov, ulen, ...);
  ⋮
  release_sock(sk);
  return ret;
}
```

Problem: Advisory locking

Solution: Mandatory locking

```
int udp_sendmsg(struct sock *sk, struct msghdr *msg, ...)
{
  ⋮
  lock_sock(sk);
  if (unlikely(sk->pending)) {
    /* Socket is already corked while preparing it */
    /* ... which is an evident application bug. --ANK */
    release_sock(sk);
    LIMIT_NETDEBUG(KERN_DEBUG "udp cork app bug 2");
    return -EINVAL;
  }
  ret = ip_append_data(sk, msg->msg_iov, ulen, ...);
  ⋮
  release_sock(sk);
  return ret;
}
```

```
int udp_sendmsg(struct sock *sk, struct msghdr *msg, ...)
{
  ⋮
  lock_sock(sk);
  if (unlikely(sk→pending)) {
    /* Socket is already corked while preparing it */
    /* ... which is an evident application bug. –ANK */
    release_sock(sk);
    LIMIT_NETDEBUG(KERN_DEBUG "udp cork app bug 2");
    return -EINVAL;
  }
  ret = ip_append_data(sk, msg→msg_iov, ulen, ...);
  ⋮
  release_sock(sk);
  return ret;
}
```

Problem: Advisory locking

Solution: Mandatory locking

Better solution: Static mandatory locking

```
int udp_sendmsg(struct sock *sk, struct msghdr *msg, ...)
{
  ⋮
  lock_sock(sk);
  if (unlikely(sk→pending)) {
    /* Socket is already corked while preparing it */
    /* ... which is an evident application bug. –ANK */
    release_sock(sk);
    LIMIT_NETDEBUG(KERN_DEBUG "udp cork app bug 2");
    return −EINVAL;
  }
  ret = ip_append_data(sk, msg→msg_iov, ulen, ...);
  ⋮
  release_sock(sk);
  return ret;
}
```

Problem: Advisory locking

Solution: Mandatory locking

Better solution: Static mandatory locking

There's a language for that: Chalice

(Leino et al. 2009)

```
type bid_msg  = Request of string
              | Offer of int
              | Reject
              | Accept
              | Resource of resource
type bid_chan = bid_msg chan
```

```
type bid_msg  = Request of string
              | Offer of int
              | Reject
              | Accept
              | Resource of resource
type bid_chan = bid_msg chan

let seller ch =
  let Request name = recv ch
   in match get name with
       | None           →
       send Reject ch
       | Some (r, price) →
       send (Offer price) ch;
       match recv ch with
         | Accept →
         send (Resource r) ch
         | Reject → ()
```

```
type bid_msg  = Request of string
              | Offer of int
              | Reject
              | Accept
              | Resource of resource
type bid_chan = bid_msg chan
```

```
let seller ch =
  let Request name = recv ch
   in match get name with
      | None             →
        send Reject ch
      | Some (r, price) →
        send (Offer price) ch;
        match recv ch with
        | Accept →
          send (Resource r) ch
        | Reject → ()
```

```
let buyer name limit ch =
  send (Request name) ch;
  let rec loop () =
    match recv ch with
    | Reject       → None
    | Offer price →
      if price ≤ limit then
        send Accept ch;
        let Resource r = recv ch
         in Some r
      else
        send (Offer limit) ch;
        loop ()
    in loop ()
```

```
type bid_msg  = Request of string
              | Offer of int
              | Reject
              | Accept
              | Resource of resource
type bid_chan = bid_msg chan
```

```
let seller ch =
  let Request name = recv ch
   in match get name with
      | None              ->
        send Reject ch
      | Some (r, price) ->
        send (Offer price) ch;
        match recv ch with
        | Accept ->
          send (Resource r) ch
        | Reject -> ()
```

```
let buyer name limit ch =
  send (Request name) ch;
  let rec loop () =
    match recv ch with
    | Reject       -> None
    | Offer price ->
      if price ≤ limit then
        send Accept ch;
        let Resource r = recv ch
         in Some r
      else
        send (Offer limit) ch;
        loop ()
  in loop ()
```

```
type bid_msg  = Request of string
              | Offer of int
              | Reject
              | Accept
              | Resource of resource
type bid_chan = bid_msg chan
```

```
let seller ch =
  let Request name = recv ch
   in match get name with
      | None          →
        send Reject ch
      | Some (r, price) →
        send (Offer price) ch;
        match recv ch with
        | Accept →
          send (Resource r) ch
        | Reject → ()
```

```
let buyer name limit ch =
  send (Request name) ch;
  let rec loop () =
    match recv ch with
    | Reject      → None
    | Offer price →
      if price ≤ limit then
        send Accept ch;
        let Resource r = recv ch
         in Some r
      else
        send (Offer limit) ch;
        loop ()
  in loop ()
```

```
type bid_msg  = Request of string
              | Offer of int
              | Reject
              | Accept
              | Resource of resource
type bid_chan = bid_msg chan

let seller ch =
  let Request name = recv ch
   in match get name with
      | None           →
        send Reject ch
      | Some (r, price) →
        send (Offer price) ch;
        match recv ch with
        | Accept →
          send (Resource r) ch
        | Reject → ()

let buyer name limit ch =
  send (Request name) ch;
  let rec loop () =
    match recv ch with
    | Reject      → None
    | Offer price →
      if price ≤ limit then
        send Accept ch;
        let Resource r = recv ch
         in Some r
      else
        send (Offer limit) ch;
        loop ()
  in loop ()
```

```
type bid_msg  = Request of string
              | Offer of int
              | Reject
              | Accept
              | Resource of resource
type bid_chan = bid_msg chan
```

```
let seller ch =
  let Request name = recv ch
   in match get name with
      | None             →
        send Reject ch
      | Some (r, price) →
        send (Offer price) ch;
        match recv ch with
        | Accept →
          send (Resource r) ch
        | Reject → ()
```

```
let buyer name limit ch =
  send (Request name) ch;
  let rec loop () =
    match recv ch with
    | Reject      → None
    | Offer price →
      if price ≤ limit then
        send Accept ch;
        let Resource r = recv ch
         in Some r
      else
        send (Offer limit) ch;
        loop ()
  in loop ()
```

```
type bid_msg  = Request of string
              | Offer of int
              | Reject
              | Accept
              | Resource of resource
type bid_chan = bid_msg chan
```

```
let seller ch =
  let Request name = recv ch
   in match get name with
      | None           →
        send Reject ch
      | Some (r, price) →
        send (Offer price) ch;
        match recv ch with
        | Accept →
          send (Resource r) ch
        | Reject → ()
```

```
let buyer name limit ch =
  send (Request name) ch;
  let rec loop () =
    match recv ch with
    | Reject      → None
    | Offer price →
      if price ≤ limit then
        send Accept ch;
        let Resource r = recv ch
         in Some r
      else
        send (Offer limit) ch;
        loop ()
   in loop ()
```

```
type bid_msg  = Request of string
              | Offer of int
              | Reject
              | Accept
              | Resource of resource
type bid_chan = bid_msg chan
```

```
let seller ch =
  let Request name = recv ch
   in match get name with
      | None            →
        send Reject ch
      | Some (r, price) →
        send (Offer price) ch;
        match recv ch with
        | Accept →
          send (Resource r) ch
        | Reject → ()
```

Pattern match failure

```
let buyer name limit ch =
  send (Request name) ch;
  let rec loop () =
    match recv ch with
    | Reject      → None
    | Offer price →
      if price ≤ limit then
        send Accept ch;
        let Resource r = recv ch
         in Some r
      else
        send (Offer limit) ch;
        loop ()
  in loop ()
```

```
type bid_msg  = Request of string
              | Offer of int
              | Reject
              | Accept
              | Resource of resource
type bid_chan = bid_msg chan
```

```
let seller ch =
  let Request name = recv ch
   in match get name with
      | None           →
        send Reject ch
      | Some (r, price) →
        send (Offer price) ch;
        match recv ch with
        | Accept →
          send (Resource r) ch
        | Reject → ()
```

```
let buyer name limit ch =
  send (Request name) ch;
  let rec loop () =
    match recv ch with
     | Reject       → None
     | Offer price →
       if price ≤ limit then
         send Accept ch;
         let Resource r = recv ch
          in Some r
       else
         send (Offer limit) ch;
         loop ()
    in loop ()
```

```
type bid_msg  = Request of string
              | Offer of int
              | Reject
              | Accept
              | Resource of resource
type bid_chan = bid_msg chan
```

Problem: Simple channel is too permissive

```
let seller ch =
  let Request name = recv ch
   in match get name with
      | None              →
        send Reject ch
      | Some (r, price) →
        send (Offer price) ch;
        match recv ch with
        | Accept →
          send (Resource r) ch
        | Reject → ()
```

```
let buyer name limit ch =
  send (Request name) ch;
  let rec loop () =
    match recv ch with
    | Reject        → None
    | Offer price →
      if price ≤ limit then
        send Accept ch;
        let Resource r = recv ch
         in Some r
      else
        send (Offer limit) ch;
        loop ()
  in loop ()
```

```
type bid_msg  = Request of string
              | Offer of int
              | Reject
              | Accept
              | Resource of resource
type bid_chan = bid_msg chan
```

Problem: Simple channel is too permissive

Solution: Session types

```
let seller ch =
  let Request name = recv ch
   in match get name with
      | None →
        send Reject ch
      | Some (r, price) →
        send (Offer price) ch;
        match recv ch with
        | Accept →
          send (Resource r) ch
        | Reject → ()
```

```
let buyer name limit ch =
  send (Request name) ch;
  let rec loop () =
    match recv ch with
    | Reject        → None
    | Offer price →
      if price ≤ limit then
        send Accept ch;
        let Resource r = recv ch
         in Some r
      else
        send (Offer limit) ch;
        loop ()
  in loop ()
```

Problem: Simple channel is too permissive

Solution: Session types

There's a language for that: Sing#

(Fähndrich et al. 2006)

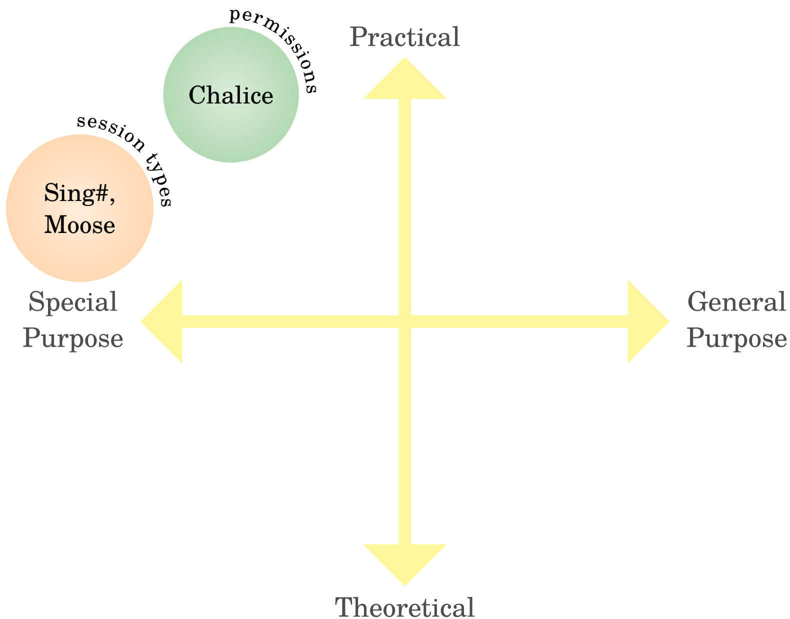
```
type bid_msg  = Request of string
              | Offer of int
              | Reject
              | Accept
              | Resource of resource
type bid_chan = bid_msg chan

let seller ch =
  let Request name = recv ch
   in match get name with
      | None →
        send Reject ch
      | Some (r, price) →
        send (Offer price) ch;
        match recv ch with
          | Accept →
            send (Resource r) ch
          | Reject → ()

let buyer name limit ch =
  send (Request name) ch;
  let rec loop () =
    match recv ch with
     | Reject      → None
     | Offer price →
       if price ≤ limit then
         send Accept ch;
         let Resource r = recv ch
          in Some r
       else
         send (Offer limit) ch;
         loop ()
  in loop ()
```

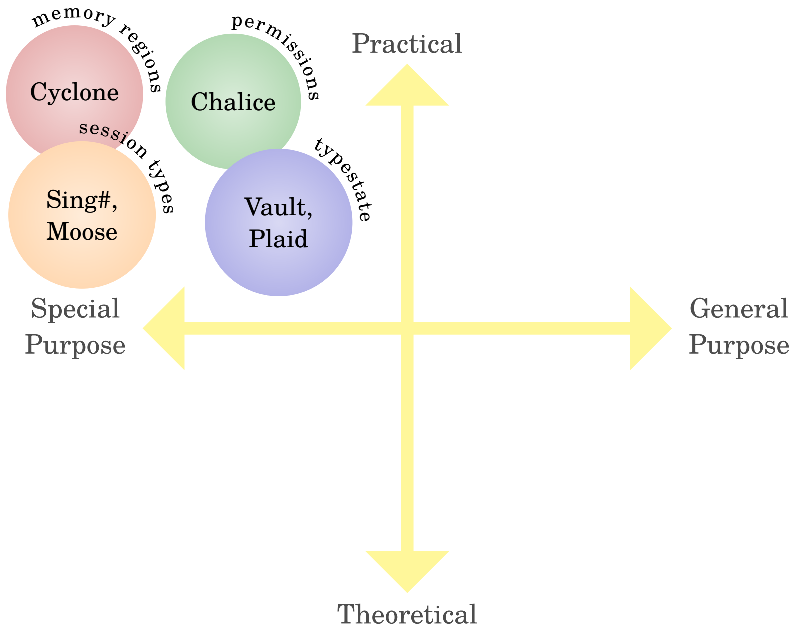
What's It Gonna Be?

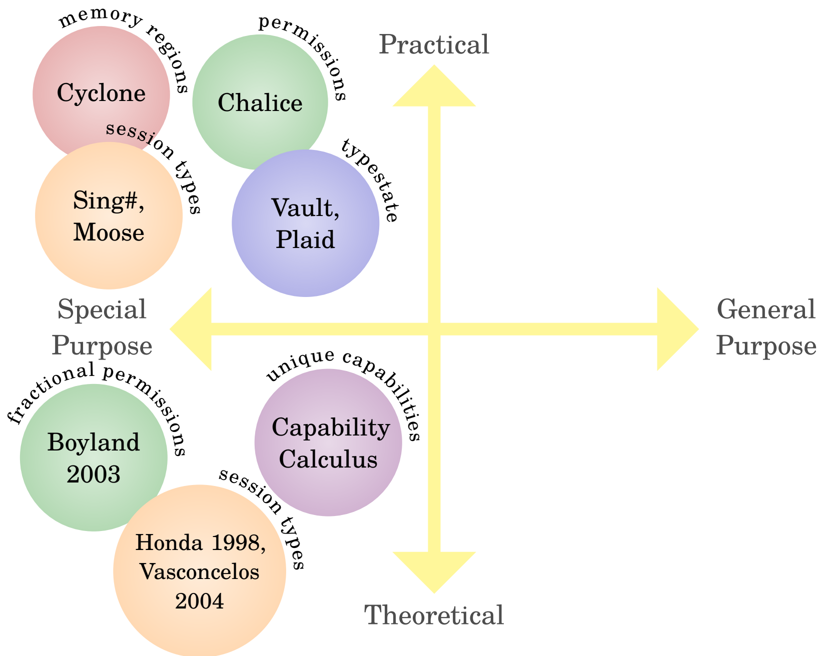
Problem:	Locking	Message passing
Solution:	Static permissions in Chalice	Session types in Sing#

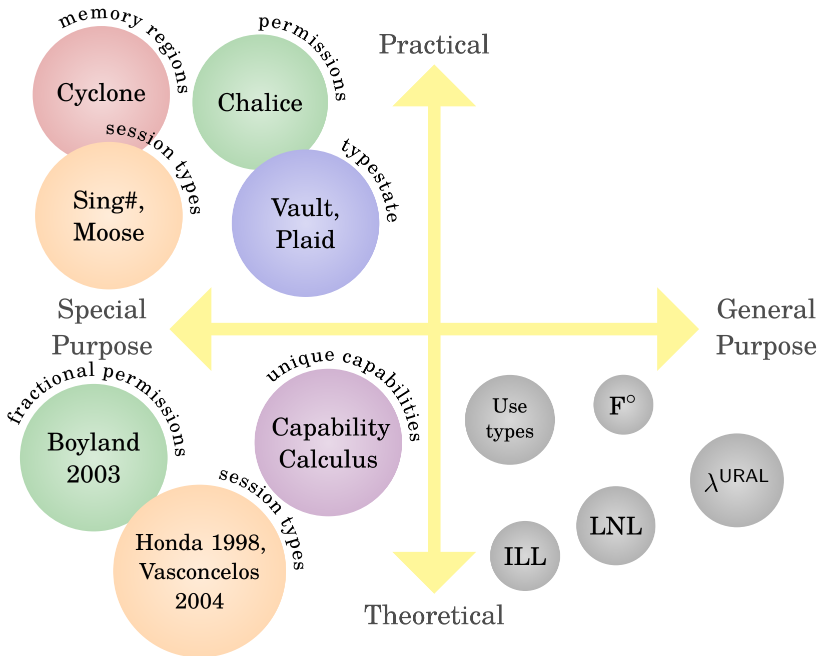
What's It Gonna Be?

Problem:	Locking	Message passing	Both
Solution:	Static permissions in Chalice	Session types in Sing#	?

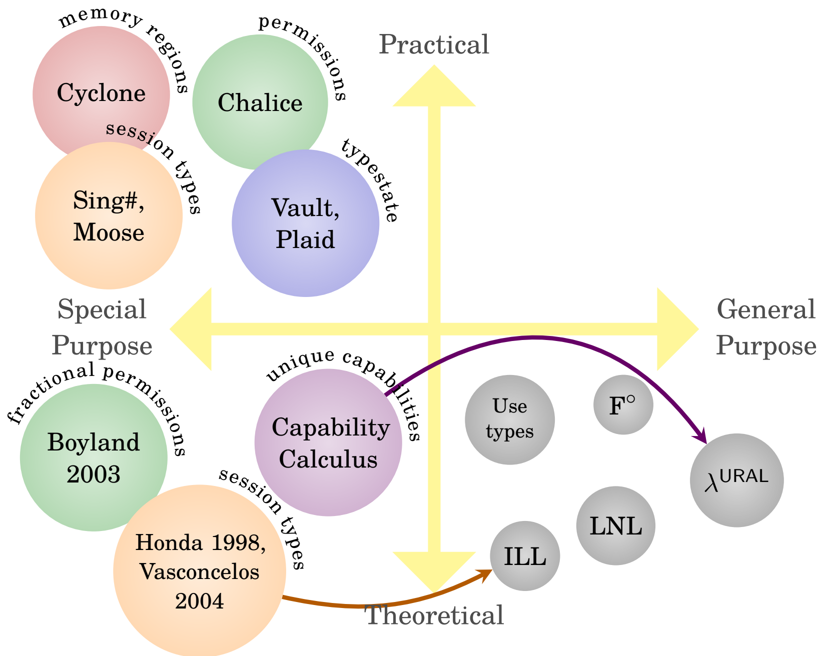

permissions
Practical
Chalice
session types
Sing#,
Moose
Special
Purpose
General
Purpose
Theoretical

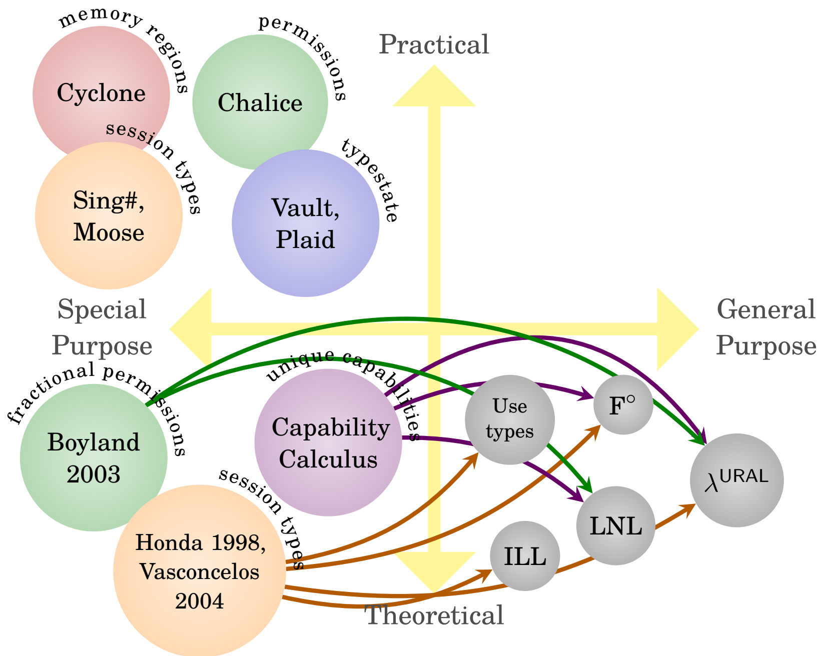

memory regions
Cyclone
permissions
Chalice
session types
Sing#,
Moose
typestate
Vault,
Plaid
Practical
Special
Purpose
General
Purpose
Theoretical


memory regions
Cyclone
permissions
Chalice
Practical
session types
Sing#,
Moose
typestate
Vault,
Plaid
Special
Purpose
General
Purpose
unique capabilities
Capability
Calculus
fractional permissions
Boyland
2003
session types
Honda 1998,
Vasconcelos
2004
Theoretical

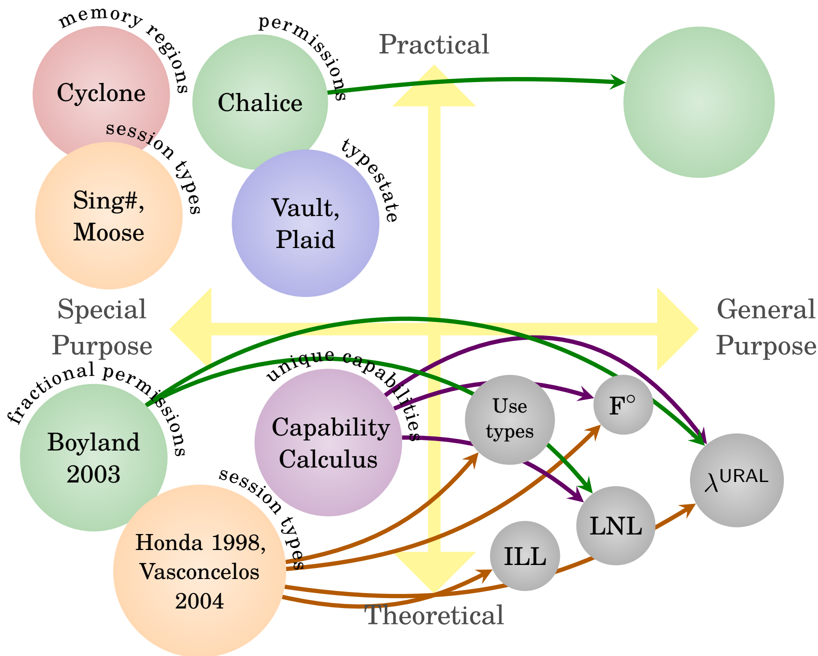


memory regions
Cyclone
permissions
Chalice
Practical
session types
Sing#,
Moose
typestate
Vault,
Plaid
Special
Purpose
General
Purpose
fractional permissions
Boyland
2003
unique capabilities
Capability
Calculus
Use
types
F°
λ^{URAL}
session types
Honda 1998,
Vasconcelos
2004
LNL
ILL
Theoretical

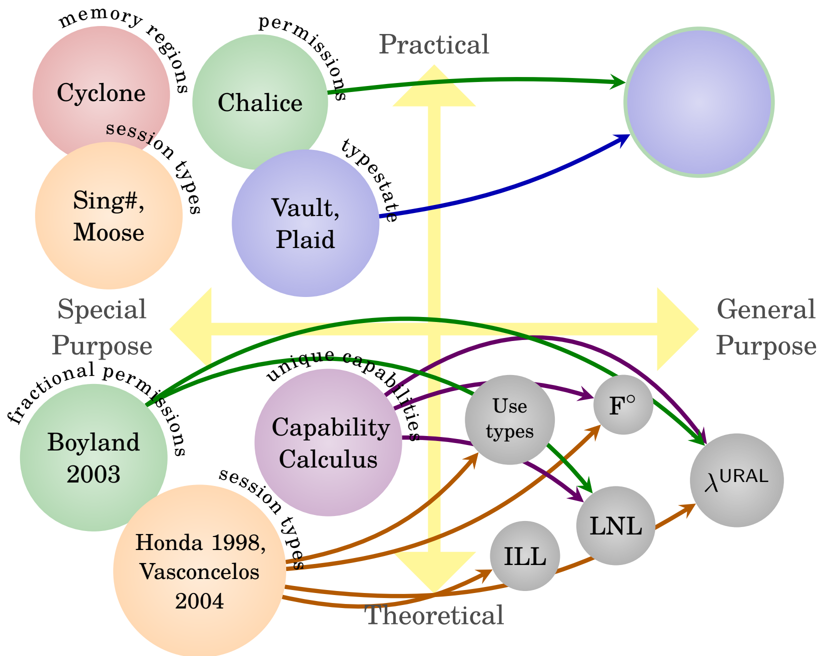

Practical
Theoretical
Special Purpose
General Purpose
memory regions
Cyclone
permissions
Chalice
session types
Sing#, Moose
typestate
Vault, Plaid
fractional permissions
Boyland 2003
unique capabilities
Capability Calculus
session types
Honda 1998, Vasconcelos 2004
Use types
F°
λ^URAL
LNL
ILL

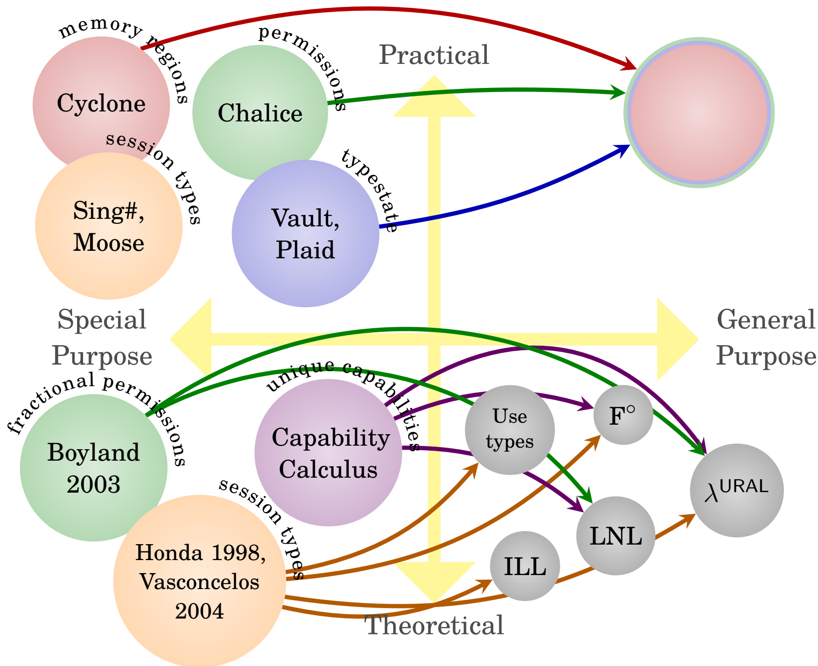


Practical
Theoretical
Special Purpose
General Purpose
memory regions
Cyclone
permissions
Chalice
session types
Sing#, Moose
typestate
Vault, Plaid
fractional permissions
Boyland 2003
unique capabilities
Capability Calculus
session types
Honda 1998, Vasconcelos 2004
Use types
F°
LNL
ILL
λ^URAL

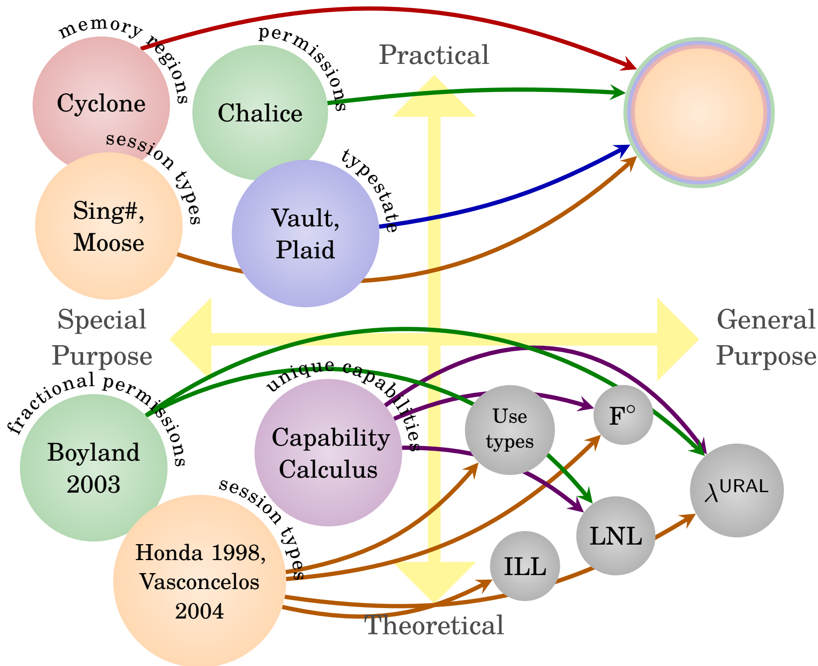


memory regions
Cyclone
permissions
Chalice
Practical
session types
Sing#,
Moose
typestate
Vault,
Plaid
Special
Purpose
General
Purpose
unique capabilities
fractional permissions
Boyland
2003
Capability
Calculus
Use
types
F°
λ^{URAL}
session types
Honda 1998,
Vasconcelos
2004
LNL
ILL
Theoretical

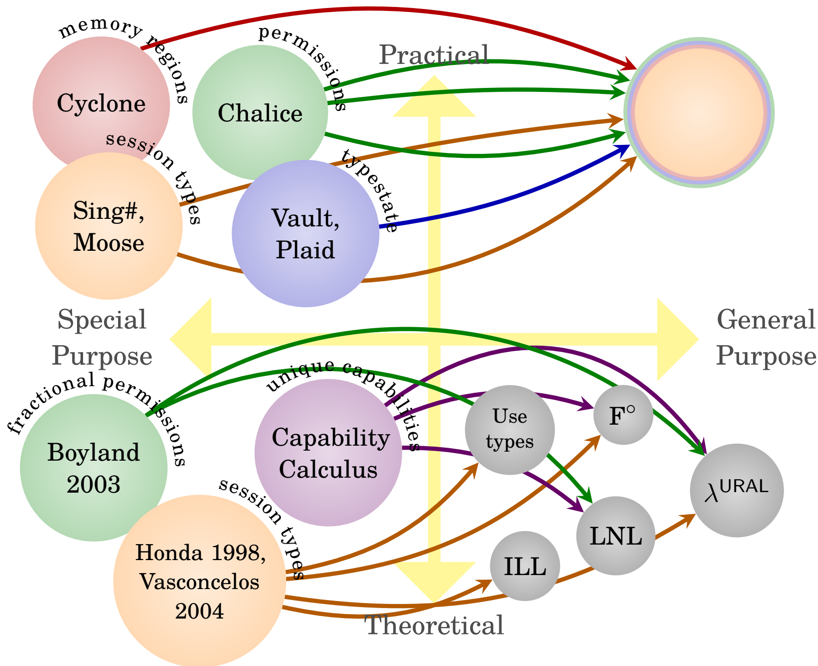

memory regions
Cyclone
permissions
Chalice
Practical
session types
Sing#,
Moose
typestate
Vault,
Plaid
Special
Purpose
General
Purpose
fractional permissions
unique capabilities
Boyland
2003
Capability
Calculus
Use
types
F°
λ^{URAL}
session types
Honda 1998,
Vasconcelos
2004
LNL
ILL
Theoretical

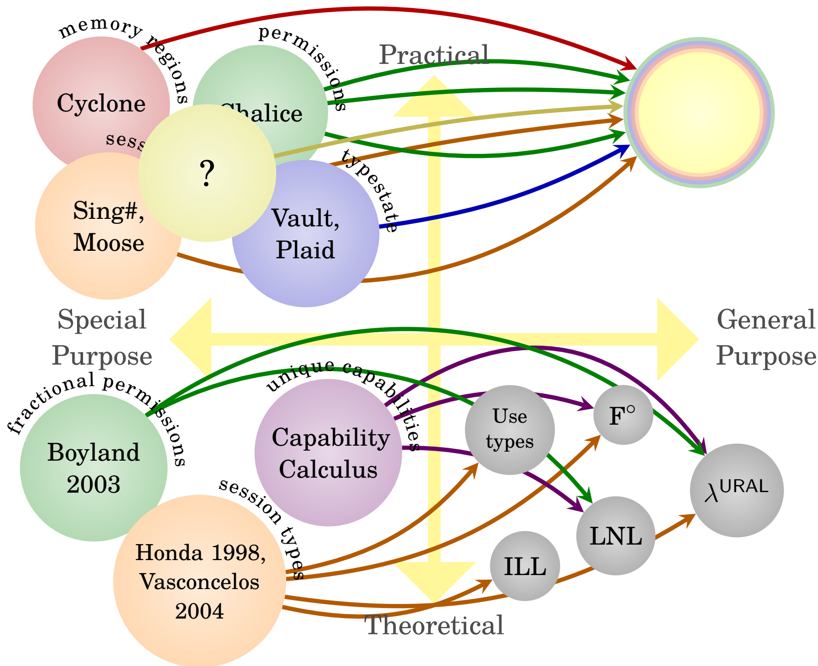


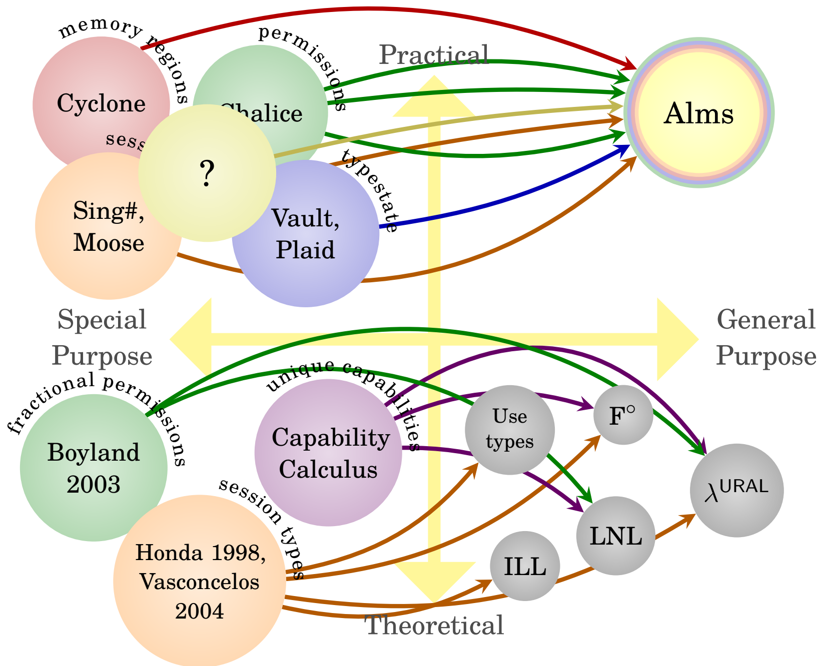
memory regions
Cyclone
permissions
Chalice
Practical
session types
Sing#,
Moose
typestate
Vault,
Plaid
Special
Purpose
General
Purpose
fractional permissions
unique capabilities
Boyland
2003
Capability
Calculus
Use
types
F°
λ^URAL
session types
Honda 1998,
Vasconcelos
2004
LNL
ILL
Theoretical



memory regions
Cyclone
permissions
Chalice
Practical
session types
Sing#, Moose
typestate
Vault, Plaid
Special Purpose
General Purpose
fractional permissions
unique capabilities
Boyland 2003
Capability Calculus
Use types
F°
λ^URAL
session types
Honda 1998, Vasconcelos 2004
LNL
ILL
Theoretical


memory regions
Cyclone
permissions
Chalice
Practical
session types
Sing#,
Moose
typestate
Vault,
Plaid
Special
Purpose
General
Purpose
fractional permissions
unique capabilities
Boyland
2003
Capability
Calculus
Use
types
F°
λ^{URAL}
session types
Honda 1998,
Vasconcelos
2004
LNL
ILL
Theoretical


memory regions
permissions
Practical
Cyclone
Chalice
sess
?
typestate
Sing#,
Moose
Vault,
Plaid
Special
Purpose
General
Purpose
unique capabilities
fractional permissions
Capability
Calculus
Use
types
F°
Boyland
2003
λ^URAL
session types
Honda 1998,
Vasconcelos
2004
LNL
ILL
Theoretical

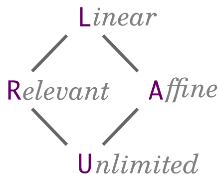


memory regions
permissions
Practical
Cyclone
Chalice
Alms
?
typestate
Sing#,
Moose
Vault,
Plaid
Special
Purpose
General
Purpose
fractional permissions
unique capabilities
Use
types
F°
Boyland
2003
Capability
Calculus
λ^{URAL}
session types
Honda 1998,
Vasconcelos
2004
LNL
ILL
Theoretical

Thesis

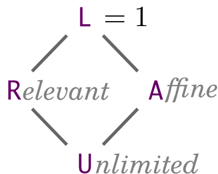
A programming language with
general-purpose substructural types
can be practical and expressive.

Thesis



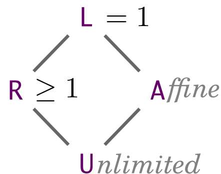
A programming language with general-purpose substructural types can be practical and expressive.

Thesis



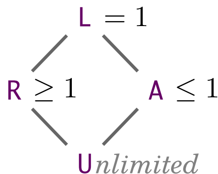
A programming language with
general-purpose substructural types
can be practical and expressive.

Thesis



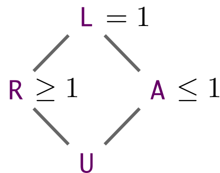
A programming language with general-purpose substructural types can be practical and expressive.

Thesis



A programming language with
general-purpose substructural types
can be practical and expressive.

Thesis



A programming language with general-purpose substructural types can be practical and expressive.

Thesis

A
|
U

A programming language with
general-purpose substructural types
can be practical and expressive.

Thesis

A
|
U

A programming language with general-purpose substructural types can be practical and expressive.

Thesis

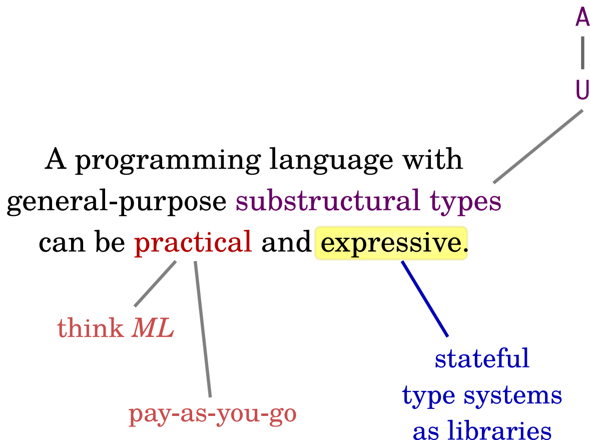
A
U

A programming language with general-purpose substructural types can be practical and expressive.

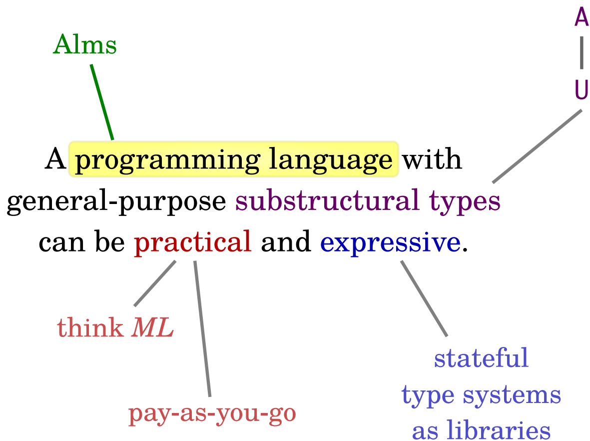
think *ML*

pay-as-you-go

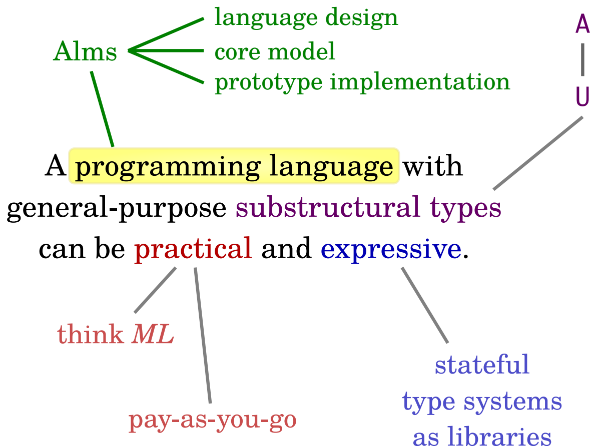
Thesis



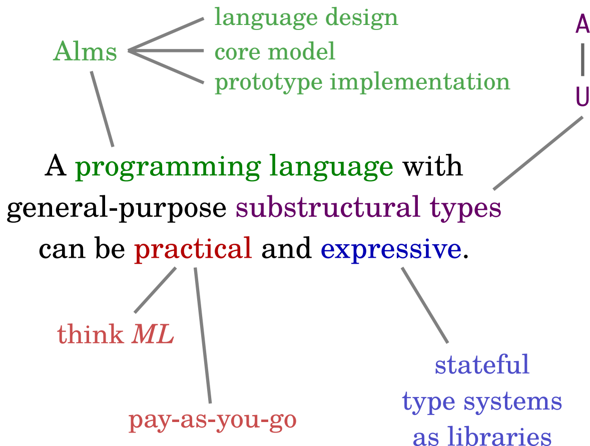
Thesis



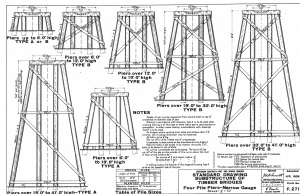
Thesis



Thesis

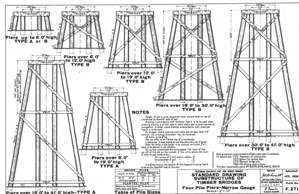


Alms By Example



Alms By Example

or Your Language Is a Library in My Language



Alms vs. OCaml

	OCaml	Alms
Affine types	No	Yes
Impredicative, rank-n	Sort of	Yes
Type inference	Yes	Yes
Modules	Yes	Yes
Opaque signatures	Yes	Yes
Algebraic data types	Yes	Yes
Pattern matching	Yes	Yes
Concurrency	Yes	Yes
Exceptions	Yes	Yes
Functors, classes, …	Yes	No

Example: Mutual Exclusion

```
let deposit (arr: int array) (acct: int) (amt: int) =
  Array.set arr acct (Array.get arr acct + amt)
```

Unlimited Arrays to Affine Arrays

```
module Array : sig
  type 'a array
  val new : int → 'a → 'a array
  val set : 'a array → int → 'a → unit
  val get : 'a array → int → 'a
end
```

Unlimited Arrays to Affine Arrays

```
module Array : sig
  type 'a array : U
  val new : int → 'a → 'a array
  val set : 'a array → int → 'a → unit
  val get : 'a array → int → 'a
end
```

Unlimited Arrays to Affine Arrays

```
module Array : sig
  type 'a array : U
  val new : int -U-> 'a -U-> 'a array
  val set : 'a array -U-> int -U-> 'a -U-> unit
  val get : 'a array -U-> int -U-> 'a
end
```

Unlimited Arrays to Affine Arrays

```
module Array : sig
  type 'a array : U
  val new : int -U-> 'a -U-> 'a array
  val set : 'a array -U-> int -U-> 'a -U-> unit
  val get : 'a array -U-> int -U-> 'a
end

module type AF_ARRAY = sig
  type 'a array : A
  val new : int -U-> 'a -U-> 'a array
  val set : 'a array -U-> int -A-> 'a -A-> 'a array
  val get : 'a array -U-> int -A-> 'a × 'a array
end
```

Unlimited Arrays to Affine Arrays

```
module Array : sig
  type 'a array : U
  val new : int -U-> 'a -U-> 'a array
  val set : 'a array -U-> int -U-> 'a -U-> unit
  val get : 'a array -U-> int -U-> 'a
end

module type AF_ARRAY = sig
  type 'a array : A
  val new : int -U-> 'a -U-> 'a array
  val set : 'a array -U-> int -A-> 'a -A-> 'a array
  val get : 'a array -U-> int -A-> 'a × 'a array
end
```

Unlimited Arrays to Affine Arrays

```
module Array : sig
  type 'a array : U
  val new : int -U-> 'a -U-> 'a array
  val set : 'a array -U-> int -U-> 'a -U-> unit
  val get : 'a array -U-> int -U-> 'a
end
```

```
module type AF_ARRAY = sig
  type 'a array : A
  val new : int -U-> 'a -U-> 'a array
  val set : 'a array -U-> int -A-> 'a -A-> 'a array
  val get : 'a array -U-> int -A-> 'a × 'a array
end
```

Unlimited Arrays to Affine Arrays

```
module Array : sig
  type 'a array : U
  val new : int → 'a → 'a array
  val set : 'a array → int → 'a → unit
  val get : 'a array → int → 'a
end

module type AF_ARRAY = sig
  type 'a array : A
  val new : int → 'a → 'a array
  val set : 'a array → int → 'a → 'a array
  val get : 'a array → int → 'a × 'a array
end
```

Unlimited Arrays to Affine Arrays

```
module Array : sig
  type 'a array : U
  val new : int → 'a → 'a array
  val set : 'a array → int → 'a → unit
  val get : 'a array → int → 'a
end

module type AF_ARRAY = sig
  type 'a array : A
  val new : int → 'a → 'a array
  val set : 'a array → int → 'a → 'a array
  val get : 'a array → int → 'a × 'a array
end
```

Unlimited Arrays to Affine Arrays

```
module Array : sig
  type 'a array : U
  val new : int → 'a → 'a array
  val set : 'a array → int → 'a → unit
  val get : 'a array → int → 'a
end

module type AF_ARRAY = sig
  type 'a array : A
  val new : int → 'a → 'a array
  val set : 'a array → int → 'a → 'a array
  val get : 'a array → int → 'a × 'a array
end
```

```
module AfArray : AF_ARRAY = struct



end
```

Unlimited Arrays to Affine Arrays

```
module Array : sig
  type 'a array : U
  val new : int → 'a → 'a array
  val set : 'a array → int → 'a → unit
  val get : 'a array → int → 'a
end

module type AF_ARRAY = sig
  type 'a array : A
  val new : int → 'a → 'a array
  val set : 'a array → int → 'a → 'a array
  val get : 'a array → int → 'a × 'a array
end
module AfArray : AF_ARRAY = struct
  type 'a array = 'a Array.array     (* U ⊑ A *)




end
```

Unlimited Arrays to Affine Arrays

```
module Array : sig
  type 'a array : U
  val new : int → 'a → 'a array
  val set : 'a array → int → 'a → unit
  val get : 'a array → int → 'a
end

module type AF_ARRAY = sig
  type 'a array : A
  val new : int → 'a → 'a array
  val set : 'a array → int → 'a → 'a array
  val get : 'a array → int → 'a × 'a array
end
module AfArray : AF_ARRAY = struct
  type 'a array = 'a Array.array
  let new = Array.new


end
```

Unlimited Arrays to Affine Arrays

```
module Array : sig
  type 'a array : U

  val new : int → 'a → 'a array
  val set : 'a array → int → 'a → unit
  val get : 'a array → int → 'a
end

module type AF_ARRAY = sig
  type 'a array : A

  val new : int → 'a → 'a array
  val set : 'a array → int → 'a → 'a array
  val get : 'a array → int → 'a × 'a array
end
```

```
module AfArray : AF_ARRAY = struct
  type 'a array = 'a Array.array

  let new = Array.new
  let set arr ix v = Array.set arr ix v; arr


end
```

Unlimited Arrays to Affine Arrays

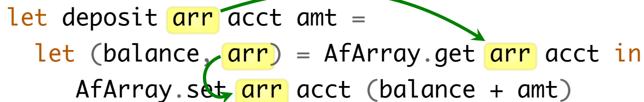
```
module Array : sig
  type 'a array : U

  val new : int → 'a → 'a array
  val set : 'a array → int → 'a → unit
  val get : 'a array → int → 'a
end

module type AF_ARRAY = sig
  type 'a array : A

  val new : int → 'a → 'a array
  val set : 'a array → int → 'a → 'a array
  val get : 'a array → int → 'a × 'a array
end
```

```
module AfArray : AF_ARRAY = struct
  type 'a array = 'a Array.array

  let new = Array.new
  let set arr ix v = Array.set arr ix v; arr
  let get arr ix = (Array.get arr ix, arr)
end
```

Using Affine Arrays

```
let deposit arr acct amt =
  let (balance, arr) = AfArray.get arr acct in
     AfArray.set arr acct (balance + amt)
```

Using Affine Arrays

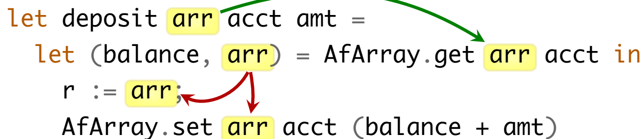


```
let deposit arr acct amt =
  let (balance, arr) = AfArray.get arr acct in
    AfArray.set arr acct (balance + amt)
```

Using Affine Arrays

```
let deposit arr acct amt =
  let (balance, arr) = AfArray.get arr acct in
    r := arr;
    AfArray.set arr acct (balance + amt)
```

Using Affine Arrays



Using Affine Arrays

```
let deposit arr acct amt =
  let (balance, arr) = AfArray.get arr acct in
    r := arr;
    AfArray.set arr acct (balance + amt)
```

```
Type error at afarray_test.alms:2:17-20:
    Qualifier inequality unsatisfiable:
    Attempted to use an affine type where only
    an unlimited type is permitted.
```

Affine Capabilities

```
module type CAP_ARRAY = sig
  type ('a,'t) array : U
  type 't cap : A
  val new : int → 'a → ∃'t. ('a,'t) array × 't cap
  val set : ('a,'t) array → int → 'a → 't cap → 't cap
  val get : ('a,'t) array → int → 't cap → 'a × 't cap
end
```

Affine Capabilities

```
module type CAP_ARRAY = sig
  type ('a,'t) array : U
  type 't cap : A
  val new : int → 'a → ∃'t. ('a,'t) array × 't cap
  val set : ('a,'t) array → int → 'a → 't cap → 't cap
  val get : ('a,'t) array → int → 't cap → 'a × 't cap
end
```

Affine Capabilities

```
module type CAP_ARRAY = sig
  type ('a,'t) array : U
  type 't cap : A
  val new : int → 'a → ∃'t. ('a,'t) array × 't cap
  val set : ('a,'t) array → int → 'a → 't cap → 't cap
  val get : ('a,'t) array → int → 't cap → 'a × 't cap
end
```

Affine Capabilities

```
module type CAP_ARRAY = sig
  type ('a,'t) array : U
  type 't cap : A
  val new : int → 'a → ∃'t. ('a,'t) array × 't cap
  val set : ('a,'t) array → int → 'a → 't cap → 't cap
  val get : ('a,'t) array → int → 't cap → 'a × 't cap
end
```

Affine Capabilities

```
module type CAP_ARRAY = sig
  type ('a,'t) array : U
  type 't cap : A
  val new : int → 'a → ∃'t. ('a,'t) array × 't cap
  val set : ('a,'t) array → int → 'a → 't cap → 't cap
  val get : ('a,'t) array → int → 't cap → 'a × 't cap
end
```

Affine Capabilities

```
module type CAP_ARRAY = sig
  type ('a,'t) array : U
  type 't cap : A
  val new : int → 'a → ∃'t. ('a,'t) array × 't cap
  val set : ('a,'t) array → int → 'a → 't cap → 't cap
  val get : ('a,'t) array → int → 't cap → 'a × 't cap
end
```

Affine Capabilities

```
module type CAP_ARRAY = sig
  type ('a,'t) array : U
  type 't cap : A
  val new : int → 'a → ∃'t. ('a,'t) array × 't cap
  val set : ('a,'t) array → int → 'a → 't cap → 't cap
  val get : ('a,'t) array → int → 't cap → 'a × 't cap
end

module CapArray : CAP_ARRAY = struct
  type ('a,'t) array = 'a Array.array
  type 't cap        = unit
  let new size init  = (Array.new size init, ())
  let set arr ix v _ = Array.set arr ix v
  let get arr ix _   = (Array.get arr ix, ())
end
```

Affine Capabilities

```
module type CAP_ARRAY = sig
  type ('a,'t) array : U
  type 't cap : A
  val new : int → 'a → ∃'t. ('a,'t) array × 't cap
  val set : ('a,'t) array → int → 'a → 't cap → 't cap
  val get : ('a,'t) array → int → 't cap → 'a × 't cap
end

module CapArray : CAP_ARRAY = struct
  type ('a,'t) array = 'a Array.array
  type 't cap        = unit
  let new size init  = (Array.new size init, ())
  let set arr ix v _ = Array.set arr ix v
  let get arr ix _   = (Array.get arr ix, ())
end
```

Affine Capabilities

```
module type CAP_ARRAY = sig
  type ('a,'t) array : U
  type 't cap : A
  val new : int → 'a → ∃'t. ('a,'t) array × 't cap
  val set : ('a,'t) array → int → 'a → 't cap → 't cap
  val get : ('a,'t) array → int → 't cap → 'a × 't cap
end

module CapArray : CAP_ARRAY = struct
  type ('a,'t) array = 'a Array.array
  type 't cap        = unit
  let new size init  = (Array.new size init, ())
  let set arr ix v _ = Array.set arr ix v
  let get arr ix _   = (Array.get arr ix, ())
end
```

Static Mandatory Locking

```
module type LOCK_ARRAY = sig
  type ('a,'t) array : U
  type 't cap : A
  val new : int → 'a → ∃'t. ('a,'t) array
  val set : ('a,'t) array → int → 'a → 't cap → 't cap
  val get : ('a,'t) array → int → 't cap → 'a × 't cap
  val acquire : ('a,'t) array → 't cap
  val release : ('a,'t) array → 't cap → unit
end
```

Static Mandatory Locking

```
module type LOCK_ARRAY = sig
  type ('a,'t) array : U
  type 't cap : A
  val new : int → 'a → ∃'t. ('a,'t) array
  val set : ('a,'t) array → int → 'a → 't cap → 't cap
  val get : ('a,'t) array → int → 't cap → 'a × 't cap
  val acquire : ('a,'t) array → 't cap
  val release : ('a,'t) array → 't cap → unit
end
```

Static Mandatory Locking

```
module type LOCK_ARRAY = sig
  type ('a,'t) array : U
  type 't cap : A
  val new : int → 'a → ∃'t. ('a,'t) array
  val set : ('a,'t) array → int → 'a → 't cap → 't cap
  val get : ('a,'t) array → int → 't cap → 'a × 't cap
  val acquire : ('a,'t) array → 't cap
  val release : ('a,'t) array → 't cap → unit
end
```

Static Mandatory Locking

```
module type LOCK_ARRAY = sig
  type ('a,'t) array : U
  type 't cap : A
  val new : int → 'a → ∃'t. ('a,'t) array
  val set : ('a,'t) array → int → 'a → 't cap → 't cap
  val get : ('a,'t) array → int → 't cap → 'a × 't cap
  val acquire : ('a,'t) array → 't cap
  val release : ('a,'t) array → 't cap → unit
end

let deposit arr acct amt =
  let arrcap = acquire arr in
  let (balance, arrcap) = get arr acc arrcap in
  let arrcap = set arr acct (balance + amt) arrcap in
    release arrcap
```

Static Mandatory Locking

```
module type LOCK_ARRAY = sig
  type ('a,'t) array : U
  type 't cap : A
  val new : int → 'a → ∃'t. ('a,'t) array
  val set : ('a,'t) array → int → 'a → 't cap → 't cap
  val get : ('a,'t) array → int → 't cap → 'a × 't cap
  val acquire : ('a,'t) array → 't cap
  val release : ('a,'t) array → 't cap → unit
end

let deposit arr acct amt =
  let arrcap = acquire arr in
  let (balance, arrcap) = get arr acc arrcap in
  let arrcap = set arr acct (balance + amt) arrcap in
    release arrcap
```

Static Mandatory Locking

```
module type LOCK_ARRAY = sig
  type ('a,'t) array : U
  type 't cap : A
  val new : int → 'a → ∃'t. ('a,'t) array
  val set : ('a,'t) array → int → 'a → 't cap → 't cap
  val get : ('a,'t) array → int → 't cap → 'a × 't cap
  val acquire : ('a,'t) array → 't cap
  val release : ('a,'t) array → 't cap → unit
end

let deposit arr acct amt =
  let arrcap = acquire arr in
  let (balance, arrcap) = get arr acc arrcap in
  let arrcap = set arr acct (balance + amt) arrcap in
    release arrcap
```

Static Mandatory Locking

```
module type LOCK_ARRAY = sig
  type ('a,'t) array : U
  type 't cap : A
  val new : int → 'a → ∃'t. ('a,'t) array
  val set : ('a,'t) array → int → 'a → 't cap → 't cap
  val get : ('a,'t) array → int → 't cap → 'a × 't cap
  val acquire : ('a,'t) array → 't cap
  val release : ('a,'t) array → 't cap → unit
end

let deposit arr acct amt =
  let arrcap = acquire arr in
  let (balance, arrcap) = get arr acc arrcap in
  let arrcap = set arr acct (balance + amt) arrcap in
    release arrcap
```

Static Mandatory Locking

```
module type LOCK_ARRAY = sig
  type ('a,'t) array : U
  type 't cap : A
  val new : int → 'a → ∃'t. ('a,'t) array
  val set : ('a,'t) array → int → 'a → 't cap → 't cap
  val get : ('a,'t) array → int → 't cap → 'a × 't cap
  val acquire : ('a,'t) array → 't cap
  val release : ('a,'t) array → 't cap → unit
end

let deposit arr acct amt =
  let !arrcap = acquire arr in
  let (balance, arrcap) = get arr acc arrcap in
  let arrcap = set arr acct (balance + amt) arrcap in
    release arrcap
```

Static Mandatory Locking

```
module type LOCK_ARRAY = sig
  type ('a,'t) array : U
  type 't cap : A

  val new : int → 'a → ∃'t. ('a,'t) array
  val set : ('a,'t) array → int → 'a → 't cap → 't cap
  val get : ('a,'t) array → int → 't cap → 'a × 't cap

  val acquire : ('a,'t) array → 't cap
  val release : ('a,'t) array → 't cap → unit
end

let deposit arr acct amt =
  let !arrcap = acquire arr in
  let (balance, arrcap) = get arr acc arrcap in
  let arrcap = set arr acct (balance + amt) arrcap in
    release arrcap
```

Static Mandatory Locking

```
module type LOCK_ARRAY = sig
  type ('a,'t) array : U
  type 't cap : A
  val new : int → 'a → ∃'t. ('a,'t) array
  val set : ('a,'t) array → int → 'a → 't cap → 't cap
  val get : ('a,'t) array → int → 't cap → 'a × 't cap
  val acquire : ('a,'t) array → 't cap
  val release : ('a,'t) array → 't cap → unit
end

let deposit arr acct amt =
  let !arrcap = acquire arr in
  let balance = get arr acc arrcap in
  let arrcap = set arr acct (balance + amt) arrcap in
    release arrcap
```

Static Mandatory Locking

```
module type LOCK_ARRAY = sig
  type ('a,'t) array : U
  type 't cap : A
  val new : int → 'a → ∃'t. ('a,'t) array
  val set : ('a,'t) array → int → 'a → 't cap → 't cap
  val get : ('a,'t) array → int → 't cap → 'a × 't cap
  val acquire : ('a,'t) array → 't cap
  val release : ('a,'t) array → 't cap → unit
end

let deposit arr acct amt =
  let !arrcap = acquire arr in
  let balance = get arr acc arrcap in
  let arrcap = set arr acct (balance + amt) arrcap in
    release arrcap
```

Static Mandatory Locking

```
module type LOCK_ARRAY = sig
  type ('a,'t) array : U
  type 't cap : A
  val new : int → 'a → ∃'t. ('a,'t) array
  val set : ('a,'t) array → int → 'a → 't cap → 't cap
  val get : ('a,'t) array → int → 't cap → 'a × 't cap
  val acquire : ('a,'t) array → 't cap
  val release : ('a,'t) array → 't cap → unit
end

let deposit arr acct amt =
  let !arrcap = acquire arr in
  let balance = get arr acc arrcap in
  set arr acct (balance + amt) ▷ arrcap;
  release arrcap
```

Static Mandatory Locking

```
module type LOCK_ARRAY = sig
  type ('a,'t) array : U
  type 't cap : A
  val new : int → 'a → ∃'t. ('a,'t) array
  val set : ('a,'t) array → int → 'a → 't cap → 't cap
  val get : ('a,'t) array → int → 't cap → 'a × 't cap
  val acquire : ('a,'t) array → 't cap
  val release : ('a,'t) array → 't cap → unit
end

let deposit arr acct amt =
  let !arrcap = acquire arr in
  let balance = get arr acc arrcap in
  set arr acct (balance + amt) ▷ arrcap;
  release ◁ arrcap
```

Static Mandatory Locking

```
module type LOCK_ARRAY = sig
  type ('a,'t) array : U
  type 't cap : A
  val new : int → 'a → ∃'t. ('a,'t) array
  val set : ('a,'t) array → int → 'a → 't cap → 't cap
  val get : ('a,'t) array → int → 't cap → 'a × 't cap
  val acquire : ('a,'t) array → 't cap
  val release : ('a,'t) array → 't cap → unit
end

let deposit arr acct amt =
  let !arrcap = acquire arr in
  let balance = get arr acc arrcap in
  set arr acct (balance + amt) ▷ arrcap;
  release ◁ arrcap
```

Static Mandatory Locking

```
module type LOCK_ARRAY = sig
  type ('a,'t) array : U
  type 't cap : A
  val new : int → 'a → ∃'t. ('a,'t) array
  val set : ('a,'t) array → int → 'a → 't cap → 't cap
  val get : ('a,'t) array → int → 't cap → 'a × 't cap
  val acquire : ('a,'t) array → 't cap
  val release : ('a,'t) array → 't cap → unit
end

let deposit arr acct amt =
  let !arrcap = acquire arr in
  let balance = get arr acc arrcap in
  set arr acct (balance + amt) ▷ arrcap;
  release ◁ arrcap
```

Problem: Advisory locking

Better solution: Static mandatory locking

Static Mandatory Locking

```
module type LOCK_ARRAY = sig
  type ('a,'t) array : U
  type 't cap : A

  val new : int → 'a → ∃'t. ('a,'t) array
  val set : ('a,'t) array → int → 'a → 't cap → 't cap
  val get : ('a,'t) array → int → 't cap → 'a × 't cap
  val acquire : ('a,'t) array → 't cap
  val release : ('a,'t) array → 't cap → unit
end

let deposit arr acct amt =
  let !arrcap = acquire arr in
  let balance = get arr acct arrcap in
  set arr acct (balance + amt) ▷ arrcap;
  release ◁ arrcap
```

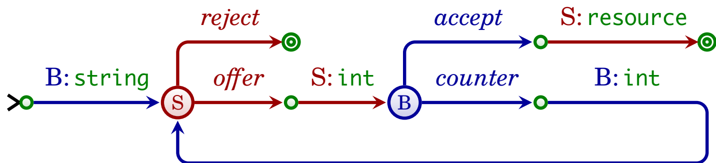
Problem: Advisory locking

Better solution: Static mandatory locking

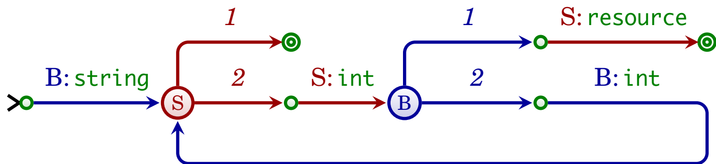
It's a library in Alms

(and it's not far to read/write locks, fractional permissions, ...)

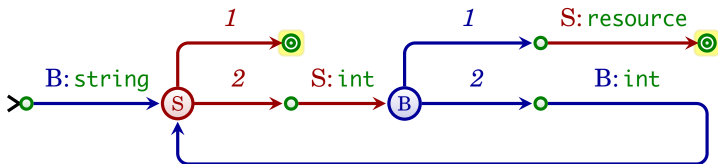
Example: Session Types



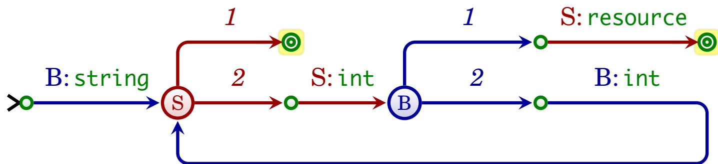
Example: Session Types



Example: Session Types

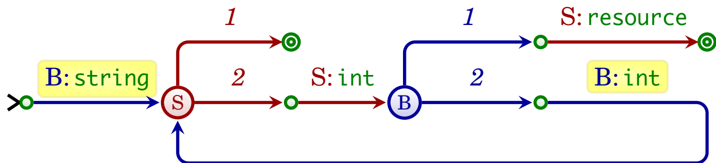


Example: Session Types



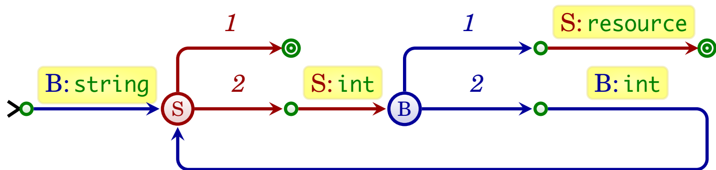

```
type done
```

Example: Session Types



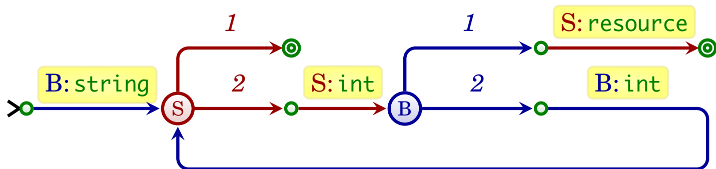

```
type done
```

Example: Session Types



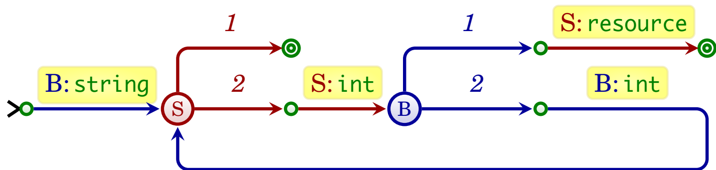

```
type done
```

Example: Session Types




```
type done
type 'a send              (* send an 'a *)
type 'a recv              (* receive an 'a *)
type ('a,'s) seq          (* do 'a, then 's *)
```

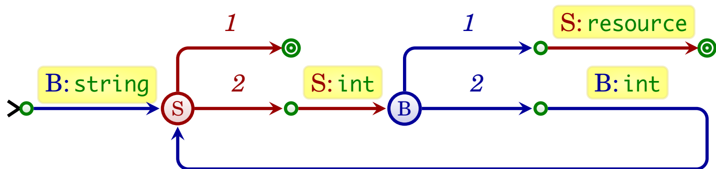
Example: Session Types




```
type done
type 'a send                (* send an 'a *)
type 'a recv                (* receive an 'a *)
type ('a,'s) seq            (* do 'a, then 's *)
```

```
(* Examples:
 *  (string recv, done) seq
 *  (int send, (string recv, done) seq) seq
```

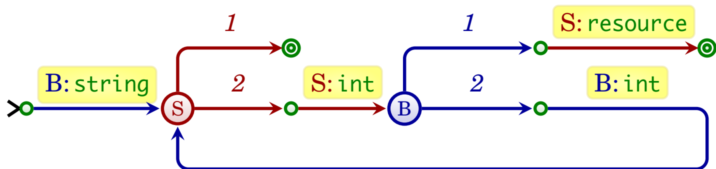
Example: Session Types




```
type done
type 'a (!)              (* send an 'a *)
type 'a (?)              (* receive an 'a *)
type ('a,'s) (;)         (* do 'a, then 's *)
```

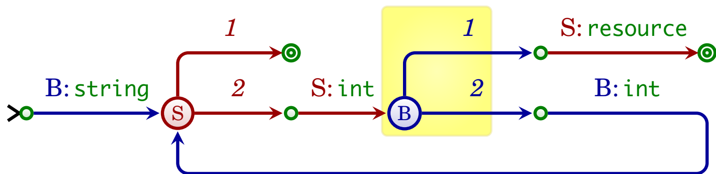
```
(* Examples:
 *  (string (?), done) (;)
 *  (int (?), (string (?), done) (;)) (;)
```

Example: Session Types



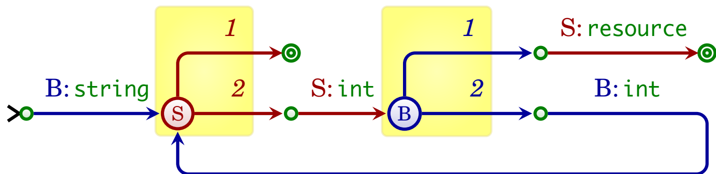

```
type done
type ! 'a                    (* send an 'a *)
type ? 'a                    (* receive an 'a *)
type 'a ; 's                 (* do 'a, then 's *)


(* Examples:
 *   ?string; done
 *   !int; ?string; done
```

Example: Session Types



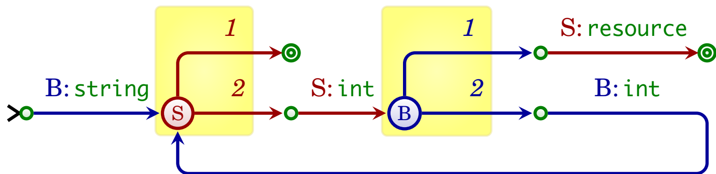

```
type done
type ! 'a                    (* send an 'a *)
type ? 'a                    (* receive an 'a *)
type 'a ; 's                 (* do 'a, then 's *)
```

Example: Session Types



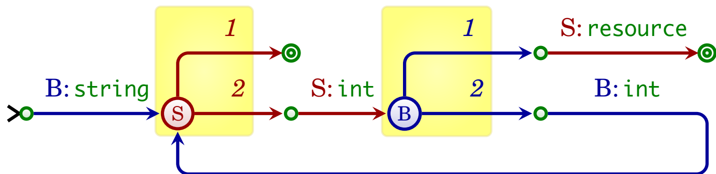

```
type done
type ! 'a              (* send an 'a *)
type ? 'a              (* receive an 'a *)
type 'a ; 's           (* do 'a, then 's *)
```

Example: Session Types



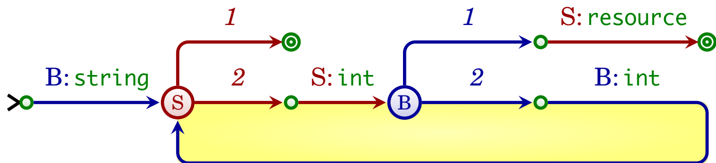

```
type done
type ! 'a            (* send an 'a *)
type ? 'a            (* receive an 'a *)
type 'a ; 's         (* do 'a, then 's *)
type 'r + 's         (* choose between 'r and 's *)
type 'r & 's         (* other side chooses *)
```

Example: Session Types



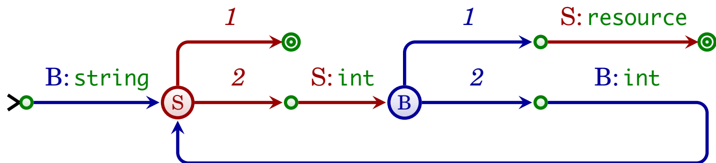

```
type done
type !  'a              (* send an 'a *)
type ?  'a              (* receive an 'a *)
type 'a ; 's            (* do 'a, then 's *)
type 'r + 's            (* choose between 'r and 's *)
type 'r & 's            (* other side chooses *)

(* Examples:
 *   (?string; done) + done
 *   !string; (done & (?int; done))
```

Example: Session Types



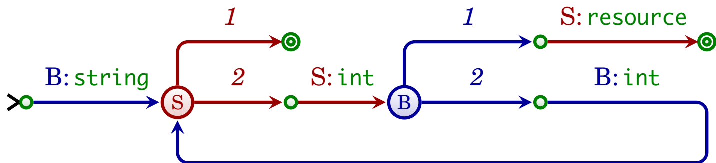

```
type done
type ! 'a                  (* send an 'a *)
type ? 'a                  (* receive an 'a *)
type 'a ; 's               (* do 'a, then 's *)
type 'r + 's               (* choose between 'r and 's *)
type 'r & 's               (* other side chooses *)
```

Example: Session Types



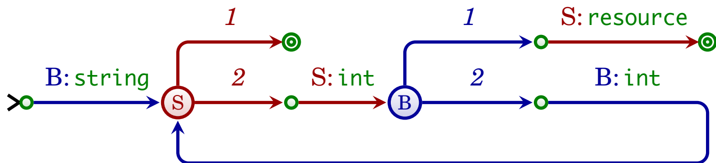

```
type done
type ! 'a                      (* send an 'a *)
type ? 'a                      (* receive an 'a *)
type 'a ; 's                   (* do 'a, then 's *)
type 'r + 's                   (* choose between 'r and 's *)
type 'r & 's                   (* other side chooses *)

type prot    = !string; offer
 and
```

Example: Session Types




```
type done
type ! 'a                 (* send an 'a *)
type ? 'a                 (* receive an 'a *)
type 'a ; 's              (* do 'a, then 's *)
type 'r + 's              (* choose between 'r and 's *)
type 'r & 's              (* other side chooses *)

type prot     = !string; offer
 and offer    = done & (?int; respond)
 and
```

Example: Session Types




```
type done
type ! 'a                    (* send an 'a *)
type ? 'a                    (* receive an 'a *)
type 'a ; 's                 (* do 'a, then 's *)
type 'r + 's                 (* choose between 'r and 's *)
type 'r & 's                 (* other side chooses  *)

type prot     = !string; offer
 and offer    = done & (?int; respond)
 and respond = (?resource; done) + (!int; offer)
```

Session Type Operations

```
module type SESSION_TYPES = sig
  type done
  type !'a               (* send an 'a *)
  type ?'a               (* receive an 'a *)
  type 'a ; 's           (* do 'a, then 's *)
  type 'r + 's           (* choose between 'r and 's *)
  type 'r & 's           (* other side chooses  *)

  ⋮
end
```

Session Type Operations

```
module type SESSION_TYPES = sig
  type done
  type ! 'a                   (* send an 'a *)
  type ? 'a                   (* receive an 'a *)
  type 'a ; 's                (* do 'a, then 's *)
  type 'r + 's                (* choose between 'r and 's *)
  type 'r & 's                (* other side chooses  *)
  type 's chan : A

  ⋮
end
```

Session Type Operations

```
module type SESSION_TYPES = sig
  type done
  type !‾ 'a                   (* send an 'a *)
  type ?‾ 'a                   (* receive an 'a *)
  type 'a ;‾ 's                (* do 'a, then 's *)
  type 'r +‾ 's                (* choose between 'r and 's *)
  type 'r &‾ 's                (* other side chooses  *)

  type 's chan : A

  val send   : 'a → (!‾'a;‾ 's) chan → 's chan
  val recv   : (?‾'a;‾ 's) chan → 'a × 's chan

  ⋮
end
```

Session Type Operations

```
module type SESSION_TYPES = sig
  type done
  type ! 'a                     (* send an 'a *)
  type ? 'a                     (* receive an 'a *)
  type 'a ; 's                  (* do 'a, then 's *)
  type 'r + 's                  (* choose between 'r and 's *)
  type 'r & 's                  (* other side chooses *)

  type 's chan : A

  val send   : 'a → (!'a; 's) chan → 's chan
  val recv   : (?'a; 's) chan → 'a × 's chan
  val sel1   : ('s + 't) chan → 's chan
  val sel2   : ('s + 't) chan → 't chan

  ⋮
end
```

Session Type Operations

```
module type SESSION_TYPES = sig
  type done
  type ! 'a                   (* send an 'a *)
  type ? 'a                   (* receive an 'a *)
  type 'a ; 's                (* do 'a, then 's *)
  type 'r + 's                (* choose between 'r and 's *)
  type 'r & 's                (* other side chooses  *)

  type 's chan : A

  val send   : 'a → (!'a; 's) chan → 's chan
  val recv   : (?'a; 's) chan → 'a × 's chan
  val sel1   : ('s + 't) chan → 's chan
  val sel2   : ('s + 't) chan → 't chan
  val follow : ('s & 't) chan → ('s chan, 't chan) either
  ⋮
end
```

Buyer, Take 2

```
type prot     = !string; offer
 and offer    = done & (?int; respond)
 and respond = (?resource; done) + (!int; offer)

let buyer name limit !ch =
  send name ▷ ch;
  let rec loop () =
    match follow ◁ ch with
    | Left ch  → None
    | Right ch →
      if recv ch ≤ limit then
        sel1 ▷ ch; Some (recv ch)
      else
        sel2 ▷ ch; send limit ▷ ch;
        loop ()
    in loop ()
```

Buyer, Take 2

```
type prot     = !string; offer
 and offer    = done & (?int; respond)
 and respond = (?resource; done) + (!int; offer)

let buyer name limit !ch =
  send name ▷ ch;
  let rec loop () =
    match follow ◁ ch with
    | Left ch  → None
    | Right ch →
      if recv ch ≤ limit then
        sel1 ▷ ch; Some (recv ch)
      else
        sel2 ▷ ch; send limit ▷ ch;
        loop ()
    in loop ()
```

Buyer, Take 2

```
type prot     = !string; offer
 and offer    = done & (?int; respond)
 and respond = (?resource; done) + (!int; offer)

let buyer name limit !ch =
  send name ▷ ch;
  let rec loop () =
    match follow ◁ ch with
    | Left ch   → None
    | Right ch  →
      if recv ch ≤ limit then
        sel1 ▷ ch; Some (recv ch)
      else
        sel2 ▷ ch; send limit ▷ ch;
        loop ()
    in loop ()
```

Buyer, Take 2

```
type prot     = !string; offer
 and offer    = done & (?int; respond)
 and respond = (?resource; done) + (!int; offer)

let buyer name limit !ch =
  send name ▷ ch;
  let rec loop () =
    match follow ◁ ch with
    | Left ch   → None
    | Right ch  →
      if recv ch ≤ limit then
        sel1 ▷ ch; Some (recv ch)
      else
        sel2 ▷ ch; send limit ▷ ch;
        loop ()
    in loop ()
```

Buyer, Take 2

```
type prot     = !string; offer
 and offer    = done & (?int; respond)
 and respond = (?resource; done) + (!int; offer)

let buyer name limit !ch =
  send name ▷ ch;
  let rec loop () =
    match follow ◁ ch with
    | Left ch  → None
    | Right ch →
      if recv ch ≤ limit then
        sel1 ▷ ch; Some (recv ch)
      else
        sel2 ▷ ch; send limit ▷ ch;
        loop ()
    in loop ()
```

Buyer, Take 2

```
type prot     = !string; offer
 and offer    = done & (?int; respond)
 and respond = (?resource; done) + (!int; offer)

let buyer name limit !ch =
  send name ▷ ch;
  let rec loop () =
    match follow ◁ ch with
    | Left ch  → None
    | Right ch →
      if recv ch ≤ limit then
        sel1 ▷ ch; Some (recv ch)
      else
        sel2 ▷ ch; send limit ▷ ch;
        loop ()
    in loop ()
```

Buyer, Take 2

```
type prot     = !string; offer
 and offer    = done & (?int; respond)
 and respond = (?resource; done) + (!int; offer)

let buyer name limit !ch =
  send name ▷ ch;
  let rec loop () =
    match follow ◁ ch with
    | Left ch   → None
    | Right ch  →
      if recv ch ≤ limit then
        sel1 ▷ ch; Some (recv ch)
      else
        sel2 ▷ ch; send limit ▷ ch;
        loop ()
    in loop ()
```

Buyer, Take 2

```
type prot     = !string; offer
 and offer    = done & (?int; respond)
 and respond = (?resource; done) + (!int; offer)

let buyer name limit !ch =
  send name ▷ ch;
  let rec loop () =
    match follow ◁ ch with
    | Left ch  → None
    | Right ch →
      if recv ch ≤ limit then
        sel1 ▷ ch; Some (recv ch)
      else
        sel2 ▷ ch; send limit ▷ ch;
        loop ()
    in loop ()
```

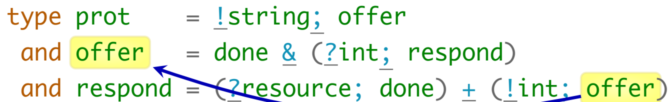
Buyer, Take 2

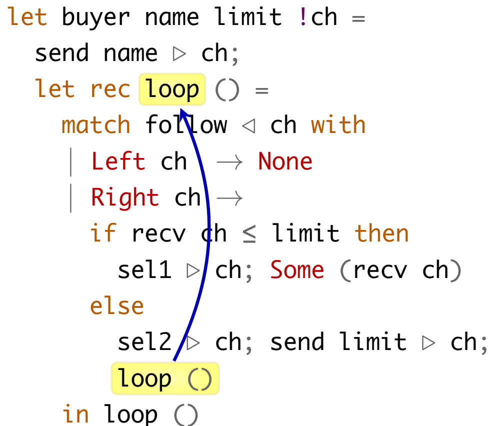
```
type prot     = !string; offer
 and offer    = done & (?int; respond)
 and respond = (?resource; done) + (!int; offer)

let buyer name limit !ch =
  send name ▷ ch;
  let rec loop () =
    match follow ◁ ch with
    | Left ch   → None
    | Right ch  →
      if recv ch ≤ limit then
        sel1 ▷ ch; Some (recv ch)
      else
        sel2 ▷ ch; send limit ▷ ch;
        loop ()
    in loop ()
```

Buyer, Take 2

```
type prot     = !string; offer
 and offer    = done & (?int; respond)
 and respond = (?resource; done) + (!int; offer)

let buyer name limit !ch =
  send name ▷ ch;
  let rec loop () =
    match follow ◁ ch with
    | Left ch  → None
    | Right ch →
      if recv ch ≤ limit then
        sel1 ▷ ch; Some (recv ch)
      else
        sel2 ▷ ch; send limit ▷ ch;
        loop ()
    in loop ()
```

Buyer, Take 2





Buyer, Take 2

```
type prot     = !string; offer
 and offer    = done & (?int; respond)
 and respond = (?resource; done) + (!int; offer)

let buyer name limit !(ch: prot chan) =
  send name ▷ ch;
  let rec loop () =
    match follow ◁ ch with
    | Left ch  → None
    | Right ch →
      if recv ch ≤ limit then
        sel1 ▷ ch; Some (recv ch)
      else
        sel2 ▷ ch; send limit ▷ ch;
        loop ()
    in loop ()
```

Buyer, Take 2

```
type prot     = !string; offer
 and offer    = done & (?int; respond)
 and respond = (?resource; done) + (!int; offer)

let buyer name limit !(ch: prot chan) =
  send name ▷ ch;
  let rec loop () =
    match follow ◁ ch with
    | Left ch  → None
    | Right ch →
      if recv ch ≤ limit then
        sel2 ▷ ch; Some (recv ch)
      else
        sel1 ▷ ch; send limit ▷ ch;
        loop ()
    in loop ()
```

Buyer, Take 2

```
type prot     = !string; offer
 and offer    = done & (?int; respond)
 and respond  = (?resource; done) + (!int; offer)

let buyer name limit !(ch: prot chan) =
  send name ▷ ch;
  let rec loop () =
    match follow ◁ ch wi
    | Left ch  → None
    | Right ch →
      if recv ch ≤ limit then
        sel2 ▷ ch; Some (recv ch)
      else
        sel1 ▷ ch; send limit ▷ ch;
        loop ()
    in loop ()
```

```
Type error at buyer.alms:12:31-33:
  Cannot subtype:
    actual:   !int; offer
    expected: ?'_a; '_b
```

Buyer, Take 2

```
type prot     = !string; offer
 and offer    = done & (?int; respond)
 and respond = (?resource; done) + (!int; offer)

let buyer name limit !(ch: prot chan) =
  send name ▷ ch;
  let rec loop () =
    match follow ◁ ch with
    | Left ch  → None
    | Right ch →
      if recv ch ≤ limit then
        sel1 ▷ ch; Some (recv ch)
      else
        sel2 ▷ ch; send limit ▷ ch;
        loop ()
    in loop ()
```

Buyer, Take 2

```
type prot     = !string; offer
 and offer    = done & (?int; respond)
 and respond = (?resource; done) + (!int; offer)

let buyer name limit !(ch: prot chan) =
  send name ▷ ch;
  let rec loop () =
    match follow ◁ ch with
    | Left ch  → None
    | Right ch →
      if recv ch ≤ limit then
        sel1 ▷ ch; Some (recv ch)
      else
        sel2 ▷ ch; send limit ▷ ch;
        loop ()
    in loop ()
```

Problem: Weak channels

Better solution: Session types

Buyer, Take 2

```
type prot     = !string; offer
 and offer    = done & (?int; respond)
 and respond = (?resource; done) + (!int; offer)

let buyer name limit !(ch: prot chan) =
  send name ▷ ch;
  let rec loop () =
    match follow ◁ ch with
    | Left ch   → None
    | Right ch →
      if recv ch ≤ limit then
        sel1 ▷ ch; Some (recv ch)
      else
        sel2 ▷ ch; send limit ▷ ch;
        loop ()
  in loop ()
```

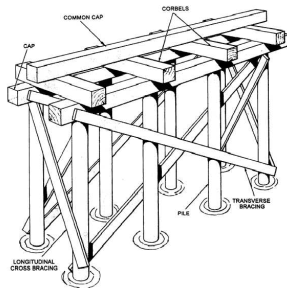
Problem: Weak channels

Better solution: Session types

There are two libraries to choose from in Alms

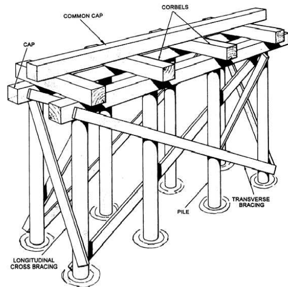
(or you can roll your own)

Design Pragmatics



Design Pragmatics

or How to Be Affine without Being Awkward



The Problem

Uncurry in ML:

```
λf (x,y) → f x y
  : ('a → 'b → 'c) → 'a × 'b → 'c
```

The Problem

Uncurry in ML:

```
λf (x,y) → f x y
  : ('a → 'b → 'c) → 'a × 'b → 'c
```

Uncurry in ILL (Bierman 1993):

The Problem

Uncurry in ML:

```
λf (x,y) → f x y
  : ('a → 'b → 'c) → 'a × 'b → 'c
```

Uncurry in ILL (Bierman 1993):

```
λf → promote (λp →
                 let (x, y) = derelict p in
                   derelict (derelict f x) y)
  : !('a ⊸ !('b ⊸ 'c)) ⊸ !(!('a ⊗ 'b) ⊸ 'c)
```

The Problem

Uncurry in ML:

```
λf (x,y) → f x y
  : ('a → 'b → 'c) → 'a × 'b → 'c
```

Uncurries in ILL (Bierman 1993):

```
λf → promote (λp →
                let (x, y) = derelict p in
                  derelict (derelict f x) y)
  : !('a ⊸ !('b ⊸ 'c)) ⊸ !(!('a ⊗ 'b) ⊸ 'c)
```

```
λf p → let (x, y) = derelict p in
         derelict (f x) y
  : ('a ⊸ !('b ⊸ 'c)) ⊸ !('a ⊗ 'b) ⊸ 'c
```

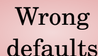
The Problem

Uncurry in ML:

```
λf (x,y) → f x y
  : ('a → 'b → 'c) → 'a × 'b → 'c
```

Uncurries in ILL (Bierman 1993):

```
λf → promote (λp →
                  let (x, y) = derelict p in
                    derelict (derelict f x) y)
  : !('a ⊸ !('b ⊸ 'c)) ⊸ !(!('a ⊗ 'b) ⊸ 'c)

λf p → let (x, y) = derelict p in
          derelict (f x) y
  : ('a ⊸ !('b ⊸ 'c)) ⊸ !('a ⊗ 'b) ⊸ 'c
```



The Problem

Uncurry in ML:

```
λf (x,y) → f x y
  : ('a → 'b → 'c) → 'a × 'b → 'c
```

Uncurries in ILL (Bierman 1993):

```
λf → promote (λp →
                  let (x, y) = derelict p in
                   derelict (derelict f x) y)
  : !('a ⊸ !('b ⊸ 'c)) ⊸ !(!('a ⊗ 'b) ⊸ 'c)

λf p → let (x, y) = derelict p in
            derelict (f x) y
  : ('a ⊸ !('b ⊸ 'c)) ⊸ !('a ⊗ 'b) ⊸ 'c
```

Wrong defaults

Frightening types

The Problem

Uncurry in ML:

```
λf (x,y) → f x y
  : ('a → 'b → 'c) → 'a × 'b → 'c
```

Uncurries in ILL (Bierman 1993):

```
λf → promote (λp →
              let (x, y) = derelict p in
               derelict (derelict f x) y)
  : !('a ⊸ !('b ⊸ 'c)) ⊸ !(!('a ⊗ 'b) ⊸ 'c)

λf p → let (x, y) = derelict p in
         derelict (f x) y
  : ('a ⊸ !('b ⊸ 'c)) ⊸ !('a ⊗ 'b) ⊸ 'c
```

Wrong defaults

Frightening types

Derelict & promote

The Problem

Uncurry in ML:

```
λf (x,y) → f x y
  : ('a → 'b → 'c) → 'a × 'b → 'c
```

Uncurries in ILL (Bierman 1993):

```
λf → promote (λp →
                 let (x, y) = derelict p in
                   derelict (derelict f x) y)
  : !('a ⊸ !('b ⊸ 'c)) ⊸ !(!('a ⊗ 'b) ⊸ 'c)
```

```
λf p → let (x, y) = derelict p in
         derelict (f x) y
  : ('a ⊸ !('b ⊸ 'c)) ⊸ !('a ⊗ 'b) ⊸ 'c
```

Too much repetition

Wrong defaults

Frightening types

Derelict & promote

The Problem

Uncurry in ML:

```
λf (x,y) → f x y
  : ('a → 'b → 'c) → 'a × 'b → 'c
```

Uncurries in ILL (Bierman 1993):

```
λf → promote (λp →
                 let (x, y) = derelict p in
                   derelict (derelict f x) y)
  : !('a ⊸ !('b ⊸ 'c)) ⊸ !(!('a ⊗ 'b) ⊸ 'c)
```

Usage kinds

Dependent kinds

Dereliction subtyping

Principal promotion

Type inference

```
et  e
d   x
  : ('a ⊸ !('b ⊸ 'c)) ⊸ !('a ⊗ 'b) ⊸ 'c
```

Too much repetition

Wrong defaults

Frightening types

Derelict & promote

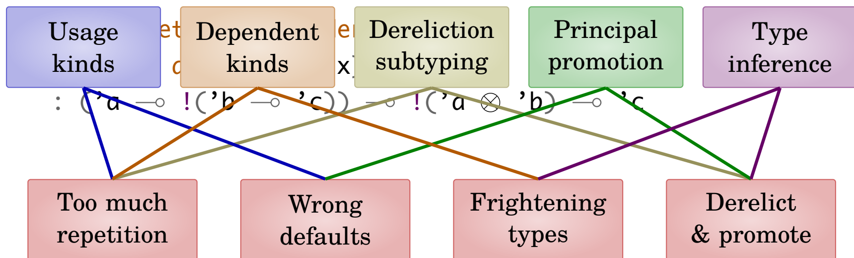
The Problem

Uncurry in ML:

```
λf (x,y) → f x y
  : ('a → 'b → 'c) → 'a × 'b → 'c
```

Uncurries in ILL (Bierman 1993):

```
λf → promote (λp →
                 let (x, y) = derelict p in
                   derelict (derelict f x) y)
  : !('a ⊸ !('b ⊸ 'c)) ⊸ !(!('a ⊗ 'b) ⊸ 'c)
```



The Problem

Uncurry in ML:

```
λf (x,y) → f x y
  : ('a → 'b → 'c) → 'a × 'b → 'c
```

Uncurries in ILL (Bierman 1993):

```
λf → promote (λp →
                 let (x, y) = derelict p in
                   derelict (derelict f x) y)
  : !('a ⊸ !('b ⊸ 'c)) ⊸ !(!('a ⊗ 'b) ⊸ 'c)
```

```
λf p → let (x, y) = derelict p in
          derelict (f x) y
  : ('a ⊸ !('b ⊸ 'c)) ⊸ !('a ⊗ 'b) ⊸ 'c
```

Uncurry in Alms:

```
λf (x,y) → f x y
```

$: (\text{“}a \xrightarrow{\text{“}d} \text{“}b \xrightarrow{A} \text{“}c) \to \text{“}a \times \text{“}b \xrightarrow{\text{“}d} \text{“}c$

The Problem

Uncurry in ML:

```
λf (x,y) → f x y
  : ('a → 'b → 'c) → 'a × 'b → 'c
```

Uncurries in ILL (Bierman 1993):

```
λf → promote (λp →
                let (x, y) = derelict p in
                  derelict (derelict f x) y)
  : !('a ⊸ !('b ⊸ 'c)) ⊸ !(!('a ⊗ 'b) ⊸ 'c)
```

```
λf p → let (x, y) = derelict p in
         derelict (f x) y
  : ('a ⊸ !('b ⊸ 'c)) ⊸ !('a ⊗ 'b) ⊸ 'c
```

Uncurry in Alms:

```
λf (x,y) → f x y
  : (`a -`d-> `b -A-> `c) → `a × `b -`d-> `c
```

The Problem

Uncurry in ML:

```
λf (x,y) → f x y
  : ('a → 'b → 'c) → 'a × 'b → 'c
```

Uncurries in ILL (Bierman 1993):

```
λf → promote (λp →
                let (x, y) = derelict p in
                  derelict (derelict f x) y)
  : !('a ⊸ !('b ⊸ 'c)) ⊸ !(!('a ⊗ 'b) ⊸ 'c)

λf p → let (x, y) = derelict p in
         derelict (f x) y
  : ('a ⊸ !('b ⊸ 'c)) ⊸ !('a ⊗ 'b) ⊸ 'c
```

Uncurry in Alms:

```
λf (x,y) → f x y
  : (`a -`d-> `b -A-> `c) → `a × `b -`d-> `c
```

The Exponential

Intuitionistic linear logic:

$$\frac{\Gamma, x{:}!\sigma, x{:}!\sigma \vdash e : \tau}{\Gamma, x{:}!\sigma \vdash e : \tau} \text{ (Contraction)}$$

The Exponential vs. Usage Kinds

Intuitionistic linear logic:

$$\frac{\Gamma, x{:}!\sigma, x{:}!\sigma \vdash e : \tau}{\Gamma, x{:}!\sigma \vdash e : \tau}\ \text{(Contraction)}$$

Alms:

$$\frac{\Gamma, x{:}\sigma, x{:}\sigma \vdash e : \tau \qquad \Gamma \vdash \sigma : \mathtt{U}}{\Gamma, x{:}\sigma \vdash e : \tau}\ \text{(Contraction)}$$

The Exponential vs. Usage Kinds

Intuitionistic linear logic:

$$\frac{\Gamma, x{:}!\sigma, x{:}!\sigma \vdash e : \tau}{\Gamma, x{:}!\sigma \vdash e : \tau} \text{ (Contraction)}$$

Alms:

$$\frac{\Gamma, x{:}\sigma, x{:}\sigma \vdash e : \tau \qquad \Gamma \vdash \sigma : \mathtt{U}}{\Gamma, x{:}\sigma \vdash e : \tau} \text{ (Contraction)}$$

$$\frac{}{\Gamma \vdash \mathtt{int} : \mathtt{U}} \qquad \frac{\Gamma \vdash \tau : \xi}{\Gamma \vdash \tau\ \mathtt{CapArray.cap} : \mathtt{A}}$$

The Exponential vs. Usage Kinds

Intuitionistic linear logic:

$$\frac{\Gamma, x{:}!\sigma, x{:}!\sigma \vdash e : \tau}{\Gamma, x{:}!\sigma \vdash e : \tau} \text{ (Contraction)}$$

Alms:

$$\frac{\Gamma, x{:}\sigma, x{:}\sigma \vdash e : \tau \qquad \Gamma \vdash \sigma : \mathtt{U}}{\Gamma, x{:}\sigma \vdash e : \tau} \text{ (Contraction)}$$

$$\frac{}{\Gamma \vdash \mathtt{int} : \mathtt{U}} \qquad \frac{\Gamma \vdash \tau : \xi}{\Gamma \vdash \tau\ \mathtt{CapArray.cap} : \mathtt{A}}$$

$$\frac{\Gamma \vdash \tau : \xi_1 \qquad \Gamma \vdash \sigma : \xi_2}{\Gamma \vdash \tau \times \sigma : \xi_1 \sqcup \xi_2} \qquad \frac{\Gamma \vdash \tau : \xi_1 \qquad \Gamma \vdash \sigma : \xi_2 \qquad \Gamma \vdash \xi}{\Gamma \vdash \tau \xrightarrow{\xi} \sigma : \xi}$$

To Dependent Kinds

Lists in OCaml:

```
type 'a list = [] | (::) of 'a × 'a list
let rec map f xs = match xs with
   | []        → []
   | x :: xs' → f x :: map f xs'
```

To Dependent Kinds

Lists in OCaml:

```
type 'a list = [] | (::) of 'a × 'a list
let rec map f xs = match xs with
  | []        → []
  | x :: xs' → f x :: map f xs'
```

Lists in System F° (Mazurak et al. 2010):

```
type ('a:U) list = [] | (::) of 'a × 'a list
let rec map f xs = match xs with
  | []        → []
  | x :: xs' → f x :: map f xs'
```

To Dependent Kinds

Lists in OCaml:

```
type 'a list = [] | (::) of 'a × 'a list
let rec map f xs = match xs with
  | []        → []
  | x :: xs'  → f x :: map f xs'
```

Lists in System F° (Mazurak et al. 2010):

```
type ('a:U) list = [] | (::) of 'a × 'a list
let rec map f xs = match xs with
  | []        → []
  | x :: xs'  → f x :: map f xs'
```

To Dependent Kinds

Lists in OCaml:

```
type 'a list = [] | (::) of 'a × 'a list
let rec map f xs = match xs with
  | []        → []
  | x :: xs' → f x :: map f xs'
```

Lists in System F° (Mazurak et al. 2010):

```
type ('a:U) listU = [] | (::) of 'a × 'a listU
let rec mapU f xs = match xs with
  | []        → []
  | x :: xs' → f x :: mapU f xs'
```

```
type ('a:A) listA = NilA | ConsA of 'a × 'a listA
let rec mapA f xs = match xs with
  | NilA            → NilA
  | ConsA (x, xs') → ConsA (f x, mapA f x xs')
```

Dependent Kinds

Lists in Alms:

```
type ‘a list = [] | (::) of ‘a × ‘a list
let rec map f xs = match xs with
  | []        → []
  | x :: xs’ → f x :: map f xs’
```

Dependent Kinds

Lists in Alms:

```
type “a list = [] | (::) of “a × “a list
let rec map f xs = match xs with
  | []        → []
  | x :: xs’ → f x :: map f xs’

(* list    : Π“a.⟨“a⟩ *)
```

Dependent Kinds

Lists in Alms:

```
type “a list = [] | (::) of “a × “a list
let rec map f xs = match xs with
  | []         → []
  | x :: xs’   → f x :: map f xs’

(* list     : Π“a.⟨“a⟩ *)
(* (×)      : Π“a.Π“b.⟨“a⟩ ⊔ ⟨“b⟩ *)
(* CapArray.cap : Π“a.A *)
(* (→)      : Π“a.Π“b.Π“c.⟨“b⟩ *)
```

$$\frac{\Gamma \vdash \tau : \xi \qquad \Gamma \vdash c : \Pi\text{“a}.\,\kappa}{\Gamma \vdash \tau\ c : \{\xi/\text{“a}\}\kappa}$$

Promotion and Dereliction

In ILL:

$$\frac{!\Gamma \vdash e : \tau}{!\Gamma \vdash \texttt{promote}\ e : !\tau}\ \textbf{(!-I)} \qquad \frac{\Gamma \vdash e : !\tau}{\Gamma \vdash \texttt{derelict}\ e : \tau}\ \textbf{(!-E)}$$

Promotion and Dereliction

In ILL:

$$\frac{!\Gamma \vdash e : \tau}{!\Gamma \vdash e : !\tau}\ (\textbf{!-I}) \qquad\qquad \frac{\Gamma \vdash e : !\tau}{\Gamma \vdash e : \tau}\ (\textbf{!-E})$$

Promotion and Dereliction

In ILL:

$$\frac{\Gamma \vdash e : \,!\tau}{\Gamma \vdash e : \tau}\ \textbf{(!-E)}$$

Dereliction Subtyping

In ILL:

$$\frac{\Gamma \vdash e : !\tau}{\Gamma \vdash e : \tau}\ (\textbf{!-E})$$

In Wadler (1991), Gay (2006), ...:

$$\frac{}{\Gamma \vdash !\tau \leq \tau} \qquad \frac{\Gamma \vdash e : \sigma \qquad \Gamma \vdash \sigma \leq \tau}{\Gamma \vdash e : \tau}$$

Dereliction Subtyping

In ILL:

$$\frac{\Gamma \vdash e : !\tau}{\Gamma \vdash e : \tau} \; (\textbf{!-E})$$

In Wadler (1991), Gay (2006), …:

$$\frac{}{\Gamma \vdash !\tau \leq \tau} \qquad \frac{\Gamma \vdash e : \sigma \qquad \Gamma \vdash \sigma \leq \tau}{\Gamma \vdash e : \tau}$$

In Alms:

$$\frac{\Gamma \vdash \sigma' \leq \sigma \qquad \Gamma \vdash \tau \leq \tau' \qquad \Gamma \vdash \xi \sqsubseteq \xi'}{\Gamma \vdash \sigma \xrightarrow{\xi} \tau \leq \sigma' \xrightarrow{\xi'} \tau'}$$

Dereliction Subtyping

$$\Gamma \vdash \texttt{workerThread} : \texttt{unit} \xrightarrow{\texttt{U}} \texttt{unit}$$
$$\Gamma \vdash \texttt{Thread.fork} : (\texttt{unit} \xrightarrow{\texttt{A}} \texttt{unit}) \xrightarrow{\texttt{U}} \texttt{thread}$$

$$\frac{}{\Gamma \vdash \texttt{Thread.fork workerThread} : \texttt{thread}}$$

Dereliction Subtyping

$\mathcal{A}\colon\ \Gamma \vdash \texttt{workerThread} : \texttt{unit} \xrightarrow{\texttt{U}} \texttt{unit}$

$\mathcal{B}\colon\ \Gamma \vdash \texttt{Thread.fork} : (\texttt{unit} \xrightarrow{\texttt{A}} \texttt{unit}) \xrightarrow{\texttt{U}} \texttt{thread}$

$$\frac{}{\Gamma \vdash \texttt{Thread.fork workerThread} : \texttt{thread}}$$

Dereliction Subtyping

$\mathcal{A}$: $\Gamma \vdash \texttt{workerThread} : \texttt{unit} \xrightarrow{\texttt{U}} \texttt{unit}$

$\mathcal{B}$: $\Gamma \vdash \texttt{Thread.fork} : (\texttt{unit} \xrightarrow{\texttt{A}} \texttt{unit}) \xrightarrow{\texttt{U}} \texttt{thread}$

$$\frac{\mathcal{B}}{\Gamma \vdash \texttt{Thread.fork workerThread} : \texttt{thread}}$$

Dereliction Subtyping

$\mathcal{A}: \ \Gamma \vdash \texttt{workerThread} : \texttt{unit} \xrightarrow{\texttt{U}} \texttt{unit}$

$\mathcal{B}: \ \Gamma \vdash \texttt{Thread.fork} : (\texttt{unit} \xrightarrow{\texttt{A}} \texttt{unit}) \xrightarrow{\texttt{U}} \texttt{thread}$

$$\mathcal{B} \ \frac{\overline{\Gamma \vdash \texttt{workerThread} : \texttt{unit} \xrightarrow{\texttt{A}} \texttt{unit}}}{\Gamma \vdash \texttt{Thread.fork workerThread} : \texttt{thread}}$$

Dereliction Subtyping

$\mathcal{A}: \ \Gamma \vdash \texttt{workerThread} : \texttt{unit} \xrightarrow{\texttt{U}} \texttt{unit}$

$\mathcal{B}: \ \Gamma \vdash \texttt{Thread.fork} : (\texttt{unit} \xrightarrow{\texttt{A}} \texttt{unit}) \xrightarrow{\texttt{U}} \texttt{thread}$

$$\dfrac{\mathcal{B} \qquad \dfrac{\mathcal{A}}{\Gamma \vdash \texttt{workerThread} : \texttt{unit} \xrightarrow{\texttt{A}} \texttt{unit}}}{\Gamma \vdash \texttt{Thread.fork workerThread} : \texttt{thread}}$$

Dereliction Subtyping

$\mathcal{A}: \; \Gamma \vdash \texttt{workerThread} : \texttt{unit} \xrightarrow{\texttt{U}} \texttt{unit}$

$\mathcal{B}: \; \Gamma \vdash \texttt{Thread.fork} : (\texttt{unit} \xrightarrow{\texttt{A}} \texttt{unit}) \xrightarrow{\texttt{U}} \texttt{thread}$

$$\dfrac{\mathcal{B} \qquad \dfrac{\mathcal{A} \qquad \dfrac{}{\Gamma \vdash \texttt{unit} \xrightarrow{\texttt{U}} \texttt{unit} \leq \texttt{unit} \xrightarrow{\texttt{A}} \texttt{unit}}}{\Gamma \vdash \texttt{workerThread} : \texttt{unit} \xrightarrow{\texttt{A}} \texttt{unit}}}{\Gamma \vdash \texttt{Thread.fork workerThread} : \texttt{thread}}$$

Dereliction Subtyping

$$\mathcal{A}:\ \Gamma \vdash \texttt{workerThread} : \texttt{unit} \xrightarrow{\texttt{U}} \texttt{unit}$$

$$\mathcal{B}:\ \Gamma \vdash \texttt{Thread.fork} : (\texttt{unit} \xrightarrow{\texttt{A}} \texttt{unit}) \xrightarrow{\texttt{U}} \texttt{thread}$$

$$\dfrac{\mathcal{B} \qquad \dfrac{\mathcal{A} \qquad \dfrac{\cdots \qquad \Gamma \vdash \texttt{U} \sqsubseteq \texttt{A}}{\Gamma \vdash \texttt{unit} \xrightarrow{\texttt{U}} \texttt{unit} \leq \texttt{unit} \xrightarrow{\texttt{A}} \texttt{unit}}}{\Gamma \vdash \texttt{workerThread} : \texttt{unit} \xrightarrow{\texttt{A}} \texttt{unit}}}{\Gamma \vdash \texttt{Thread.fork workerThread} : \texttt{thread}}$$

Principal Promotion

In ILL:

$$\frac{!\Gamma \vdash e : \tau}{!\Gamma \vdash \texttt{promote}\, e : !\tau}$$

In Alms:

$$\frac{\Gamma, x{:}\sigma \vdash e : \tau \qquad \langle \Gamma|_{\mathbf{FV}(\boldsymbol{\lambda} x.\, e)} \rangle = \xi}{\Gamma \vdash \boldsymbol{\lambda} x.\, e : \sigma \xrightarrow{\xi} \tau}$$

Principal Promotion

In ILL:

$$\frac{!\Gamma \vdash e : \tau}{!\Gamma \vdash \texttt{promote}\, e : !\tau}$$

In Alms:

$$\frac{\Gamma, x{:}\sigma \vdash e : \tau \qquad \langle \Gamma|_{\mathbf{FV}(\boldsymbol{\lambda} x.\, e)} \rangle = \xi}{\Gamma \vdash \boldsymbol{\lambda} x.\, e : \sigma \xrightarrow{\xi} \tau}$$

Theorem: Alms's type system finds the least qualifier for every typable function.

Type Inference

```
#- λx → x
```

Type Inference

```
#- λx → x
it : “a → “a
```

Type Inference

```
#- λx → x
it : “a → “a

#- λx → (x, x)
```

Type Inference

```
#- λx → x
it : “a → “a

#- λx → (x, x)
it : 'a → 'a × 'a
```

Type Inference

```
#- λx → x
it : “a → “a

#- λx → (x, x)
it : ’a → ’a × ’a

#- λ(x, y) → (y, x)
```

Type Inference

```
#- λx → x
it : “a → “a

#- λx → (x, x)
it : ’a → ’a × ’a

#- λ(x, y) → (y, x)
it : “a × “b → “b × “a
```

Type Inference

```
#- λf x → f x
```

Type Inference

```
#- λf x → f x
it : (“a —“c→ “b) → “a —“c→ “b
```

Type Inference

```
#- λf x → f x
it : (“a –“c→ “b) → “a –“c→ “b

#- λf x → f (f x)
it : (“a → “a) → “a → “a

#- λf x → f (x, x)
it : ('a × 'a –“c→ “b) → 'a –“c→ “b

#- λf x → let y = f x in (y, y)
it : (“a –“c→ 'b) → “a –“c→ 'b × 'b

#- λf x → (f x, f x)
```

Type Inference

```
#- λf x → f x
it : (“a —“c→ “b) → “a —“c→ “b

#- λf x → f (f x)
it : (“a → “a) → “a → “a

#- λf x → f (x, x)
it : (’a × ’a —“c→ “b) → ’a —“c→ “b

#- λf x → let y = f x in (y, y)
it : (“a —“c→ ’b) → “a —“c→ ’b × ’b

#- λf x → (f x, f x)
it : (’a → “b) → ’a → “b × “b
```

Type Inference

```
#- let rec foldr f z xs = match xs with
#=   | [] → []
#=   | x::xs' → f x (foldr f z xs)
foldr : ('a → “b list ᴬ→ “b list) →
         “c →  'a list → “b list
```

Type Inference

```
#- let rec foldr f z xs = match xs with
#=    | [] → []
#=    | x::xs' → f x (foldr f z xs)
foldr : ('a → “b list →A “b list) →
          “c →  'a list → “b list
```

Type Inference

```
#- let rec foldr f z xs = match xs with
#=    | [] → []
#=    | x::xs’ → f x (foldr f z xs)
foldr : (’a → “b list -A→ “b list) →
          “c →  ’a list → “b list
```

```
#- let rec foldr f z xs = match xs with
#=    | [] → []
#=    | x::xs’ → f x (foldr f z xs’)
foldr : (“a → “b list -A→ “b list) →
          “c →  “a list → “b list
```

Constraints for Type Inference

$$
\begin{array}{rclll}
C & ::= & \top & & \text{trivial} \\
 & | & C \wedge C' & & \text{conjunction} \\
 & | & \sigma \leq \tau & & \text{subtype} \\
 & | & \exists \alpha . C & & \text{freshness}
\end{array}
$$

Constraints for Type Inference

$$
\begin{array}{llll}
C & ::= & \top & \text{trivial} \\
 & \mid & C \wedge C' & \text{conjunction} \\
 & \mid & \sigma \leq \tau & \text{subtype} \\
 & \mid & \exists \alpha . C & \text{freshness}
\end{array}
$$

$$\llbracket \Gamma \vdash e_1\ e_2 : \tau \rrbracket =$$

Constraints for Type Inference

$$
\begin{array}{rclll}
C & ::= & \top & & \text{trivial} \\
 & | & C \wedge C' & & \text{conjunction} \\
 & | & \sigma \leq \tau & & \text{subtype} \\
 & | & \exists \alpha.\, C & & \text{freshness}
\end{array}
$$

$$
\begin{aligned}
[\![\Gamma \vdash e_1\ e_2 : \tau]\!] = \exists \alpha.\, \exists \beta.\, & [\![\Gamma \vdash e_1 : \alpha]\!] \\
\wedge\, & [\![\Gamma \vdash e_2 : \beta]\!] \\
\wedge\, & (\alpha \leq \beta \rightarrow \tau)
\end{aligned}
$$

Constraints for Type Inference

$$
\begin{array}{rcll}
C & ::= & \top & \text{trivial} \\
& | & C \wedge C' & \text{conjunction} \\
& | & \sigma \leq \tau & \text{subtype} \\
& | & \exists \alpha .\, C & \text{freshness}
\end{array}
$$

$$
\begin{aligned}
[\![\Gamma \vdash e_1\, e_2 : \tau]\!] = \exists \alpha .\, \exists \beta .\, & [\![\Gamma \vdash e_1 : \alpha]\!] \\
& \wedge [\![\Gamma \vdash e_2 : \beta]\!] \\
& \wedge (\alpha \leq \beta \to \tau)
\end{aligned}
$$

Constraints for Type Inference

$$\begin{array}{rcll} C & ::= & \top & \text{trivial} \\ & | & C \wedge C' & \text{conjunction} \\ & | & \sigma \leq \tau & \text{subtype} \\ & | & \exists \alpha .\, C & \text{freshness} \end{array}$$

$$\begin{aligned} [\![\Gamma \vdash e_1\ e_2 : \tau]\!] = \exists \alpha .\, \exists \beta .\, & [\![\Gamma \vdash e_1 : \alpha]\!] \\ \wedge\, & [\![\Gamma \vdash e_2 : \beta]\!] \\ \wedge\, & (\alpha \leq \beta \to \tau) \end{aligned}$$

Constraints for Type Inference

$$
\begin{array}{rcll}
C & ::= & \top & \text{trivial} \\
 & | & C \wedge C' & \text{conjunction} \\
 & | & \sigma \leq \tau & \text{subtype} \\
 & | & \exists \alpha.\, C & \text{freshness}
\end{array}
$$

$$
\begin{aligned}
[\![\Gamma \vdash e_1\ e_2 : \tau]\!] = \exists \alpha.\, \exists \beta.\, & [\![\Gamma \vdash e_1 : \alpha]\!] \\
\wedge\, & [\![\Gamma \vdash e_2 : \beta]\!] \\
\wedge\, & (\alpha \leq \beta \to \tau)
\end{aligned}
$$

Constraints for Type Inference

$$
\begin{array}{rcll}
C & ::= & \top & \text{trivial} \\
  & | & C \wedge C' & \text{conjunction} \\
  & | & \sigma \leq \tau & \text{subtype} \\
  & | & \exists \alpha.\, C & \text{freshness}
\end{array}
$$

$$
\begin{aligned}
[\![\Gamma \vdash e_1\ e_2 : \tau]\!] = \exists \alpha.\, \exists \beta.\, & [\![\Gamma \vdash e_1 : \alpha]\!] \\
\wedge\, & [\![\Gamma \vdash e_2 : \beta]\!] \\
\wedge\, & (\alpha \leq \beta \to \tau)
\end{aligned}
$$

$$
\begin{aligned}
[\![\Gamma \vdash \lambda x.\, e : \tau]\!] = \exists \alpha.\, \exists \beta.\, & [\![\Gamma, x{:}\alpha \vdash e : \beta]\!] \\
\wedge\, & (\alpha \to \beta \leq \tau)
\end{aligned}
$$

Constraints for Type Inference

$$
\begin{array}{rcll}
C & ::= & \top & \text{trivial} \\
 & | & C \wedge C' & \text{conjunction} \\
 & | & \sigma \leq \tau & \text{subtype} \\
 & | & \exists \alpha.\, C & \text{freshness}
\end{array}
$$

$$
\begin{aligned}
[\![\Gamma \vdash e_1\; e_2 : \tau]\!] = \exists \alpha.\, \exists \beta.\, & [\![\Gamma \vdash e_1 : \alpha]\!] \\
\wedge\, & [\![\Gamma \vdash e_2 : \beta]\!] \\
\wedge\, & (\alpha \leq \beta \to \tau)
\end{aligned}
$$

$$
\begin{aligned}
[\![\Gamma \vdash \lambda x.\, e : \tau]\!] = \exists \alpha.\, \exists \beta.\, & [\![\Gamma, x{:}\alpha \vdash e : \beta]\!] \\
\wedge\, & (\alpha \to \beta \leq \tau)
\end{aligned}
$$

Constraints for Type Inference

$$
\begin{array}{rcll}
C & ::= & \top & \text{trivial} \\
& | & C \wedge C' & \text{conjunction} \\
& | & \sigma \leq \tau & \text{subtype} \\
& | & \exists \alpha.\, C & \text{freshness}
\end{array}
$$

$$
\begin{aligned}
[\![\Gamma \vdash e_1\ e_2 : \tau]\!] = \exists \alpha.\, \exists \beta.\, & [\![\Gamma \vdash e_1 : \alpha]\!] \\
\wedge\, & [\![\Gamma \vdash e_2 : \beta]\!] \\
\wedge\, & (\alpha \leq \beta \to \tau)
\end{aligned}
$$

$$
\begin{aligned}
[\![\Gamma \vdash \lambda x.\, e : \tau]\!] = \exists \alpha.\, \exists \beta.\, & [\![\Gamma, x{:}\alpha \vdash e : \beta]\!] \\
\wedge\, & (\alpha \to \beta \leq \tau)
\end{aligned}
$$

Constraints for Type Inference

$$
\begin{array}{rcll}
C & ::= & \top & \text{trivial} \\
 & | & C \wedge C' & \text{conjunction} \\
 & | & \sigma \leq \tau & \text{subtype} \\
 & | & \exists \alpha .\, C & \text{freshness} \\
 & | & \xi_1 \sqsubseteq \xi_2 & \text{subqualifier}
\end{array}
$$

$$
\begin{aligned}
[\![\Gamma \vdash e_1\ e_2 : \tau]\!] = \exists \alpha .\, \exists \beta .\, & [\![\Gamma \vdash e_1 : \alpha]\!] \\
\wedge\ & [\![\Gamma \vdash e_2 : \beta]\!] \\
\wedge\ & (\alpha \leq \beta \to \tau)
\end{aligned}
$$

$$
\begin{aligned}
[\![\Gamma \vdash \lambda x .\, e : \tau]\!] = \exists \alpha .\, \exists \beta .\, & [\![\Gamma, x{:}\alpha \vdash e : \beta]\!] \\
\wedge\ & (\alpha \to \beta \leq \tau)
\end{aligned}
$$

Constraints for Type Inference

$$\begin{array}{rcll} C & ::= & \top & \text{trivial} \\ & | & C \wedge C' & \text{conjunction} \\ & | & \sigma \leq \tau & \text{subtype} \\ & | & \exists \alpha.\, C & \text{freshness} \\ & | & \xi_1 \sqsubseteq \xi_2 & \text{subqualifier} \end{array}$$

$$\begin{aligned} [\![\Gamma \vdash e_1\ e_2 : \tau]\!] = \exists \alpha. \exists \beta. \exists \gamma.\ & [\![\Gamma \vdash e_1 : \alpha]\!] \\ & \wedge [\![\Gamma \vdash e_2 : \beta]\!] \\ & \wedge (\alpha \leq \beta \to \tau) \end{aligned}$$

$$\begin{aligned} [\![\Gamma \vdash \lambda x.\, e : \tau]\!] = \exists \alpha. \exists \beta.\ & [\![\Gamma, x{:}\alpha \vdash e : \beta]\!] \\ & \wedge (\alpha \to \beta \leq \tau) \end{aligned}$$

Constraints for Type Inference

$$
\begin{array}{rcll}
C & ::= & \top & \text{trivial} \\
  & \mid & C \land C' & \text{conjunction} \\
  & \mid & \sigma \leq \tau & \text{subtype} \\
  & \mid & \exists \alpha.\, C & \text{freshness} \\
  & \mid & \xi_1 \sqsubseteq \xi_2 & \text{subqualifier}
\end{array}
$$

$$
\begin{aligned}
[\![\Gamma \vdash e_1\ e_2 : \tau]\!] = \exists \alpha.\, \exists \beta.\, \exists \gamma.\, & [\![\Gamma \vdash e_1 : \alpha]\!] \\
& \land [\![\Gamma \vdash e_2 : \beta]\!] \\
& \land (\alpha \leq \beta \xrightarrow{\gamma} \tau)
\end{aligned}
$$

$$
\begin{aligned}
[\![\Gamma \vdash \lambda x.\, e : \tau]\!] = \exists \alpha.\, \exists \beta.\, & [\![\Gamma, x{:}\alpha \vdash e : \beta]\!] \\
& \land (\alpha \to \beta \leq \tau)
\end{aligned}
$$

Constraints for Type Inference

$$
\begin{array}{rcll}
C & ::= & \top & \text{trivial} \\
& | & C \wedge C' & \text{conjunction} \\
& | & \sigma \leq \tau & \text{subtype} \\
& | & \exists \alpha.\, C & \text{freshness} \\
& | & \xi_1 \sqsubseteq \xi_2 & \text{subqualifier}
\end{array}
$$

$$
\begin{aligned}
[\![\Gamma \vdash e_1\ e_2 : \tau]\!] = \exists \alpha.\, \exists \beta.\, \exists \gamma.\, & [\![\Gamma \vdash e_1 : \alpha]\!] \\
\wedge\ & [\![\Gamma \vdash e_2 : \beta]\!] \\
\wedge\ & (\alpha \leq \beta \xrightarrow{\gamma} \tau)
\end{aligned}
$$

$$
\begin{aligned}
[\![\Gamma \vdash \lambda x.\, e : \tau]\!] = \exists \alpha.\, \exists \beta.\, \exists \gamma.\, & [\![\Gamma, x{:}\alpha \vdash e : \beta]\!] \\
\wedge\ & (\alpha \xrightarrow{\gamma} \beta \leq \tau)
\end{aligned}
$$

Constraints for Type Inference

$$\begin{array}{rcll} C & ::= & \top & \text{trivial} \\ & \mid & C \wedge C' & \text{conjunction} \\ & \mid & \sigma \leq \tau & \text{subtype} \\ & \mid & \exists \alpha.\, C & \text{freshness} \\ & \mid & \xi_1 \sqsubseteq \xi_2 & \text{subqualifier} \end{array}$$

$$\begin{aligned} [\![\Gamma \vdash e_1\ e_2 : \tau]\!] = \exists\alpha.\,\exists\beta.\,\exists\gamma.\, & [\![\Gamma \vdash e_1 : \alpha]\!] \\ & \wedge [\![\Gamma \vdash e_2 : \beta]\!] \\ & \wedge (\alpha \leq \beta \xrightarrow{\gamma} \tau) \end{aligned}$$

$$\begin{aligned} [\![\Gamma \vdash \lambda x.\, e : \tau]\!] = \exists\alpha.\,\exists\beta.\,\exists\gamma.\, & [\![\Gamma, x{:}\alpha \vdash e : \beta]\!] \\ & \wedge (\alpha \xrightarrow{\gamma} \beta \leq \tau) \\ & \wedge \langle\gamma\rangle \sqsubseteq \langle \Gamma|_{\mathbf{FV}(\lambda x.e)} \rangle \end{aligned}$$

Constraints for Type Inference

$$
\begin{array}{rcll}
C & ::= & \top & \text{trivial} \\
  & \mid & C \wedge C' & \text{conjunction} \\
  & \mid & \sigma \leq \tau & \text{subtype} \\
  & \mid & \exists \alpha.\, C & \text{freshness} \\
  & \mid & \xi_1 \sqsubseteq \xi_2 & \text{subqualifier}
\end{array}
$$

$$
\begin{aligned}
[\![\Gamma \vdash e_1\; e_2 : \tau]\!] = \exists\alpha.\, \exists\beta.\, \exists\gamma.\, & [\![\Gamma \vdash e_1 : \alpha]\!] \\
\wedge\, & [\![\Gamma \vdash e_2 : \beta]\!] \\
\wedge\, & (\alpha \leq \beta \xrightarrow{\gamma} \tau)
\end{aligned}
$$

$$
\begin{aligned}
[\![\Gamma \vdash \lambda x.\, e : \tau]\!] = \exists\alpha.\, \exists\beta.\, \exists\gamma.\, & [\![\Gamma, x{:}\alpha \vdash e : \beta]\!] \\
\wedge\, & (\alpha \xrightarrow{\gamma} \beta \leq \tau) \\
\wedge\, & \langle\gamma\rangle \sqsubseteq \langle \Gamma|_{\mathbf{FV}(\lambda x.e)} \rangle \\
\wedge\, & \langle\alpha\rangle \sqsubseteq \langle occ_x(e) \rangle
\end{aligned}
$$

More Practical Issues

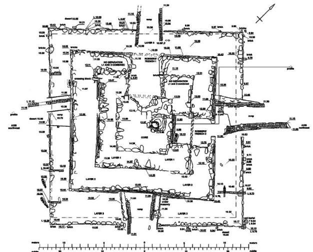
Interoperation with a conventional language (ESOP 2010)

(using behavioral contracts)

Substructural types and control effects (OOPSLA 2011)

(using a type-and-effect system)

Conclusion



Related Work: Theory

Foundations

- Linear logic (Girard 1987)
- Intuitionistic linear logic (Bierman 1993)
- λ^{URAL} (Ahmed et al. 2005)

Pragmatics

- Use types (Wadler 1991)
- "Uniqueness typing simplified" (de Vries et al. 2008)
- System F° (Mazurak et al. 2010)

Related Work: Stateful Type Systems

Permissions

- Fractional permissions (Boyland 2003)
- Chalice (Leino et al. 2009)

Session types (Honda et al. 1998)

- Functional session types (Vasconcelos et al. 2004)
- Sing# (Fähndrich et al. 2006)
- Session types to ILL (Pucella and Heller 2008)

Related Work: The Competition

Fine (Swamy et al. 2010)

F* (Swamy et al. 2011)

AdjS (Krishnaswami 2012)

Future Work

Improve the run-time story

Combine affine and uniqueness types

Explore the design space

Future Work

Improve the run-time story

(Exploit affine types for efficiency?)

Combine affine and uniqueness types

Explore the design space

Future Work

Improve the run-time story

(Exploit affine types for efficiency?)

Combine affine and uniqueness types

(Similar machinery, different meanings)

Explore the design space

Future Work

Improve the run-time story

(Exploit affine types for efficiency?)

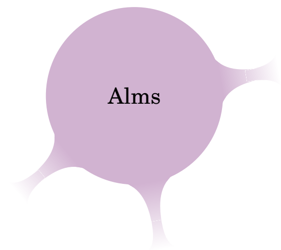
Combine affine and uniqueness types

(Similar machinery, different meanings)

Explore the design space

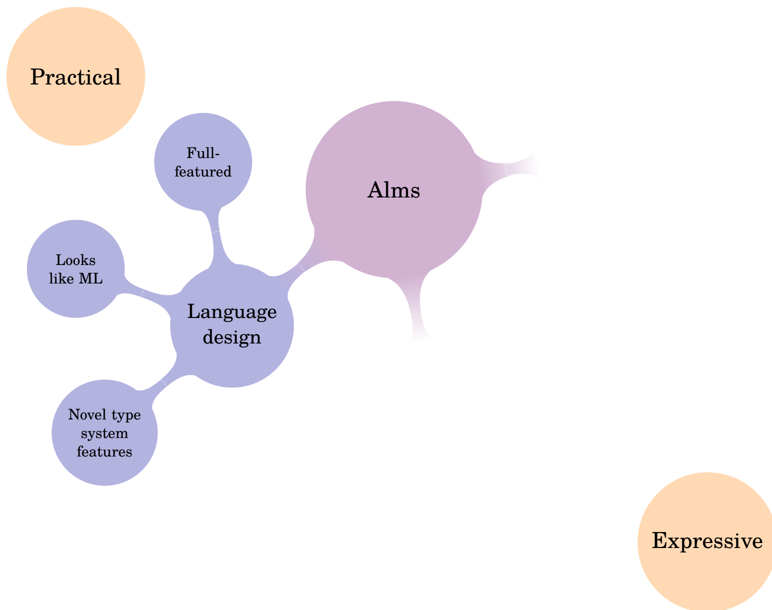
(Why stop with ML?)

Contributions

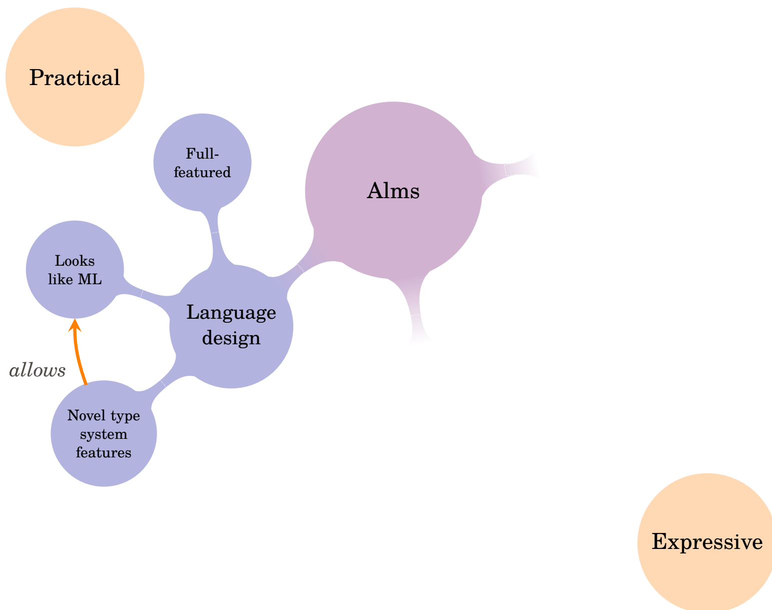


Contributions

Contributions



Contributions



Contributions

Practical

Full-featured

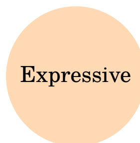
Alms

Looks like ML

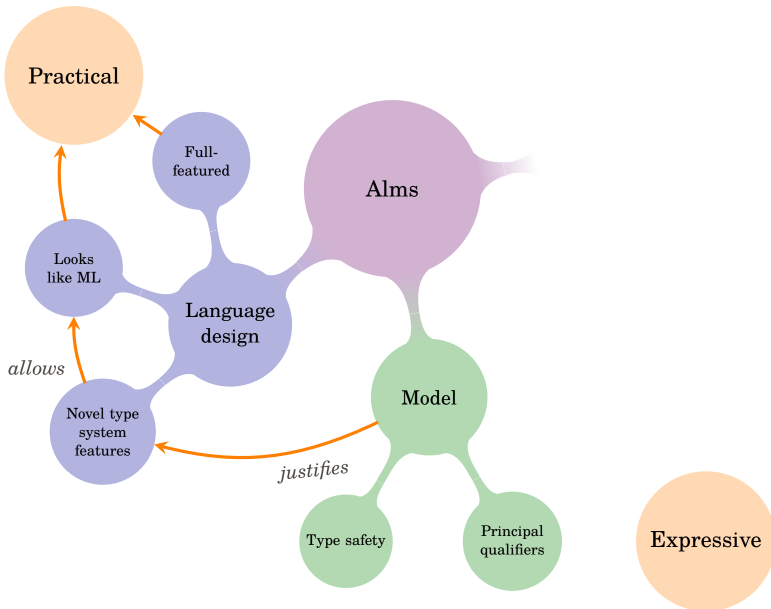
Language design

allows

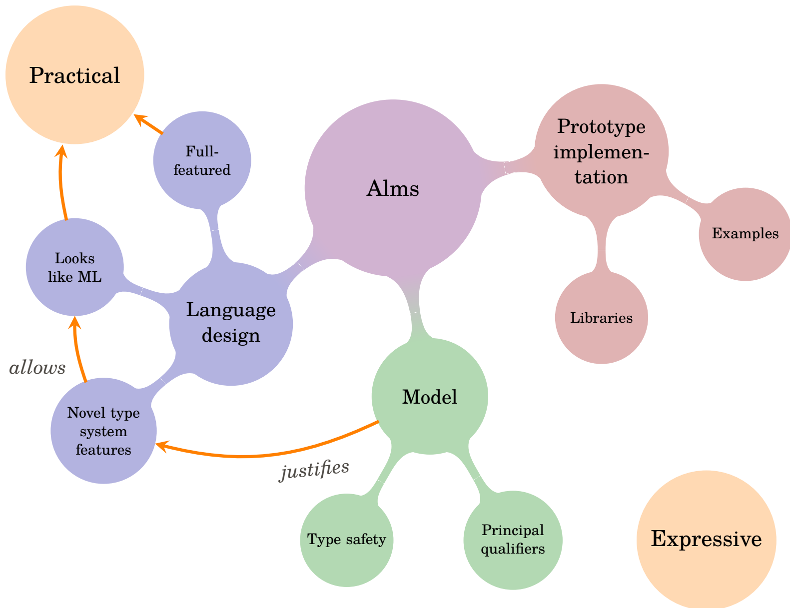
Novel type system features



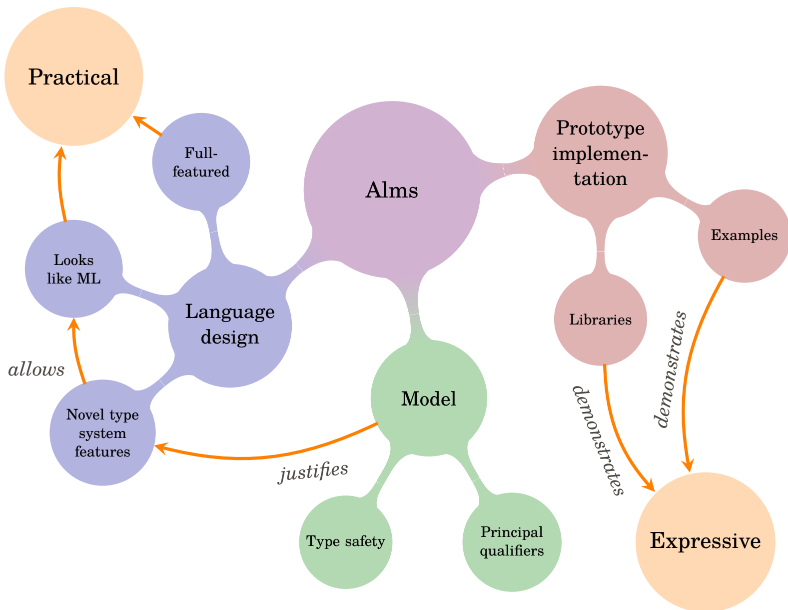
Contributions



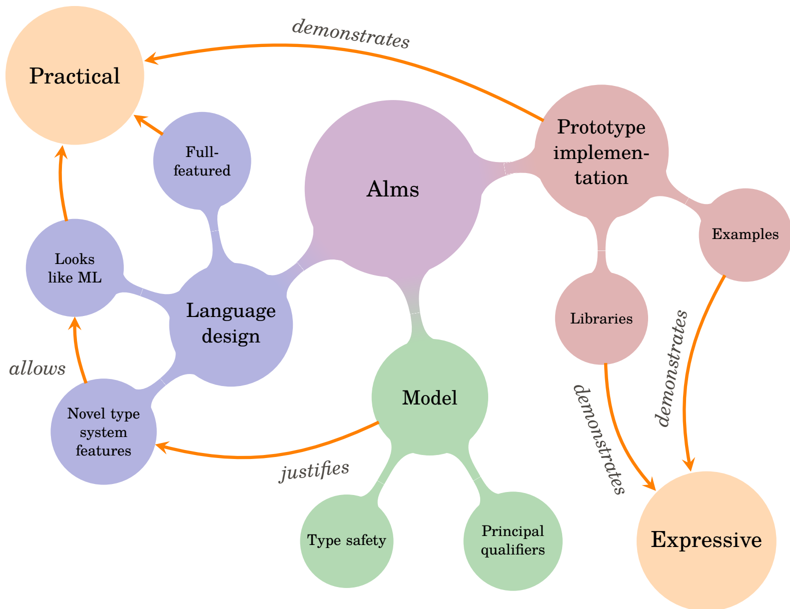
Contributions



Contributions



Contributions



Contributions