Asynchronous Liquid Separation Types

Johannes Kloos Rupak Majumdar Viktor Vafeiadis

Max Planck Institute for Software Systems

ECOOP 2015

Introducing asynchronous concurrency

Example of asynchronous execution:

Task A

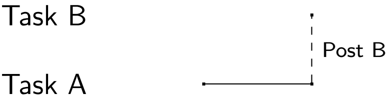
Introducing asynchronous concurrency

Example of asynchronous execution:

Task A

Introducing asynchronous concurrency

Example of asynchronous execution:



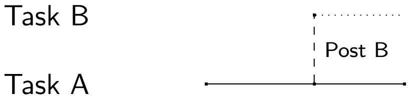
Task B

Post B

Task A

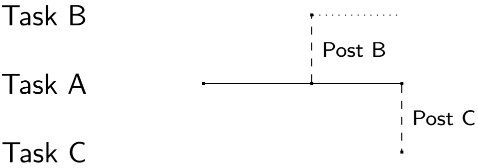
Introducing asynchronous concurrency

Example of asynchronous execution:



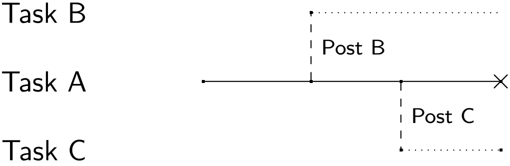
Introducing asynchronous concurrency

Example of asynchronous execution:



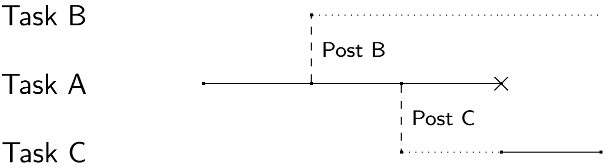
Introducing asynchronous concurrency

Example of asynchronous execution:



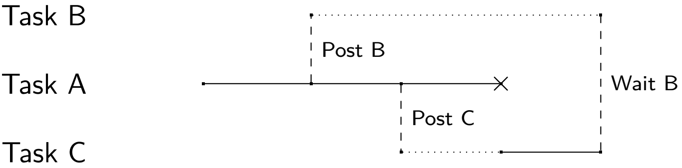
Introducing asynchronous concurrency

Example of asynchronous execution:



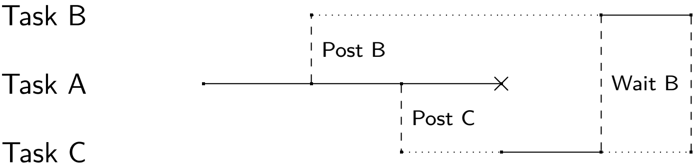
Introducing asynchronous concurrency

Example of asynchronous execution:



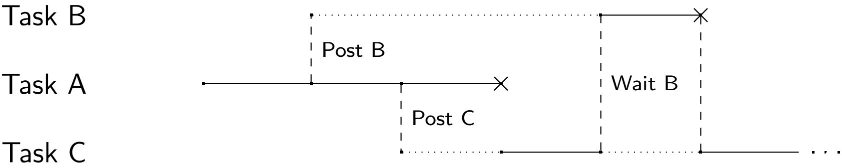
Introducing asynchronous concurrency

Example of asynchronous execution:



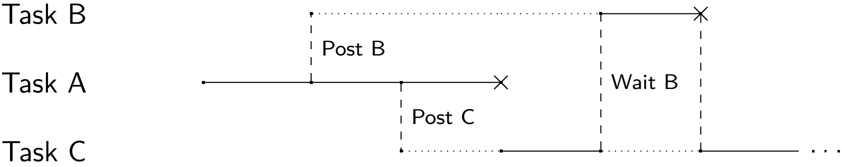
Introducing asynchronous concurrency

Example of asynchronous execution:



Introducing asynchronous concurrency

Example of asynchronous execution:



Used in:

- Web applications, AJAX
- Android programming model
- Operation systems: Interrupt handling

Asynchronous concurrency in OCaml

```
let rec copy ins outs =
    if eof ins then () else
    let fill = post (read ins) in
    let buf = wait fill in
    let drain = post (write outs buf) in
    wait drain;
    copy ins outs
```

Asynchronous concurrency in OCaml

```
let rec copy ins outs =
    if eof ins then () else
    let fill = post (read ins) in
    let buf = wait fill in
    let drain = post (write outs buf) in
    wait drain;
    copy ins outs
```

1. Create task to read from input stream.

Asynchronous concurrency in OCaml

```
let rec copy ins outs =
    if eof ins then () else
    let fill = post (read ins) in
    let buf = wait fill in
    let drain = post (write outs buf) in
    wait drain;
    copy ins outs
```

1. Create task to read from input stream.
2. Wait for results of read task.

Asynchronous concurrency in OCaml

```
let rec copy ins outs =
    if eof ins then () else
    let fill = post (read ins) in
    let buf = wait fill in
    let drain = post (write outs buf) in
    wait drain;
    copy ins outs
```

1. Create task to read from input stream.
2. Wait for results of read task.
3. Use task to write data to output stream.

Asynchronous concurrency in OCaml

```
let rec copy ins outs =
    if eof ins then () else
    let fill = post (read ins) in
    let buf = wait fill in
    let drain = post (write outs buf) in
    wait drain;
    copy ins outs
```

1. Create task to read from input stream.
2. Wait for results of read task.
3. Use task to write data to output stream.
4. Copy more data.

Asynchronous concurrency in OCaml

```
let rec copy ins outs =
    if eof ins then () else
    let fill = post (read ins) in
    let buf = wait fill in
    let drain = post (write outs buf) in
    wait drain;
    copy ins outs
```

1. Create task to read from input stream.
2. Wait for results of read task.
3. Use task to write data to output stream.
4. Copy more data.
5. If end-of-file, terminate.

Asynchronous concurrency in OCaml

```
let rec copy ins outs =
    if eof ins then () else
    let fill = post (read ins) in
    let buf = wait fill in
    let drain = post (write outs buf) in
    wait drain;
    copy ins outs
```

How can we reason about such programs?

Challenge: Typing asynchronous code

Consider the code fragment `post (read ins)`.

read: *stream* → *buffer*

Challenge: Typing asynchronous code

Consider the code fragment `post` (read ins).

read: *stream* → *buffer*
`post` (read ins): *promise buffer*

Challenge: Typing asynchronous code

Consider the code fragment `post` (read ins).

read: *stream* → *buffer*
`post` (read ins): *promise buffer*

Introduce subtyping. Suppose *initbuffer* <: *buffer*.

Challenge: Typing asynchronous code

Consider the code fragment `post` (read ins).

read: *stream* $\rightarrow$ *buffer*
`post` (read ins): *promise buffer*

Introduce subtyping. Suppose *initbuffer* $<:$ *buffer*.

read: *stream* $\rightarrow$ *initbuffer*

Challenge: Typing asynchronous code

Consider the code fragment `post` (read ins).

read: *stream* → *buffer*
`post` (read ins): *promise buffer*

Introduce subtyping. Suppose $\mathit{initbuffer} <: \mathit{buffer}$.

read: *stream* → *initbuffer*
`post` (read ins): *promise initbuffer*

Challenge: Typing asynchronous code

Consider the code fragment `post` (read ins).

read: *stream* → *buffer*
`post` (read ins): *promise buffer*

Introduce subtyping. Suppose *initbuffer* <: *buffer*.

read: *stream* → *initbuffer*
`post` (read ins): *promise initbuffer*

In the paper: refinement typing, using liquid types

Challenge: Typing with mutable state

Another copying loop, with buffers on the heap (no concurrency):

```
let copy_buf ins outs =
    let rec do_copy buf =
        if eof ins then () else
        read ins buf;
        write outs buf;
        do_copy buf
    in do_copy (make_buffer ())
```

Challenge: Typing with mutable state

Another copying loop, with buffers on the heap (no concurrency):

```
let copy_buf ins outs =
    let rec do_copy buf =
        if eof ins then () else
        read ins buf;
        write outs buf;
        do_copy buf
    in do_copy (make_buffer ())
```

1. Main copying loop gets a pre-allocated buffer

Challenge: Typing with mutable state

Another copying loop, with buffers on the heap (no concurrency):

```
let copy_buf ins outs =
    let rec do_copy buf =
        if eof ins then () else
        read ins buf;
        write outs buf;
        do_copy buf
    in do_copy (make_buffer ())
```

1. Main copying loop gets a pre-allocated buffer
2. Read fills this buffer, write drains it.

Challenge: Typing with mutable state

Another copying loop, with buffers on the heap (no concurrency):

```
let copy_buf ins outs =
    let rec do_copy buf =
        if eof ins then () else
        read ins buf;
        write outs buf;
        do_copy buf
    in do_copy (make_buffer ())
```

1. Main copying loop gets a pre-allocated buffer
2. Read fills this buffer, write drains it.
3. The copying loop is called with a fresh buffer.

Challenge: Typing with mutable state

Another copying loop, with buffers on the heap (no concurrency):

```
let copy_buf ins outs =
    let rec do_copy buf =
        if eof ins then () else
        read ins buf;
        write outs buf;
        do_copy buf
    in do_copy (make_buffer ())
```

Can we make use subtyping to ensure: Only initialized buffers are written?

Subtyping and mutation don't mix well

Easier example:

```
let r = ref 0 in
assert (!r = 0); (* The content of r is 0 *)
r := 1;
assert (!r = 1) (* The content of r is 1 *)
```

Subtyping and mutation don’t mix well

Easier example:

```
let r = ref 0 in
assert (!r = 0); (* The content of r is 0 *)
r := 1;
assert (!r = 1) (* The content of r is 1 *)
```

Can the type system prove that the assertions hold?

Subtyping and mutation don't mix well

Easier example:

```
let r = ref 0 in
assert (!r = 0); (* The content of r is 0 *)
r := 1;
assert (!r = 1) (* The content of r is 1 *)
```

Can the type system prove that the assertions hold?
Suppose $int_{=0} <: int$ and $int_{=1} <: int$.

Subtyping and mutation don't mix well

Easier example:

```
let r = ref 0 in
assert (!r = 0); (* The content of r is 0 *)
r := 1;
assert (!r = 1) (* The content of r is 1 *)
```

Can the type system prove that the assertions hold?
Suppose $\mathit{int}_{=0} <: \mathit{int}$ and $\mathit{int}_{=1} <: \mathit{int}$.

- r:*ref int*: Can't prove anything
- r:$\mathit{ref}\ \mathit{int}_{=1}$: Not true – initially, !r is 0
- r:$\mathit{ref}\ \mathit{int}_{=0}$: Not true

Subtyping and mutation don't mix well

Easier example:

```
let r = ref 0 in
assert (!r = 0); (* The content of r is 0 *)
r := 1;
assert (!r = 1) (* The content of r is 1 *)
```

Can the type system prove that the assertions hold?
Suppose $\mathit{int}_{=0} <: \mathit{int}$ and $\mathit{int}_{=1} <: \mathit{int}$.

- r:*ref int*: Can't prove anything
- r:$\mathit{ref}\ \mathit{int}_{=1}$: Not true – initially, !r is 0
- r:$\mathit{ref}\ \mathit{int}_{=0}$: Not true

We need some form of strong updates!

Program logics work well with updating state

$\{\ \textit{emp}\ \}$

`let` r = `ref` 0 `in`

$\{\ r \mapsto 0\ \}$

assert (!r = 0);

$\{\ r \mapsto 0\ \}$

r := 1;

$\{\ r \mapsto 1\ \}$

assert (!r = 1);

$\{\ r \mapsto 1\ \}$

The situation

Subtyping, refinement types:

- Expressive
- In the guise of liquid types: Provide strong automation.

The situation

Subtyping, refinement types:

- Expressive
- In the guise of liquid types: Provide strong automation.
- But: Don't deal with mutable state out-of-the box.

The situation

Subtyping, refinement types:

- Expressive
- In the guise of liquid types: Provide strong automation.
- But: Don't deal with mutable state out-of-the box.

Program logics:

The situation

Subtyping, refinement types:

- Expressive
- In the guise of liquid types: Provide strong automation.
- But: Don't deal with mutable state out-of-the box.

Program logics:

- Are good at dealing with mutable state

The situation

Subtyping, refinement types:

- Expressive
- In the guise of liquid types: Provide strong automation.
- But: Don't deal with mutable state out-of-the box.

Program logics:

- Are good at dealing with mutable state
- Also work well with concurrency

The situation

Subtyping, refinement types:

- Expressive
- In the guise of liquid types: Provide strong automation.
- But: Don't deal with mutable state out-of-the box.

Program logics:

- Are good at dealing with mutable state
- Also work well with concurrency
- But: Automation is much harder.

The situation

Subtyping, refinement types:

- Expressive
- In the guise of liquid types: Provide strong automation.
- But: Don't deal with mutable state out-of-the box.

Program logics:

- Are good at dealing with mutable state
- Also work well with concurrency
- But: Automation is much harder.

Can we combine both?

Our Contribution

Combine type systems and program logics to reason about OCaml programs with mutable state and asynchronous concurrency.

Our Contribution

Combine type systems and program logics to reason about OCaml programs with mutable state and asynchronous concurrency.

Liquid type + Concurrent Separation Logic
=
Asynchronous Liquid Separation Types (ALST)

Outline

- The type system
- Type inference
- Some larger examples

Outline

- The type system
- Type inference
- Some larger examples

Typing the copying loop with buffers

```
let copy_buf ins outs =
    let rec do_copy buf =
        if eof ins then () else
        read ins buf;
        write outs buf;
        do_copy buf
    in do_copy (make_buffer ())
```

Typing the copying loop with buffers

```
let copy_buf ins outs =
    let rec do_copy buf =
        if eof ins then () else
        read ins buf;
        write outs buf;
        do_copy buf
    in do_copy (make_buffer ())
```

Traditional typing judgment

$$\underset{\text{Environment}}{\underset{\uparrow}{\Gamma}} \vdash \underset{\text{Expression}}{\underset{\uparrow}{e}} : \underset{\text{Type}}{\underset{\uparrow}{\tau}}$$

Extended typing judgment

$$\underset{\substack{\uparrow \\ \text{Environment}}}{\Gamma} \quad \overset{\substack{\text{Precondition} \\ \downarrow}}{\langle\ \text{pre}\ \rangle} \quad \vdash \underset{\substack{\uparrow \\ \text{Expression}}}{\text{e}} : \quad \underset{\substack{\uparrow \\ \text{Type}}}{\tau} \quad \overset{\substack{\text{Postcondition} \\ \downarrow}}{\langle\ \text{post}\ \rangle}$$

Typing the example

ins: *stream*, buf: *ref buffer* $\langle \qquad\qquad \rangle$

$\vdash$ read ins buf: *unit* $\langle \qquad\qquad \rangle$

Typing the example

$$\begin{array}{l} \text{ins: } \mathit{stream}, \text{ buf: } \mathit{ref}\ \ \mathit{buffer} \quad \langle \mu \mapsto \mathit{buffer} \rangle \\ \vdash \text{ read ins buf: } \quad \mathit{unit} \quad \langle \mu \mapsto \mathit{initbuffer} \rangle \end{array}$$

- Precondition: Content of heap cell μ has type *buffer*
  Postcondition: Content of heap cell μ has type *initbuffer*

Typing the example

ins: *stream*, buf: ref_μ *buffer* $\langle \mu \mapsto buffer \rangle$
$\vdash$ read ins buf: *unit* $\langle \mu \mapsto initbuffer \rangle$

- Precondition: Content of heap cell μ has type *buffer*
  Postcondition: Content of heap cell μ has type *initbuffer*
- buf is a refernce to μ

Typing the example

ins: *stream*, buf: ref_μ *buffer* $\langle \mu \mapsto \mathit{buffer} \rangle$

$\vdash$ read ins buf: *unit* $\langle \mu \mapsto \mathit{initbuffer} \rangle$

- Precondition: Content of heap cell μ has type *buffer*
  Postcondition: Content of heap cell μ has type *initbuffer*
- buf is a refernce to μ
- Note: Memory cells have types here, subtyping allowed!

We can even handle concurrency

Consider modified code fragment:

```
post (read ins buf)
```

We can even handle concurrency

Consider modified code fragment:

```
post (read ins buf)
```

Known: Type of read ins buf

$\text{ins}: \textit{stream}, \text{buf}: \textit{ref}_\mu\ \textit{buffer} \quad \langle \mu \mapsto \textit{buffer} \rangle$

$\vdash \text{read ins buf}: \quad \textit{unit} \quad \langle \mu \mapsto \textit{initbuffer} \rangle$

We can even handle concurrency

Consider modified code fragment:

```
post (read ins buf)
```

Known: Type of read ins buf

$\text{ins}: \mathit{stream}, \text{buf}: \mathit{ref}_\mu\ \mathit{buffer} \quad \langle \mu \mapsto \mathit{buffer} \rangle$

$\vdash \text{read ins buf}: \quad \mathit{unit} \quad \langle \mu \mapsto \mathit{initbuffer} \rangle$

The post expression:

$\text{ins}: \mathit{stream}, \text{buf}: \mathit{ref}_\mu\ \mathit{buffer} \quad \langle \mu \mapsto \mathit{buffer} \rangle$

$\vdash \texttt{post}\ (\text{read ins buf}):$

$\langle \qquad\qquad\qquad\qquad \rangle$

We can even handle concurrency

Consider modified code fragment:

```
post (read ins buf)
```

Known: Type of read ins buf

$\text{ins: } \mathit{stream}, \text{ buf: } \mathit{ref}_\mu\ \mathit{buffer} \quad \langle \mu \mapsto \mathit{buffer} \rangle$

$\vdash \text{read ins buf: } \quad \mathit{unit} \quad \langle \mu \mapsto \mathit{initbuffer} \rangle$

The post expression:

$\text{ins: } \mathit{stream}, \text{ buf: } \mathit{ref}_\mu\ \mathit{buffer} \quad \langle \mu \mapsto \mathit{buffer} \rangle$

$\vdash \texttt{post} \text{ (read ins buf):}$

$\quad \mathit{promise}_\pi\ \mathit{unit} \quad \langle \qquad\qquad\qquad\qquad \rangle$

We can even handle concurrency

Consider modified code fragment:

`post` (read ins buf)

Known: Type of read ins buf

ins: *stream*, buf: ref_μ *buffer* $\langle \mu \mapsto \mathit{buffer} \rangle$
$\vdash$ read ins buf: *unit* $\langle \mu \mapsto \mathit{initbuffer} \rangle$

The post expression:

ins: *stream*, buf: ref_μ *buffer* $\langle \mu \mapsto \mathit{buffer} \rangle$
$\vdash$ `post` (read ins buf):
$\mathit{promise}_\pi$ *unit* $\langle \mathit{Wait}(\pi, \mu \mapsto \mathit{initbuffer}) \rangle$

Package resources produced by task in *wait permission*

A non-trivial example

Earlier example, slightly extended.

```
let copy_buf ins outs =
    let rec do_copy buf =
        if eof ins then () else
        wait (post (read ins buf));
        wait (post (write outs buf));
        do_copy buf
    in do_copy (make_buffer ())
```

A non-trivial example

Earlier example, slightly extended.

```
let copy_buf ins outs =
    let rec do_copy buf =
        if eof ins then () else
        wait (post (read ins buf));
        wait (post (write outs buf));
        d
    in do_
```

Given read: $\mathit{stream} \to \mathit{ref}_\mu\ \mathit{buffer}\ \langle \mu \mapsto \mathit{buffer} \rangle \to$
$\mathit{unit}\ \langle \mu \mapsto \mathit{initbuffer} \rangle$

$\ldots \langle \mu \mapsto \mathit{buffer} \rangle \vdash \text{read ins buf} : \mathit{unit}\langle \mu \mapsto \mathit{initbuffer} \rangle$

A non-trivial example

Earlier example, slightly extended.

```
let copy_buf ins outs =
    let rec do_copy buf =
        if eof ins then () else
        wait (post (read ins buf));
        wait (post (write outs buf));
        do_copy buf
    in do_copy (n
```

$$\ldots\langle \mu \mapsto \mathit{buffer}\rangle \vdash \texttt{post (read ins buf)}:$$
$$\mathit{promise}_\pi\ \mathit{unit}\ \langle \mathit{Wait}(\pi, \mu \mapsto \mathit{initbuffer})\rangle$$

A non-trivial example

Earlier example, slightly extended.

```
let copy_buf ins outs =
    let rec do_copy buf =
        if eof ins then () else
        wait (post (read ins buf));
        wait (post (write outs buf));
        do_copy buf
    in do_copy
```

$$\ldots\langle \mu \mapsto \mathit{buffer}\rangle \vdash \texttt{wait}\ \ (\text{post (read ins buf)}) : \mathit{unit}\langle \mu \mapsto \mathit{initbuffer}\rangle$$

A non-trivial example

Earlier example, slightly extended.

```
let copy_buf ins outs =
    let rec do_copy buf =
        if eof ins then () else
        wait (post (read ins buf));
        wait (post (write outs buf));
        do_copy buf
    in do_copy (make
```

$\ldots \langle \mu \mapsto \mathit{buffer} \rangle \vdash \mathit{read,\ then\ write}:$

$\mathit{unit}\langle \mu \mapsto \mathit{initbuffer} \rangle$

A non-trivial example

Earlier example, slightly extended.

```
let copy_buf ins outs =
    let rec do_copy buf =
        if eof ins then () else
        wait (post (read ins buf));
        wait (post (write outs buf));
        do_copy buf
    in do_copy (make_buffer ())
```

$\ldots \langle \mu \mapsto \mathit{buffer} \rangle \vdash \mathit{loop\ body}: \mathit{unit} \langle \mu \mapsto \mathit{buffer} \rangle$

A non-trivial example

Earlier example, slightly extended.

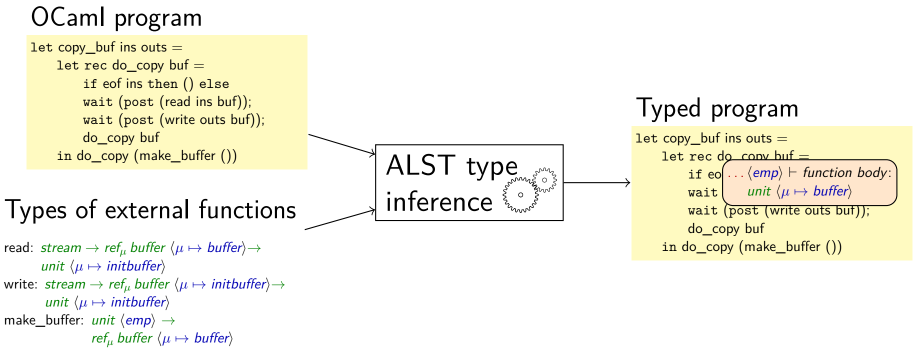
```
let copy_buf ins outs =
    let rec do_copy buf =
        if eof ins then () else
        wait (post (read ins buf));
        wait (post (write outs buf));
        do_copy buf
    in do_copy (make_buffer ())
```

$\boxed{\ldots \langle \mathit{emp} \rangle \vdash \mathit{function\ body} \colon \mathit{unit}\ \langle \mu \mapsto \mathit{buffer} \rangle}$

Outline

- The type system
- Type inference
- Some larger examples

What type inference does



OCaml program

```
let copy_buf ins outs =
    let rec do_copy buf =
        if eof ins then () else
        wait (post (read ins buf));
        wait (post (write outs buf));
        do_copy buf
    in do_copy (make_buffer ())
```

Types of external functions

read: $\textit{stream} \to \textit{ref}_\mu\ \textit{buffer}\ \langle \mu \mapsto \textit{buffer} \rangle \to \textit{unit}\ \langle \mu \mapsto \textit{initbuffer} \rangle$

write: $\textit{stream} \to \textit{ref}_\mu\ \textit{buffer}\ \langle \mu \mapsto \textit{initbuffer} \rangle \to \textit{unit}\ \langle \mu \mapsto \textit{initbuffer} \rangle$

make_buffer: $\textit{unit}\ \langle \textit{emp} \rangle \to \textit{ref}_\mu\ \textit{buffer}\ \langle \mu \mapsto \textit{buffer} \rangle$

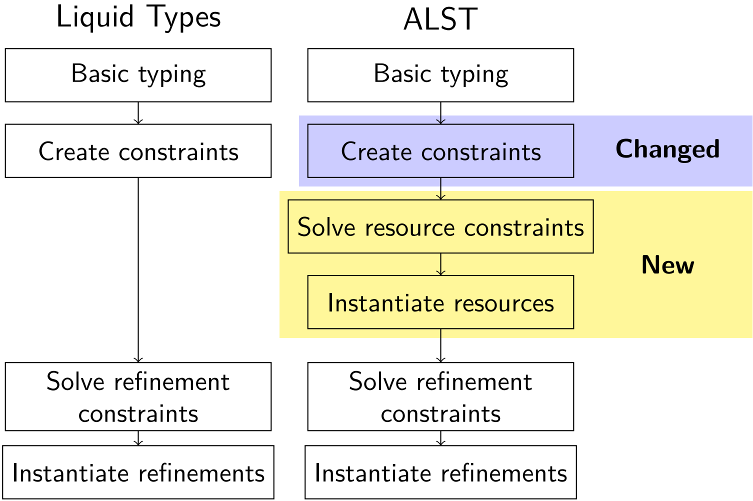
ALST type inference

Typed program

```
let copy_buf ins outs =
    let rec do_copy buf =
        if eo
        wait
        wait (post (write outs buf));
        do_copy buf
    in do_copy (make_buffer ())
```

$\ldots \langle \textit{emp} \rangle \vdash \textit{function body}: \textit{unit}\ \langle \mu \mapsto \textit{buffer} \rangle$

From liquid types to ALST



Liquid Types

Basic typing

Create constraints

Solve refinement constraints

Instantiate refinements

ALST

Basic typing

Create constraints

Changed

Solve resource constraints

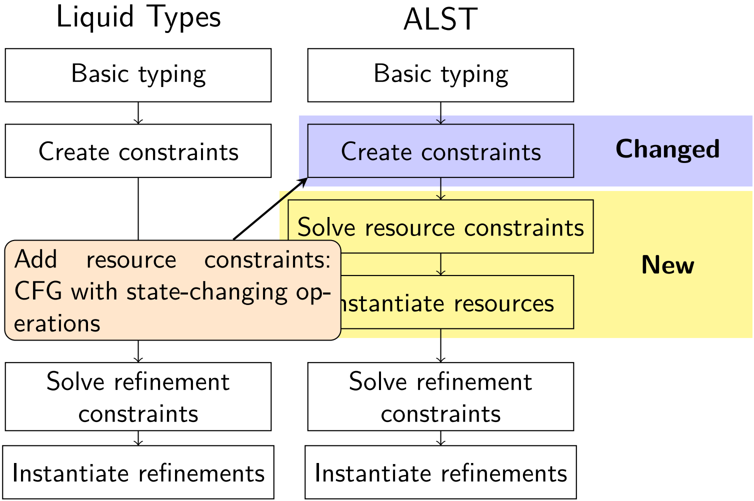
Instantiate resources

New

Solve refinement constraints

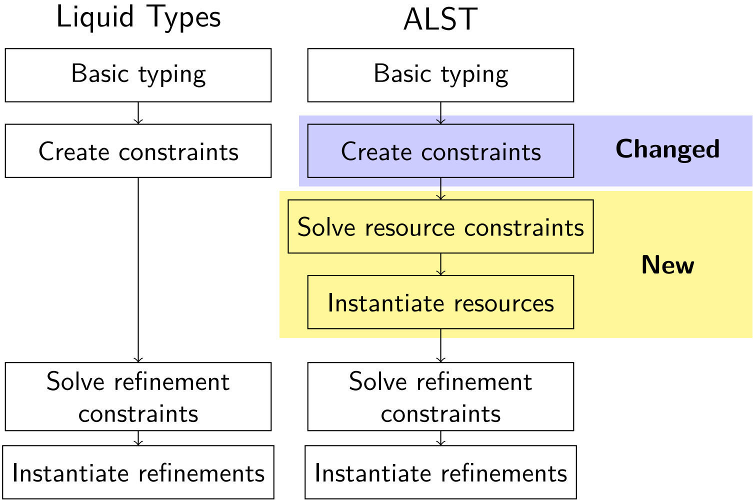
Instantiate refinements

From liquid types to ALST



Add resource constraints: CFG with state-changing operations

From liquid types to ALST



Liquid Types

Basic typing

Create constraints

Solve refinement constraints

Instantiate refinements

ALST

Basic typing

Create constraints

Changed

Solve resource constraints

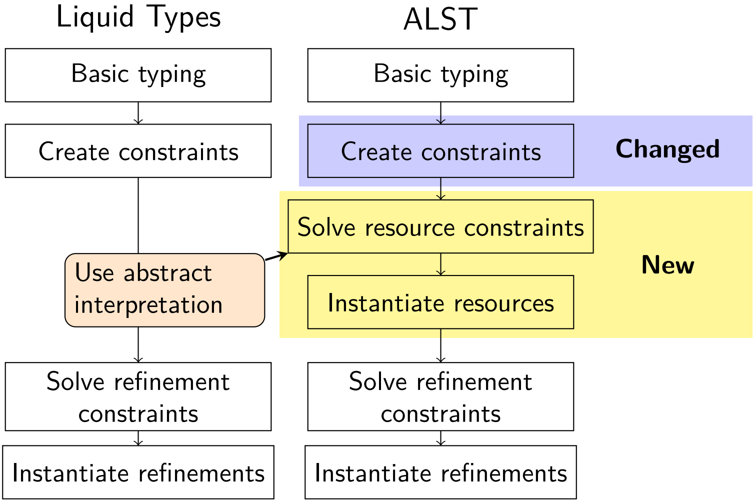
Instantiate resources

New

Solve refinement constraints

Instantiate refinements

From liquid types to ALST



Fundamental limitations

- No way to express pre-/post-condition polymorphism.
  Cannot provide a single useful type for this:

```
let rec iter f l = match l with
  | x::l -> f x; iter f l
  | [] -> ()
```

Fundamental limitations

- No way to express pre-/post-condition polymorphism.
  Cannot provide a single useful type for this:

```
let rec iter f l = match l with
   | x::l −> f x; iter f l
   | [] −> ()
```

  This is much harder than it seems: Pre-/post-condition of iter would depend on those of f.

Fundamental limitations

- No way to express pre-/post-condition polymorphism.
  Cannot provide a single useful type for this:

  ```
  let rec iter f l = match l with
    | x::l -> f x; iter f l
    | [] -> ()
  ```

  This is much harder than it seems: Pre-/post-condition of iter would depend on those of f.
  Workaround: Consider partially-applied versions of iter.

Fundamental limitations

- No way to express pre-/post-condition polymorphism.
  Cannot provide a single useful type for this:

```
let rec iter f l = match l with
  | x::l −> f x; iter f l
  | [] −> ()
```

  This is much harder than it seems: Pre-/post-condition of iter would depend on those of f.
  Workaround: Consider partially-applied versions of iter.

- No way to handle unbounded task creation.

```
let rec make_tasks l = match l with
  | x::l −> post (f x) :: make_tasks l
  | [] −> []
```

Outline

- The type system
- Type inference
- Some larger examples

Outline: Larger examples

- The type system
- Type inference
- Some larger examples
  - Several variants of copying loops.
  - Coordination code for a parallel SAT solver.
  - Code from the MirageOS FAT file system.

Outline: Larger examples

- The type system
- Type inference
- Some larger examples
    - Several variants of copying loops.
    - Coordination code for a parallel SAT solver.
    - Code from the MirageOS FAT file system.

The MirageOS FAT example

Code taken from an old version of MirageOS:

```
type state = { format: ...; mutable fat: fat_type }

let update_directory_containing x path =
  post (let c = Fat_entry.follow_chain x.format x.fat in ...)

let update x = ...
  update_directory_containing x path;
  x.fat <- List.fold_left update_allocations x.fat fat_allocations
```

The MirageOS FAT example

Code taken from an old version of MirageOS:

```
type state = { format: ...; mutable fat: fat_type }

let update_directory_containing x path =
  post (let c = Fat_entry.follow_chain x.format x.fat in ...)

let update x = ...
  update_directory_containing x path;
  x.fat <- List.fold_left update_allocations x.fat fat_allocations
```

ALST output: `Access to resource` ξ_{42} `that is wrapped in a wait permission.`

The MirageOS FAT example

Code taken from an old version of MirageOS:

```
type state = { format: ...; mutable fat: fat_type }

let update_directory_containing x path =
  post (let c = Fat_entry.follow_chain x.format x.fat in ...)

let update x = ...
  update_directory_containing x path;
  x.fat <- List.fold_left update_allocations x.fat fat_allocations
```

ALST output: `Access to resource` ξ_{42} `that is wrapped in a wait permission.`

This code has a race condition!

Summary

Asynchronous Liquid Separation Types allow reasoning about OCaml programs with asynchronous concurrency and mutable state.
They are an extension of liquid types with concurrent separation logic and allow type inference.

`http://plv.mpi-sws.org/ALSTypes`
Contact: Johannes Kloos, `jkloos@mpi-sws.org`

Related Work

Other approaches to the same problem:

1. Low-level liquid types (Rondon, Kawaguchi, Jhala; POPL 2010)
   Applies the liquid types machinery to C programs.
   No reasoning about first-class functions, no concurrency
2. Hoare Type Theory (Nanevski, Morrisett, Birkedal; JFP 2007)
   A more ambitious way of combining types and program logics.
   Geared towards type-checking and interactive theorem proving, little automated inference.
3. Mezzo (Protzenko; PhD thesis, INRIA 2014)
   Takes a radically different approach based on permissions.
   Does not provide type inference.