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

Task A

Post B

Used in:
- Web applications, AJAX
- Android programming model
- Operation systems: Interrupt handling
Introducing asynchronous concurrency

Example of asynchronous execution:

Task B

......

Post B

Task A
Introducing asynchronous concurrency

Example of asynchronous execution:

Task B

Post B

Task A

Post C

Task C

Used in:
- Web applications, AJAX
- Android programming model
- Operation systems: Interrupt handling
Introducing asynchronous concurrency

Example of asynchronous execution:

Task B

Post B

Task A

Post C

Task C
Introducing asynchronous concurrency

Example of asynchronous execution:

Task B

Task A

Task C

Post B

Post C
Introducing asynchronous concurrency

Example of asynchronous execution:

Task B

Task A

Task C

Post B

Wait B

Post C
Introducing asynchronous concurrency

Example of asynchronous execution:

Task B

Task A

Task C

Post B

Wait B

Post C

Used in:
- Web applications, AJAX
- Android programming model
- Operation systems: Interrupt handling
Introducing asynchronous concurrency

Example of asynchronous execution:

Task B

Task A

Task C

Post B

Wait B

Post C

...
Introducing asynchronous concurrency

Example of asynchronous execution:

Task B

Task A

Task C

Post B

Post C

Wait B

Used in:

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

```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.
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

```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.
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).

\[
\text{read: } \text{stream } \to \text{buffer}
\]
Challenge: Typing asynchronous code

Consider the code fragment post (read ins).

read: \textit{stream} \rightarrow \textit{buffer}
post (read ins): \textit{promise buffer}
Challenge: Typing asynchronous code

Consider the code fragment post (read ins).

\[
\text{read: } \text{stream} \rightarrow \text{buffer} \\
\text{post (read ins): } \text{promise buffer}
\]

Introduce subtyping. Suppose \( \text{initbuffer} <: \text{buffer} \).
Consider the code fragment post (read ins).

read: \textit{stream} \rightarrow \textit{buffer}  
post (read ins): \textit{promise buffer}

Introduce subtyping. Suppose \textit{initbuffer} \textless;: \textit{buffer}. 

read: \textit{stream} \rightarrow \textit{initbuffer}
Challenge: Typing asynchronous code

Consider the code fragment post (read ins).

\[
\text{read: } \text{stream} \rightarrow \text{buffer} \\
\text{post (read ins): promise buffer}
\]

Introduce subtyping. Suppose \( \text{initbuffer} <: \text{buffer} \).

\[
\text{read: } \text{stream} \rightarrow \text{initbuffer} \\
\text{post (read ins): promise initbuffer}
\]
Challenge: Typing asynchronous code

Consider the code fragment post (read ins).

\[
\text{read: } \text{stream} \rightarrow \text{buffer} \\quad \text{post (read ins): } \text{promise buffer}
\]

Introduce subtyping. Suppose \textit{initbuffer} \textless: \textit{buffer}.

\[
\text{read: } \text{stream} \rightarrow \text{initbuffer} \\quad \text{post (read ins): } \text{promise initbuffer}
\]

In the paper: refinement typing, using liquid types
Another copying loop, with buffers on the heap (no concurrency):

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

```ocaml
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 ())
```

- Main copying loop gets a pre-allocated buffer
Another copying loop, with buffers on the heap (no concurrency):

```ocaml
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):

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

```ml
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:

```ocaml
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.
```

`r : ref int`: Can’t prove anything

`r : ref int = 1`: Not true – initially, `!r` is 0

`r : ref int = 0`: Not true

We need some form of strong updates!
Subtyping and mutation don’t mix well

Easier example:

```ocaml
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:

```ocaml
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 \( int_{\leq 0} <: int \) and \( int_{\geq 1} <: int \).
Subtyping and mutation don’t mix well

Easier example:

```ocaml
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 \( \text{int}_{=0} <: \text{int} \) and \( \text{int}_{=1} <: \text{int} \).

- \( r:\text{ref int} \): Can’t prove anything
- \( r:\text{ref int}_{=1} \): Not true – initially, \(!r\) is 0
- \( r:\text{ref int}_{=0} \): Not true
Subtyping and mutation don't mix well

Easier example:

```plaintext
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 \textit{int}_0 <: \textit{int} and \textit{int}_1 <: \textit{int}.

- \textit{r:ref int}: Can't prove anything
- \textit{r:ref int}_1: Not true – initially, !r is 0
- \textit{r:ref int}_0: Not true

We need some form of strong updates!
Program logics work well with updating state

```
{ emp }
let r = ref 0 in
{ r ↦→ 0 }
assert (!r = 0);
{ r ↦→ 0 }
r := 1;
{ r ↦→ 1 }
assert (!r = 1);
{ r ↦→ 1 }
```
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.

\[
\text{Liquid type} + \text{Concurrent Separation Logic} = \text{Asynchronous Liquid Separation Types (ALST)}
\]
Outline

- The type system
- Type inference
- Some larger examples
Outline

- The type system
- Type inference
- Some larger examples
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 ())
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

\[ \Gamma \vdash e : \tau \]

- Environment
- Expression
- Type
Extended typing judgment

\[ \Gamma \vdash e : \tau \]

<table>
<thead>
<tr>
<th>Environment</th>
<th>Expression</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \langle \text{pre} \rangle )</td>
<td>( \vdash e : \tau )</td>
<td>( \langle \text{post} \rangle )</td>
</tr>
</tbody>
</table>
Typing the example

\[\text{ins: \textit{stream}, buf: \textit{ref buffer}} \quad \langle \quad \rangle\]

\[\vdash \text{read ins buf: \textit{unit}} \quad \langle \quad \rangle\]
Typing the example

\[
\text{ins: } \textit{stream}, \textbf{buf: } \textit{ref } \textit{buffer} \quad \langle \mu \mapsto \textit{buffer} \rangle \\
\vdash \text{read ins buf: } \textit{unit} \quad \langle \mu \mapsto \textit{initbuffer} \rangle
\]

- Precondition: Content of heap cell \( \mu \) has type \textit{buffer}
- Postcondition: Content of heap cell \( \mu \) has type \textit{initbuffer}
Typing the example

\[
\text{ins: } stream, \text{ buf: } \text{ref}_{\mu} \text{ buffer} \quad \langle \mu \mapsto \text{buffer} \rangle
\]
\[
\vdash \text{read ins buf: } \text{unit} \quad \langle \mu \mapsto \text{initbuffer} \rangle
\]

- Precondition: Content of heap cell \( \mu \) has type \( \text{buffer} \)
- Postcondition: Content of heap cell \( \mu \) has type \( \text{initbuffer} \)
- \( \text{buf} \) is a reference to \( \mu \)
Typing the example

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

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

- Precondition: Content of heap cell \( \mu \) has type \textit{buffer}
- Postcondition: Content of heap cell \( \mu \) has type \textit{initbuffer}
- \textit{buf} is a reference to \( \mu \)
- \textbf{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

- `ins: stream, buf: ref_\mu buffer \langle \mu \mapsto buffer \rangle`
- `\vdash read ins buf: unit \langle \mu \mapsto initbuffer \rangle`
We can even handle concurrency

Consider modified code fragment:

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

Known: Type of read ins buf

- `ins: stream, buf: refμ buffer  ⟨μ ↦→ buffer⟩`
- `⊢ read ins buf: unit  ⟨μ ↦→ initbuffer⟩`

The post expression:

- `ins: stream, buf: refμ buffer  ⟨μ ↦→ buffer⟩`
- `⊢ post (read ins buf):
  ⟨⟩`
We can even handle concurrency

Consider modified code fragment:

\[
\text{post (read ins buf)}
\]

Known: Type of read ins buf

\[
\begin{align*}
\text{ins: } & \text{stream}, \text{buf: } \text{ref}_\mu \text{ buffer} \quad \langle \mu \mapsto \text{buffer}\rangle \\
\vdash & \text{read ins buf: } \text{unit} \quad \langle \mu \mapsto \text{initbuffer}\rangle
\end{align*}
\]

The post expression:

\[
\begin{align*}
\text{ins: } & \text{stream}, \text{buf: } \text{ref}_\mu \text{ buffer} \quad \langle \mu \mapsto \text{buffer}\rangle \\
\vdash & \text{post (read ins buf): } \\
& \text{promise}_\pi \text{ unit} \quad \langle \rangle
\end{align*}
\]
We can even handle concurrency

Consider modified code fragment:

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

Known: Type of `read ins buf`

- `ins: stream, buf: ref_\mu buffer \langle \mu \mapsto buffer \rangle`
- `\vdash read ins buf: \textit{unit} \langle \mu \mapsto initbuffer \rangle`

The post expression:

```
ins: stream, buf: ref_\mu buffer \langle \mu \mapsto buffer \rangle
\vdash post (read ins buf):
  \textit{promise}_\pi \textit{unit} \langle \text{Wait}(\pi, \mu \mapsto initbuffer) \rangle
```

Package resources produced by task in `wait permission`
A non-trivial example

Earlier example, slightly extended.

```ml
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.

```ocaml
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 ());
```

Given read: 

```
...⟨μ ↦→ buffer⟩ ⊢ read ins buf : unit⟨μ ↦→ initbuffer⟩
```
A non-trivial example

Earlier example, slightly extended.

```ocaml
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 ()
    ...⟨μ↦→buffer⟩ ⊢ post (read ins buf):
    promise_π unit ⟨Wait(π, μ ↦→ initbuffer)⟩
```

J. Kloos (MPI-SWS)
Asynchronous Liquid Separation Types
ECOOP 2015 15 / 23
A non-trivial example

Earlier example, slightly extended.

```ocaml
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 ()
```

\[ \langle \mu \mapsto \text{buffer} \rangle \vdash \text{wait (post (read ins buf))}: \text{unit} \langle \mu \mapsto \text{initbuffer} \rangle \]
A non-trivial example

Earlier example, slightly extended.

```ocaml
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 ()

...\langle \mu \mapsto \text{buffer} \rangle \vdash \text{read, then write:}
unit\langle \mu \mapsto \text{initbuffer} \rangle
```
A non-trivial example

Earlier example, slightly extended.

```ocaml
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 ())
```

\[ \langle \mu \mapsto \text{buffer} \rangle \vdash \text{loop body:} \]

\[ \text{unit} \langle \mu \mapsto \text{buffer} \rangle \]
A non-trivial example

Earlier example, slightly extended.

```ocaml
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 ()
... ⟨emp⟩ ⊢ function body: unit ⟨μ ↦ buffer⟩
```
Outline

- The type system
- Type inference
- Some larger examples
OCaml program

```ocaml
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**: \( \text{stream} \rightarrow \text{ref}_{\mu} \text{buffer} \langle \mu \mapsto \text{buffer} \rangle \rightarrow \text{unit} \langle \mu \mapsto \text{initbuffer} \rangle \)
- **write**: \( \text{stream} \rightarrow \text{ref}_{\mu} \text{buffer} \langle \mu \mapsto \text{initbuffer} \rangle \rightarrow \text{unit} \langle \mu \mapsto \text{initbuffer} \rangle \)
- **make_buffer**: \( \text{unit} \langle \text{emp} \rangle \rightarrow \text{ref}_{\mu} \text{buffer} \langle \mu \mapsto \text{buffer} \rangle \)

Typed program

```ocaml
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 ())
```

**ALST type inference**
From liquid types to ALST

Liquid Types
- Basic typing
- Create constraints
- Solve refinement constraints
- Instantiate refinements

ALST
- Basic typing
- Create constraints
- Solve resource constraints
- Instantiate resources
- Solve refinement constraints
- Instantiate refinements

Changed

New
From liquid types to ALST

**Liquid Types**
- Basic typing
- Create constraints
- Add resource constraints: CFG with state-changing operations
- Solve refinement constraints
- Instantiate refinements

**ALST**
- Basic typing
- Create constraints
- Solve resource constraints
- Instantiate resources
- Solve refinement constraints
- Instantiate refinements

**Changed**
- Add resource constraints: CFG with state-changing operations

**New**
From liquid types to ALST

Liquid Types
- Basic typing
- Create constraints
- Solve refinement constraints
- Instantiate refinements

ALST
- Basic typing
- Create constraints
- Solve resource constraints
- Instantiate resources

New

Changed
From liquid types to ALST

**Liquid Types**
- Basic typing
- Create constraints
- Use abstract interpretation
- Solve refinement constraints
- Instantiate refinements

**ALST**
- Basic typing
- Create constraints
- Solve resource constraints
- Instantiate resources
- Solve refinement constraints
- Instantiate refinements

**Changed**

**New**

J. Kloos (MPI-SWS)
No way to express pre-/post-condition polymorphism. Cannot provide a single useful type for this:

```ocaml
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:

```ocaml
let rec make_tasks l = match l with
    | x::l -> post (f x) :: make_tasks l
    | [] -> []
```
Fundamental limitations

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

  ```ocaml
  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:

```ocaml
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.
Workarounds: Consider partially-applied versions of iter.
Fundamental limitations

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

```ml
definition 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.

```ml
definition 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:

```ocaml
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
```

This code has a race condition!
The MirageOS FAT example

Code taken from an old version of MirageOS:

```ocaml
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 $\xi_{42}$ that is wrapped in a wait permission.
The MirageOS FAT example

Code taken from an old version of MirageOS:

```ocaml
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!
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.

   Takes a radically different approach based on permissions.
   Does not provide type inference.