Asynchronous Liquid Separation Types

Johannes Kloos, Rupak Majumdar, and Viktor Vafeiadis

Max Planck Institute for Software Systems, Germany
{ jkloos, rupak, viktor }@mpi-sws.org

Abstract

We present a refinement type system for reasoning about asynchronous programs manipulating shared mutable state. Our type system guarantees the absence of races and the preservation of user-specified invariants using a combination of two ideas: refinement types and concurrent separation logic. Our type system allows precise reasoning about programs using two ingredients. First, our types are indexed by sets of resource names and the type system tracks the effect of program execution on individual heap locations and task handles. In particular, it allows making strong updates to the types of heap locations. Second, our types track ownership of shared state across concurrently posted tasks and allow reasoning about ownership transfer between tasks using permissions. We demonstrate through several examples that these two ingredients, on top of the framework of liquid types, are powerful enough to reason about correct behavior of practical, complex, asynchronous systems manipulating shared heap resources.

We have implemented type inference for our type system and have used it to prove complex invariants of asynchronous OCaml programs. We also show how the type system detects subtle concurrency bugs in a file system implementation.

1998 ACM Subject Classification F.3.1 Specifying and Verifying and Reasoning about Programs, D.2.4 Software/Program Verification

Keywords and phrases Liquid Types, Asynchronous Parallelism, Separation Logic, Type Systems

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2015

1 Introduction

Asynchronous programming is a common programming idiom used to handle concurrent interactions. It is commonly used not only in low-level systems code, such as operating systems kernels and device drivers, but also in internet services, in programming models for mobile applications, in GUI event loops, and in embedded systems.

An asynchronous program breaks the logical units that comprise its functionality into atomic, non-blocking, sections called tasks. Each task performs some useful work, and then schedules further tasks to continue the work as necessary. At run time, an application-level co-operative scheduler executes these tasks in a single thread of control. Since tasks execute atomically and the scheduler is co-operative, asynchronous programs must ensure that each task runs for a bounded time. Thus, blocking operations such as I/O are programmed in an asynchronous way: an asynchronous I/O operation returns immediately whether or not the data is available, and returns a promise that is populated once the data is available. Conversely, a task can wait on a promise, and the scheduler will run such a task once the data is available. In this way, different logical units of work can make simultaneous progress.

In recent years, many programming languages provide support for asynchronous programming, either as native language constructs (e.g., in Go [8] or Rust [30]) or as libraries (e.g., libevent [18] for C, Async [20] and Lwt [33] for OCaml). These languages and libraries allow the programmer to write and compose tasks in a monadic style and a type checker

enforces some basic invariants about the data being passed around. However, reasoning about finer-grained invariants, especially involving shared resources, remains challenging. Furthermore, the loose coupling between different enabled handlers which may interact through shared resources gives rise to many subtle bugs in these programs.

In this paper, we focus on asynchronous programs written in OCaml, whose type system already guarantees basic memory safety, and seek to extend the guarantees that can be provided to the programmers. Specifically, we would like to be able to automatically verify basic correctness properties such as race-freedom and the preservation of user-supplied invariants. To achieve this goal, we combine two well-known techniques, *refinement types* and *concurrent separation logic*, into a type system we call *asynchronous liquid separation types* (ALS types).

Refinement types [7, 35] are good at expressing invariants that are needed to prove basic correctness properties. For example, to ensure that that array accesses never go out of bounds, one can use types such as $\{x : int \mid 0 \leq x < 7\}$. Moreover, in the setting of *liquid types* [29], many such refinement types can be inferred automatically, relieving the programmer from having to write any annotations besides the top-level specifications. However, existing refinement type systems do not support concurrency and shared state.

On the other hand, concurrent separation logic (CSL) [22] is good at reasoning about concurrency: its rules can handle strong updates in the presence of concurrency. Being an expressive logic, CSL can, in principle, express all the invariants expressible via refinement types, but in doing so, gives up on automation. Existing fully automated separation logic tools rely heavily on shape analysis (e.g. [5, 3]) and can find invariants describing the pointer layout of data structures on the heap, but not arbitrary properties of their content.

Our combination of the two techniques inherits the benefits of each. In addition, using liquid types, we automate the search for refinement type annotations over a set of user-supplied predicates using an SMT solver. Given a program and a set of predicates, our implementation can automatically infer rich data specifications in terms of these predicates for asynchronous OCaml programs, and can prove the preservation of user-supplied invariants, as well as the absence of memory errors, such as array out of bounds accesses, and concurrency errors, such as data races. This is achieved by extending the type inference procedure of liquid types, adding a step that derives the structure of the program's heap and information about ownership of resources using an approach based on abstract interpretation. Specifically, our system was able to infer a complex invariant in a parallel SAT solve and detect a subtle concurrency bug in a file system implementation.

Because of limited space, we have omitted proofs and technical details in this version of the paper. A full version can be found on our website [14].

Outline The remainder of our paper is structured as follows:

§2 We introduce a small ML-like language with asynchronous tasks, and present a number of small examples motivating the main features of our type system.

§3 We give a formal presentation of our type system and state the type safety theorem.

§4 We discuss how we perform type inference by extending the liquid typing algorithm [29] with a static analysis.

§5 We evaluate our type inference implementation on a number of case studies that include a file system and a parallel SAT solver. We discuss a subtle concurrency error uncovered in an asynchronous file system implementation.

§6 We discuss limitations of ALS types.

§7 We discuss related work.

$$
\begin{array}{rrl}
c & \text{Constants} & x, f \quad \text{Variables} \\
\ell \in \mathsf{Locs} & \text{Heap locations} & p \in \mathsf{Tasks} \quad \text{Task handles} \\
v \in \mathsf{Values} & ::= c \mid x \mid \lambda x.\, e \mid \mathbf{rec}\, f\, x\, e \mid \ell \mid p & \\
e & ::= v \mid e\, e \mid \mathbf{ref}\ e \mid\ !e \mid e\ \texttt{:=}\ e \mid \mathbf{post}\ e \mid \mathbf{wait}\ e \mid \mathbf{if}\ e\ \mathbf{then}\ e\ \mathbf{else}\ e & \\
t \in \mathsf{TaskStates} & ::= \mathsf{run}\colon e \mid \mathsf{done}\colon v & \\
H & := \mathsf{Locs} \rightharpoonup \mathsf{Values} & \text{Heaps} \\
P & := \mathsf{Tasks} \rightharpoonup \mathsf{TaskStates} & \text{Task buffers}
\end{array}
$$

Figure 1 The core calculus.

$$
\textsc{EL-Post}\ \frac{p \text{ fresh w.r.t. } P, p_r}{(\mathbf{post}\ e, H, P) \hookrightarrow_{p_r} (p, H, P[p \mapsto \mathsf{run}\colon e])}
\qquad
\textsc{EL-WaitDone}\ \frac{P(p) = \mathsf{done}\colon v}{(\mathbf{wait}\ p, H, P) \hookrightarrow_{p_r} (v, H, P)}
$$

$$
\textsc{EG-Local}\ \frac{(e, H, P) \hookrightarrow_{p_r} (e', H', P') \qquad p_r \notin \operatorname{dom} P}{(H, P[p_r \mapsto \mathsf{run}\colon e], p_r) \hookrightarrow (H', P'[p_r \mapsto \mathsf{run}\colon e'], p_r)}
$$

$$
\textsc{EG-WaitRun}\ \frac{P(p_2) = \mathsf{run}\colon \_ \qquad P(p_1) = \mathsf{run}\colon \mathcal{C}[\mathbf{wait}\ p] \qquad P(p) = \mathsf{run}\colon \_}{(H, P, p_1) \hookrightarrow (H, P, p_2)}
$$

$$
\textsc{EG-Finished}\ \frac{P(p_2) = \mathsf{run}\colon \_ \qquad P(p_1) = \mathsf{run}\colon v \qquad p_1 \neq p_2}{(H, P, p_1) \hookrightarrow (H, P[p_1 \mapsto \mathsf{done}\colon v], p_2)}
$$

Figure 2 Small-step semantics.

2 Examples and Overview

2.1 A core calculus for asynchronous programming

For concreteness, we base our formal development on a small λ calculus with recursive functions, ML-style references, and two new primitives for asynchronous concurrency: **post** e that creates a new task that evaluates the expression e and returns a handle to that task; and **wait** e that evaluates e to get a task handle p, waits for the completion of task with handle p, and returns the value that the task yields. Figure 1 shows the core syntax of the language; for readability in examples, we use standard syntactic sugar (e.g., let).

The semantics of the core calculus is largely standard, and is presented in a small-step operational fashion. We have two judgments: (1) the *local semantics*, $(e, H, P) \hookrightarrow_{p_r} (e', H', P')$, that describe the evaluation of the active task, p_r, and (2) the *global semantics*, $(H, P, p) \hookrightarrow (H', P', p')$, that describe the evaluation of the system as a whole. Figure 2 shows the local semantics rules for posts and waits, as well as the global semantic rules.

In more detail, local configurations consist of the expression being evaluated, e, the heap, H, and the task buffer, P. We model heaps as partial maps from locations to values, and task buffers as partial maps from task handles to task states. A task state can be either a running task containing the expression yet to be evaluated, or a finished task containing some value. We assume that the current process being evaluated is not in the task buffer, $p_r \notin \operatorname{dom} P$. Evaluation of a **post** e expression generates a new task with the expression e, while **wait** p reduces only if the referenced task has finished, in which case its value is returned. For the standard primitives, we follow the OCaml semantics. In particular, evaluation uses right-to-left call-by-value reduction.

Global configurations consist of the heap, H, the task buffer, P, and the currently active task, p. As the initial configuration of an expression e, we take $(\emptyset, [p_0 \mapsto \mathsf{run}: e], p_0)$. A global step is either a local step (EG-Local), or a scheduling step induced by the wait instruction when the task waited for is still running (EG-WaitRun) or the termination of a task (EG-Finished). In these cases, some other non-terminated task p_2 is selected as the active task.

2.2 Promise types

We now illustrate our type system using simple programs written in the core calculus. They can be implemented easily in OCaml using libraries such as Lwt [33] and Async [20]. If expression e has type α, then **post** e has type $\mathsf{promise}\,\alpha$, a promise for the value of type α that will eventually be computed. If the type of e is $\mathsf{promise}\,\alpha$, then **wait** e types as α.

As a simple example using these operations, consider the following function that copies data from an input stream `ins` to an output stream `outs`:

```
let rec copy1 ins outs =
  let buf = wait (post (read ins)) in
  let _ = wait (post (write outs buf buf)) in
  if eof ins then () else copy1 ins outs
```

where the read and write operations have the following types:

```
read: stream → buffer      write: stream → buffer → unit
```

and `eof` checks if `ins` has more data. The code above performs (potentially blocking) reads and writes asynchronously[1]. It posts a task for reading and blocks on its return, then posts a task for writing and blocks on its return, and finally, calls itself recursively if more data is to be read from the stream. By posting `read` and `write` tasks, the asynchronous style enables other tasks in the system to make progress: the system scheduler can run other tasks while `copy1` is waiting for a read or write to complete.

2.3 Refinement types

In the above program, the ML type system provides coarse-grained invariants that ensure that the data type eventually returned from `read` is the same data type passed to `write`. To verify finer-grained invariants, in a sequential setting, one can augment the type system with *refinement types* [35, 29]. For example, in a refinement type, one can write $\{\nu : \mathtt{int} \mid \nu \geq 0\}$ for refinement of the integer type that only allows non-negative values. In general, a refinement type of the form $\{\nu : \tau \mid p(\nu)\}$ is interpreted as a subtype of τ where the values v are exactly those values of τ that satisfy the predicate $p(v)$. A subtyping relation between types $\{\nu : \tau \mid \rho_1\}$ and $\{\nu : \tau \mid \rho_2\}$ can be described informally as "all values that satisfy ρ_1 must also satisfy ρ_2"; this notion is made precise in Section 3.

For purely functional asynchronous programs, the notion of type refinements carries over transparently, and allows reasoning about finer-grain invariants. For example, suppose we know that the read operation always returns buffers whose contents have odd parity. We can express this by refining the type of `read` to $\mathtt{stream} \to \mathsf{promise}\,\{\nu : \mathtt{buffer} \mid \mathrm{odd}(\nu)\}$. Dually, we can require that the write operation only writes buffers whose contents have odd parity

[1] We assume that reading an empty input stream simply results in an empty buffer; an actual implementation will have to guard against I/O errors

by specifying the type `stream` $\to \{\nu : \texttt{buffer} \mid \text{odd}(\nu)\} \to$ promise unit. Using the types for **post** and **wait**, it is simple to show that the code still types in the presence of refinements.

Thus, for purely functional asynchronous programs, the machinery of refinement types and SMT-based implementations such as liquid types [29] generalize transparently and provide powerful reasoning tools. The situation is more complex in the presence of shared state.

2.4 Refinements and state: strong updates

Shared state complicates refinement types even in the sequential setting. Consider the following sequential version of copy, where read and write take a heap-allocated buffer:

```
let seqcp ins outs = let b = ref empty_buffer in readb ins b; writeb outs b
```

where `readb, writeb: stream` $\to$ ref `buffer` $\to$ unit.[2] As subtyping is unsound for references (see, e.g., [23, §15.5]), it is not possible to track the precise contents of a heap cell by modifying the refinement predicate in the reference type. One symptom of this unsoundness is that there can be multiple instances of a reference to a heap cell, say $x_1, \ldots, x_n$, with types ref $\tau_1, \ldots,$ ref τ_n. It can be shown that all the types $\tau_1, \ldots, \tau_n$ must be essentially the same: Suppose, for example, that $\tau_1 = \texttt{int}_{=1}$ and $\tau_2 = \texttt{int}_{\geq 0}$. Suppose furthermore that x_1 and x_2 point to the same heap cell. Then, using standard typing rules, the following piece of code would type as $\texttt{int}_{=1}$: x_2 `:= 2; !`x_1. But running the program would return 2, breaking type safety. By analogy with static analysis, we call references typed like in ordinary ML *weak references*, and updates using only weak references *weak updates*. Their type only indicates which values a heap cell can possibly take over the execution of a whole program, but not its current contents.

Therefore, to track refinements over changes in the mutable state, we modify the type system to perform *strong updates* that track such changes. For this, our type system includes preconditions and postconditions that explicitly describe the global state before and after the execution of an expression. We also augment the types of references to support strong updates, giving us *strong references*.

Resource sets To track heap cells and task handles in pre- and postconditions, we introduce *resource names* that uniquely identify each resource. At the type level, global state is described using *resource sets* that map resource names to types. Resource sets are written using a notation inspired from separation logic. For example, the resource set $\mu \mapsto \{\nu : \texttt{buffer} \mid \text{odd}(\nu)\}$ describes a heap cell that is identified by the resource name μ and contains a value of type $\{\nu : \texttt{buffer} \mid \text{odd}(\nu)\}$.

To connect references to resources, reference types are extended with indices ranging over resource names. For example, the reference ref_μ `buffer` denotes a reference that points to a heap cell with resource name μ and that contains a value of type `buffer`. In general, given a reference type $\text{ref}_\mu\, \tau$ and a resource set including $\mu \mapsto \tau'$, we ensure τ' is a subtype of τ. Types of the form $\text{ref}_\mu\, \tau$ are called *strong references*.

Full types Types and resource sets are tied together by using *full types* of the form $И\Xi.\, \tau\langle\eta\rangle$, where τ is a type, η is a resource set, and Ξ is a list of resource names that are considered "not yet bound." The $И$ binder indicates that all names in Ξ must be fresh, and therefore, distinct from all names occurring in the environment. For example, if expression e has type

[2] We write ref `buffer` instead of `buffer ref` for ease of readability.

$И\mu.\,\mathsf{ref}_\mu\,\tau\langle\mu \mapsto \tau'\rangle$, it means that e will return a reference to a newly-allocated memory cell with a fresh resource name μ, whose content has type τ', and $\tau' \preceq \tau$.

We use some notational shorthands to describe full types. We omit quantifiers if nothing is quantified: $И\cdot.\,\tau\langle\eta\rangle = \tau\langle\eta\rangle$ and $\forall\cdot.\tau = \tau$. If a full type has an empty resource set, it is identified with the type of the return value, like this: $\tau\langle\mathsf{emp}\rangle = \tau$.

To assign a full type to an expression, the global state in which the expression is typed must be given as part of the environment. More precisely, the typing judgment is given as $\Gamma;\eta \vdash e : И\Xi.\,\tau\langle\eta'\rangle$. It reads as follows: in the context Γ, when starting from a global state described by η, executing e will return a value of type τ and a global state matching η', after instantiating the resource names in Ξ. As an example, the expression **ref** `empty_buffer` would type as $\cdot;\mathsf{emp} \vdash \ldots : И\mu.\,\mathsf{ref}_\mu\,\mathtt{buffer}\langle\mu \mapsto \mathtt{buffer}\rangle$.

To type functions properly, we need to extend function types to capture the functions' effects. For example, consider an expression e that types as $\Gamma, x : \tau_x;\eta \vdash e : \varphi$. If we abstract it to a function, its type will be $x{:}\,\tau_x\langle\eta\rangle \to \varphi$, describing a function that takes an argument of type τ_x and, if executed in a state matching η, will return a value of full type φ.

Furthermore, function types admit name quantification. Consider the expression e given by `!`x `+ 1`. Its type is $\mu, x : \mathsf{ref}_\mu\,\mathtt{int};\mu \mapsto \mathtt{int} \vdash e : \mathtt{int}\langle\mu \mapsto \mathtt{int}\rangle$. By lambda abstraction,

$$\mu;\mathsf{emp} \vdash \lambda x.e : \tau\langle\mathsf{emp}\rangle \text{ with } \tau = x{:}\,\mathsf{ref}_\mu\,\mathtt{int}\langle\mu \mapsto \mathtt{int}\rangle \to \mathtt{int}\langle\mu \mapsto \mathtt{int}\rangle.$$

To allow using this function with arbitrary references, the name μ can be universally quantified:

$$\cdot;\mathsf{emp} \vdash \lambda x.e : \tau\langle\mathsf{emp}\rangle \text{ with } \tau = \forall\mu.(x{:}\,\mathsf{ref}_\mu\,\mathtt{int}\langle\mu \mapsto \mathtt{int}\rangle \to \mathtt{int}\langle\mu \mapsto \mathtt{int}\rangle).$$

In the following, if a function starts with empty resource set as a precondition, we omit writing the resource set: $x{:}\,\tau_x\langle\mathsf{emp}\rangle \to \varphi$ is written as $x : \tau_x \to \varphi$.

As an example, the type of the `readb` function from above would be:

$$\mathtt{readb} : \mathtt{stream} \to \forall\mu.(b{:}\,\mathsf{ref}_\mu\,\mathtt{buffer}\langle\mu \mapsto \mathtt{buffer}\rangle \to \mathsf{unit}\langle\mu \mapsto \{\nu : \mathtt{buffer} \mid \mathrm{odd}\ \nu\}\rangle$$

$$\mathtt{writeb} : \mathtt{stream} \to \forall\mu.(b{:}\,\mathsf{ref}_\mu\,\mathtt{buffer}\langle\mu \mapsto \{\nu : \mathtt{buffer} \mid \mathrm{odd}\ \nu\}\rangle \to \mathsf{unit}\langle\mu \mapsto \{\nu : \mathtt{buffer} \mid \mathrm{odd}\ \nu\}\rangle$$

2.5 Asynchrony and shared resources

The main issue in ensuring safe strong updates in the presence of concurrency is that aliasing control now needs to extend across task boundaries: if task 1 modifies a heap location, all other tasks with access to that location must be aware of this. Otherwise, the following race condition may be encountered: suppose task 1 and task 2 are both scheduled, and heap location ξ_1 contains the value 1. During its execution, task 1 modifies ξ_1 to hold the value 2, whereas task 2 outputs the content of ξ_1. Depending on whether task 1 or task 2 is run first by the scheduler, the output of the program differs. A precise definition of race conditions for asynchronous programs can be found in [26].

To understand the interaction between asynchronous calls and shared state, consider the more advanced implementation of the copying loop that uses two explicitly allocated buffers and a double buffering strategy:

```
let copy2 ins outs =
  let buf1 = ref empty_buffer and buf2 = ref empty_buffer in
  let loop bufr bufw =
    let drain_bufw = post (writeb outs bufw) in
```

```
  if eof ins then wait drain_bufw else
    let fill_bufr = post (readb ins bufr) in
    wait drain_bufw; wait fill_bufr; loop bufw bufr
in wait (post (readb ins buf1));
loop buf2 buf1
```

where `readb` : `stream` → ref `buffer` → unit and `writeb` : `stream` → ref `buffer` → unit. The double buffered copy pre-allocates two buffers, `buf1` and `buf2` that are shared between the reader and the writer. After an initial read to fill `buf1`, the writes and reads are pipelined so that at any point, a read and a write occur concurrently.

A key invariant is that the buffer on which `writeb` operates and the buffer on which the concurrent `readb` operates are distinct. Intuitively, the invariant is maintained by ensuring that there is exactly one owner for each buffer at any time. The main loop transfers ownership of the buffers to the tasks it creates and regains ownership when the tasks terminate.

Our type system explicitly tracks resource ownership and transfer. As in concurrent separation logic, resources describe the *ownership* of heap cells by tasks. The central idea of the type system is that at any point in time, each resource is owned by at most one task. This is implemented by explicit notions of resource ownership and resource transfer.

Ownership and transfer In the judgment $\Gamma; \eta \vdash e : \varphi$, the task executing e owns the resources in η, meaning that for any resource in η, no other existing task will try to access this resource. When a task p_1 creates a new task p_2, p_1 may relinquish ownership of some of its resources and pass them to p_2; this is known as resource transfer. Conversely, when task p_1 waits for task p_2 to finish, it may also acquire the resources that p_2 holds.

In the double-buffered copying loop example, multiple resource transfers take place. Consider the following slice e_{once} of the code:

```
let task = post (writeb outs buf2) in readb ins buf1; wait task
```

Suppose this code executes in task p_1, and the task created by the **post** statement is p_2. Suppose further that `buf1` has type ref_{μ_1} `buffer` and `buf2` has type $\mathsf{ref}_{\mu_2}\{\nu : \texttt{buffer} \mid \text{odd } \nu\}$. Initially, p_1 has ownership of μ_1 and μ_2. After executing the **post** statement, p_1 passes ownership of μ_2 to p_2 and keeps ownership of μ_1. After executing **wait** `task`, p_1 retains ownership of μ_1, but also regains μ_2 from the now-finished p_2.

Wait permissions The key idea to ensure that resource transfer is performed correctly is to use *wait permissions*. A wait permission is of the form $\mathsf{Wait}(\pi, \eta)$. It complements a promise by stating which resources (namely the resource set η) may be gained from the terminating task identified by the name π. In contrast to promises, a wait permission may only be used once, to avoid resource duplication. In the following, we use the abbreviations $\mathsf{B} := \texttt{buffer}$ and $\mathsf{B}_{\mathsf{odd}} := \{\nu : \texttt{buffer} \mid \text{odd } \nu\}$. Consider again the code slice e_{once} from above. Using ALS types, it types as follows:

$$\Gamma; \mu_r \mapsto \mathsf{B} * \mu_w \mapsto \mathsf{B}_{\mathsf{odd}} \vdash e_{\text{once}} : \varphi \text{ with } \varphi = \mathsf{unit}\langle \mu_r \mapsto \mathsf{B}_{\mathsf{odd}} * \mu_w \mapsto \mathsf{B}_{\mathsf{odd}} \rangle$$

To illustrate the details of resource transfer, consider a slice of e_{once} where the preconditions have been annotated as comments:

```
(* μw ↦ Bodd * μr ↦ B*)
let drain_bufw = post (writeb outs bufw) in
(* Wait(πw, μw ↦ Bodd) * μr ↦ B*)
let fill_bufr = post (readb ins bufr) in
```

ρ	Refinement expressions	$\tau ::= \{\nu : \beta \mid \rho\} \mid x{:}\tau\langle\eta\rangle \to \varphi \mid \mathsf{ref}_\mu\,\tau \mid \mathsf{promise}_\pi\,\tau \mid \forall\xi.\tau$
β	Base types	$\eta ::= \mathsf{emp} \mid \mu \mapsto \tau \mid \mathsf{Wait}(\pi, \eta) \mid \eta * \eta$
μ, π, ξ	Resource names	$\varphi ::= И\Xi.\,\tau\langle\eta\rangle$
$\Xi ::= \cdot \mid \Xi, \xi$		$\Gamma ::= \cdot \mid \Gamma, x : \tau \mid \Gamma, \xi$

Figure 3 Syntax of ALS types.

```
(* Wait(π_w, μ_w ↦ B_odd) * Wait(π_r, μ_r ↦ B_odd)*)
wait drain_bufw;
(* μ_w ↦ B_odd * Wait(π_r, μ_r ↦ B_odd)*)
wait fill_bufr;
(* μ_w ↦ B_odd * μ_r ↦ B_odd *)
loop bufw bufr
```

Note how the precondition of **wait** `drain_bufw` contains a wait permission $\mathsf{Wait}(\pi_r, \mu_r \mapsto \mathsf{B}_{\mathsf{odd}})$. The resource set $\mu_r \mapsto \mathsf{B}_{\mathsf{odd}}$ describes the postcondition of `readb ins bufr`, and this is the resource set that will be returned by a **wait**.

2.6 Detecting concurrency pitfalls

We now indicate how our type system catches common errors. Consider the following incorrect code that has a race condition:

```
let task = post (writeb outs buf1) in readb ins buf1; wait task
```

Suppose `buf1` types as $\mathsf{ref}_\mu\,\mathsf{B}$. For the code to type check, both p_1 and p_2 would have to own μ. This is, however, not possible by the properties of resource transfer because resources cannot be duplicated. Thus, our type system rejects this incorrect program.

Similarly, suppose the call to the main loop incorrectly passed the same buffer twice: `loop buf1 buf1`. Then, `loop buf1 buf1` would have to be typed with precondition $\mu_1 \mapsto \mathsf{B} * \mu_1 \mapsto \mathsf{B}_{\mathsf{odd}}$. But this resource set is not wellformed, so this code does not type check.

Finally, suppose the order of the buffers was swapped in the initial call to the loop: `loop buf1 buf2`. Typing `loop buf1 buf2` requires a precondition $\mu_1 \mapsto \mathsf{B}_{\mathsf{odd}} * \mu_2 \mapsto \mathsf{B}$. But previous typing steps have established that the precondition will be $\mu_1 \mapsto \mathsf{B} * \mu_2 \mapsto \mathsf{B}_{\mathsf{odd}}$, and even by subtyping, these two resource sets could be made to match only if $\ldots \vdash \mathsf{B} \preceq \mathsf{B}_{\mathsf{odd}}$. But since this is not the case, the buggy program will again not type check.

3 The Type System

We now describe the type system formally. The ALS type system has two notions of types: *value types* and *full types* (see Figure 3). Value types, τ, express the (effect-free) types that values have, whereas full types, φ, are used to type expressions: they describe the type of the computed value and also the heap and task state at the end of the computation.

In order to describe (the local view of) the mutable state of a task, we use *resource sets*, denoted by η, which describe the set of resource names owned by the task. A resource name associates an identifier with physical resources (e.g., heap cells or task ids) that uniquely identifies it in the context of a typing judgment. In the type system, ξ, μ, and π stand for resource names. We use μ for resource names having to do with heap cells, π for resource names having to do with tasks, and ξ where no distinction is made. Resource names are distinct from "physical names" like pointers to heap cells and task handles. This is needed

to support situations in which a name can refer to more than one object, for example, when typing weak references that permit aliasing.

There are five cases for value types τ:

1. Base types $\{\nu : \beta \mid \rho\}$ are type refinements over primitive types β with refinement ρ. Their interpretation is as in liquid types [29].
2. Reference types $\mathsf{ref}_\mu\,\tau$ stand for references to a heap cell that contains a value whose type is a subtype of τ. The type is indexed by a parameter μ, which is a resource name identifying the heap cell.
3. Promise types $\mathsf{promise}_\pi\,\tau$ stand for promises [15] of a value τ. A promise type can be forced, using **wait**, to yield a value of type τ.
4. Arrow types of the form $x{:}\,\tau\langle\eta\rangle \to \varphi$ stand for types of function that may have side effects, and summarize both the interface and the possible side effects of the function. In particular, a function of the above form takes one argument x of (value) type τ. If executed in a global state that matches resource set η, it will, if it terminates, yield a result of full type φ.
5. Resource quantifications of the form $\forall\xi.\tau$ provide polymorphism of names. The type $\forall\xi.\tau$ can be instantiated to any type $\tau[\xi'/\xi]$, as long as this introduces no resource duplications.

Next, consider full types. A full type $\varphi = И\Xi.\,\tau\langle\eta\rangle$ consists of three parts that describe the result of a computation: a list of resource name bindings Ξ, a value type τ, and a resource set η (introduced below). If an expression e is typed with φ, this means that if it reduces to a value, that value has type τ, and the global state matches η. The list of names Ξ describes names that are allocated during the reduction of e and occur in τ or η. The operator $И$ acts as a binder; each element of Ξ is to be instantiated by a fresh resource name.

Finally, consider resource sets η. Resource sets describe the heap cells and wait permissions owned by a task. They are given in a separation logic notation and consists of a separating conjunction of points-to facts and wait permissions. Points-to facts are written as $\mu \mapsto \tau$ and mean that for the memory location(s) associated with the resource name μ, the values residing in those memory locations can be typed with value type τ, similar to Alias Types [32]. The resource emp describes that no heap cells or wait permissions are owned. Conjunction $\eta_1 * \eta_2$ means that the resources owned by a task can be split into two disjoint parts, one described by η_1 and the other by η_2. The notion of disjointness is given in terms of the *name sets* of η: The name set of η is defined as $\mathsf{Names}(\mathsf{emp}) = \varnothing$, $\mathsf{Names}(\mu \mapsto \tau) = \{\mu\}$ and $\mathsf{Names}(\eta_1 * \eta_2) = \mathsf{Names}(\eta_1) \cup \mathsf{Names}(\eta_2)$. The resources owned by η are then given by $\mathsf{Names}(\eta)$, and the resources of η_1 and η_2 are disjoint iff $\mathsf{Names}(\eta_1) \cap \mathsf{Names}(\eta_2) = \varnothing$.

A resource of the form $\mathsf{Wait}(\pi, \eta)$ is called a *wait permission*. A wait permission describes the fact that the process indicated by π will hold the resources described by η upon termination, and the owner of the wait permissions may acquire theses resources by waiting for the task. Wait permissions are used to ensure that no resource is lost or duplicated in creating and waiting for a task, and to carry out resource transfers. For wellformedness, we demand that $\pi \notin \mathsf{Names}(\eta)$, and define $\mathsf{Names}(\mathsf{Wait}(\pi, \eta)) = \mathsf{Names}(\eta) \cup \{\pi\}$.

Lastly, $*$ is treated as an associative and commutative operator with unit emp, and resources that are the same up to associativity and commutativity are identified. For example, $\mu_1 \mapsto \tau_1 * \mathsf{Wait}(\pi, \mu_2 \mapsto \tau_2 * \mu_3 \mapsto \tau_3)$ and $\mu_1 \mapsto \tau_1 * \mathsf{Wait}(\pi, \mu_3 \mapsto \tau_3 * (\mathsf{emp} * \mu_2 \mapsto \tau_2))$ are considered the same resource.

$$\frac{[\![\Gamma]\!] \models \forall \nu. [\![\rho_1]\!] \implies [\![\rho_2]\!] \quad \Gamma \vdash \{\nu : \beta \mid \rho_1\}\ \mathrm{wf} \quad \Gamma \vdash \{\nu : \beta \mid \rho_2\}\ \mathrm{wf}}{\Gamma \vdash \{\nu : \beta \mid \rho_1\} \preceq \{\nu : \beta \mid \rho_2\}} \qquad \frac{\Gamma \vdash \tau_2 \preceq \tau_1 \quad \Gamma, x : \tau_2 \vdash \eta_2 \preceq \eta_1 \quad \Gamma, x : \tau_2 \vdash \varphi_1 \preceq \varphi_2}{\Gamma \vdash x{:}\,\tau_1\langle\eta_1\rangle \to \varphi_1 \preceq x{:}\,\tau_2\langle\eta_2\rangle \to \varphi_2}$$

$$\frac{\Gamma \vdash \tau\ \mathrm{wf}}{\Gamma \vdash \tau \preceq \tau} \qquad \frac{\Gamma, \xi \vdash \tau_1 \preceq \tau_2}{\Gamma \vdash \forall \xi. \tau_1 \preceq \forall \xi. \tau_2} \qquad \frac{\Gamma \vdash \tau_1 \preceq \tau_2 \quad \Gamma \vdash \mu}{\Gamma \vdash \mu \mapsto \tau_1 \preceq \mu \mapsto \tau_2} \qquad \frac{\vdash \Gamma\ \mathrm{wf}}{\Gamma \vdash \eta \preceq \eta}$$

$$\frac{\Gamma \vdash \eta_1 \preceq \eta_1' \quad \Gamma \vdash \eta_2 \preceq \eta_2' \quad \Gamma \vdash \eta_1 * \eta_2\ \mathrm{wf} \quad \Gamma \vdash \eta_1' * \eta_2'\ \mathrm{wf}}{\Gamma \vdash \eta_1 * \eta_2 \preceq \eta_1' * \eta_2'} \qquad \frac{\Xi \subseteq \Xi' \quad \Gamma, \Xi \vdash \tau_1 \preceq \tau_2 \quad \Gamma, \Xi \vdash \eta_1 \preceq \eta_2}{\Gamma \vdash И\Xi.\, \tau_1\langle\eta_1\rangle \preceq И\Xi'.\, \tau_2\langle\eta_2\rangle}$$

Figure 4 Subtyping rules. The notations $[\![\cdot]\!]$ and $\models$ are defined in [29].

3.1 Typing rules

The connection between expressions and types is made using the typing rules of the core calculus. The typing rules use auxiliary judgments to describe wellformedness and subtyping. There are four types of judgments used in the type system: wellformedness, subtyping, value typing and expression typing. Wellformedness provides three judgments, one for each kind of type: wellformedness of value types $\Gamma \vdash \tau$ wf, of resources $\Gamma \vdash \eta$ wf and of full types $\Gamma \vdash \varphi$ wf. Subtyping judgments are of the form $\Gamma \vdash \tau_1 \preceq \tau_2$, $\Gamma \vdash \eta_1 \preceq \eta_2$ and $\Gamma \vdash \varphi_1 \preceq \varphi_2$. Finally, value typing statements are of the form $\Gamma \vdash v : \tau$, while expression typing statements are of the form $\Gamma; \eta \vdash e : \varphi$.

The typing environment Γ is a list of variable bindings of the form $x : \tau$ and resource name bindings ξ. We assume that all environments are wellformed, i.e., no name or variable is bound twice and in all bindings of the form $x : \tau$, the type τ is wellformed.

The wellformedness rules are straightforward; details can be found in the full version [14]. They state that all free variables in a value type, resource or full type are bound in the environment, and that no name occurs twice in any resource, i.e., for each subexpression $\eta_1 * \eta_2$, the names in η_1 and η_2 are disjoint, and for each subexpression $\mathsf{Wait}(\pi, \eta)$, we have $\pi \notin \mathsf{Names}(\eta)$.

The subtyping judgments are defined in Figure 4. Subtyping judgments describe that a value, resource, or full type is a subtype of another object of the same kind. Subtyping of base types is performed by semantic subtyping of refinements (i.e., by logical implication), as in liquid types [29]. References are invariant under subtyping to ensure type safety.

Arrow type subtyping follows the basic pattern of function type subtyping: arguments —including the resources— are subtyped contravariantly, while results are subtyped covariantly.

Resource subtyping is performed pointwise: $\Gamma \vdash \eta_1 \preceq \eta_2$ holds if the wait permissions in η_1 are the same as in η_2, if μ points to τ_1 in η_1, then it points to τ_2 in η_2 where $\Gamma \vdash \tau_1 \preceq \tau_2$, and if μ points to τ_2 in η_2, it points to some τ_1 in η_1 with $\Gamma \vdash \tau_1 \preceq \tau_2$.

3.2 Value and expression typing

Figure 5 shows some of the value and expression typing rules. Value typing, $\Gamma \vdash v : \tau$, assigns a value type τ to a value v in the environment Γ, whereas expression typing, $\Gamma; \eta \vdash e : \varphi$ assigns, given an initial resource η, called the *precondition*, and an environment Γ, a full type φ to an expression e. The value typing rules and the subtyping rules are standard,

$$\textsc{TV-Const}\ \frac{\vdash \Gamma\ \text{wf}}{\Gamma \vdash c : \text{typeof}(c)} \qquad \textsc{TV-Var}\ \frac{\vdash \Gamma\ \text{wf} \qquad \Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \textsc{TV-Lambda}\ \frac{\Gamma, x : \tau; \eta \vdash e : \varphi}{\Gamma \vdash \lambda x.e : x{:}\,\tau\langle\eta\rangle \to \varphi}$$

$$\textsc{T-Value}\ \frac{\Gamma \vdash v : \tau}{\Gamma; \mathsf{emp} \vdash v : \tau\langle\mathsf{emp}\rangle} \qquad \textsc{TV-Subtype}\ \frac{\Gamma \vdash \tau \preceq \tau' \qquad \Gamma \vdash v : \tau}{\Gamma \vdash v : \tau'} \qquad \textsc{T-Subtype}\ \frac{\Gamma \vdash \varphi \preceq \varphi' \qquad \Gamma; \eta \vdash e : \varphi}{\Gamma; \eta \vdash e : \varphi'}$$

$$\textsc{TV-ForallIntro}\ \frac{\Gamma, \xi \vdash v : \tau}{\Gamma \vdash v : \forall\xi.\tau} \qquad \textsc{T-ForallElim}\ \frac{\Gamma; \eta \vdash e : И\Xi.\, \forall\xi.\tau\langle\eta'\rangle \qquad \Gamma \vdash И\Xi.\, \tau[\xi'/\xi]\langle\eta'\rangle\ \text{wf}}{\Gamma; \eta \vdash e : И\Xi.\, \tau[\xi'/\xi]\langle\eta'\rangle} \qquad \textsc{T-Frame}\ \frac{\Gamma; \eta_1 \vdash e : И\Xi.\, \tau\langle\eta_2\rangle \qquad \Gamma \vdash \eta_1 * \eta\ \text{wf} \qquad \Gamma, \Xi \vdash \eta_2 * \eta\ \text{wf}}{\Gamma; \eta_1 * \eta \vdash e : И\Xi.\, \tau\langle\eta_2 * \eta\rangle}$$

$$\textsc{T-Ref}\ \frac{\Gamma; \eta_1 \vdash e : И\Xi.\, \tau\langle\eta_2\rangle \qquad \Gamma, \Xi \vdash \tau' \preceq \tau \qquad \mu\ \text{fresh resource name variable}}{\Gamma; \eta_1 \vdash \mathbf{ref} e : И\Xi, \mu.\, \mathsf{ref}_\mu\, \tau'\langle\eta_2 * \mu \mapsto \tau\rangle} \qquad \textsc{T-WRef}\ \frac{\Gamma; \eta_1 \vdash e : И\Xi.\, \tau\langle\eta_2\rangle \qquad \mu\ \text{weak}}{\Gamma; \eta_1 \vdash \mathbf{ref} e : И\Xi, \mu.\, \mathsf{ref}_\mu\, \tau'\langle\eta_2\rangle}$$

$$\textsc{T-Read}\ \frac{\Gamma; \eta_1 \vdash e : И\Xi.\, \mathsf{ref}_\mu\, \tau'\langle\eta_2 * \mu \mapsto \tau\rangle}{\Gamma; \eta_1 \vdash\, !e : И\Xi.\, \tau\langle\eta_2 * \mu \mapsto \tau\rangle} \qquad \textsc{T-WRead}\ \frac{\Gamma; \eta_1 \vdash e : И\Xi.\, \mathsf{ref}_\mu\, \tau\langle\eta_2\rangle \qquad \mu\ \text{weak}}{\Gamma; \eta_1 \vdash\, !e : И\Xi.\, \tau\langle\eta_2\rangle}$$

$$\textsc{T-Write}\ \frac{\Gamma; \eta_1 \vdash e_2 : И\Xi_1.\, \tau_2\langle\eta_2\rangle \qquad \Gamma, \Xi_1, \Xi_2 \vdash \tau_2 \preceq \tau \qquad \Gamma, \Xi_1; \eta_2 \vdash e_1 : И\Xi_2.\, \mathsf{ref}_\mu\, \tau\langle\eta_3 * \mu \mapsto \tau_1\rangle}{\Gamma; \eta_1 \vdash e_1 := e_2 : И\Xi_1, \Xi_2.\, \mathsf{unit}\langle\eta_3 * \mu \mapsto \tau_2\rangle} \qquad \textsc{T-WWrite}\ \frac{\Gamma; \eta_1 \vdash e_2 : И\Xi_1.\, \tau\langle\eta_2\rangle \qquad \Gamma, \Xi_1; \eta_2 \vdash e_1 : И\Xi.\, \mathsf{ref}_\mu\, \tau\langle\eta_3\rangle \qquad \mu\ \text{weak}}{\Gamma; \eta_1 \vdash e_1 := e_2 : И\Xi.\, \mathsf{unit}\langle\eta_3\rangle}$$

$$\textsc{T-Post}\ \frac{\Gamma; \eta \vdash e : И\Xi.\, \tau\langle\eta'\rangle \qquad \pi\ \text{fresh resource name variable}}{\Gamma; \eta \vdash \mathbf{post}\, e : И\Xi, \pi.\, \mathsf{promise}_\pi\, \tau\langle\mathsf{Wait}(\pi, \eta')\rangle} \qquad \textsc{T-WaitTransfer}\ \frac{\Gamma; \eta \vdash e : И\Xi.\, \mathsf{promise}_\pi\, \tau\langle\eta_1 * \mathsf{Wait}(\pi, \eta_2)\rangle}{\Gamma; \eta \vdash \mathbf{wait}\, e : И\Xi.\, \tau\langle\eta_1 * \eta_2\rangle}$$

$$\textsc{T-App}\ \frac{\Gamma; \eta_1 \vdash e_2 : И\Xi_1.\, \tau_x\langle\eta_2\rangle \qquad \Gamma, \Xi_1; \eta_2 \vdash e_1 : И\Xi_2.\, (x{:}\,\tau_x\langle\eta_4\rangle \to И\Xi_3.\, \tau\langle\eta_5\rangle)\langle\eta_3\rangle \qquad \Gamma, \Xi_1 \vdash \eta_3 \preceq \eta_4 * \eta_i \qquad \Gamma \vdash И\Xi_1, \Xi_2, \Xi_3.\, \tau\langle\eta_5 * \eta_i\rangle\ \text{wf}}{\Gamma; \eta_1 \vdash e_1\, e_2 : И\Xi_1, \Xi_2, \Xi_3.\, \tau\langle\eta_5 * \eta_i\rangle}$$

Figure 5 Value and expression typing.

and typing a value as an expression gives them types as an effect-free expression: From an empty precondition η, they yield a result of type τ with empty postcondition and no name allocation.

The rules TV-ForallIntro and T-ForallElim allow for the quantification of resource names for function calls. This is used to permit function signatures that are parametric in the resource names, and can therefore be used with arbitrary heap and task handles as arguments. The typing rules are based on the universal quantification rules for Indexed Types [37], and are similar to the quantification employed in alias types.

The rule T-Frame implements the frame rule from separation logic [27] in the context

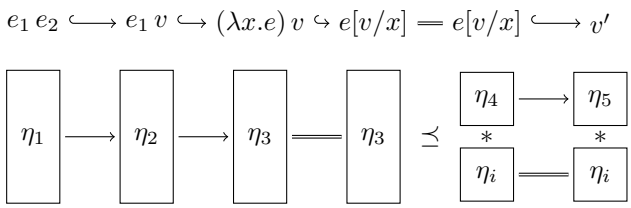


Figure 6 Transformation of the global state as modeled by the T-App rule. Upper row: expression reduction steps, lower row: The corresponding resources of the global state.

of ALS types. It allows adjoining a resource that is left invariant by the execution of an expression e to the pre-condition and the post-condition.

The typing rules T-Ref, T-Read and T-Write type the memory access operations. The typing rules implement strong heap updates using the pre- and post-conditions. This is possible because separate resources for pre- and post-conditions that are tied to specific global states are used, whereas the type of the reference only describes an upper bound for the type of the actual cell contents. A similar approach is used in low-level liquid types [28]. Additionally, the rules T-WRef, T-WRead and T-WWrite allow for weak heap updates, using a subset of locations that is marked as weak and never occur in points-to facts.

It is important to note how the evaluation order affects these typing rules. For example, when evaluating $e_1 := e_2$, we first reduce to $e_1 := v$, and then to $\ell := v$. Therefore T-Write types e_2 with the initial η_1 precondition and uses the derived postcondition, η_2, as a precondition for typing e_1.

The typing rules T-Post and T-WaitTransfer serve the dual purpose of providing the proper return type to the concurrency primitives (`post` and `wait`) and to control the transfer of resource ownership between tasks.

T-Post types task creation using an expression of the form **post** e. For an expression e that yields a value of type τ and a resource η, it gives a promise that if evaluating the expression e terminates, waiting for the task will yield a value of type τ, and additionally, if some task acquires the resources of the task executing e, it will receive exactly the resources described by η.

T-WaitTransfer types expressions that wait for the termination of a task with resource transfer. It states that if e returns a promise for a value τ and a corresponding wait permission $\mathsf{Wait}(\pi, \eta_2)$ yielding a resource η_2, as well as some additional resource η_1, then **wait** e yields a value of type τ, and the resulting global state has a resource $\eta_1 * \eta_2$. In particular, the postcondition describes the union of the postcondition of e, without the wait permission $\mathsf{Wait}(\pi, \eta_2)$, and the postcondition of the task that e refers to, as given by the wait permission.

Finally, T-App types function applications under the assumption that the expression is evaluated from right to left, as in OCaml. The first two preconditions on the typing of e_1 and e_2 are standard up to the handling of resources, while the wellformedness condition ensures that the variable x does not escape its scope. The resource manipulation of T-App is illustrated in Fig. 6. Resources are chosen in such a way that they describe the state transformation of first reducing e_2 to a value, then e_1 and finally the β-redex of $e_1 e_2$.

The type system contains several additional rules for handling if–then–else expressions and for dealing with weak references. These rules are completely standard and can be found in the full version [14].

3.3 Type safety

The type system presented above enjoys type safety in terms of a global typing relation. The details can be found in the the full version [14]; here, only the notion of global typing and the type safety statement are sketched.

We need the following three functions. The *global type* γ is a function that maps heap locations to value types and task identifiers to full types. For heap cells, it describes the type of the reference to that heap cell, and for a task, the postcondition type of the task. The *global environment* ψ is a function that maps heap locations to value types and task identifiers to resources. For heap cells, it describes the precise type of the cell content, and for a task, the precondition of the task. The *name mapping* χ is a function that maps heap locations and task identifiers to names. It is used to connect the heap cells and tasks to their names used in the type system.

For the statement of type safety, we need three definitions:

1. Given γ, ψ and χ, we say that γ, ψ and χ type a global configuration, written $\psi, \chi \vdash (H, P, p) : \gamma$, when:
   - For all $\ell \in \operatorname{dom} H$, $\Gamma_\ell \vdash H(\ell) : \psi(\ell)$,
   - For all $p \in \operatorname{dom} P$, $\Gamma_p; \psi(p) \vdash P(p) : \gamma(p)$

   where the Γ_ℓ and Γ_p environments are defined in the full version [14]. In other words, the heap cells can be typed with their current, precise type, as described by ψ, while the tasks can be typed with the type give by γ, using the precondition from ψ.
2. γ, ψ and χ are *wellformed*, written (γ, ψ, χ) wf, if a number of conditions are fulfilled. The intuition is that on one hand, a unique view of resource ownership can be constructed from the three functions, and on the other hand, different views of resources (e.g., the type of a heap cell as given by a precondition compared with the actual type of the heap cell) are compatible.
3. For two partial functions f and g, f extends g, written $g \sqsubseteq f$, if $\operatorname{dom} g \subseteq \operatorname{dom} f$ and $f(x) = g(f)$ for all $x \in \operatorname{dom} g$.
   Given two global type γ and γ', and two name maps χ and χ', we say that (γ, χ) *specializes to* (γ', χ'), written $(\gamma, \chi) \rhd (\gamma', \chi')$, when the following holds: $\chi \sqsubseteq \chi'$, $\gamma \mid_{\mathsf{Locs}} \sqsubseteq \gamma' \mid_{\mathsf{Locs}}$, $\operatorname{dom} \gamma \subseteq \operatorname{dom} \gamma'$ and for all task identifiers $p \in \operatorname{dom} \gamma$, $\gamma'(p)$ specializes γ in the following sense: Let $\varphi = \text{И}\Xi.\, \tau\langle\eta\rangle$ and $\varphi' = \text{И}\Xi'.\, \tau'\langle\eta'\rangle$ be two full types. Then φ' specializes φ if there is a substitution σ such that $\text{И}\Xi'.\, \tau\sigma\langle\eta\sigma\rangle = \varphi'$, i.e., φ' can be gotten from φ by instantiating some names.

The following theorem follows using a standard preservation/progress argument.

▶ **Theorem 1** (Type safety). *Consider a global configuration (H, P, p) that is typed as $\psi, \chi \vdash (H, P, p) : \gamma$. Suppose that (γ, ψ, χ) wf.*

Then for all (H', P', p') such that $(H, P, p) \hookrightarrow^* (H', P', p)$, there are γ', ψ', χ' such that $\psi', \chi' \vdash (H', P', p') : \gamma'$, (γ', ψ', χ') wf and $(\gamma, \psi) \rhd (\gamma', \psi')$.

Furthermore, if (H', P', p') cannot take a step, then all processes in P' have terminated, in the sense that the expressions of all tasks have reduced to values.

4 Type Inference

To infer ALS types, we extend the liquid type inference algorithm. The intention was to stay as close to the original algorithm as possible. The liquid type inference consists of the four steps depicted on the left of Figure 7:

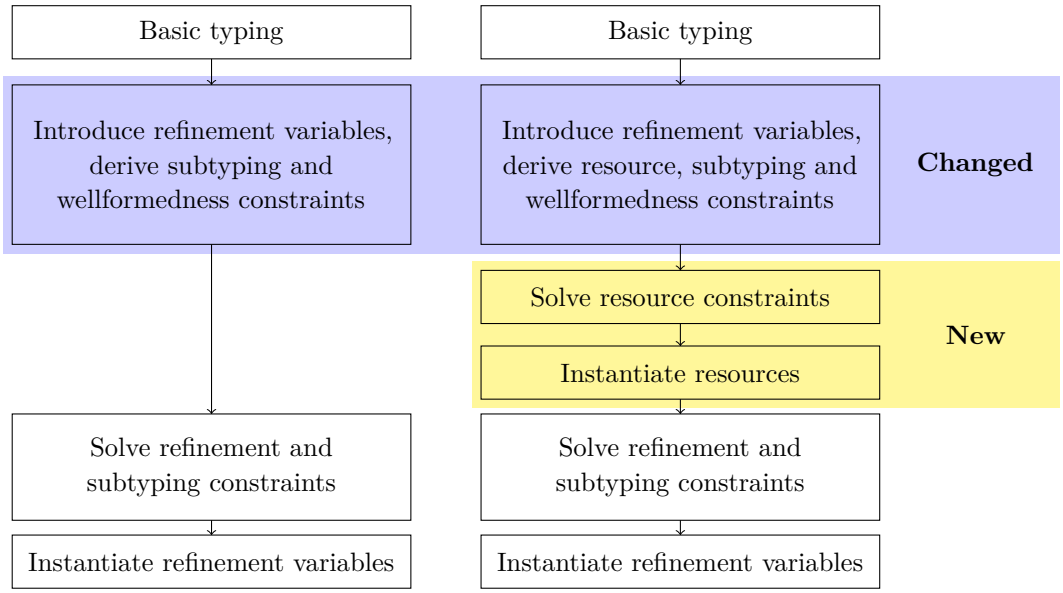


Figure 7 High-level overview of the liquid type inference procedure (left), and the modified procedure presented in this section (right). The changes are highlighted.

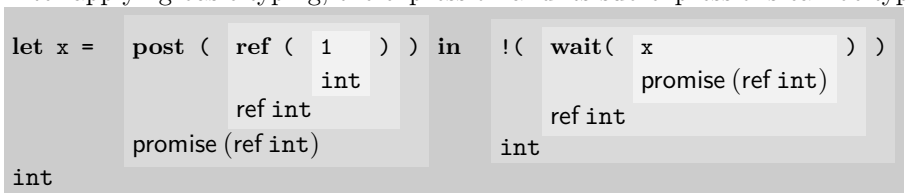
1. Basic typing assigns plain OCaml types to the expression that is being typed using the Hindley-Milner algorithm.
2. The typing derivation is processed to add refinements to the types. In those cases where a clear refinement is known (e.g., for constants), that refinement is added to the type. In all other cases, a refinement variable is added to the type. In the latter case, additional constraints are derived that limit the possible instantiations of the refinement variables. For example, consider the typing of an application $e_1 e_2$. Suppose e_1 has the refined type $x : \{\nu : \texttt{int} \mid \nu \geq 0\} \to \{\nu : \texttt{int} \mid \nu = x + 1\}$, and e_2 has refined type $\{\nu : \texttt{int} \mid \nu \geq 5\}$. From step 1, $e_1 e_2$ has type `int`. In this step, this type is augmented to $\{\nu : \texttt{int} \mid \rho\}$, where ρ is a refinement variable, and two constraints are produced:
   - $\vdash x : \{\nu : \texttt{int} \mid \nu \geq 0\} \to \{\nu : \texttt{int} \mid \nu = x + 1\} \preceq x : \{\nu : \texttt{int} \mid \nu \geq 5\} \to \{\nu : \texttt{int} \mid \rho\}$, describing that the function type should be specialized taking the more precise type of the argument into account,
   - $\vdash \{\nu : \texttt{int} \mid \rho\}$ wf, describing that $\{\nu : \texttt{int} \mid \rho\}$ should be wellformed. In particular, the instantiation of ρ may not mention the variable x.
3. The constraints from the second step are solved relative to a set of user-provided predicates. In the example, one possible solution for ρ would be $\nu \geq 6$.
4. The solutions from the third step are substituted for the refinement variables. In the example, $e_1 e_2$ would therefore get the type $\{\nu : \texttt{int} \mid \nu \geq 6\}$.

The details of this procedure are described in [29]. For ALS types, the procedure is extended to additionally derive the resources that give preconditions and postconditions for the expressions. This involves a new type of variables, *resource variables*, which are placeholders for pre- and post-conditions. This is depicted on the right-hand side of Figure 7.

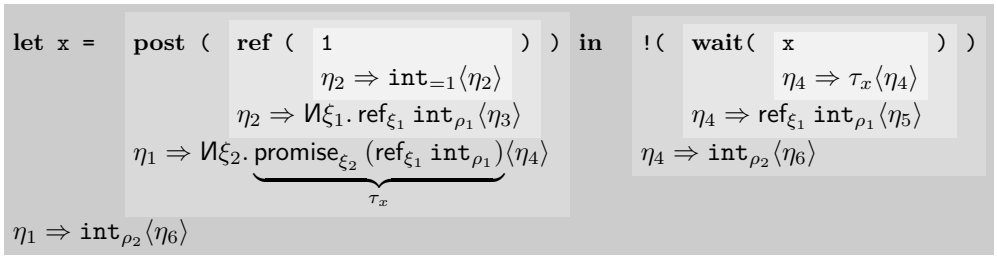
Several steps are identical to the algorithm above; the constraint derivation step has been modified, whereas the steps dealing with resource variables are new. We sketch the working of the algorithm by way of a small example. Consider the following expression:

```
let x = post (ref 1) in !(wait x)
```

After applying basic typing, the expression and its sub-expressions can be typed as follows:



The second step then derives the following ALS typing derivation. In this step, each precondition and postcondition gets a new resource variable:

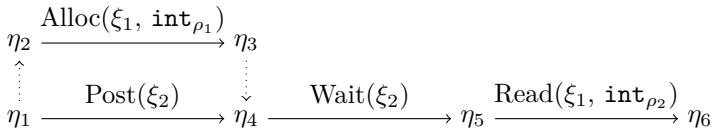


Here, an expression e types as $\eta \Rightarrow И\Xi.\, \tau\langle\eta'\rangle$ iff, for some environment Γ and some Ξ', $\Gamma; \eta \vdash e : И\Xi, \Xi'.\, \tau\langle\eta'\rangle$. The η_i occurring in the derivation are all variables.

Three types of constraints are derived: subtyping and wellformedness constraints (for the refinement variables), and resource constraints (for the resource variables). For the first two types of constraints, the following constraints are derived:

- $\vdash \texttt{int}_{=1} \preceq \texttt{int}_{\rho_1}$, derived from the typing of **ref** `1`: The reference type $\texttt{int}_{\rho_1}$ must allow a cell content of type $\texttt{int}_{=1}$.
- $x : \tau_x \vdash \texttt{int}_{\rho_2} \preceq \texttt{int}_{\rho_1}$, derived from the typing of `(wait x)`: The cell content type of the cell ξ_1 must be a subtype of the type of the reference.
- $\vdash \texttt{int}_{\rho_2}$ wf, which derives from the **let** expression: The type $\texttt{int}_{\rho_2}$ must be wellformed outside the **let** expression, and therefore, must not contain the variable x.

The refinement constraints can be represented by a constraint graph representing heap accesses, task creation and finalization, and function calls. For the example, we get the following constraint graph:



Here, $\text{Alloc}(\xi_1, \texttt{int}_{\rho_1})$ stands for "Allocate a cell with name ξ_1 containing data of type $\texttt{int}_{\rho_1}$" and so on.

To derive the correct resources for the resource variables, we make use of the following observation. Given an η, say, $\eta = \mathbf{wait}(\pi_1, \mathbf{wait}(\pi_2, \mu \mapsto \tau))$, each name occurring in this resource has a unique sequence of task names π associated with it that describe in which way it is enclosed by wait permissions. This sequence is called its *wait prefix*. In the example, μ is enclosed in a wait permissions for π_2, which is in turn enclosed by one for π_1, so the wait prefix for μ is $\pi_1\pi_2$. For π_2, it is π_1, while for π_1, it is the empty sequence ϵ.

It is easy to show that a resource η can be uniquely reconstructed from the wait prefixes for all the names occurring in η, and the types of the cells occurring in η. In the inference algorithm, the wait prefixes and the cell types for each resource variable are derived independently.

First, the algorithm derives wait prefixes for each refinement variable by applying abstract interpretation to the constraint graph. For this, the wait prefixes are embedded in a lattice

Names $\to \{U, W\} \cup \{p \mid p \text{ prefix}\}$, where $U \sqsubset p \sqsubset W$ for all prefixes p. Here, U describes that a name is unallocated, whereas W describes that a name belongs to a weak reference.

In the example, the following mapping is calculated:

$$\begin{array}{ccccccc} \eta_2 : \bot, \bot & \xrightarrow{\text{Alloc}(\xi_1,\, \texttt{int}_{\rho_1})} & \eta_3 : \epsilon, \bot & & & & \\ \vdots\uparrow & & \vdots\downarrow & & & & \\ \eta_1 : \bot, \bot & \xrightarrow{\text{Post}(\xi_2)} & \eta_4 : \xi_2, \epsilon & \xrightarrow{\text{Wait}(\xi_2)} & \eta_5 : \epsilon, \bot & \xrightarrow{\text{Read}(\xi_1,\, \texttt{int}_{\rho_2})} & \eta_6 : \epsilon, \bot \end{array}$$

The mapping is read as follows: If $\eta : w_1, w_2$, then ξ_1 has wait prefix w_1 and ξ_2 has wait prefix w_2.

In this step, several resource usage problems can be detected:

- A name corresponding to a wait permission is not allowed to be weak, because that would mean that there are two tasks sharing a name, which would break the resource transfer semantics.
- When waiting for a task with name π, π must have prefix ϵ: The waiting task must possess the wait permission.
- When reading or writing a heap cell with name μ, μ must have prefix ϵ or be weak, by a similar argument.

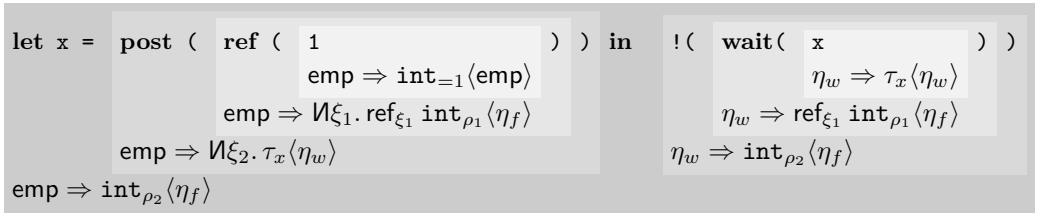
Second, the algorithm derives cell types for each refinement variable. This is done by propagating cell types along the constraint graph; if a cell can be seen to have multiple refinements $\{\nu : \tau \mid \rho_1\}, \ldots, \{\nu : \tau \mid \rho_n\}$, a new refinement variable ρ is generated and subtyping constraints $\Gamma \vdash \{\nu : \tau \mid \rho_1\} \preceq \{\nu : \tau \mid \rho\}, \ldots, \Gamma \vdash \{\nu : \tau \mid \rho_n\} \preceq \{\nu : \tau \mid \rho\}$ are added to the constraint set. In the example, the following mapping is calculated for cell ξ_1 (where $\bot$ stands for "cell does not exist"):

$$\begin{array}{ccccccc} \eta_2 : \bot & \xrightarrow{\text{Alloc}(\xi_1,\, \texttt{int}_{\rho_1})} & \eta_3 : \texttt{int}_{=1} & & & & \\ \vdots\uparrow & & \vdots\downarrow & & & & \\ \eta_1 : \bot & \xrightarrow{\text{Post}(\xi_2)} & \eta_4 : \texttt{int}_{=1} & \xrightarrow{\text{Wait}(\xi_2)} & \eta_5 : \texttt{int}_{=1} & \xrightarrow{\text{Read}(\xi_1,\, \texttt{int}_{\rho_2})} & \eta_6 : \texttt{int}_{=1} \end{array}$$

Additionally, a new subtyping constraint is derived: $x : \tau_x \vdash \texttt{int}_{=1} \preceq \texttt{int}_{\rho_2}$. Using this information, instantiations for the resource variables can be computed:

$$\eta_1, \eta_2 : \mathsf{emp} \qquad \eta_3, \eta_5, \eta_6 : \xi_1 \mapsto \texttt{int}_{=1} \qquad \eta_4 : \mathsf{Wait}(\xi_2, \xi_1 \mapsto \texttt{int}_{=1})$$

These instantiations are then substituted wherever resource variables occur, both in constraints and in the type of the expression. We get the following type for the expression (using $\eta_f = \xi_1 \mapsto \texttt{int}_{=1}$, $\eta_w := \mathsf{Wait}(\xi_2, \xi_1 \mapsto \texttt{int}_{=1})$ and τ_x from above):



Additionally, some further subtyping and wellformedness constraints are introduced to reflect the relationship between cell types, and to give lower bounds on the types of reads. In the example, one new subtyping constraint is introduced: $x : \tau_x \vdash \texttt{int}_{=1} \preceq \texttt{int}_{\rho_2}$, stemming from the read operation $\text{Read}(\xi_1, \texttt{int}_{\rho_2})$ that was introduced for the reference access $\texttt{!}(\textbf{wait}\ x)$. It indicates that the result of the read has a typing $\texttt{int}_{\rho_2}$ that subsumes that cell content type, $\texttt{int}_{=1}$.

At this point, it turns out that, when using this instantiation of resource variables, the resource constraints are fulfilled as soon as the subtyping and wellformedness constraints are fulfilled. The constraints handed to the liquid type constraint solver are:

$$\vdash \texttt{int}_{=1} \preceq \texttt{int}_{\rho_1} \qquad x : \tau_x \vdash \texttt{int}_{\rho_2} \preceq \texttt{int}_{\rho_1} \qquad \vdash \texttt{int}_{\rho_2} \text{ wf} \qquad x : \tau_x \vdash \texttt{int}_{=1} \preceq \texttt{int}_{\rho_2}$$

This leads to the instantiation of ρ_1 and ρ_2 with the predicate $\nu = 1$.

5 Case Studies

We have extended the existing liquid type inference tool, `dsolve`, to handle ALS types. Below, we describe our experiences on several examples taken from the literature and real-world code.

In general, the examples make heavy use of external functions. For this reason, some annotation work will always be required. In many cases, it turns out that only few functions will have to be explicitly annotated with ALS types. In the examples, we state how many annotations were used in each case.

Our implementation only supports liquid type annotations on external functions but not ALS types. We work around this by giving specifications of abstract purely functional versions of functions, and providing an explicit wrapper implementation that implement the correct interface. For example, suppose we want to provide the following external function:

$$\texttt{write}\colon \texttt{stream} \to \mathsf{ref}_\xi\, \texttt{buffer}\langle \xi \mapsto \{\nu : \texttt{buffer} \mid \nu \text{ odd}\}\rangle \to (\mathsf{unit}\langle \xi \mapsto \texttt{buffer}\rangle)$$

We implement this by providing an external function

$$\texttt{write\_sync}\colon \texttt{stream} \to \{\nu : \texttt{buffer} \mid \nu \text{ odd}\} \to \texttt{buffer}$$

and a wrapper implementation

```
let write s b = b := write_sync s (!b)
```

The wrapper code is counted separately from annotation code.

5.1 The double-buffering example, revisited

Our first example is the double-buffering copy loop from Section 2. We consider three versions of the code:

1. The copying loop, exactly as given.
2. A version of the copying loop in which an error has been introduced. Instead of creating a task that writes a full buffer to the disk, i.e., **post** `(Writer.write outs buffer_full)`, we post a task that tries to write the read buffer: **post**`(Writer.write outs buffer_empty)`.
3. Another version of the copying loop. This time, the initial call to the main loop is incorrect: the buffers are switched, so that the loop would try to write the empty buffer while reading into the full buffer.

We expect the type check to accept the first version of the example and to detect the problems in the other two versions.

We use the following ALS type annotations:

$$\begin{aligned}
\texttt{write} &: s : \texttt{stream} \to \forall \mu.\mathsf{ref}_\mu\ \texttt{buffer}\langle \mu \mapsto \{\nu : \texttt{buffer} \mid \mathrm{odd}(\nu)\}\rangle \to \\
&\qquad \mathsf{unit}\langle \mu \mapsto \{\nu : \texttt{buffer} \mid \mathrm{odd}(\nu)\}\rangle \\
\texttt{read} &: s : \texttt{stream} \to \forall \mu.\mathsf{ref}_\mu\ \texttt{buffer}\langle \mu \mapsto \texttt{buffer}\rangle \to \\
&\qquad \mathsf{unit}\langle \mu \mapsto \{\nu : \texttt{buffer} \mid \mathrm{odd}(\nu)\}\rangle \\
\texttt{make_buffer} &: \mathsf{unit} \to \mathit{M}\mu.\ \mathsf{ref}_\mu\ \texttt{buffer}\langle \mu \mapsto \{\nu : \texttt{buffer} \mid \neg\mathrm{odd}(\nu)\}\rangle
\end{aligned}$$

The main use of annotations is to introduce the notion of a buffer with odd parity. Using a predicate `odd`, we can annotate the contents of a buffer cell to state whether it has odd parity or not. For example, the function `read` has type[3].

$$s : \texttt{stream} \to b : \mathsf{ref}_\xi\ \texttt{buffer}\langle \xi \mapsto \texttt{buffer}\rangle \to \mathsf{unit}\langle \xi \mapsto \{\nu : \texttt{buffer} \mid \texttt{odd}(\nu)\}\rangle.$$

We discuss the results in turn. For the first example, `dsolve` takes roughly 0.8s. As expected, dsolve derives types for this example. For instance, the type for the main copying loop, `copy2`, is exactly the one given in Section 2 up to α-renaming.

For the second example, the bug is detected in 0.3s. while calculating the resources. In particular, consider the following part of the code:

```
9   let rec copy buf_full buf_empty =
10    let drain = post (write outs buf_empty) in
11    if eof ins then
12      wait drain
13    else begin
14      let fill = post (read ins buf_empty) in
15        wait fill; wait drain; copy buf_empty buf_full
16    end
```

The tool detects an error at line 14: a resource which corresponds to the current instance of `buf_empty`, is accessed by two different tasks at the same time. This corresponds to a potential race condition, and it is, in fact, exactly the point where we introduced the bug.

For the third example, `dsolve` takes about 0.8s. Here, an error is detected in a more subtle way. The derived type of `copy` is:

$$\begin{aligned}
\forall \mu_1.\texttt{buf_full} : \mathsf{ref}_{\mu_1}\ \texttt{buffer} \to \forall \mu_2.\texttt{buf_empty} : \mathsf{ref}_{\mu_2}\ \texttt{buffer} \\
\langle \mu_2 \mapsto \texttt{buffer} * \mu_1 \mapsto \{\nu : \texttt{buffer} \mid \mathrm{odd}(\nu)\}\rangle \to \mathsf{unit}\langle \ldots \rangle
\end{aligned}$$

In particular, in the initial call `copy buf2 buf1`, it must hold that `buf2` corresponds to any buffer, and `buf1` corresponds to a buffer with odd parity. To enforce this, dsolve introduces a subtyping constraint $\vdash \mu_1 \mapsto \{\nu : \texttt{buffer} \mid \rho\} \preceq \mu_1 \mapsto \{\nu : \texttt{buffer} \mid \rho'\}$, where ρ is the predicate that is derived for the content of the cell μ_1 at the moment when `copy` is actually called, and ρ' is the predicate from the function precondition, i.e., $\rho' = \mathrm{odd}(\nu)$. For ρ, dsolve derives the instantiation $\rho = \neg\mathrm{odd}(\nu)$. Therefore, the following subtyping constraint is asserted:

$$\vdash \mu_1 \mapsto \{\nu : \texttt{buffer} \mid \neg\mathrm{odd}(\nu)\} \preceq \mu_1 \mapsto \{\nu : \texttt{buffer} \mid \mathrm{odd}(\nu)\}$$

This constraint entails that for every ν, $\neg\mathrm{odd}(\nu)$ implies $\mathrm{odd}(\nu)$, which leads to a contradiction. Thus, dsolve detects a subtyping error, which points to the bug in the code.

[3] Strictly speaking, `read` is a wrapper function, so it is not annotated with a type. Nevertheless, this is the type that it derives from its abstract implementation, `read_impl`

$\texttt{Reader.read}$: $\texttt{Reader.t} \to$
$\quad \forall \mu, b{:}\, \mathsf{ref}_\mu\, \texttt{string}\langle \mu \mapsto \texttt{string}\rangle \to \mathit{И}\pi.\, \mathsf{promise}_\pi\, \texttt{result}\langle \mu \mapsto \texttt{string}\rangle,$
$\texttt{Writer.write}$:
$\quad \texttt{int} \to \texttt{Writer.t} \to \forall \mu, b{:}\, \mathsf{ref}_\mu\, \texttt{string}\langle \mu \mapsto \texttt{string}\rangle \to \mathsf{unit}\langle \mu \mapsto \texttt{string}\rangle,$
$\texttt{Writer.flushed}$: $\texttt{Writer.t} \to \mathit{И}\pi.\, \mathsf{promise}_\pi\, \mathsf{unit}\langle \mathsf{emp}\rangle.$

Figure 8 Types for asynchronous I/O functions in the Async library

5.2 Another asynchronous copying loop

The "Real World OCaml" book [20, Chapter 18] contains an example of an asynchronous copying loop in monadic style:

```
let rec copy_block buffer r w =
  Reader.read r buffer >>= function
  | `Eof -> return ()
  | `Ok bytes_read ->
    Writer.write w buffer ~len:bytes_read;
    Writer.flushed w >>= fun () -> copy_blocks buffer r w
```

where the functions `Reader.read`, `Writer.write` and `Writer.flushed` have the types given in Figure 8. One possible implementation of `Reader.read` is the following:

```
let read stream buffer = post (sync_read stream buffer)
```

where `sync_read` is typed as `stream` → ref `buffer` → `int`, returning the number of bytes read. In practice, this function is implemented as an I/O primitive by the Async library, making use of operating system facilities for asynchronous I/O to ensure that this operation never blocks the execution of runnable tasks. The same holds for `Writer.write` and `Writer.flushed`.

By running `dsolve` on the example, we expect the following type for `copy_block`:

$$\forall \mu.b : \mathsf{ref}_\mu\, \texttt{string} \to r : \texttt{Reader.t} \to w : \texttt{Writer.t}\langle \mu \mapsto \texttt{string}\rangle \to$$
$$\mathit{И}\pi.\, \mathsf{unit}\langle \mathsf{Wait}(\pi, \mu \mapsto \texttt{string})\rangle$$

To be able to type this function, it needs to be rewritten in **post/wait** style. In this and all following examples, we use a specific transformation: In the Async and Lwt libraries, tasks are represented using an asynchronous monad with operators `return` and `bind`, the latter often written in infix form as `>>=`. A task is built by threading together the computations performed by the monad. For example, the following code reads some data from a Reader and, as soon as the reader is finished, transforms the data by applying the function `f`:

```
Reader.read stream >>= fun x -> return (f x)
```

This code can be translated to the post/wait style as follows:

```
post (let x = wait (Reader.read stream) in f x)
```

The idea is that the monadic value above corresponds to a single task to be posted, which evaluates each binding in turn. In general, a monad expression e_1 `>>=` e_2 `>>=` ... `>>=` e_n can be translated to:

```
post (let x_1 = wait e_1 in
      let x_2 = wait (e_2 x_1) in
      ...
      let x_n = wait (e_n x_{n-1}) in
      x_n)
```

The expression `return` e then translates to **post** e. Additionally, we use the "`return` rewriting law" `return` e_1 `>>=` $e_2 \equiv e_2\ e_1$ to simplify the expressions a bit further.

Running `dsolve` on the example takes about 0.1s, and derives the expected type for `copy_block`.

5.3 Coordination in a parallel SAT solver

The next example is a simplified version of an example from X10 [34]. It models the coordination between tasks in a parallel SAT solver. There are two worker tasks running in parallel and solving the same CNF instance. Each of the tasks works on its own state. A central coordinator keeps global state in the form of an updated CNF. The worker tasks can poll the coordinator for updates; this is implemented by the worker task returning POLL. The coordinator will then restart the worker with a newly-created task.

We use two predicates, `sat` and `equiv`. It holds that $\texttt{sat}(c)$ iff c is satisfiable. We introduce `res_ok cnf res` as an abbreviation for $(res = \texttt{SAT} \Rightarrow \texttt{sat}(cnf)) \wedge (res = \texttt{UNSAT} \Rightarrow \neg\texttt{sat}(cnf))$. The predicate `cnf_equiv cnf1 cnf2` holds if `cnf1` and `cnf2` are equivalent. Denote by $\texttt{cnf}_{\equiv c}$ the type $\{\nu : \texttt{cnf} \mid \texttt{cnf\_equiv c}\ \nu\}$.

```
1  (* Interface of helper functions. *)
2  type cnf
3  type worker_result = SAT | UNSAT | POLL
4  val worker: c:cnf → ∀μ.ref_μ cnf⟨μ ↦ cnf_≡c⟩ → {ν : worker_result | res_ok c ν}⟨μ ↦ cnf_≡c⟩
5  val update: c:cnf → ∀μ.ref_μ cnf⟨μ ↦ cnf_≡c⟩ → cnf_≡c⟨μ ↦ cnf_≡c⟩
6  (* The example code *)
7  let parallel_SAT c =
8    let buffer1 = ref c and buffer2 = ref c in
9    let rec main c1 worker1 worker 2 =
10     if * then (* non-deterministic choice; in practice, use a select *)
11       match wait worker1 with
12       | SAT -> discard worker2; true
13       | UNSAT -> discard worker2; false
14       | POLL ->
15         let c2 = update c1 buffer1 in
16         let w = post (worker c2 buffer1) in
17         main c2 w worker2
18     else
19       ... (* same code, with roles switched *)
20   in main (post (worker c buffer1)) (post (worker c buffer2))
```

Here, **discard** can be seen as a variant of **wait** that just cancels a task. The annotations used in the example are given in the first part of the code, "Interface of helper functions".

For this example, we expect a type for `parallel_SAT` along the lines of

$$c : \texttt{cnf}\langle\texttt{emp}\rangle \to \{\nu : \texttt{bool} \mid \mathrm{sat}(c) \Leftrightarrow \nu\}\langle\ldots\rangle.$$

Executing `dsolve` on this example takes roughly 9.8s, of which 8.7s are spent in solving subtyping constraints. The type derived for `parallel_SAT` is (after cleaning up some irrelevant refinements):

$$c : \texttt{cnf}\langle\texttt{emp}\rangle \to \mathsf{И}\mu_1, \mu_2.\, \{\nu : \texttt{bool} \mid \nu = \mathrm{sat}(c)\}\langle\mu_1 \mapsto \texttt{cnf}_{\equiv c} * \mu_2 \mapsto \texttt{cnf}_{\equiv c}\rangle$$

This type is clearly equivalent to the expected type.

5.4 The MirageOS FAT file system

Finally, we considered a version of the MirageOS [16] FAT file system code,[4] in which we wanted to check if our tool could detect any concurrency errors. Indeed, using ALS types, we discovered a concurrency bug with file writing commands: the implementation has a race condition with regard to the in-memory cached copy of the file allocation table.

For this, we split up the original file so that each module inside it resides in its own file. We consider the code that deals with the FAT and directory structure, which makes heavy use of concurrency, and treat all other modules as simple externals. Since the primary goal of the experiment was to check whether the code has concurrency errors, we do not provide any type annotations.

Running `dsolve` takes about 4.4s and detects a concurrency error: a resource is accessed even though it is still wrapped in a wait permission. Here is a simplified view of the relevant part of the code:

```
1
2   type state = { format: ...; mutable fat: fat_type }
3   ...
4   let update_directory_containing x path =
5     post (... let c = Fat_entry.follow_chain x.format x.fat ... in ...)
6   ...
7   let update x ... =
8     ...
9     update_directory_containing x path;
10    x.fat <- List.fold_left update_allocations x.fat fat_allocations
11    ...
```

In this example, `x.fat` has the reference type ref_μ `fat_type`. By inspecting the implementation of `update_directory_containing`, it is clear that this function needs to have (read) access to `x.fat`. Therefore, the type of e_1 :=`update_directory_containing x path` will be along the lines of $\Gamma; \mu \mapsto \texttt{fat\_type} * \eta \vdash e_1 : \mathit{И}\pi, \Xi.\, \tau\langle \mathsf{Wait}(\pi, \mu \mapsto \texttt{fat\_type} * \eta')\rangle$. Moreover, by inspection of e_2 :=`x.fat <- List.fold_left ... x.fat fat_allocations`, one notices that it needs to have access to memory cell μ, i.e., its type will be along the lines of $\Gamma; \mu \mapsto \texttt{fat\_type} * \eta'' \vdash e_2 : \varphi$. But for $e_1\,;\, e_2$ to type, the postcondition of e_1 would have to match the precondition of e_2: In particular, in both, μ should have the same wait prefix. But this is clearly not the case: in the postcondition of e_1, μ is wrapped in a wait permission for π, while in the precondition of e_2, it outside all wait permissions.

By analyzing the code, one finds that this corresponds to an concurrency problem: The code in `update_directory_containing` runs in its own task that is never being waited for. Therefore, it can be arbitrarily delayed. But since it depends on the state of the FAT at the time of invocation to do its work, while the arbitrary delay may case the FAT data structure to change significantly before this function is actually run.

6 Limitations

A major limitation of ALS types is that it enforces a strict ownership discipline according to which data is owned by a single process and ownership can only be passed at task creation

[4] The code in question can be found on GitHub at `https://github.com/mirage/ocaml-fat/blob/9d7abc383ebd9874c2d909331e2fb3cc08d7304b/lib/fs.ml`

or termination. This does not allow us to type programs that synchronize using shared variables. Consider the following program implementing a mutex:

```
let rec protected_critical_section mutex data =
  if !mutex then
    mutex := false;
    (* Code modifying the reference data, posting and waiting for tasks. *)
    mutex := true;
  else
    wait (post ()); (* yield *)
    protected_critical_section mutex data

let concurrent_updates mutex data =
  post (protected_critical_section mutex data);
  post (protected_critical_section mutex data)
```

The function `concurrent_updates` does not type check despite being perfectly safe: there is a race on the mutex and on the data protected by the mutex. Similarly, we do not support other synchronization primitives such as semaphores and mailboxes (and the implicit ownership transfer associated with them). One could extend the ALS type system with ideas from separation logic to handle more complex sharing.

Also, the type system cannot deal with functions that are all both higher-order and state-changing. For example, consider the function `List.iter`, which can be given as

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

As it turns out, there is no way to provide a sufficiently general type of `iter` that allows arbitrary heap transformations of f: There is no single type that encompasses `let acc = ref 0 in iter (fun x -> acc := x + !acc) l` and `iter print l` – they have very different footprints, which is not something that can be expressed in the type system. Since our examples do not require higher-order functions with effects, we type higher-order functions in such a way that the argument functions have empty pre- and post-condition.

Finally, we do not support OCaml's object-oriented features.

7 Related Work

7.1 Dependent types

This work fits into the wider framework of dependent types. Dependent and refinement types are a popular technique to statically reason about subtle invariants in programs. There is a wide range of levels of expressivity and decidability in the different kinds of dependent types. For instance, indexed types [38, 35, 36] allow adding an "index" to a type, which can be used, for instance, to describe bounds on the value of a numerical type. While the expressivity of this approach is limited, type inference and type checking can be completely automated. Similarly, the refinement types in [17] provide a way to reason about the state of a value, e.g., whether a file handle is able to perform a given operation, by providing a way to associate predicates with types.

On the other end of the expressivity scale are languages such as Agda [21], Coq [19] and Cayenne [1] in which types are expressions. For example, types in Agda can encode formulas in an intuitionistic higher-order logic, and type inference is undecidable.

We design our type system with automation in mind. We build up on liquid types [29], whose refinement types are based on predicates in some SMT-solvable logic. This allows reasonably automatic inference of subtle invariants. While low-level liquid types [28] provide some support for dealing with heap state, this is only for an imperative setting. In particular, only basic C-style data types are supported, and concurrency is not considered. Our work builds upon these foundations by extending liquid types with a way to handle state and concurrency in a functional setting.

7.2 Aliasing and concurrency

Another stream of directly related work are logics for reasoning about the heap and concurrency. Separation logic [27] is a logic that is specifically designed to deal with the aliasing problem when reasoning about heaps. It achieves this by introducing a separating conjunction operator *, where $\varphi_1 * \varphi_2$ means that φ_1 and φ_2 must hold on disjoint portions of the heap. Building on that, concurrent separation logic [22] introduces the notion of resource ownership to separation logic, and provides reasoning principles for passing resources between tasks. The resource passing scheme is more general than that of our type system; adapting it is left as future work.

Permissions have been used in separation logic to reason about programs with locks [9, 10] and programs with dynamic thread creation [6]. For instance, Gotsman et al. [9] introduce a Locked predicate that indicates that a task holds a lock on a given heap cell, which can be used to control whether certain operations such as unlocking may be performed. Dodds et al. [6] use permissions to extend rely-guarantee reasoning into a setting with dynamically generated threads by way of fork-join parallelism. Their approach is quite similar to our use of wait permissions: Upon forking a thread t, they generate a permission $\mathsf{Thread}(t, T)$, where T is a formula describing the postcondition of the thread t, namely the states that are assumed when the thread terminates. Upon waiting, these resources are transferred to the thread that joins t.

Our reason for not directly using CSL is twofold. Since we are using a functional language, the type system already gives us a lot of information that we would have to re-derive when performing proofs in CSL. Second, most work on the automatization of separation logic [5, 2, 11] focuses on the derivation of complex heap shapes. This is not a priority in our work, since in OCaml, instead of using pointer-based representations, complex data structures would be represented using inductively-defined data types.

Our type system is based on Alias Types [32]. They provide a type system that allows precise reasoning about heaps in the presence of aliasing. The key idea of alias types is to track resource constraints that describe the relation between pointers and the contents of heap cells. These resource constraints describe separate parts of the heap, and may either describe a unique heap cell (allowing strong updates), or a summary of an arbitrary number of heap cells. The Calculus of Capabilities [4] takes a similar approach. low-level liquid types [28] uses a similar approach to reason about heap state, and extends it with a "version control" mechanism to allow for temporary reasoning about a heap cell described by a summary using strong updates.

Another type system-based approach to handling mutable state is explored in Mezzo [24]. In Mezzo, the type of a variable is interpreted as a permission to access the heap. For instance, as explained in [25], after executing the assignment to x in **let** `x = (2, "foo")` **in** `...`, a new permission is available: x@(`int`, `string`), meaning that using x, one may access a heap location containing a pair consisting of an `int` and a `string`. Without further annotation, these permissions are not duplicable, quite similar to our points-to facts. In certain situations,

e.g., the types of function arguments, permissions can be annotated as *consumed*, meaning that the corresponding heap cell is not accessible in the given form anymore, or *duplicable*, meaning the heap cell can be aliased without any issues (e.g., for an immutable heap cell). There are also additional features for explicit aliasing information and control. ALS and Mezzo address similar goals, but in two different ways: While both present a type system-based approach for handling mutable state and concurrency, ALS uses a straightforward extension of the OCaml type system, while Mezzo provides an entirely new typing approach. The result of this is that ALS's expressivity is somewhat limited, but requires little annotation and has powerful type inference, whereas Mezzo is very expressive, but requires more annotations and has only limited type inference.

7.3 Static analysis

Static analysis of asynchronous programming models have been studied in the model checking community [31, 12, 13]. These results focus on finite-state imperative programs without dynamic allocation. In contrast, our type system works with unbounded data domains and dynamically allocated heap data.

8 Conclusion

Asynchronous liquid separation types add the ability to reason about shared state in concurrent processes to liquid types, thereby increasing the expressivity of liquid types while retaining automated inference. In our preliminary experiments, we have shown that ALS types allow us to detect race conditions and prove user-specified data invariants for asynchronous code written in a functional style. It will be interesting to extend ALS to reason about common synchronization idioms in asynchronous code as well as to the full OCaml programming language.

References

1 L. Augustsson. Cayenne—a language with dependent types. In *Advanced Functional Programming*, pages 240–267. Springer, 1999.
2 J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO*, pages 115–137, 2005.
3 C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. *J. ACM*, 58(6):26, 2011.
4 K. Crary, D. Walker, and J. G. Morrisett. Typed memory management in a calculus of capabilities. In *POPL 1999*, pages 262–275, 1999.
5 D. Distefano, P. W. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *TACAS 2006*, pages 287–302, 2006.
6 M. Dodds, X. Feng, M. J. Parkinson, and V. Vafeiadis. Deny-guarantee reasoning. In *ESOP 2009*, pages 363–377, 2009.
7 T. Freeman and F. Pfenning. Refinement types for ML. In *PLDI*, pages 268–277, 1991.
8 The Go programming language. `http://golang.org/`.
9 A. Gotsman, J. Berdine, B. Cook, N. Rinetzky, and M. Sagiv. Local reasoning for storable locks and threads. In *APLAS 2007*, pages 19–37, 2007.
10 A. Hobor, A. W. Appel, and F. Z. Nardelli. Oracle semantics for concurrent separation logic. In *ESOP 2008*, pages 353–367, 2008.
11 B. Jacobs, J. Smans, and F. Piessens. A quick tour of the VeriFast program verifier. In *APLAS 2010*, pages 304–311, 2010.

12 R. Jhala and R. Majumdar. Interprocedural analysis of asynchronous programs. In *POPL*, pages 339–350, 2007.

13 A. Kaiser, D. Kroening, and T. Wahl. Dynamic cutoff detection in parameterized concurrent programs. In *CAV 2010*, 2010.

14 J. Kloos, R. Majumdar, and V. Vafeiadis. Asynchronous liquid separation types. `http://plv.mpi-sws.org/ALStypes/`. Full version.

15 B. Liskov and L. Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In *PLDI 1988*, pages 260–267, 1988.

16 A. Madhavapeddy and D. J. Scott. Unikernels: the rise of the virtual library operating system. *Commun. ACM*, 57(1):61–69, 2014.

17 Y. Mandelbaum, D. Walker, and R. Harper. An effective theory of type refinements. In *ICFP 2003*, pages 213–225, 2003.

18 N. Mathewson. Fast portable non-blocking network programming with Libevent. `http://http://www.wangafu.net/~nickm/libevent-book/`.

19 The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2012. Version 8.4.

20 Y. Minsky, A. Madhavapeddy, and H. Jason. *Real-World OCaml*. O'Reilly Media, 2013.

21 U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

22 P. W. O'Hearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, 2007.

23 B. C. Pierce. *Types and programming languages*. MIT Press, 2002.

24 F. Pottier and J. Protzenko. Programming with permissions in Mezzo. In *ICFP 2013*, pages 173–184. ACM, 2013.

25 J. Protzenko. Introduction to Mezzo. Series of two blog posts, starting at `http://gallium.inria.fr/blog/introduction-to-mezzo`, Jan 2013.

26 V. Raychev, M. T. Vechev, and M. Sridharan. Effective race detection for event-driven programs. In *OOPSLA 2013*, pages 151–166, 2013.

27 J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS 2002*, pages 55–74, 2002.

28 P. Rondon, M. Kawaguchi, and R. Jhala. Low-level liquid types. In *POPL 2010*, 2010.

29 P. M. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *PLDI 2008*, 2008.

30 The Rust programming language. `http://www.rust-lang.org/`.

31 K. Sen and M. Viswanathan. Model checking multithreaded programs with asynchronous atomic methods. In *CAV*, LNCS 4144, pages 300–314. Springer, 2006.

32 F. Smith, D. Walker, and J. G. Morrisett. Alias types. In *ESOP 2000*, pages 366–381, 2000.

33 J. Vouillon. Lwt: a cooperative thread library. In *ML 2008*, pages 3–12, 2008.

34 SatX10: A scalable plug & play parallel solver. `http://x10-lang.org/component/content/article/37-community/applications/207-satx10.html`.

35 H. Xi. Imperative programming with dependent types. In *LICS 2000*, 2000.

36 H. Xi. Dependent ML: An approach to practical programming with dependent types. *J. Funct. Program.*, 17(2):215–286, 2007.

37 H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *PLDI 1998*, pages 249–257, 1998.

38 H. Xi and F. Pfenning. Dependent types in practical programming. In *POPL 1999*, 1999.