Heap-based reasoning about asynchronous programs

Vom Fachbereich Informatik der
Technischen Universität Kaiserslautern
zur Verleihung des akademischen Grades
Doktor der Ingenieurwissenschafen (Dr.-Ing.)

genehmigte Dissertation

von

Johannes Kloos

Datum der wissenschaftlichen Aussprache: 14.06.2018

Dekan:	Stefan Deßloch
Berichterstatter:	Rupak Majumdar
Berichterstatter:	Ranjit Jhala
Berichterstatter:	Viktor Vafeiadis

D 386

Contents

Summary

Asynchronous concurrency is a wide-spread way of writing programs that deal with many short tasks. It is the programming model behind event-driven concurrency, as exemplified by GUI applications, where the tasks correspond to event handlers, web applications based around JavaScript, the implementation of web browsers, but also of server-side software or operating systems.

This model is widely used because it provides the performance benefits of concurrency together with easier programming than multi-threading.

While there is ample work on how to implement asynchronous programs, and significant work on testing and model checking, little research has been done on handling asynchronous programs that involve heap manipulation, nor on how to automatically optimize code for asynchronous concurrency.

This thesis addresses the question of how we can reason about asynchronous programs while considering the heap, and how to use this this to optimize programs. The work is organized along the main questions: (i) How can we reason about asynchronous programs, without ignoring the heap? (ii) How can we use such reasoning techniques to optimize programs involving asynchronous behavior? (iii) How can we transfer these reasoning and optimization techniques to other settings?

The unifying idea behind all the results in the thesis is the use of an appropriate model encompassing global state and a promise-based model of asynchronous concurrency. For the first question, We start from refinement type systems for sequential programs and extend them to perform precise resource-based reasoning in terms of heap contents, known outstanding tasks and promises. This extended type system is known as Asynchronous Liquid Separation Types, or ALST for short. We implement ALST in for OCaml programs using the Lwt library.

For the second question, we consider a family of possible program optimizations, described by a set of rewriting rules, the DWFM rules. The rewriting rules are type-driven: We only guarantee soundness for programs that are well-typed under ALST. We give a soundness proof based on a semantic interpretation of ALST that allows us to show behavior inclusion of pairs of programs.

For the third question, we address an optimization problem from industrial practice: Normally, JavaScript files that are referenced in an HTML file are be loaded synchronously, i.e., when a `script` tag is encountered, the browser must suspend parsing, then load and execute the script, and only after will it continue parsing HTML. But in practice, there are numerous JavaScript files for which asynchronous loading would be perfectly sound. First, we sketch a hypothetical optimization using the DWFM rules and a static analysis.

To actually implement the analysis, we modify the approach to use a dynamic analysis. This analysis, known as JSDefer, enables us to analyze real-world web pages, and provide experimental evidence for the efficiency of this transformation.

Zusammenfassung

Asynchrone Nebenläufigkeit ist ein gängiger Weg, Programm zu schreiben, die aus vielen kleinen Tätigkeiten ("Tasks") bestehen. Es ist das Programmiermodell hinter ereignisgetriebener Nebenläufigkeit, wie in: GUI-Anwendungen, wo die Tasks die Ereignis-Behandlung erledigen; Web-Anwendungen auf Basis von JavaScript; Web-Browsern sowie Servern und Betriebssystemen. Das Modell ist weit verbreitet, da es die Leistungsvorteile der Nebenläufigkeit mit einem einfacheren Programmiermodell als Multi-Threading verbindet.

Zwar gibt es viele Arbeiten über die Implementierung asynchroner Programme, wie auch über Testen und Model Checking, doch wurde wenig in die Richtung geforscht, wie man mit asynchronen Programmen, die den Heap manipulieren, umgeht, noch wie man Programme automatisch für asynchrone Nebenläufigkeit optimieren kann.

Diese Dissertation behandelt die Frage, wie man über asynchrone Programme unter Einbeziehung des Heaps argumentieren kann, und wie man damit Programm optimiert. Die Arbeit ist auf drei Grundfragen aufgebaut: (i) Wie kann man über asynchrone Programme argumentieren, ohne den Heap zu ignorieren? (ii) Wie kann man mit solchen Techniken Programm mit asynchronem Verhalten optimieren? (iii) Wie kann man solche Techniken auf andere Gebiete übertragen?

Die vereinheitlichende Idee hinter allen Resultaten der Dissertation ist die Verwendung eines geeigneten Modells, das den globalen Programmzustand und ein promisebasiertes Modell asynchroner Nebenläufigkeit enthält. Für die erste Frage gehen wir von Verfeinerungs-Typsystemen für sequentielle Programme aus und erweitern sie um ein präzises Resourcenmodell aus Heap-Inhalt, bekanntermaßen ausstehenden Tasks und Promises. Dieses erweiterte Typsystem heißt Asynchronous Liquid Separation Types, kurz ALST. Wir implementieren ALST für OCaml-Programme mit der Lwt-Bibliothek.

Für die zweite Frage betrachten wir eine Familie potentieller Programm-Optimierungen in Form von Rewriting-Regeln, die DWFM-Regeln. Die regeln sind typgetrieben: Wir garantieren Korrektheit nur für Programme, die ALST-wohltypisiert sind. Wir führen einen Korrektheitsbeweis auf Basis einer semantischen Interpretation von ALST, die uns erlaubt, Verhaltensinklusion von Programmpaaren zu zeigen.

Für die dritte Frage gehen wir ein Optimierungsproblem aus der Praxis an: Normalerweise werden JavaScript-Dateien, die in einer HTML-Datei verlinkt sind, synchron geladen, d.h., wenn ein `script`-Tag gelesen wird, muss der Browser anhalten, das Skript laden und ausführen, und darf dann erst das HTML-Parsing fortsetzen. Aber in der Praxis gibt es viele JavaScript-Dateien, für die asynchrones Laden unproblematisch wäre. Wir zeigen zuerst eine hypothetische Optimierung auf Basis von DWFM und einer statischen Analyse.

Um die Analyse tatsächlich zu implementieren, ändern wir den Ansatz, indem wir eine dynamische Analyse verwenden. Diese Analyse, JSDefer, erlaubt uns, realistische Webseiten zu analysieren, und liefert experimentelle Daten für die Effizienz dieser Transformation.

Acknowledgements

The work that went into this PhD thesis could not have been completed without the help of a great number of individualsl.

First of all, I thank my supervisor Rupak Majumdar for his continued guidance and support during my PhD time. He was a wise mentor and a great supervisor, and helped me learn a lot of what I know today. It has to be said that the depth of his knowledge is truly awe-inspiring.

Second, I wish to thank my co-author and reviewer Viktor Vafeiadis. He was always willing to have a scientific discussion, and would provide me with important insights.

I also wish to thank my third reviewer, Prof. Ranjit Jhala, and my PhD committee, Professors Schweitzer, Poetzsch-Heffter and Schneider, for the work they put into reviewing my thesis and hold my PhD defense.

Next, I want to thank all the many people that accompanied me on my way through my PhD, whether at MPI-SWS or at Instart Logic and Menlo Park Hacker Home. On one hand, I am grateful for all the scientific discussions, but also for the friendship and cameraderie. In alphabetical order, they include: Anton, Azalea, Burcu, Clark, Derek, Dmitry, Ezgi, Filip, Frank, Hai, Helen, Isabel, Ivan, Janno, Laura, Manuel, Maria, Marisol, Marko D., Marko H, Michael, Mitra, Ori, Parvez, Ralf, Rayna, Ruzica, Sadegh, Sebastian, Silke, Soham, Utkarsh and Yunjun.

Furthermore, I am grateful to all the groups that would give me the opportunity to present my work. Aside from people already mentioned, I want to thank Philippa Gardner and her group for two great visits during the last year of my PhD.

Finally, I am grateful to my parents and sisters who would always be there for me when I needed them.

1. Introduction

Asynchronous concurrency is a wide-spread way of writing programs that deal with many short tasks. It is the programming model behind event-driven concurrency, as exemplified by GUI applications, where the tasks correspond to event handlers, web applications based around JavaScript, the implementation of web browsers, but also of server-side software or operating systems.

It provides a middle-ground between sequential programming and multi-threading: the program starts by executing an initial task. During its execution, this task may create further tasks, which are put in a buffer. This is called posting tasks. Once the running tasks terminates or explicitly relinquishes control, a runnable task is taken from the task buffer and executed. This new task may again post tasks, and after it runs to completion or relinquishes control, yet another task is chosen to execute. In a sense, this form of concurrency is a form of cooperative multi-tasking.

This model is widely used because it provides the performance benefits of concurrency (although, to a lesser degree than a carefully engineered multi-threading program) together with easier programming than multi-threading (although writing asynchronous code is not as simple as writing sequential code).

In recent years, asynchronous concurrency and related models have become widely used. We list some of the best-known examples:

- Microsoft's C# language [Hejlsberg et al., 2003] provides two primitives, `async` and `await`, that post a task and wait for a task's completion, respectively. Tasks can be run in multiple threads, providing actual parallelism.
- IBM's X10 language [Charles et al., 2005] also provides `async` and `finish` constructs. Here, task are run in parallel on nodes of a distributed system.
- TAME [Krohn et al., 2007] and TaskJava [Fischer et al., 2007] both deal with event-based programming, where programs react to events from the environment. Their general approach is to model an event by posting a task. They only schedule one task at a time.
- JavaScript uses asynchronous concurrency for event handling [see, e.g., WHATWG, 2016, chapter 8.1.4]. In contrast to the examples above, it does not provide language support for doing so, relying on explicit callbacks instead. Only one task is run at a time.
- Various libraries, such as Lwt for OCaml [Vouillon, 2008] or the ES6 Promise library for JavaScript [ES6, section 25.4], provide an alternative model for asynchronous concurrency, based on primitives for creating and chaining tasks. The aforementioned

libraries are one task at a time, while other examples (such as C++-11's `std::async`) allow for multi-threaded execution of tasks.

- Android's execution model is another example of asynchronous programming. Tasks correspond to intents in Android, see Intents.

1.1. How does asynchronous programming differ from other models?

Asynchronous programming provides its own set of challenges compared to sequential and multi-threaded programming. We illustrate this by considering how asynchronous programs expose concurrency effects, but (due to their coarser form of concurrency) allows some programs to be safe that wouldn't be so under multi-threading

Here is an example of JavaScript code that has a race condition. This cannot occur in sequential code.

```
var x = 1;
setTimeout(function () { x = x * 2; }, 100);
setTimeout(function () { x = x + 3; }, 99);
setTimeout(function () { console.log(x) }, 100);
```

The `setTimeout` function posts the function given as first argument to the task buffer when the time given in the second parameter (in milliseconds) has elapsed. Due to some constraints of the JavaScript task model, which ensures that the first and third task will be run in order, this program can produce the values 2, 5 or 8. Note that the race condition only manifests because multiple tasks can be scheduled in different orders in this program.

In contrast, since tasks are essentially atomic units of execution, some programs that would be incorrect under a multi-threaded model are correct in the asynchronous model. For instance, the following code is perfectly fine:

```
function transferMoney(account1, account2, amount) {
    if (amount >= 0 && balance[account1] >= amount) {
        balance[account1] -= amount;
        balance[account2] += amount;
    }
}
setTimeout(function () { transferMoney(acc1, acc2, 200); }, 100);
setTimeout(function () { transferMoney(acc1, acc3, 700); }, 99);
```

Suppose the balance of acc1 is initially 800. In asynchronous concurrency, the balance will never become negative: Either the first task is executed first, leaving the balance at 600. Then when the second task is run, the if condition will ensure that the operation fails. Or the second task is executed first, so when the first task runs, its operation fails. While there is a race, it is an intended race.

1.2. State of the art

Having seen that the model has become quite widespread, but behaves differently from other common models, there has been a significant amount of research into making it safe and easy to use asynchronous programming.

A major area of research has been to develop analysis methods to ensure correctness and safety of asynchronous programs. This includes adaptations of techniques from the analysis of sequential and multi-threaded software, as well as specialized analyses for questions specific to asynchronous programs.

One very active area of research are tools to identify concurrency problems. While asynchronous programs do not have the full interleaving behavior of multi-threaded code, similar issues still apply. The two examples above show that there is a need for specific analysis techniques for asynchronous programs. Starting with the example of race conditions, there has been significant research in race detection techniques for asynchronous and event-driven programs, including Raychev et al. [2013], Maiya et al. [2014] and Hsiao et al. [2014].

Another area of specifically targeted analyses is May-happen-in-parallel analysis. This analysis is used to calculate whether, for a given pair of tasks, it is possible that both of them are in the task buffer and schedulable; we say that these tasks may happen in parallel. It is, in a sense, the complement to race analysis: If two tasks cannot happen in parallel, they will never race. This analysis is useful for situations where several tasks will be run at the same time, e.g., in multi-core settings: If may-happen-in-parallel analysis indicates that any two tasks that may happen in parallel do not interfere with each other (e.g., by races on the same shared variable), it is safe to schedule both of them to run at the same time.

A different area that has seen a lot of activity is model-checking and testing. One strand of research is to apply standard exploration techniques for sequential programs with some form of constrained schedule optimization; this gives an exploration of all program behaviors for certain classes of task schedules [Qadeer and Rehof, 2005, Emmi et al., 2011, Bouajjani et al., 2017]. These approaches are quite effective at bug-finding, but necessarily give under-approximations of program behavior. Another stand of research abstracts asynchronous programs into a Petri net model, using this abstraction to check reachability and termination questions [Sen and Viswanathan, 2006, Ganty et al., 2009].

One area that has not been discussed so far are deductive methods to prove correctness of asynchronous programs. To the best of our knowledge, the only publication to address this (outside the work presented in this thesis) is [Gavran et al., 2015], which extends rely-guarantee reasoning to asynchronous programs. That being said, it is possible to apply standard program logic techniques from the multi-threaded world to asynchronous programs, e.g., Deny-Guarantee reasoning [Dodds et al., 2009].

Finally, there is a small body of work in optimizing programs to be more asynchronous (e.g., by making turning function calls into memoizing futures [Molitorisz et al., 2012]), although the scope of this work is quite limited.

Summarizing the work that has been done so far, there is significant work on the implementation side of asynchronous programming, and there are plenty of specifically

targeted analyses for such programs, as well as work that analyzes finite-state abstractions. Additionally, there is some work on optimizations for asynchrony, but it doesn't go very far.

1.3. Contributions of this thesis

While there is an impressive body of research on the analysis of asynchronous programs, there are still important gaps in the existing work.

The first gap is that there is little support for reasoning about asynchronous programs for which mutable state plays a significant role. For instance, consider a web page which uses JavaScript to manage some document (e.g., something along the lines of Google Docs). This program cannot be easily modeled as a finite-state system, meaning that we cannot apply Petri Net-based techniques to show correctness properties. To prove such properties, a natural approach would be the application of deductive proof techniques. But the only technique that is specialized for asynchronous programs, using the rely-guarantee approach, has not been automated, and it is not clear that this could be done easily.

The second gap is that the translation and optimization schemes for asynchronous programs that were mentioned above do not come with soundness proofs, even though soundness is not obvious for some important steps in these transformations. We find that this is due to a lack of good proof techniques for showing some form of refinement between asynchronous programs.

The final, lesser gap is that the optimization schemes are focused on local optimizations, making only simple transformation steps. There is little published on making significant parts of programs asynchronous, while this may well be an interesting form of optimization.

We address these points along the line of three leading questions:

- How can we reason about asynchronous programs, without ignoring the heap?
- How can we show that existing program optimizations for asynchrony are correct?
- How can we scale reasoning and optimization to apply to real-world software?

This thesis provides answers to all three questions. The unifying idea behind all of these results is the use of a appropriate model of global state (usually, the heap) and a promise-based model of asynchronous concurrency. This turns out to be sufficiently strong to perform interesting reasoning, while at the same time giving rise to tractable analysis problems.

The first question is addressed by adapting techniques from the multi-threaded world, using concurrent separation logic and deny-guarantee reasoning as a base. In particular, we introduce the idea of *wait permissions*, which summarizes the postcondition of a task. This gives us a program logic for asynchronous programs.

Building on this work, we show that we can build an automated program analysis in a convenient but realistic setting, namely OCaml programs using the Lwt library: We extend the Liquid Types dependent type system with pre- and postconditions (forming a Hoare type system) and permit wait permissions in the pre- and postconditions. This

allows us to reason about resource transfer and tasks on the type level, yielding a type system called Asynchronous Liquid Separation Types (ALST for short). ALST supports automated inference of rich types that carry significant specification information.

The second question is addressed by extending the program logic from above to reason about refinements between asynchronous programs. Taking ideas from the field of logical relations in general, and Turon's CaReSL in particular, we introduce the proof technique of *delayed refinement*. This proof technique allows us to show that given two well-typed programs, every observable behavior of the first can be replicated by the second. Here, well-typed means that both programs must have the same type in the ALST type system. Using delayed simulation, we show that a number of basic rewriting rules (the DontWaitForMe system) used in the program optimizations for asynchronous programs are sound.

The third question is addressed by applying ideas from the previous sections to the analysis of web pages. We consider a problem from industrial practice: For reasons of soundness, JavaScript files that are referenced in an HTML file must be loaded synchronously, i.e., when a `script` tag is encountered, the browser must suspend parsing, then load the script and execute it, and only after that may it continue parsing the HTML file. But in practice, there are numerous JavaScript files for which asynchronous loading would be perfectly sound.

To do so, we provide a dynamic analysis, which makes it actually feasible to perform on real-world web pages, to infer which pages can be loaded asynchronously. The transformation builds on DontWaitForMe. We also evaluate the efficiency gained by performing the transformation.

In summary, this thesis has the following contributions:

- Separation logic-based reasoning about asynchronous programs using *wait permissions* (this was introduced in Kloos and Majumdar [2018], but was already implicit in Kloos et al. [2015]);
- The ALST type system, which allows for easy automatic analysis of asynchronous programs [Kloos et al., 2015];
- Delayed refinement as a proof technique for showing observational refinement between asynchronous programs [Kloos and Majumdar, 2018];
- A proof of soundness for key parts of optimizations for asynchronous programs, i.e., the DontWaitForMe rules [Kloos and Majumdar, 2018];
- A new optimization for web pages that introduces asynchronous script loading [Kloos et al., 2017];
- Experimental results on the behavior of web pages, including the performance impact of asynchronous loading [Kloos et al., 2017].

The thesis is structured along the lines of the three questions. Chapter 3 is devoted to ALST. Chapter 4 includes delayed refinement and the soundness proofs of DontWaitForMe.

Chapter 5 describes our web page optimization, JSDefer, and gives experimental results. Details of proofs are given in the appendices.

2. Related work

Below, I will review related work. I start with an overview of the programming model, and provide a simple classification. Next, I discuss previous work on analyzing asynchronous programs. Since, as mentioned above, this is unsatisfying, I also discuss work similar to ours in other settings, mostly with a view towards analysis of and reasoning about concurrent programs. Initially, I will be talking about the relatively well-explored areas of static analysis, dynamic analysis and model checking. Since these approaches do not readily apply to heap-manipulating programs, I discuss related research in deductive verification and type systems, focusing on how interesting properties are proved for multi-threaded programs and what of this research can be adapted to asynchronous programs. After that, I move on to the question of program optimization for asynchronous programs: I will first give context in terms of existing asynchrony optimizations and parallelisation. Next, I discuss work on optimization for the web, focusing on scripts and script loading. After this, I give an overview of techniques for proving program optimizations sound. To finish up, I discuss some aspects of automated program analysis.

2.1. Asynchronous and event-driven programming

There are plenty of examples of languages and libraries to support an asynchronous programming style. In the following, I will give a sample of existing work, classified by the programming interface. For each class, I describe the interface using one example, and list the others. Additionally, I will further classify each instance by analyzing how tasks can enter the task buffer, and if more than one task can be active. For the first sub-classification, I consider two classes: *Direct posting* of tasks, using some primitive that adds a task to the task buffer directly, and *event-based* task creation, where tasks are added to the task buffer as a way to handle the occurrence of some event in the environment. For the second sub-classification, I consider two classes: *Single-threaded* asynchronous concurrency, in which at most one task can be running at a given time, and *multi-threaded* asynchronous concurrency, in which multiple task can be running at the same time. In the single-threaded case, one usually allows I/O operations to run in parallel with the execution of tasks, meaning that such programs are not fully equivalent to single-threaded programs.

One way to enable asynchronous concurrency is by providing language extensions. This will often allow a near-sequential programming style. For instance, the TAME language [Krohn et al., 2007] provides the `twait` construct that suspends a currently running task to wait for some event, allowing further tasks to execute during the wait. An example from the OKWS webserver [OKWS] contains the following code:

```
twait {
  x->drain_to_network(req, connector::cnc (mkevent (r), ev, &outc));
}
```

This construct invisibly decomposes the code being executed into two tasks: One that gets started to perform the draining of data to the network, and a second one that waits for the first task to complete and performs further work. TAME task enter the task buffer as event completion handlers (i.e., it is event-based), and apart from I/O operations, only one task can be active. Similar constructs are available in C# [Hejlsberg et al., 2003] (direct posting, multi-threaded), Go [go] (direct posting, multi-threaded), X10 [Charles et al., 2005] (direct posting, multi-threaded) and TaskJava [Fischer et al., 2007] (both direct posting and events, single-threaded).

A similar style, which can be implemented without language extensions in functional programming languages, is the monadic style. We show an example using the OCaml library Lwt [Vouillon, 2008].

```
Lwt.bind (Lwt_io.read_line Lwt_io.stdin)
  (fun line -> Lwt.return (process line))
```

Here, `Lwt_io.read_line` returns a promise for a string, and `Lwt.bind` builds a task that, upon the completion of the read, received the string and may process it. Lwt uses direct posting and is single-threaded. Similar libraries include the Async library for OCaml [async (OCaml)] (direct posting, single-threaded), async for Haskell [Marlow, 2012] (direct posting, multi-threaded) and the JavaScript promise library [ES6, Archibald, 2017] (direct posting, single-threaded).

A different style is provided by callback-based approaches. A classical example of this style is JavaScript without using the promise library, for instance when using XMLHttpRequests [XMLHttpRequest].

```
function reqListener () {
  console.log(this.responseText);
}

var oReq = new XMLHttpRequest();
oReq.addEventListener("load", reqListener);
oReq.open("GET", "http://www.example.org/example.txt");
oReq.send();
```

This code fragment starts a request for downloading a resource from the network, and upon completion, posts a task to execute the function `reqListener`.

Other instances of this model include various GUI libraries such as Swing [Swing] and GTK+ [GTK]. All the examples given here are event-based and single-threaded. Android uses a similar model, *intents* [Intents], to model interactions between apps. An intent can be seen as an indirect call to a callback method, and provides a single-threaded, event-based model.

2.2. Analysis and testing of asynchronous programs

As already stated in the introduction, a number of specific analyses and testing techniques exist for concurrent and asynchronous programs.

We have mentioned race analysis and may-happen-in-parallel analysis before. Race analysis in particular is an active field; for a sample of the work in this field, consider race analysis for web applications [Raychev et al., 2013, Mutlu et al., 2015], Android [Maiya et al., 2014, Hsiao et al., 2014] or software-defined networks [El-Hassany et al., 2016], Most of these analyses are based on dynamic analysis, using a dynamically constructed happens-before relation to find potential races.

May-happen-in-parallel analysis [Naumovich and Avrunin, 1998, Naumovich et al., 1999a] identifies which statements of a program can execute in parallel. It is used as a building block for the static analysis of concurrent software [Naumovich et al., 1999b]. While the initial work was focused on multi-threaded programming models, later work [Agarwal et al., 2007, Flores-Montoya et al., 2013, Lee et al., 2012] extends the analysis to settings with asynchronous calls.

A counterpart analysis to may-happen-in-parallel analysis is *concurrency robustness analysis* [Bouajjani et al., 2017]. This analysis checks whether any execution of an asynchronous program in a multi-threaded execution model can be realized as an execution in (a restricted form of) the single-threaded model. This allows showing properties of asynchronous programs with much simpler analyses that do not need to take care of multiple interacting threads. It can also be seen as a relative to linearizability analyses, which allow reducing concurrent to sequential programs.

Jhala and Majumdar [2007] discuss another approach to the static analysis of asynchronous programs. In their work, the reduce the data flow problem to a standard sequential data flow problem, introducing a family of counters in the reduction. By bounding the counters appropriately with some parameter k, one finds that standard sequential analysis techniques now yields over- or under-approximations of the solution to the dataflow problem. Finally, they prove that for any problem instance, there is a k such that over- and under-approximation coincide.

Another notable direction in the analysis of concurrent programs is model-checking. Here, the key issue is that due to the non-determinism of task scheduling, many different possible executions exist, most of which only differ in the scheduling details, but with equal outcome.

For multi-threaded programs, various approaches exist. Partial order reduction [Godefroid, 1990] allows ruling out redundant execution paths, and can help with reducing scheduling non-determinism. On its own, it is usually not sufficient to deal with multi-threaded and asynchronous programs.

If one is interested in bug finding, one approach is to use bounded model-checking. One well-known example is the CBMC model checker for multi-threaded C programs [Wu et al., 2013, Witkowski et al., 2007].

Another direction for limiting the search depth are various strategies that control the scheduler. The observation here is that "most concurrency bugs are shallow": It only takes very simple schedules to uncover most bugs. For this reason, several classes of

bounded schedulers were introduced. Context-bounding [Qadeer and Rehof, 2005] bounds the number of context switches that may be applied during an execution to a given number. Delay bounding [Emmi et al., 2011], in contrast, starts from a given schedule and allows deviating from it by delaying a given number of scheduling decisions to later steps. Another technique is preemption bounding [Musuvathi and Qadeer, 2007]. A combination of bounded search with partial order reduction techniques is described in [Coons et al., 2013].

An entirely different approach to model-checking concurrent programs is thread-modular model checking [Flanagan and Qadeer, 2003, Henzinger et al., 2003]. The idea here is that threads communicate only through some (small) global state that evolves in a specified way. Hence, the model checking problem for the multi-threaded program is reduced to finding an environment assumption that abstracts the possible evolutions of global state, and model-checking each thread as a single-threaded program separately. This work assumes that only a fixed number of threads exist.

The first two approaches have also been applied to asynchronous programs. One example of bounded model checking for asynchronous programs is BBS [Majumdar and Wang, 2015].

Some scheduler bounding strategies are particularly applicable to or even designed for the analysis of asynchronous programs. In particular, delay-bounding applies naturally to asynchronous programs by limiting the scheduling of tasks using delays. The R4 model checker [Jensen et al., 2015] contains a bounding strategy the number of conflict reversals, where two tasks conflict with each other either by one disabling the other, or having a race. A conflict reversal means executing two conflicting tasks in a different order from the original schedule.

Yet another direction is to reduce model-checking for multi-threaded and asynchronous programs to infinite-state systems with a tractable analysis. One common approach is to use Boolean or other finite-state abstractions of the programs being analyzed and encode the analysis problem as a well-structured transition system, most commonly a Petri net.

For multi-threaded programs, one such approach is described in Kaiser et al. [2012]. For asynchronous programs, Sen and Viswanathan [2006], Ganty et al. [2009] show reductions to Petri nets for checking safety and liveness. Sen and Viswanathan [2006] reduce to multiset push-down automata instead, while Kaiser et al. [2010] use thread transition systems.

While some of these approaches are highly efficient (in particular, bounded model checking and restricting schedules), while others offer nice theoretical guarantees, they all assume that the program state is relatively small to be able to work. This is often achieved by heavy abstraction and does not work very well with heap-manipulating programs.

2.3. Deductive verification and semantics

The most common formal way to describe the semantics of concurrent and asynchronous programs is by way of small-step semantics. This is particularly true for asynchronous execution models. One exception is the work of Abadi and Plotkin [2010], which provides

denotational semantics for asynchronous programs. In their model, they give the semantics of an asynchronous by describing each task as a state transforming function, an execution trace as the sequence of state transforming functions for the tasks of the execution, and the semantics of the program as the set of traces (i.e., sequences of functions). For the languages considered in this thesis, we use the following semantics: For OCaml, we follow the semantics given by the OCaml compiler, with our core calculus being a standard lambda calculus with references and asynchronous operations; for JavaScript, we follow the ES6 standard [ES6], and for HTML, the WHATWG standard [WHATWG, 2016]. Alternative, more formalized versions of the JavaScript semantic include a lambda calculus encoding, λ_{JS}, [Guha et al., 2010], direct small-step semantics [Maffeis et al., 2008] and a mechanized specification using big-step semantics [Bodin et al., 2014]. Other specifications for HTML mostly focus on specific aspects. Featherweight Firefox [Bohannon and Pierce, 2010] focuses models the event model, while [Bichhawat et al., 2014] models information flows for security. EventRacer [Raychev et al., 2013] has formalized the task model of HTML pages to detect race conditions.

For deductive reasoning about concurrent and asynchronous programs, there are two main approaches: Rely-guarantee based reasoning and separation logic.

Rely-Guarantee reasoning was introduced by Owicki and Gries [1976]: Here, we assume that each threads has two relations associated to it, the *rely* relation, which describes what kinds of state changes it expects the environment to perform on shared states, and the *guarantee* relation, which describes which state changes it will perform on the environment itself. In the proof, the guarantee condition turns up as additional proof obligation, while the postconditions are only given up to a sequence of rely steps. In the rule for parallel composition, the rely and guarantee conditions of the threads being composed get checked for compatibility. A variation of rely-guarantee reasoning for asynchronous programs has been described by Gavran et al. [2015]. According to the authors, it is unlikely that this approach can be automated easily.

Separation logic was originally introduced by Reynolds [2002] for sequential programs, but quickly adapted to Concurrent Separation Logic [O'Hearn, 2007, Brookes, 2007, Brookes and O'Hearn, 2016], which allows thread-local reasoning, with shared data being exchanged by explicitly acquiring and releasing it using synchronization operations. At acquisition, the data will satisfy some pre-specified invariant, and on release, it must again satisfy the invariant. From the point of view of the logic, at any point, only a single thread can access any piece of shared data. This may be relaxed a bit by introducing fractional permissions [Boyland, 2013], but concurrent writes cannot be performed in this setting. More complex forms of resource transfer and ownership can be described. For instance, Concurrent Abstract Predicates [Dinsdale-Young et al., 2010] allow reasoning about data structures which provide a high-level "fiction of disjointness", e.g., by providing a data structure which models a map, where separate tasks own disjoint slices of the map, while the actual implementation may require significant data sharing. The RGSep logic [Vafeiadis and Parkinson, 2007] combines separation logic with rely-guarantee reasoning.

The entire field of program logics deals with extensions of (concurrent) separation logic to handle additional constructs, interaction patterns and use cases. Due to the numerous

publications in this field, I will not attempt to give a full summary of the field, pointing to an overview article [Brookes and O'Hearn, 2016] for a reasonably complete picture.

I will cherry-pick some pieces of work that form the basis of this thesis. Deny-guarantee reasoning [Dodds et al., 2009] introduces a way to reason about multi-threaded programs with dynamically generated tasks. This work is using a fork-join model of concurrency. Upon forking a thread t, a thread generates 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 joining, these resources are transferred to the thread that joins t. We will adapt these permissions into wait permissions, which are the analogue for asynchronous programs.

CaReSL [Turon et al., 2013a] combines ideas from program logics and logical relations to build a program logic for reasoning about program refinement. We will describe it in more detail below, when discussing methods for proving relations between programs.

Abstract Separation Logic [Calcagno et al., 2007] gives a more general view on program logics. It describes the construction of separation logics that are independent of specific programming models, allowing the establishment of many basic concepts such as the frame rule and concurrency rule as consequences of how the logic is set up, instead of difficult lemmas in the logic's soundness proof.

Several frameworks have been developed to define program logics in an abstract way. The Views framework [Dinsdale-Young et al., 2013] builds a separation logic for a given language (given by atomic commands Atom and machine states S, with an interpretation of atomic commands as relations over S), high-level view of the state (given by a "view semi-group" $(\mathsf{View}, *)$ and axiomatic semantics of atomic operations), and a reification function from view semi-group elements to machine states $\ulcorner - \urcorner : \mathsf{View} \to \mathcal{P}(S)$ such that a basic soundness condition for atomic commands is satisfied. From this, one automatically receives a corresponding program logic for the given language. Note that $\ulcorner - \urcorner$ need not be injective or consider all of the information contained in the view: It is perfectly permissible, and often desirable, to have additional virtual information in the view that is not reflected in the machine state; this kind of information is known as *ghost state*.

Iris [Krebbers et al., 2017] is another framework for defining program logics. It takes a reductionist approach, reducing all constructions to a basic, powerful model built on step-indexed Kripke semantics. It provides multiple layers. The lowest is the "algebra layer", in which algebraic structures called CMRAs are defined. These CMRAs are used to model abstract and machine states and their connection in the higher layers. Building on the algebra layers, the "base logic layer" defines a family of separation logics, parameterized by CMRAs, and provides various constructions useful for modeling complex behaviors, such as invariant properties, distributed transition systems and resource transfers. The third layer, the "program logic layer", uses the features of the base logic layer to define a weakest precondition generator (over a language that can be given by the user), deriving Hoare triples from it and allowing reasoning about programs. We use Iris to mechanize the results from Chapter 4.

2.4. Type systems

Type systems have long been used to ensure properties about programs. Reynolds [1983] noticed that types can be used to reason about abstraction. Donahue and Demers [1985] make precise the notion of strongly-typed programs, in particular requiring that a strongly typed program be representation independent, in particular implying the substitution property: One may always replace a variable by the value it currently holds without changing program behavior. Wright and Felleisen [1994] discuss this further, stating that a type system is sound if a well-typed program cannot cause type errors.

Dependent and refinement types are a popular technique to statically reason about more 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 [Xi and Pfenning, 1999, Xi, 2000, 2007] 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 [Mandelbaum et al., 2003] 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.

Liquid Types [Rondon et al., 2008] takes this approach even further: It augments base types β with a first-order *refinement predicate* ρ, giving types of the form $\{\beta \mid \rho\}$, that contain values of type β satisfying ρ. The predicates are chosen from SMT-solvable theories, allowing the derivation of strong specifications for functional programs. Liquid Types derives these specifications automatically, given a program (well-typed using normal OCaml types), a set of predicates and (optionally) interface specifications for external functions. It started a line of research including Liquid Types for C programs, including basic reasoning about the heap [Rondon et al., 2010], parallelisation analysis (by adding an effect system) [Kawaguchi et al., 2012] and Abstract Refinement Types (allowing for polymorphism on the specification level) [Vazou et al., 2013, 2015].

Taking expressiveness to even higher levels, languages such as Agda [Norell, 2007], Coq [The Coq development team, 2012] and Cayenne [Augustsson, 1999] have types which are expressions in the language itself. For example, types in Agda can encode formulas in an intuitionistic higher-order logic, and type inference is undecidable.

Hoare Type Theory [Nanevski et al., 2006] combines type-based reasoning with program logic reasoning. It introduces types of the form $\{P\}x : A\{Q\}$, which state that if an expression is evaluated starting from a state matching P, on termination, it will yield a value x of type A and a postcondition matching Q. We will construct a Hoare Type Theory version of Liquid Types in Chapter 3. In contrast to low-level Liquid Types, we support values of arbitrary type on the heap, as well as concurrency.

Our type system is based on Alias Types [Smith et al., 2000]. 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 [Crary et al., 1999]

takes a similar approach. Low-level liquid types 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 Pottier and Protzenko [2013]. In Mezzo, the type of a variable is interpreted as a permission to access the heap. For instance, as explained in [Protzenko, 2013], 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.

A less-extreme version of handling mutable state in the type system is used by the Rust programming language [Matsakis and II, 2014]: They provide various pointer types, encoding information about mutability, and enforce a strict ownership discipline: Normally, a value can only be mutated through an exclusive reference. If a shared reference is held, mutation requires the explicit use of unsafe operations.

ALST, Mezzo and Rust are different points in an expressivity continuum with regard to ownership: ALST is the most strict and limited, but also the most automatic of these systems, using a straightforward extension of the OCaml type system with the resulting powerful type inference. On the other hand, Mezzo provides an entirely new typing approach, but requires more annotations and has only limited type inference. Rust lies between these two extremes.

2.5. Optimization and parallelisation

In the second and third part of the thesis, we consider various program optimizations: In Chapter 4, we prove the soundness of schemes to introduce asynchronous concurrency to programs, while in Chapter 5, we construct a new optimization that provides asynchronous loading of scripts on web pages.

Examples of transformations considered in Chapter 4 include Fuhrer and Saraswat [2009], Markstrum et al. [2009], Molitorisz et al. [2012], Surendran and Sarkar [2016]; most of these transformations performing only simple optimizations, such as moving pure function calls into asynchronous tasks. We instead consider an abstract program rewriting system that contains a set of basic operations from which such optimizations can be built, and prove this general system sound.

A closely related family of optimizations is parallelisation: Given a program, rewrite it to make use of multiple threads and/or hardware parallelism. The *Bernstein criteria* [Bernstein, 1966] state that two blocks A and B of code are parallelizable if A neither reads nor writes memory cells that B writes, and vice versa.

Raza et al. [2009] describe a parallelisation using separation logic, adding labels

describing regions of memory, and tracking the evolution of labels. These labels are used to control aliasing in non-trivial regions that may be unfolded differently at different points in time. Type-based approaches to parallelisation include Deterministic Parallel Java [Bocchino Jr., 2011] and Liquid Effects [Kawaguchi et al., 2012].

Another approach to parallelisation, introduced by [Rinard and Diniz, 1996], analyzes parts of the program for *commutativity*. Two functions A and B commute if, starting from the same program state, executing $A; B$ and $B; A$ gives the same state. In [Aleen and Clark, 2009] the analysis is extended to commutativity up to observational equivalence.

In another direction, Cerny et al. [Cerný et al., 2013] describe program transformations that can be used to fix concurrency problems. Yet another approach is to provide parallelisation operators to the user, for instance in TAME [Krohn et al., 2007] or TaskJava [Fischer et al., 2007].

In Chapter 5, we consider optimizations for the web. For websites, one important aspect is performance. One key ingredient of website performance is front-end performance: How long does it take to load and display the page, and how responsive is it [Souders, 2008]? User studies suggest that the Time-to-Render metric, which measures the delay between the first byte being sent on the network and the first pixel displayed on the screen, best reflects user perception of page performance [Gao et al., 2017].

In this work, we focus on the effect of script loading times on page performance. Google's guidelines on improving display times [BlockingJS], recommends using the `async` and `defer` attributes of script tags to speed up page loading by making the loading of scripts more efficient.

The question of asynchronous JavaScript loading has been addressed before [Lipton et al., 2010, Kuhn et al., 2014, FAINBERG et al., 2011]; these works focus on implementing the asynchronous loading procedure, not addressing the question of which scripts can be loaded asynchronously.

Another technique to improve script loading times is to make the scripts themselves smaller. Besides compression and code optimization [Google, 2016], one may replace function bodies with stubs that, download the function implementation from the network [Livshits and Kiciman, 2008]. Asynchronous loading complements these techniques.

2.6. Proving refinement and behavior inclusion

To prove the soundness of program optimizations, one usually shows that the optimization does not change the observable behavior of the program. The three best-known techniques for doing this are simulation, reduction and contextual refinement.

Simulation [Lynch and Vaandrager, 1995] is probably the most widely used criterion. It has been successfully applied to the verification of whole compilers [Leroy, 2009], to perform translation validation [Pnueli et al., 1998] and in many other settings. Sadly, it does not scale well to programs with highly complex control flow: The approach is that one goes from a pair of related pre-states to a pair of related post-states, where the relation is a nice first-order relation between concrete states. In the setting of asynchronous programs, one would have to add some way to deal with outstanding executions, which

makes the definition significantly more difficult and brittle.

Another approaches to reason about concurrent programs is reduction, as defined by Lipton [1975]. It reasons about pairs of programs by introducing the notions of left- and right-movers, i.e., program statements that can be made to execute earlier or later than their actual position in the program, and using this to produce an essentially-sequential version of the program. To the best of our knowledge, no version dealing with asynchronous programs has been published.

The last approach, which we apply in Chapter 4, is to use logical relations and contextual refinement. Logical relations were introduced by Girard [1971], Plotkin [1973] and Reynolds [1983]. They have long been used to reason about the relationship between pairs of programs (see, e.g., the work of Pitts and Stark [1993], Ritter and Pitts [1995], Ahmed [2006], Jung and Tiuryn [1993], Dreyer et al. [2012] or more recent work [Turon et al., 2013b] that build logical relations for concurrent programs), but is somewhat hampered by having to construct precise semantic models for the problem at hand, which often requires advanced mathematical techniques.

A simpler-to-use approach, CaReSL, was pioneered by Turon et al. [2013a], who combined ideas from program logics and logical relations to build a program logic for reasoning about program refinement. Instead of constructing the relations between programs and program states directly, as in logical relations, they express the relationship properties in terms of a program logic, solving the model construction issue by reducing it to the model of the underlying concurrent separation logic. Both CaReSL and RGSIM [Liang et al., 2014] model reasoning about pairs of programs by having an assertion for the evaluation state of one of the programs, allowing the use of standard unary Hoare triples for the proofs. Krogh-Jespersen et al. [2017] showed contextual refinement of parallelisation for higher-order programs; they encoded a variant of CaReSL in Iris to perform this proof.

As it turns out, we cannot directly use the proof techniques of CaReSL and related work, due to the issue of having to proof relatedness of states between different program points (more on this in Chapter 4; compare [Raza et al., 2009], which faced a related problem). For this reason, we introduce our notion of delayed refinement as a strengthening of contextual refinement.

2.7. Automated program analysis

Separation logic has been automated to some extent, e.g., in VeriFast [Jacobs et al., 2010] or in Chalice [Leino, 2010]. The Infer project [Infer] uses Separation Logic as one building block for a static analyzer for large real-world software projects. Most work of this work focuses on the derivation of complex heap shapes. This is not a priority in our work: In OCaml, instead of using pointer-based representations, complex data structures would be represented using inductively-defined data types. For the JavaScript part, an in-depth knowledge of the heap structure is not required, since we are content with a decent over-approximation of a script's footprint.

For JavaScript programs, the most common approach is dynamic analysis. The dominant approach is to use instrumented browsers (see, e.g., Raychev et al. [2013] and

Bichhawat et al. [2014]). Alternatively, source-to-source translations can be employed for instrumentation; Jalangi [Sen et al., 2013] is an example of this approach. Static analysis has been attempted, but seems to be unscalable [Jensen et al., 2009, 2011, 2012, Kashyap et al., 2014]. Nevertheless, type-based approaches have achieved some practical success; in particular, Typescript [Bierman et al., 2014] and flow [Facebook, 2016] have become widely used in practice as useful programming tools. Other type-based approaches include Thiemann [2005] and Chugh and Jhala [2011].

Finally, the JaVerT verification engine [Santos et al., 2018] allows deductive reasoning about JavaScript. It translates JavaScript to the JSIL intermediate language and combines with specialized logics, such as the DOM logic of Raad et al. [2016], to reason about web applications.

3. Asynchronous Liquid Separation Types

The material in this chapter is taken from the paper "Asynchronous Liquid Separation Types", presented at ECOOP 2015 [Kloos et al., 2015].

In this chapter, 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* (ALST).

Refinement types [Freeman and Pfenning, 1991, Xi, 2000] 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* [Rondon et al., 2008], 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) [O'Hearn, 2007] 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. Distefano et al. [2006], Calcagno et al. [2011]) 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.

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

Figure 3.1.: The core calculus.

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

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

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

Figure 3.2.: Small-step semantics.

3.1. Examples and Overview

3.1.1. 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 3.1 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 3.2 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 $(\emptyset, [p_0 \mapsto \mathsf{run}\colon e], p_0)$. 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.

3.1.2. 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 [Vouillon, 2008] and Async [Minsky et al., 2013]. 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` $\to$ `buffer` `write: stream` $\to$ `buffer` $\to$ `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.

3.1.3. 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* [Xi, 2000, Rondon et al., 2008]. For example, in a refinement

[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

type, one can write $\{\nu : \texttt{int} \mid \nu \geq 0\}$ 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 3.2.

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 $\texttt{stream} \to \mathsf{promise}\,\{\nu : \texttt{buffer} \mid \mathrm{odd}(\nu)\}$. Dually, we can require that the write operation only writes buffers whose contents have odd parity by specifying the type $\texttt{stream} \to \{\nu : \texttt{buffer} \mid \mathrm{odd}(\nu)\} \to \mathsf{promise}\,\mathsf{unit}$. 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 [Rondon et al., 2008] generalize transparently and provide powerful reasoning tools. The situation is more complex in the presence of shared state.

3.1.4. 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` $\to \mathsf{ref}\,$`buffer` $\to \mathsf{unit}$.[2] As subtyping is unsound for references (see, e.g., [Pierce, 2002, §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 $\mathsf{ref}\,\tau_1, \ldots, \mathsf{ref}\,\tau_n$. 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 $\texttt{int}_{=1}$: `x2 := 2; !x1`. 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*.

[2] We write $\mathsf{ref}\,$`buffer` instead of `buffer` ref for ease of readability.

Resource expressions 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 expressions* that map resource names to types. Resource expressions are written using a notation inspired from separation logic. For example, the resource expression $\mu \mapsto \{\nu : \texttt{buffer} \mid \mathrm{odd}(\nu)\}$ describes a heap cell that is identified by the resource name μ and contains a value of type $\{\nu : \texttt{buffer} \mid \mathrm{odd}(\nu)\}$.

To connect references to resources, reference types are extended with indices ranging over resource names. For example, the reference $\mathsf{ref}_\mu\,\texttt{buffer}$ 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 $\mathsf{ref}_\mu\,\tau$ and a resource expression including $\mu \mapsto \tau'$, we ensure τ' is a subtype of τ. Types of the form $\mathsf{ref}_\mu\,\tau$ are called *strong references*.

Full types Types and resource expressions are tied together by using *full types* of the form $\mathsf{И}A.\,\tau\langle\eta\rangle$, where τ is a type, η is a resource expression, and A is a list of resource names that are considered "not yet bound." The $\mathsf{И}$ binder indicates that all names in A must be fresh, and therefore, distinct from all names occurring in the environment. For example, if expression e has type $\mathsf{И}\mu.\,\mathsf{ref}_\mu\,\tau\langle\mu \mapsto \tau'\rangle$, 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: $\mathsf{И}\cdot.\,\tau\langle\eta\rangle = \tau\langle\eta\rangle$ and $\forall\cdot.\tau = \tau$. If a full type has an empty resource expression, 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 $\Gamma; \eta \vdash e : \mathsf{И}A.\,\tau\langle\eta'\rangle$. 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 A. As an example, the expression `ref empty_buffer` would type as $\cdot; \mathsf{emp} \vdash \ldots : \mathsf{И}\mu.\,\mathsf{ref}_\mu\,\texttt{buffer}\langle\mu \mapsto \texttt{buffer}\rangle$.

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 $\mu, x : \mathsf{ref}_\mu\,\texttt{int}; \mu \mapsto \texttt{int} \vdash e : \texttt{int}\langle\mu \mapsto \texttt{int}\rangle$. By lambda abstraction,

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

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

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

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

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

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

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

3.1.5. 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 Raychev et al. [2013].

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` $\to$ ref `buffer` $\to$ unit and `writeb` : `stream` $\to$ ref `buffer` $\to$ 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 also 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 expression 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
(* Wait(πw, μw ↦ Bodd) * Wait(πr, μr ↦ Bodd) *)
wait drain_bufw;
(* μw ↦ Bodd * Wait(πr, μr ↦ Bodd) *)
wait fill_bufr;
(* μw ↦ Bodd * μr ↦ Bodd *)
loop bufw bufr
```

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

3.1.6. 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
```

$$
\begin{array}{rl}
\rho & \text{Refinement expressions} \\
\beta & \text{Base types} \\
\mu, \pi, \xi & \text{Resource names} \\
A & ::= \cdot \mid A, \xi \\
\tau & ::= \{\nu : \beta \mid \rho\} \mid x{:}\,\tau\langle\eta\rangle \to \varphi \mid \mathsf{ref}_\mu\,\tau \mid \mathsf{promise}_\pi\,\tau \mid \forall\xi.\tau \\
\eta & ::= \mathsf{emp} \mid \mu \mapsto \tau \mid \mathsf{Wait}(\pi, \eta) \mid \eta * \eta \\
\varphi & ::= \mathsf{И}A.\,\tau\langle\eta\rangle \\
\Gamma & ::= \cdot \mid \Gamma, x : \tau \mid \Gamma, \xi
\end{array}
$$

Figure 3.3.: Syntax of ALS types.

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 expression 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 expressions 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.

3.2. 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 3.3). 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 expressions*, 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.

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 [Liskov and Shrira, 1988] 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 $\varphi = \mathsf{И}A.\,\tau\langle\eta\rangle$ consists of three parts that describe the result of a computation: a list of resource name bindings A, 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 A describes names that are allocated during the reduction of e and occur in τ or η. The operator $\mathsf{И}$ acts as a binder; each element of A is to be instantiated by a fresh resource name.

Finally, consider resource expressions η. Resource expressions 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 [Smith et al., 2000]. 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

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

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

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

Figure 3.4.: Subtyping rules. The notations $[\![\cdot]\!]$ and $\models$ are defined in Rondon et al. [2008].

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.

3.2.1. 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 appendix. 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 3.4. 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. 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 $\Gamma \vdash \tau_1 \preceq \tau_2$.

3.2.2. Value and expression typing

Figure 3.5 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, 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 [Xi and Pfenning, 1998], and are similar to the quantification employed in alias types.

The rule T-Frame implements the frame rule from separation logic [Reynolds, 2002] in the context 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 [Rondon et al., 2010]. 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,

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

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

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

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

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

$$\textsc{T-Write}\ \frac{\Gamma; \eta_1 \vdash e_2 : И A_1.\, \tau_2\langle\eta_2\rangle \qquad \Gamma, A_1, A_2 \vdash \tau_2 \preceq \tau \qquad \Gamma, A_1; \eta_2 \vdash e_1 : И A_2.\, \mathsf{ref}_\mu\, \tau\langle\eta_3 * \mu \mapsto \tau_1\rangle}{\Gamma; \eta_1 \vdash e_1 := e_2 : И A_1, A_2.\, \mathsf{unit}\langle\eta_3 * \mu \mapsto \tau_2\rangle} \qquad \textsc{T-WWrite}\ \frac{\Gamma; \eta_1 \vdash e_2 : И A_1.\, \tau\langle\eta_2\rangle \qquad \Gamma, A_1; \eta_2 \vdash e_1 : И A.\, \mathsf{ref}_\mu\, \tau\langle\eta_3\rangle \qquad \mu\ \text{weak}}{\Gamma; \eta_1 \vdash e_1 := e_2 : И A.\, \mathsf{unit}\langle\eta_3\rangle}$$

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

$$\textsc{T-WaitTransfer}\ \frac{\Gamma; \eta \vdash e : И A.\, \mathsf{promise}_\pi\, \tau\langle\eta_1 * \mathsf{Wait}(\pi, \eta_2)\rangle}{\Gamma; \eta \vdash \mathtt{wait}\ e : И A.\, \tau\langle\eta_1 * \eta_2\rangle}$$

$$\textsc{T-App}\ \frac{\Gamma; \eta_1 \vdash e_2 : И A_1.\, \tau_x\langle\eta_2\rangle \qquad \Gamma, A_1; \eta_2 \vdash e_1 : И A_2.\, (x{:}\,\tau_x\langle\eta_4\rangle \to И A_3.\, \tau\langle\eta_5\rangle)\langle\eta_3\rangle \qquad \Gamma, A_1 \vdash \eta_3 \preceq \eta_4 * \eta_i \qquad \Gamma \vdash И A_1, A_2, A_3.\, \tau\langle\eta_5 * \eta_i\rangle\ \text{wf}}{\Gamma; \eta_1 \vdash e_1\, e_2 : И A_1, A_2, A_3.\, \tau\langle\eta_5 * \eta_i\rangle}$$

Figure 3.5.: Value and expression typing.

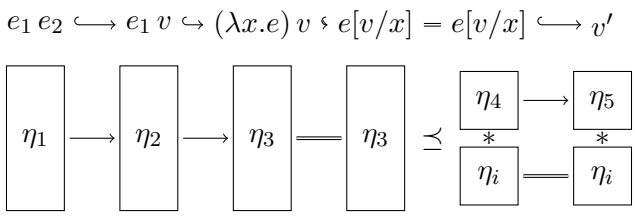


Figure 3.6.: Transformation of the global state as modeled by the T-App rule. Upper row: expression reduction steps, lower row: The corresponding resources of the global state.

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 $\texttt{wait}\, e$ 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. 3.6. 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 appendix.

3.2.3. 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 appendix; 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 appendix. 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 $\varphi = \mathit{И}A.\, \tau\langle\eta\rangle$ and $\varphi' = \mathit{И}A'.\, \tau'\langle\eta'\rangle$ be two full types. Then φ' specializes φ if there is a substitution σ such that $\mathit{И}A'.\, \tau\sigma\langle\eta\sigma\rangle = \varphi'$, 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.

3.3. 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 3.7:

1. Basic typing assigns plain OCaml types to the expression that is being typed using the Hindley-Milner algorithm.

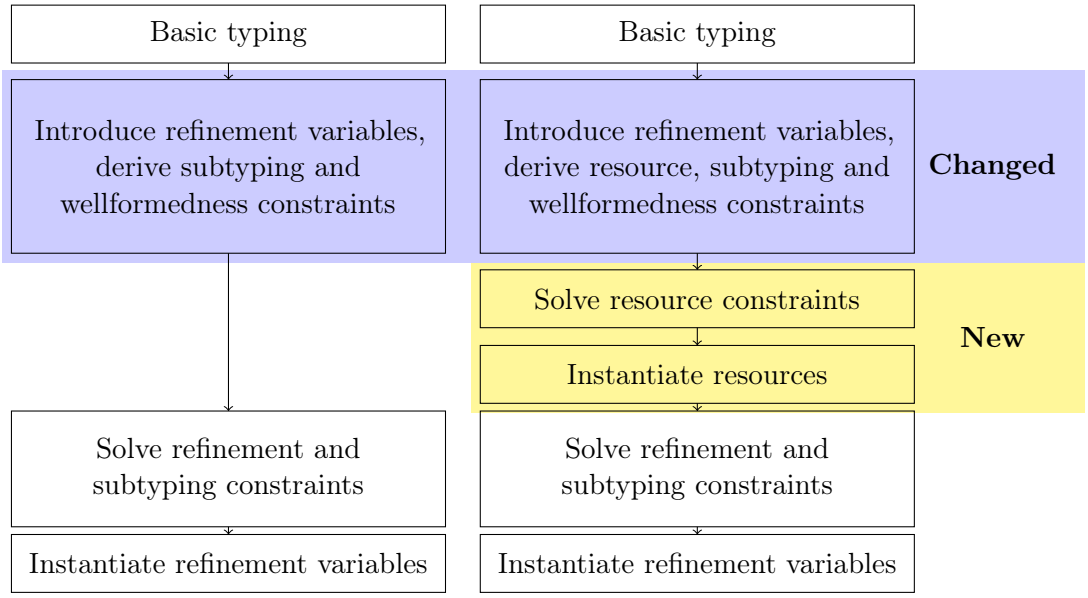


Figure 3.7.: High-level overview of the liquid type inference procedure (left), and the modified procedure presented in this section (right). The changes are highlighted.

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 $\{\nu : \texttt{int} \mid \nu \geq 6\}$.

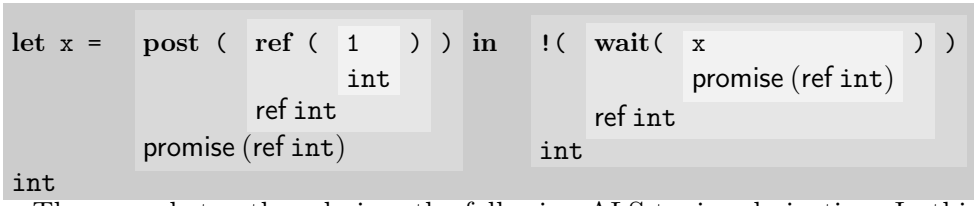
The details of this procedure are described in Rondon et al. [2008]. For ALS types, the procedure is extended to additionally derive the resources that give preconditions

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 3.7.

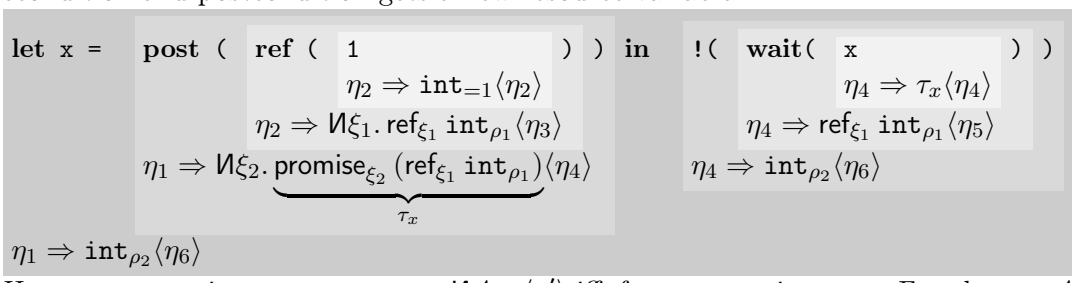
Several steps are identical to the algorithm above; the constraint derivation step has been modified, whereas the steps dealing with resource variables are new. We sketch the working of the algorithm by way of a small example. Consider the following expression:

```
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:

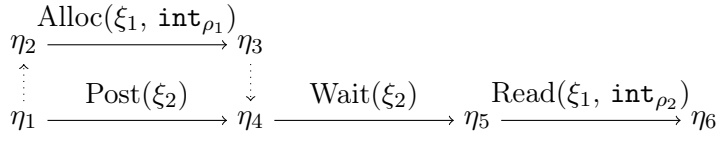


Here, an expression e types as $\eta \Rightarrow \text{И}A.\, \tau\langle\eta'\rangle$ iff, for some environment Γ and some A', $\Gamma; \eta \vdash e : \text{И}A, A'.\, \tau\langle\eta'\rangle$. The η_i occurring in the derivation are all variables.

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, Alloc(ξ_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, $\eta = \texttt{wait}(\pi_1, \texttt{wait}(\pi_2, \mu \mapsto \tau))$, 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:

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

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"):

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

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

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

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 $\eta_f = \xi_1 \mapsto \mathtt{int}_{=1}$, $\eta_w := \mathsf{Wait}(\xi_2, \xi_1 \mapsto \mathtt{int}_{=1})$ and τ_x from above):

$$\begin{array}{l} \textbf{let x =}\ \ \textbf{post (}\ \ \textbf{ref (}\ \ \mathtt{1} \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{emp} \Rightarrow \mathtt{int}_{=1}\langle \mathsf{emp} \rangle \\ \qquad\qquad\qquad\qquad\quad\ \ \mathsf{emp} \Rightarrow \mathsf{И}\xi_1.\, \mathsf{ref}_{\xi_1}\, \mathtt{int}_{\rho_1}\langle \eta_f \rangle \\ \qquad\qquad\quad\ \ \mathsf{emp} \Rightarrow \mathsf{И}\xi_2.\, \tau_x\langle \eta_w \rangle \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad\ \ \textbf{)}\ \ \textbf{)}\ \ \textbf{in}\ \ \textbf{!(}\ \ \textbf{wait(}\ \ \mathtt{x} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \eta_w \Rightarrow \tau_x\langle \eta_w \rangle \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad\ \ \eta_w \Rightarrow \mathsf{ref}_{\xi_1}\, \mathtt{int}_{\rho_1}\langle \eta_f \rangle \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad\ \ \eta_w \Rightarrow \mathtt{int}_{\rho_2}\langle \eta_f \rangle \\ \mathsf{emp} \Rightarrow \mathtt{int}_{\rho_2}\langle \eta_f \rangle \end{array}$$

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: $x : \tau_x \vdash \mathtt{int}_{=1} \preceq \mathtt{int}_{\rho_2}$, stemming from the read operation $\mathrm{Read}(\xi_1, \mathtt{int}_{\rho_2})$ that was introduced for the reference access `!(wait x)`. It indicates that the result of the read has a typing $\mathtt{int}_{\rho_2}$ that subsumes that cell content type, $\mathtt{int}_{=1}$.

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:

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

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

3.4. 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.

3.4.1. The double-buffering example, revisited

Our first example is the double-buffering copy loop from Section 3.1. 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, **post** `(Writer.write outs buffer_full)`, creating a task that writes a full buffer to the disk, we write **post** `(Writer.write outs buffer_empty)`, i.e., post a task that tries to write the read buffer.
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:

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

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].

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

[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`

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 3.1 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:

$$\forall \mu_1.\texttt{buf\_full} : \mathsf{ref}_{\mu_1}\ \texttt{buffer} \to \forall \mu_2.\texttt{buf\_empty} : \mathsf{ref}_{\mu_2}\ \texttt{buffer} \\ \langle \mu_2 \mapsto \texttt{buffer} * \mu_1 \mapsto \{\nu : \texttt{buffer} \mid \mathrm{odd}(\nu)\}\rangle \to \mathsf{unit}\langle \ldots \rangle$$

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.

3.4.2. Another asynchronous copying loop

The "Real World OCaml" book [Minsky et al., 2013, 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
```

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

Figure 3.8.: Types for asynchronous I/O functions in the Async library

where the functions `Reader.read`, `Writer.write` and `Writer.flushed` have the types given in Figure 3.8. One possible implementation of `Reader.read` is the following:

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

where `sync_read` is typed as $\texttt{stream} \to \mathsf{ref}\,\texttt{buffer} \to \texttt{int}$, 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`:

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

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`.

3.4.3. Coordination in a parallel SAT solver

The next example is a simplified version of an example from [X10]. 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:
5    c:cnf → ∀μ.ref_μ cnf⟨μ ↦ cnf_≡c⟩ → {ν : worker_result | res_ok c ν}⟨μ ↦ cnf_≡c⟩
6  val update: c:cnf → ∀μ.ref_μ cnf⟨μ ↦ cnf_≡c⟩ → cnf_≡c⟨μ ↦ cnf_≡c⟩
7  (* The example code *)
8  let parallel_SAT c =
9    let buffer1 = ref c and buffer2 = ref c in
10   let rec main c1 worker1 worker 2 =
11     if * then (* non-deterministic choice; in practice, use a select *)
12       match wait worker1 with
13       | SAT -> discard worker2; true
14       | UNSAT -> discard worker2; false
15       | POLL ->
16         let c2 = update c1 buffer1 in
17         let w = post (worker c2 buffer1) in
18         main c2 w worker2
19     else
20       ... (* same code, with roles switched *)
21   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

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

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):

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

This type is clearly equivalent to the expected type.

3.4.4. The MirageOS FAT file system

Finally, we considered a version of the MirageOS [Madhavapeddy and Scott, 2014] 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 $\Gamma; \mu \mapsto$ `fat_type` $* \eta \vdash e_1 : \mathit{И}\pi, A.\, \tau\langle\mathsf{Wait}(\pi, \mu \mapsto$ `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 $\Gamma; \mu \mapsto$ `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.

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

3.5. 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 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.

4. DontWaitForMe

Although using asynchronous concurrency to improve performance and handle multiple parallel operations is an attractive approach, actually programming in this style has proved troublesome. The need to split function bodies at the point were asynchronous calls are made, and to maintain the program state correctly across task boundaries, have proved to be problematic. For this reason, various proposals, such as TaskJava [Fischer et al., 2007], TAME [Krohn et al., 2007] or X10 [Charles et al., 2005, Markstrum et al., 2009, Fuhrer and Saraswat, 2009], as well as more recent developments such as the async/await constructs of C# have been made to make asynchronous programming easier. Inherent in all of this work is the assumption that there is a simple transformation scheme from sequential to asynchronous programs: as long as we break no data dependencies, it should make no difference if certain parts of the program are run in sequence or in parallel tasks.

While all this work has provided clear efficiency improvements and is intuitively correct, none of the cited work give a proof that making parts of the program asynchronous is sound: it is not shown that these transformations do no introduce additional behaviors. In fact, this is not very surprising, since it requires relational reasoning about programs with mutable state, complex concurrency and implicit resource transfers.

In this chapter, we revisit the question of how to prove such transformation schemes sound. We make use of recent developments in the fields of program logics and logical relations. In particular, we make use of the work of Deny-Guarantee reasoning [Dodds et al., 2009] and CaReSL [Turon et al., 2013a] to reason about (pairs of) asynchronous programs.

The main notion in proving the correctness of transformations using the methods described above is *contextual refinement*. Suppose we are given some programming language with expressions e and contexts $\mathcal{C}$, where $\mathcal{C}[e]$ is the application of e to $\mathcal{C}$. Assume that we also have a type system for e, and let two expressions e and e' be given such that they both have the same type τ. We say that e is a contextual refinement of e' if for every context $\mathcal{C}$ such that $\mathcal{C}[e]$ and $\mathcal{C}[e']$ are well-typed, if $\mathcal{C}[e]$ has a terminating execution, so does $\mathcal{C}[e']$.

Surprisingly, a direct application of proof techniques for establishing contextual refinement fails when dealing with asynchronous programs. Intuitively, the reason for this is that these techniques prove that pairs of programs move from pairs of related states to pairs of related states. But in the case of asynchronous programs, we can only establish that tasks have moved to related states once both of them have completed; since we have no control over when a task completes, this leaves a gap in such proof attempts. We discuss this issue in detail in Section 4.2.2.

Our solution is to strengthen the notion of refinement. We introduce *delayed refinement*. Intuitively, instead of working on related pairs of states, we attach predicates to the states

$$e ::= c \mid x \mid \texttt{let}\, x = e\, \texttt{in}\, e \mid \texttt{if}\, e\, \texttt{then}\, e\, \texttt{else}\, e \mid \texttt{ref}\, e \mid \texttt{!}\, e \mid e := e \mid \texttt{post}\, e \mid \texttt{wait}\, e$$

Figure 4.1.: A fragment of the core calculus

on both sides of the proof. These predicates are connected in such a way that if the pair of predicates for both sides hold *at the same time*, they imply that the states are related. Thus, we recover the usual refinement criterion as a special case, and can deduce contextual refinement from delayed refinement. To define these predicates, we have to give a semantic interpretation of an ALST subset.

4.1. A core calculus and type system

In this chapter, we consider a fragment of the core calculus and type system from ALST. The reason for choosing this fragment is that reasoning about the full core calculus and type system incurs significant additional technical burden, obscuring the main points of the already intricate argument with low-level details.

In our fragment, we leave out functions (adding `let` bindings as an explicit construct) and function types. Furthermore, we drop refinements, omit the type argument from reference types (reference types are given as ref_ξ), and explicitly annotate the sets of allocated names in promises and wait permissions. The syntax of the simplified core calculus can be found in Fig. 4.1, while the semantics are the same as those given in the previous chapter (cf. Fig. 3.2). The type system is given in Fig. 4.2 (syntax and name sets) and Fig. 4.3 (typing rules). In Fig. 4.2, we also define a family of functions names from types, resource expressions and environments to sets of names, and rnames from resource expressions to sets of names. The former collects all names occurring in the given type, resource expression or environment, while the latter provides the names occurring in points-to facts and wait permissions of the resource expression. $\mathsf{rnames}(\eta)$ is always a subset of $\mathsf{names}(\eta)$; we have that $\mathsf{names}(\xi \mapsto \mathsf{ref}_\chi) = \{\xi, \chi\}$, but $\mathsf{rnames}(\xi \mapsto \mathsf{ref}_\chi) = \{\xi\}$. We now have three kinds of types: base types β, which include Boolean `bool` and the unit type unit, references ref_ξ and promises $\mathsf{promise}_{\xi,A}\,\tau$. The base types are the types of constants. A reference type ref_ξ expresses that a value of this type refers to a heap cell with logical name ξ; it makes no assertion about the content of the heap cell. The content will be constrained later on using resource expressions. Finally, $\mathsf{promise}_{\xi,A}\,\tau$ asserts that a value with this type refers to a task with logical name ξ that will, upon termination, yield a value of type τ. The set A is used for bookkeeping; it keeps track of the logical names that task ξ allocates.

Additionally, we have resource expressions. They are given as formula in a separation logic-like notation, where emp describes an empty heap fragment and $\eta_1 * \eta_2$ describes a heap fragment that is made up of two disjoint parts η_1 and η_2. The expression $\xi \mapsto \tau$ describes a fragment of a heap that has a single cell with logical name ξ, containing a value of type τ. Finally, $\mathsf{Wait}(\xi, A, \eta)$ states when the task with logical names ξ completes, it will make available the resources given by η. The names allocated by the task are given

$\beta \supseteq \{\mathtt{bool}, \mathsf{unit}\}$	Set of base types
$\xi \in \mathsf{names}$	Unique resource names
$x \in \mathsf{var}$	Variable names
$A \subseteq_{fin} \mathsf{names}$	Finite sets of (newly allocated) names
$\tau ::= \beta \mid \mathsf{ref}_\xi \mid \mathsf{promise}_{\xi,A}\,\tau$	Types
$\eta ::= \mathsf{emp} \mid \eta * \eta \mid \xi \mapsto \tau \mid \mathsf{Wait}(\xi, A, \eta)$	Resource expressions
$\mathrm{ty} : \mathsf{C} \to \tau$	Types of constants

$$
\begin{aligned}
\mathsf{names}(\beta) &= \emptyset \\
\mathsf{names}(\mathsf{ref}_\xi) &= \{\xi\} \\
\mathsf{names}(\mathsf{promise}_{\xi,A}\,\tau) &= \{\xi\} \cup (\mathsf{names}(\tau) \setminus A) \\
\mathsf{names}(\mathsf{emp}) &= \emptyset \\
\mathsf{names}(\eta_1 * \eta_2) &= \mathsf{names}(\eta_1) \cup \mathsf{names}(\eta_2) \\
\mathsf{names}(\xi \mapsto \tau) &= \{\xi\} \cup \mathsf{names}(\tau) \\
\mathsf{names}(\mathsf{Wait}(\xi, A, \eta)) &= \{\xi\} \cup (\mathsf{names}(\eta) \setminus A) \\
\mathsf{names}(x_1 : \tau_1, \ldots, x_n : \tau_n) &= \bigcup_{i=1}^{n} \mathsf{names}(\tau_i) \\
\mathsf{names}(\tau; \eta) &= \mathsf{names}(\tau) \cup \mathsf{names}(\eta) \\
\mathsf{names}(\Gamma; \eta) &= \mathsf{names}(\Gamma) \cup \mathsf{names}(\eta)
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{rnames}(\mathsf{emp}) &= \emptyset \\
\mathsf{rnames}(\eta_1 * \eta_2) &= \mathsf{rnames}(\eta_1) \cup \mathsf{rnames}(\eta_2) \\
\mathsf{rnames}(\xi \mapsto \tau) &= \{\xi\} \\
\mathsf{rnames}(\mathsf{Wait}(\xi, A, \eta)) &= \{\xi\} \cup (\mathsf{rnames}(\eta) \setminus A)
\end{aligned}
$$

Figure 4.2.: Types for the simplified core calculus.

$$\mathrm{ty}(\texttt{true}) = \mathrm{ty}(\texttt{false}) = \texttt{bool} \qquad \mathrm{ty}(\texttt{()}) = \mathsf{unit} \qquad \frac{}{\Gamma; \mathsf{emp} \vdash c : \text{И}\emptyset.\, \mathrm{ty}(c)\langle \mathsf{emp}\rangle}$$

$$\frac{x : \tau \in \Gamma}{\Gamma; \mathsf{emp} \vdash x : \text{И}\emptyset.\, \tau\langle \mathsf{emp}\rangle} \qquad \frac{\begin{array}{c}\Gamma; \eta_1 \vdash e_1 : \text{И}A_1.\, \texttt{bool}\langle \eta_2\rangle \\ \Gamma, A_1; \eta_2 \vdash e_2 : \text{И}A_2.\, \tau\langle \eta_3\rangle \qquad \Gamma, A_1; \eta_2 \vdash e_3 : \text{И}A_2.\, \tau\langle \eta_3\rangle\end{array}}{\Gamma; \eta_1 \vdash \texttt{if}\, e_1\, \texttt{then}\, e_2\, \texttt{else}\, e_3 : \text{И}A_1, A_2.\, \tau\langle \eta_3\rangle}$$

$$\frac{\Gamma; \eta_1 \vdash e : \text{И}A.\, \tau\langle \eta_2\rangle \qquad \xi \text{ fresh}}{\Gamma; \eta_1 \vdash \texttt{ref}\, e : \text{И}A, \xi.\, \mathsf{ref}_\xi\, \langle \eta_2 * \xi \mapsto \tau\rangle} \qquad \frac{\Gamma; \eta_1 \vdash e : \text{И}A.\, \mathsf{ref}_\xi\, \langle \eta_2 * \xi \mapsto \tau\rangle}{\Gamma; \eta_1 \vdash \texttt{!}\, e : \text{И}A.\, \tau\langle \eta_2 * \xi \mapsto \tau\rangle}$$

$$\frac{\begin{array}{c}\Gamma; \eta_1 \vdash e_1 : \text{И}A_1.\, \mathsf{ref}_\xi\, \langle \eta_2\rangle \\ \Gamma, A_1; \eta_2 \vdash e_2 : \text{И}A_2.\, \tau\langle \eta_3 * \xi \mapsto \tau\rangle\end{array}}{\Gamma; \eta_1 \vdash e_1 \,\texttt{:=}\, e_2 : \text{И}A_1, A_2.\, \mathsf{unit}\langle \eta_3 * \xi \mapsto \tau\rangle} \qquad \frac{\begin{array}{c}\Gamma; \eta_1 \vdash e_1 : \text{И}A_1.\, \tau_1\langle \eta_2\rangle \\ \Gamma, A_1, x : \tau_1; \eta_2 \vdash e_2 : \text{И}A_2.\, \tau_2\langle \eta_3\rangle\end{array}}{\Gamma; \eta_1 \vdash \texttt{let}\, x \,\texttt{=}\, e_1\, \texttt{in}\, e_2 : \text{И}A_1, A_2.\, \tau_2\langle \eta_3\rangle}$$

$$\frac{\Gamma; \eta \vdash e : \text{И}A.\, \tau\langle \eta'\rangle \qquad \xi \text{ fresh}}{\Gamma; \eta \vdash \texttt{post}\, e : \text{И}\{\xi\}.\, \mathsf{promise}_{\xi,A}\, \tau\langle \mathsf{Wait}(\xi, A, ,)\eta'\rangle}$$

$$\frac{\begin{array}{c}\Gamma; \eta \vdash e : \text{И}A_1.\, \mathsf{promise}_{\xi,A_2}\, \tau\langle \eta' * \mathsf{Wait}(\xi, A_2, \eta'')\rangle \\ A_1 \cap A_2 = \emptyset \qquad \xi \notin \mathsf{names}(\tau) \cup \mathsf{names}(\eta')\end{array}}{\Gamma; \eta \vdash \texttt{wait}\, e : \text{И}A_1, A_2.\, \tau\langle \eta' * \eta''\rangle}$$

T-Frame
$$\frac{\begin{array}{c}\Gamma; \eta \vdash e : \text{И}A.\, \tau\langle \eta'\rangle \qquad \mathsf{rnames}(\eta) \cap \mathsf{rnames}(\eta_f) = \emptyset \\ \mathsf{rnames}(\eta') \cap \mathsf{rnames}(\eta_f) = \emptyset \qquad A \cap \mathsf{names}(\eta_f) = \emptyset\end{array}}{\Gamma; \eta * \eta_f \vdash e : \text{И}A.\, \tau\langle \eta' * \eta_f\rangle}$$

T-WeakenM
$$\frac{\begin{array}{c}\Gamma; \eta \vdash e : \text{И}A.\, \tau\langle \eta'\rangle \\ \Gamma \subseteq \Gamma' \qquad A \subseteq A'\end{array}}{\Gamma'; \eta \vdash e : \text{И}A'.\, \tau\langle \eta'\rangle}$$

Figure 4.3.: Typing rules for the simplified core calculus.

by A. We assume that $(\eta, *, \mathsf{emp})$ is a partial commutative monoid.

The typing rules for the core calculus are given in Fig. 4.2 as well. They are largely the same as for ALST (cf. Fig. 3.5), with the necessary adjustments. In particular, the rules for `post` and `wait` now track the allocated names of the task.

4.1.1. A program logic for asynchronous programs

So far, all the reasoning tools we have provided for asynchronous programs were based on type systems, bringing with them a relatively coarse abstraction of the program behavior. While refinement typing allows us to constrain types quite strongly, we cannot use it to perform in-detail reasoning about program behavior; in particular, since the type system is a purely syntactic construction that is connected to the program semantics only by the type safety theorem, we cannot use it to perform proper semantic reasoning.

In this subsection, we introduce a program logic to reason about the semantics of

programs in the core calculus. In particular, we give an axiomatic semantics for e, in terms of a separation logic. For now, suppose we have formulas given by the following grammar (we will extend the logic later on):

$$
\begin{aligned}
& x, x_i, N \text{ Variables;} \qquad i \text{ Indices} \\
& \mathit{type} ::= \mathsf{val} \mid \mathsf{Set} \to \mathsf{val} \\
& t ::= x \mid v \mid t.i \\
& \phi ::= \top \mid \bot \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \implies \phi \mid \phi * \phi \mid \phi \mathrel{-\!\!*} \phi \mid t \doteq t \\
& \quad \mid \forall (x : \mathit{type}).\phi \mid \exists (x : \mathit{type}).\phi \mapsto_I t \mid \textsc{Wait}(t; x.\ \phi(t, x)) \mid t \equiv_E t \\
& \quad \mid t \twoheadrightarrow N \mid t \twoheadrightarrow \bot \mid \phi \Rrightarrow \phi
\end{aligned}
$$

The logic is multi-sorted, with terms having type val (values) or $E \to \mathsf{val}$ for some index set E (maps). Terms can be variables x (of any type), values v (of type val) or map references $t.i$; there is some index set E such that t is of type $E \to \mathsf{val}$, $i \in E$ and $t.i$ is of type val. For convenience, we alias program variables and logical variables.

We sketch a simplistic model of the logic. As usual for program logics, we start by defining a *separation algebra* [Calcagno et al., 2007] which describes a *configuration*.

A configuration consists of three parts:

1. A heap fragment H. It is given by a map from heap locations to values, and describes the actual contents of a part of the heap.

   For two heap fragments H and H', we define a partial function $*$ such that $H * H' = H \cup H'$ (as maps) whenever $\operatorname{dom} H \cap \operatorname{dom} H' = \emptyset$.

2. An abstract task buffer T. It is modeled as a map from task handles (i.e., the values return by `post`) to *task states*. A task state can either be $\mathsf{run}\colon e$, for some expression e, or $\mathsf{done}\colon v$ for some value v. For $\mathsf{run}\colon e$, e is the expressions that needs to be executed. For $\mathsf{done}\colon v$, the value v is the (concrete) return value of the finished task. Note that it reflects the task buffer in the small-step semantics.

   For two heap fragments T and T', we define $T * T' = T \cup T'$ whenever for all $t \in \operatorname{dom} T \cap \operatorname{dom} T'$, $T(t) = T'(t)$.

3. A *per-task data map* M. It contains, for each task p, either a placeholder value $\bot$ or a map N that describes the newly-allocated logical names and how they relate to actual heap locations and task handles that become exposed after the execution of this task. The per-task data map is *ghost state*: it is not reflected in actual program behavior, and can be constructed by tracing the execution of the task.

   For two per-task data maps M and M', we define $M * M' = M \cup M'$ whenever for all $t \in \operatorname{dom} M \cap \operatorname{dom} M'$, we have $M(t) = M(t') \neq \bot$.

Additionally, for a configuration (H, T, M), we enforce the invariant that $\operatorname{dom} M \subseteq \operatorname{dom} T$. We define separating conjunction on configurations (H, T, M) and (H', T', M') as $(H, T, M) * (H', T', M') = (H * H', T * T', M * M')$, whenever all the components are

defined. In the appendix, we give a more precise model, using an encoding in the Iris logic.

The always true formula $\top$, unsatisfiable formula $\bot$, conjunction $\wedge$, disjunction $\vee$, implication $\Longrightarrow$ and equality $t_1 \doteq t_2$ have their usual meaning.

Separating conjunction $\phi_1 * \phi_2$ is interpreted in the standard way as well: $(H, T, M) \models \phi_1 * \phi_2$ iff we can find $H_1, H_2, T_1, T_2, M_1, M_2$ such that $(H, T, M) = (H_1, T_1, M_1) * (H_2, T_2, M_2)$ and $(H_i, T_i, M_i) \models \phi_i$ for $i = 1, 2$.

Similarly, $\phi_1 \mathrel{-\!\!*} \phi_2$ is interpreted as follows: $(H, T, M) \models \phi_1 \mathrel{-\!\!*} \phi_2$ if we can find H', T', M' such that $(H, T, M) * (H', T', M')$ is defined, $(H', T', M') \models \phi_1$ and $(H, T, M) * (H', T', M') \models \phi_2$.

Quantification $\forall(x : \mathit{type}).\phi(x)$ and $\exists(x : \mathit{type}).\phi(x)$ is defined as: $\forall(x : \mathit{type}).\phi(x)$ holds on a configuration (H, T, M) if for all v of the given type, $\phi(v)$ holds on (H, T, M). $\exists(x : \mathit{type}).\phi(x)$ holds on (H, T, M) if there is a v of the given type such that $\phi(v)$ holds on (H, T, M).

The points-to predicate $t_1 \mapsto_I t_2$ indicates that for a configuration (H, T, M), we have $M(t_1) = t_2$, i.e., the heap fragment contains a cell t_1 with contents t_2; the index I of $\mapsto_I$ is used in the next section to disambiguate which heap of a pair of heaps we are talking about.

The *logical wait permission* $\textsc{Wait}(t; x.\ \phi(t, x))$ describes an asynchronous task. It brings a generalization of ALST-style wait permissions to the program logical level. Here, t is a term giving the handle of the task under discussion and ϕ describes the postcondition of this task. Note that ϕ gets two arguments: The task handle t, and the task's return value x. The semantics of $\textsc{Wait}(t; x.\ \phi(t, x))$ can be sketch as follows: Let a configuration (H, T, M) be given. If $T(t)$ is $\mathsf{done}: v$, then $\phi(t, v)$ holds. If t is $\mathsf{run}: \_$, the wait permission holds if the above properties hold after the corresponding task has terminated. These semantics also justify the following SplitWait rule:

$$\textsc{Wait}(t; x.\ \phi(t, x) * \phi'(t, x)) \equiv \textsc{Wait}(t; x.\ \phi(t, x)) * \textsc{Wait}(t; x.\ \phi'(t, x))$$

The *ghost update junctor* $\phi \Rrightarrow \phi'$ describes a valid update to the ghost state; in this case, this gives updates to the task data map by filling cells with data. Concretely, let (H, T, M) be given. Then $(H, T, M) \models \phi \Rrightarrow \phi'$ iff for all H', T', M_1, there is M_2 such that $(H', T', M_1) \models \phi$, $(H', T', M_2) \models \phi'$ and M_2 can be produced from $M * M_1$ by changing entries of the form $t \mapsto \bot$ to entries of the form $t \mapsto N$.

The *task data assertion* $t \rightarrowtriangle N$ associates a task handle t with a name map N of type $T_N := \mathsf{names} \to \mathsf{val}$. The task data assertion comes in two forms: $t \rightarrowtriangle N$ means that the task data map contains an entry with value N for task t, while $t \rightarrowtriangle \bot$ means that the task data map contains no entry for t. It holds that $t \rightarrowtriangle \bot * t \rightarrowtriangle \ldots \equiv \bot$, and $t \rightarrowtriangle N * t \rightarrowtriangle N' \equiv t \rightarrowtriangle N \wedge N \doteq N'$. Also, we have that $t \rightarrowtriangle \bot \Rrightarrow t \rightarrowtriangle N$ for any N.

Finally, $t_1 \equiv_E t_2$ is a predicate on maps: suppose $t_1 : E_1 \to \mathsf{val}$ and $t_2 : E_2 \to \mathsf{val}$ such that $E \subseteq E_1, E_2$. Then $t_1 \equiv_E t_2$ iff $t_1(i) = t_2(i)$ for all $i \in E$. We call it the *map overlap* predicate.

We define Hoare triples $\{\phi\}\ e\ \{x.\ \phi'(x)\}$ with the following semantics: given a configuration satisfying ϕ, any execution of e that reduces e to some value v will end up in a

configuration matching $\phi'(v)$[1]. Using this logic, we can give the Hoare triples of e as in Fig. 4.4. The soundness of these triples can be shown by reduction to Deny-Guarantee reasoning [Dodds et al., 2009].

4.1.2. Semantics of types

As a preparation for the next section, where we use an (extended) version of this program logic to show behavior inclusion, we use the program logic to give specifications to well-typed programs. For this, we introduce two functions, $[\![\tau]\!]_T(N, x)$ and $[\![\eta]\!]_T(N)$, that turn (syntactical) types into formulas in the logic (that allow semantic reasoning), and reduce to a safety result for the program logic. The parameter N for both functions has type $T_N = \mathsf{names} \to \mathsf{val}$ and is used to translate from logical names ξ to heap locations and task handles, while the parameter x of the first function takes the value whose type we wish to assert. We call N the *name map*.

In the following, we use an implicit typing convention: the variables N and N' have type T_N, and all others have type val.

The interpretation functions are given in Fig. 4.5. We start by discussing the meaning of the interpretation.

The first two cases, $[\![\mathtt{bool}]\!]_T$ and $[\![\mathsf{unit}]\!]_T$ are natural: the value x must map to `true` or `false` in the `bool` case, and to `()` in the unit case. The map N is not used, since no names appear in the types. The interpretations of emp and $\eta_1 * \eta_2$ are completely straightforward.

The cases for ref_ξ and $\xi \mapsto \tau$ form a pair. The interpretation of ref_ξ states that the value x is the same as location assigned to ξ in N, i.e., $x \doteq N.\xi$. The interpretation of $\xi \mapsto \tau$ then states that the heap cell associated with ξ contains a value v of type τ.

We explain the details of the interaction on an example. Consider the program `ref true`, which obviously types as $\cdot; \mathsf{emp} \vdash \mathtt{ref\ true} : \forall\!\!\forall \xi.\, \mathsf{ref}_\xi \langle \xi \mapsto \mathtt{bool} \rangle$ for some fresh name ξ. Anticipating later definitions, we wish to show:

$$\{[\![\cdot]\!]_T(N, \sigma) * [\![\mathsf{emp}]\!]_T(N)\}$$
$$(\mathtt{ref\ true})\sigma$$
$$\left\{x.\ \exists N'.\ N \equiv_{\overline{\{\xi\}}} N' * [\![\mathsf{ref}_\xi]\!]_T(N', x) * [\![\xi \mapsto \mathtt{bool}]\!]_T(N')\right\}$$

Simplifying, we have to show:

$$\{\top\}\ \mathtt{ref\ true}\ \left\{x.\ \exists N'.N \equiv_{\overline{\{\xi\}}} N' * [\![\mathsf{ref}_\xi]\!]_T(N', x) * [\![\xi \mapsto \tau]\!]_T(N')\right\}$$

Using the observation above, this is equivalent to showing:

$$\{\top\}\ \mathtt{ref\ true}\ \left\{x.\ \exists N', N \equiv_{\overline{\{\xi\}}} N' * \exists v.\ x \doteq N'.\xi \wedge x \mapsto_I v * (v \doteq \mathtt{true} \vee v \doteq \mathtt{false})\right\}$$

Using H-Alloc and H-Weaken, we reduce this to showing: For all y,

$$y \mapsto_I \mathtt{true} \vdash \exists N', v.\ N \equiv_{\overline{\{\xi\}}} N' * y \doteq N'.\xi \wedge y \mapsto_I v * (v \doteq \mathtt{true} \vee v \doteq \mathtt{false})$$

[1] In a sense, the execution of expression e terminates, but we make no statements about additional posted tasks.

$$\textsc{H-Const}\ \frac{}{\{\top\}\ c\ \{x.\ x \doteq c\}} \qquad \textsc{H-Context}\ \frac{\{\phi_1\}\ e\ \{x.\ \phi_2(x)\} \qquad \forall v, \{\phi_2(v)\}\ \mathcal{C}[v]\ \{x.\ \phi_3(x)\}}{\{\phi_1\}\ \mathcal{C}[e]\ \{x.\ \phi_3(x)\}}$$

$$\textsc{H-If}\ \frac{\{\phi_1 \wedge v \doteq \texttt{true}\}\ e_1\ \{x.\ \phi_2(x)\} \qquad \{\phi_1 \wedge v \doteq \texttt{false}\}\ e_2\ \{x.\ \phi_2(x)\}}{\{\phi_1 \wedge (v \doteq \texttt{true} \vee v \doteq \texttt{false})\}\ \texttt{if}\, v\, \texttt{then}\, e_1\, \texttt{else}\, e_2\ \{x.\ \phi_2(x)\}}$$

$$\textsc{H-Alloc}\ \frac{}{\{\top\}\ \texttt{ref}\, v\ \{x.\ x \mapsto_I v\}} \qquad \textsc{H-Read}\ \frac{}{\{\ell \mapsto_I v\}\ \texttt{!}\,\ell\ \{x.\ \ell \mapsto_I v \wedge x \doteq v\}}$$

$$\textsc{H-Write}\ \frac{}{\{\exists v', \ell \mapsto_I v'\}\ \ell := v\ \{\ell \mapsto_I v\}} \qquad \textsc{H-Let}\ \frac{\{\phi\}\ e[x/v]\ \{x.\ \phi'(x)\}}{\{\phi\}\ \texttt{let}\, x = v\, \texttt{in}\, e\ \{x.\ \phi'(x)\}}$$

$$\textsc{H-Post}\ \frac{\{\phi * p \mathrel{-\!\!\triangleright} \bot\}\ e\ \{x.\ \phi'(p,x)\}}{\{\phi\}\ \texttt{post}\, e\ \{p.\ \textsc{Wait}(p; x.\ \phi'(p,x))\}} \qquad \textsc{H-Wait}\ \frac{}{\{\textsc{Wait}(p; x.\ \phi(p,x))\}\ \texttt{wait}\, p\ \{x.\ \phi(p,x)\}}$$

$$\textsc{H-Frame}\ \frac{\{\phi\}\ e\ \{x.\ \phi'(x)\} \qquad x \notin \text{free}(\phi_f)}{\{\phi * \phi_f\}\ e\ \{x.\ \phi'(x) * \phi_f\}} \qquad \textsc{H-Weaken}\ \frac{\phi \mathrel{-\!\!*} \psi \qquad \forall x.\ \psi'(x) \mathrel{-\!\!*} \phi'(x) \qquad \{\psi\}\ e\ \{x.\ \psi'(x)\}}{\{\phi\}\ e\ \{x.\ \phi'(x)\}} \qquad \textsc{H-GhostUpdate}\ \frac{\phi \Rrightarrow \psi \qquad \forall x.\ \psi'(x) \Rrightarrow \phi'(x) \qquad \{\psi\}\ e\ \{x.\ \psi'(x)\}}{\{\phi\}\ e\ \{x.\ \phi'(x)\}}$$

Figure 4.4.: Hoare triples for reasoning about the core calculus. We just list the most important rules.

But this is easy to show by choosing N' such that $N'.\xi = y$, $N'.\xi' = N.\xi'$ for $\xi' \neq \xi$ and $v = \texttt{true}$.

The reason why we have separate interpretations of types and resource expressions is that we wish to be able to properly handle situations such as $\cdot; \xi \mapsto \tau \vdash () : \mathsf{И}\emptyset.\, \mathsf{unit}\langle \xi \mapsto \tau \rangle$, where ξ only occurs in the resource expressions, and vice versa. To make this work, we require some form of name map to associate logical names ξ with actual location constants, and keep the mapping of locations consistent between different interpretations.

We have another pair of interpretations for promises and wait permissions. We first discuss the commonalities: Suppose we want to interpret some object X (i.e., the τ in $\mathsf{promise}_{\xi,A}\,\tau$ or the η in $\mathsf{Wait}(\xi, A, \eta)$). Both interpretations have the general structure

$$(\text{side conds}) \wedge \textsc{Wait}(N.\xi; N'; x.\ N \equiv_{\text{Names}(X)\setminus A} N' \wedge [\![X]\!]_T(N', \ldots)).$$

First, we have some task $N.\xi$; this gives the task handle of a concrete task. The interpretation of the promise/wait permission asserts that we hold a logic-level wait permission for task $N.\xi$, with a unique name map N' such that N and N' coincide on all previously known names appearing in X — concretely, on $\mathsf{names}(X) \setminus A$. Using this name map, we assert that the interpretation of X (for that name map) holds when the

$$
\begin{aligned}
[\![\mathsf{bool}]\!]_T(N,x) &= x \doteq \mathtt{true} \vee x \doteq \mathtt{false} \\
[\![\mathsf{unit}]\!]_T(N,x) &= x \doteq () \\
[\![\mathsf{ref}_\xi\,]\!]_T(N,x) &= x \doteq N.\xi \\
[\![\mathsf{promise}_{\xi,A}\,\tau]\!]_T(N,x) &= N.\xi = x \wedge \textsc{Wait}(N.\xi; N'; y.\ N \equiv_{\mathsf{names}(\tau)\setminus A} N' \wedge [\![\tau]\!]_T(N',y)) \\
[\![\xi \mapsto \tau]\!]_T(N) &= \exists v.\ N.\xi \mapsto_I v * [\![\tau]\!]_T(N,v) \\
[\![\mathsf{Wait}(\xi,A,\eta)]\!]_T(N) &= \textsc{Wait}(N.\xi; N'; \_.\ N \equiv_{\mathsf{names}(\eta)\setminus A} N' \wedge [\![\eta]\!]_T(N')) \\
[\![\mathsf{emp}]\!]_T(N) &= \top \\
[\![\eta_1 * \eta_2]\!]_T(N) &= [\![\eta_1]\!]_T(N) * [\![\eta_2]\!]_T(N) \\
[\![x_1 : \tau_1; \ldots; x_n : \tau_n]\!]_T(N,\sigma) &:= \circledast_{i=1}^{n} [\![\tau_i]\!]_T(N, x_i\sigma)
\end{aligned}
$$

Figure 4.5.: Unary interpretation of type and resource expressions for type safety.

task $N.\xi$ terminates.

A (type-level) wait permission $\mathsf{Wait}(\xi, A, \eta)$ is mapped to a (logic-level) wait permission for $N.\xi$, ensuring that $[\![\eta]\!]_T(N')$ holds for the new name map N' bound by the logic-level wait permission.

The interpretation of promises consists of two parts; as before, we have a conjunct $N.\xi \doteq x$ that relates the value x to the name ξ. The other half is a (logic-level) wait permission for the task corresponding to ξ. We explain the details with an example.

Let us discuss the interplay between the pairs of interpretations. We illustrate it on a sequence of examples. In the examples, we will consider various situations in which $\cdot; \mathsf{emp} \vdash e : \text{И}A.\,\tau\langle\eta'\rangle$ holds, and try to prove that

$$\{\top\}\ e\ \{x.\ \exists N.\ [\![\tau]\!]_T(N,x) * [\![\eta']\!]_T(N)\}$$

holds. These are special cases of Theorem 2 below.

To understand the details of the interpretation of promises and wait permissions, first consider a task `post ()`. We want to have that

$$\{\top\}\ \mathtt{post}\ ()\ \{x.\ \exists N. [\![\mathsf{promise}_{\xi,\emptyset}\,\mathsf{unit}]\!]_T(N,x) * [\![\mathsf{Wait}(\xi,\emptyset,\mathsf{emp})]\!]_T(N)\}$$

holds. By unfolding the interpretations and definitions, we find that we have to show:

$$\{\top\}\ \mathtt{post}\ ()\ \left\{ x.\ \exists N.\ \begin{array}{c} N.\xi = x \wedge \\ \textsc{Wait}(N.\xi; y.\ \exists N'.\ N.\xi \twoheadrightarrow N' * N \equiv_{\emptyset\setminus\emptyset} N' \wedge y \doteq ()) * \\ \textsc{Wait}(N.\xi; y.\ \exists N'.\ N.\xi \twoheadrightarrow N' * N \equiv_{\emptyset\setminus\emptyset} N' \wedge \top) \end{array} \right\} \quad (4.1)$$

As a first step, notice that, given x, we can always choose N such that $N.\xi = x$. Thus, by the consequence rule, we can reduce to

$$\{\top\}\ \mathtt{post}\ ()\ \left\{ x.\ \begin{array}{c} \textsc{Wait}(x; y.\ \exists N'.\ N.\xi \twoheadrightarrow N' * N \equiv_{\emptyset} N' \wedge y \doteq ()) * \\ \textsc{Wait}(x; y.\ \exists N'.\ N.\xi \twoheadrightarrow N' * N \equiv_{\emptyset} N' \wedge \top) \end{array} \right\} \quad (4.2)$$

Using SplitWait and simplifying, we find it suffices to show:

$$\{\top\}\ \texttt{post}\ ()\ \ \left\{x.\ \ \textsc{Wait}(x; y.\ \exists N'.\ x \rightarrowtriangle N' * (N \equiv_\emptyset N' \wedge y \doteq ()) * (N \equiv_\emptyset N' \wedge \top))\ \right\} \tag{4.3}$$

Thus allows us to apply the H-Post rule; it remains to show:

$$\{\top * x \rightarrowtriangle \bot\}\ ()\ \ \left\{y.\ \exists N'.\ x \rightarrowtriangle N' * (N \equiv_\emptyset N' \wedge y \doteq ()) * (N \equiv_\emptyset N' \wedge \top)\right\} \tag{4.4}$$

At this point, we simplify the pre- and post-condition:

$$\{x \rightarrowtriangle \bot\}\ ()\ \ \left\{y.\ \exists N'.\ x \rightarrowtriangle N' * y \doteq ()\right\} \tag{4.5}$$

We can apply the H-GhostUpdate rule to get

$$\{x \rightarrowtriangle N\}\ ()\ \ \left\{y.\ \exists N'.\ x \rightarrowtriangle N' * y \doteq ()\right\} \tag{4.6}$$

At this point, we are basically done: Since the expression is simply a value, it suffices to show that the pre-condition implies the post-condition. But by choosing $N' := N$, this is trivial.

We have not yet explained the role of the N' logical variable, nor of the assertions $N \equiv_{\mathsf{names}(\ldots)\setminus A} N'$. We will describe the role of both of them in turn.

The reason why we introduce N' is that an asynchronous task may perform its own heap cell allocations and post its own tasks. These new heap cells and tasks are represented by the logical names given in A (compare the typing rule for `post`), so we have to somehow account for the mapping from the names in A to actual locations and task handles. This is the role of N': It provides a name map that contains at least a proper mapping for the names required to interpret τ (in $\mathsf{promise}_{\xi,A}\,\tau$) and η (in $\mathsf{Wait}(\xi, A, \eta)$). The overlap assertion is then used to ensure that we can properly merge name maps when performing a `wait`.

First, consider the following program: `post(ref ())`. We claim that

$$\{\top\}\ \texttt{post(ref ())}\ \left\{x.\ \exists N.\ [\![\mathsf{promise}_{\pi,\{\xi\}}\,\mathsf{ref}_\xi\,]\!]_T(N, x) * [\![\mathsf{Wait}(\pi, \{\xi\}, \xi \mapsto \mathsf{unit})]\!]_T(N)\right\}. \tag{4.7}$$

Now, suppose for a moment that $[\![\mathsf{promise}_{\pi,A}\,\tau]\!]_T(N, x)$ was simply defined as $N.\pi \doteq x \wedge \textsc{Wait}(x; y.\ [\![\tau]\!]_T(N, y))$, and $[\![\mathsf{Wait}(\pi, A, \eta)]\!]_T(N) = \textsc{Wait}(N.\pi, y.\ [\![\eta]\!]_T(N))$.

By arguing along the same lines as above, we find it is sufficient to show:

$$\{\top\}\ \texttt{post(ref ())}\ \{x.\ \exists N.\ N.\pi \doteq x \wedge \textsc{Wait}(x; y.\ N.\xi \doteq x \wedge \exists v. N.\xi \mapsto_I v * v \doteq ())\} \tag{4.8}$$

Thus, it would be sufficient to prove:

$$\textsc{Wait}(x; y.\ y \mapsto_I ()) \Rrightarrow \exists N.\ N.\pi = x \wedge \textsc{Wait}(x; y.\ N.\xi \doteq y \wedge \ldots) \tag{4.9}$$

The problem in trying to prove this is that we need to be able to provide an N. Due to the presence of wait permissions, we would need some form of excluded middle to give N — there is no constructive way of getting at the value of y. But it is well-known that a

form of excluded middle that holds for the kind of formulas we consider here does not hold for an important class of models of separation logic (the so-called "intuitionistic models"); cf. Ishtiaq and O'Hearn [2001, Section 9]. Since we want our proofs to hold in both intuitionistic and classical models, we need to provide N constructively.

So, we need to introduce an additional name map N' that handles the new names that running task π introduces. Suppose, as another strawman, that we had the following definitions:

$$\begin{aligned}[\![\mathsf{promise}_{\pi,A}\,\tau]\!]_T(N,x) &:= x \doteq N.\pi \wedge \textsc{Wait}(x; y.\ \exists N'.[\![\tau]\!]_T(N',t))\\ [\![\mathsf{Wait}(\pi,A,\eta)]\!]_T(N) &:= \textsc{Wait}(N.\pi; \_.\exists N'.[\![\eta]\!]_T(N')).\end{aligned}$$

While we can prove that (4.7) holds, we only get the following postcondition (N, x arbitrary):

$$\begin{aligned}&[\![\mathsf{promise}_{\pi,A}\,\mathsf{ref}_\xi\,]\!]_T(N,x) * [\![\mathsf{Wait}(\pi,A,\xi\mapsto\mathsf{unit})]\!]_T(N)\\ \equiv\ & x \doteq N.\pi \wedge \textsc{Wait}(x; y.\ (\exists N'.y \doteq N'.\xi) \wedge (\exists N'', v.\ N''.\xi \mapsto_I v * v \doteq ())))\end{aligned}$$

Note that we lose the connection between y and $N''.\xi$ in the points-to fact: by quantifying in this way, we lose the property that we give the same name map to both interpretation functions!

If we instead use the actual definition of $[\![\mathsf{promise}_{\pi,A}\,\tau]\!]_T(N,x)$ and $[\![\mathsf{Wait}(\pi,A,\eta)]\!]_T(N)$, we get:

$$\begin{aligned}&[\![\mathsf{promise}_{\pi,A}\,\mathsf{ref}_\xi\,]\!]_T(M,N,x) * [\![\mathsf{Wait}(\pi,A,\xi\mapsto\mathsf{unit})]\!]_T(M,N)\\ \equiv\ & x \doteq N.\pi \wedge \textsc{Wait}(x; y.\ \exists N'.x \twoheadrightarrow N' * N \equiv_{\dots} N' \wedge y \doteq N'.\xi \wedge \exists v.\ N'.\xi \mapsto_I v * v \doteq ())\end{aligned}$$

Note that using ghost state, we have restored the function of N'.

After the role of the logical variable is clear, we discuss the role of $N \equiv_{\dots} N'$. The key idea here is that we want to "stitch" different name maps when we perform a `wait` (this gives one half of the equation), and that we want interpretation function to be "monotone" (this gives the other half of the equation).

We illustrate this on another example. Consider the following program:

$$\begin{aligned}&\texttt{let}\ x = \texttt{ref}\ \texttt{true}\ \texttt{in}\\ &\texttt{let}\ y = \texttt{post}(x := \texttt{false})\ \texttt{in}\\ &\texttt{wait}\ y; x\end{aligned}$$

This program creates a reference x with initial value `true`, starts a task that sets the cell references by x to `false`, waits for that task to finish and returns x.

We start with the "stitching" property. For this, we will consider a very specific fragment of the program: $\texttt{wait}\ y; x$. We have

$$x : \mathsf{ref}_\xi\,, y : \mathsf{promise}_{\pi,\emptyset}\,\mathsf{unit}; \mathsf{Wait}(\pi,\emptyset,\xi\mapsto\texttt{bool}) \vdash \texttt{wait}\ y; x : \mathsf{ref}_\xi\,\xi\mapsto\texttt{bool}$$

We want to show that, for all N, x and y,

$$\begin{array}{l}\left\{[\![\mathsf{ref}_\xi\,]\!]_T(N,x) * [\![\mathsf{promise}_{\pi,\emptyset}\,\mathsf{unit}]\!]_T(N,y) * [\![\mathsf{Wait}(\pi,\emptyset,\xi\mapsto \mathtt{bool})]\!]_T(N)\right\}\\ \mathtt{wait}\,y; x\\ \left\{z.\ \exists N'.[\![\mathsf{ref}_\xi\,]\!]_T(N',x) * [\![\xi\mapsto \mathtt{bool}]\!]_T(N')\right\}\end{array}$$

Unfolding and simplifying gives us:

$$\left\{N.\xi \doteq x * \textsc{Wait}\left(x; N'; y.\ \begin{array}{l} N \equiv_{\mathsf{names}(\mathsf{unit})\setminus\emptyset} N' \wedge\\ N \equiv_{\mathsf{names}(\xi\mapsto \mathtt{bool})\setminus\emptyset} N' \wedge\\ y \doteq () \wedge\\ \exists v.\ N'.\xi \mapsto_I v * (v \doteq \mathtt{true} \vee v \doteq \mathtt{false})\end{array}\right)\right\}$$
$$\begin{array}{l}\mathtt{wait}\,y\ ;\ x\\ \left\{y.\ \exists N''.\ N \equiv_{\overline{\emptyset}} N'' * N''.\xi = x * \exists v.\ N''.\xi \mapsto_I v * (v \doteq \mathtt{true} \vee v \doteq \mathtt{false})\right\}\end{array}$$

Abbreviate $\phi(N) := \exists v.\ N.\xi \mapsto_I v * (v \doteq \mathtt{true} \vee v \doteq \mathtt{false})$. Using standard reasoning rules, we reduce the claim to showing, for all N and y:

$$\begin{array}{c} N.\xi \doteq x \wedge \exists N'.\ N \equiv_{\{\xi\}} N' \wedge \phi(N')\\ \vdash \exists N''.N \equiv_{\mathrm{Names}} N'' * N''.\xi = x * \phi(N'')\end{array}$$

where $\phi \vdash \phi'$ means that we can prove ϕ' starting from the assumption ϕ, using standard reasoning rules of separation logic and the semantics of the predicates. This states that we know that $N.\xi$ is x, and there is a N' such that $N.\xi = N'.\xi$ such that $\phi(N')$ holds. We have to prove that there is an N'' that coincides with N on all names, such that $N''.\xi = x$ and $\phi(N'')$ holds. Now, note that $\phi(N'')$ holds whenever $N''.\xi = N'.\xi$ (by definition of ϕ), but also $N'.\xi = N.\xi$ (since $N \equiv_{\{\xi\}} N'$). Thus, the overlap assertion allows us to prove that $\phi(N)$ holds. Hence, if we choose $N'' := N$, all the claims we have to prove hold.

More generally, suppose we have both a promise $\mathsf{promise}_{\xi,A}\,\tau$ and a wait permission $\mathsf{Wait}(\xi, A, \eta)$. Then the interpretations of promise and wait permission together assert that, given a name map N, there is a name map N' such that N and N' coincide on all names appearing in τ and η, except for those in A. Thus, when we wait for a task, we can define a new name map N'' such that $N''.\xi := N'.\xi$ when $\xi \in A$ and $N''.\xi := N.\xi$ otherwise. Due to the overlap assertion, if the interpretation of, say, τ holds under N', it also holds under N''; we formalize and prove this as a *locality property* in the next section. Conversely, we need to ensure that the interpretations of promises and wait permissions also enjoy the locality property themselves; to achieve this, we restrict the overlap to $\mathsf{names}(\tau) \setminus A$ and $\mathsf{names}(\eta) \setminus A$, respectively.

To summarize, the interpretation of promises consists of an (obvious) name map assertion $x \doteq N.\xi$, and a wait permission that contains: (1) a logical variable N' that contains the correct name map for the interpretation of the return type; (2) a map overlap assertion $N \equiv_{\mathsf{names}(\tau)\setminus A} N'$ for a return type τ that ensures that the name map N' from the wait permission and the current name map N overlap on names that are not newly allocated by the task and (3) the interpretation of the return type. The overlap

assertion ensures that we can piece together name maps in the `wait` case correctly. The interpretation of wait permissions simply is a wait permission with logical variable N' that ensures that the interpretation of the contained post-condition holds, using the name map N' (which is the same as for the promise), and a complementary overlap assertion.

The interpretation of environments, $[\![\Gamma]\!]_T(N, \sigma)$, applies the interpretation of types to the value substituted by σ for each variable. Using these interpretation functions, we define a semantic interpretation of typing judgments. Suppose $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$. Define

$$[\![\Gamma; \eta \vdash e : \text{M}A.\, \tau\langle\eta'\rangle]\!]_T := \forall N, \sigma.$$
$$\{[\![\Gamma]\!]_T(N, \sigma) * [\![\eta]\!]_T(N)\}\ \ e\sigma\ \ \{x.\ \exists N'.N \equiv_{\overline{A}} N' * [\![\tau]\!]_T(N', y) * [\![\eta']\!]_T(N')\}$$

This means that, given a name map and an instantiation of the variables σ, if the variables have correct types and the initial configuration satisfies η, any execution that reduces the main task to a value will yield a value y of type τ and end up in a configuration satisfying η'. Notice that we again have a name map overlap assertion; this is needed for the same reasons as for promises, above. We call this assertion the *name persistence condition.*

We only state the following theorem; the proof follows from that for Lemma 6. The theorem says that if a program is well-typed using the simplified type system given above, we can give it a specification that can be derived from the type easily.

Theorem 2 (Fundamental theorem for $[\![\cdot]\!]_T$) *Suppose that* $\Gamma; \eta \vdash e : \text{M}A.\, \tau\langle\eta'\rangle$ *holds. Then we have* $[\![\Gamma; \eta \vdash e : \text{M}A.\, \tau\langle\eta'\rangle]\!]_T$.

4.2. Relational reasoning for asynchronous programs

In this section, we extend the program logic and the semantic interpretations of types to allow for relational reasoning about asynchronous programs. Our aim is to develop a technique for showing, for a given program rewriting scheme, that if we rewrite a well-typed expression e into an expression e' with the same type, the behaviors of e' can be simulated by e.

In particular, we introduce the *DontWaitForMe* rewriting system, short *DWFM*, which provides a number of rewriting rules that form the basis of "asynchronization schemes", i.e., program optimizations that introduce asynchronous concurrency into programs. We start by introducing the rewrite rules.

4.2.1. The rewrite rules

The rules of the rewriting system are given in Fig. 4.6; we will discuss the rules one by one. For a given program e, execution context $\mathcal{C}$ and fresh variable x, we say that $\texttt{let}\, x = e\, \texttt{in}\, \mathcal{C}[x]$ is the *let expansion* of $\mathcal{C}[e]$. In the rules R-Commute, R-If-Left and R-If-Right, we assume that the program is sufficiently `let`-expanded; this can be ensured using the R-To-Anf rule.

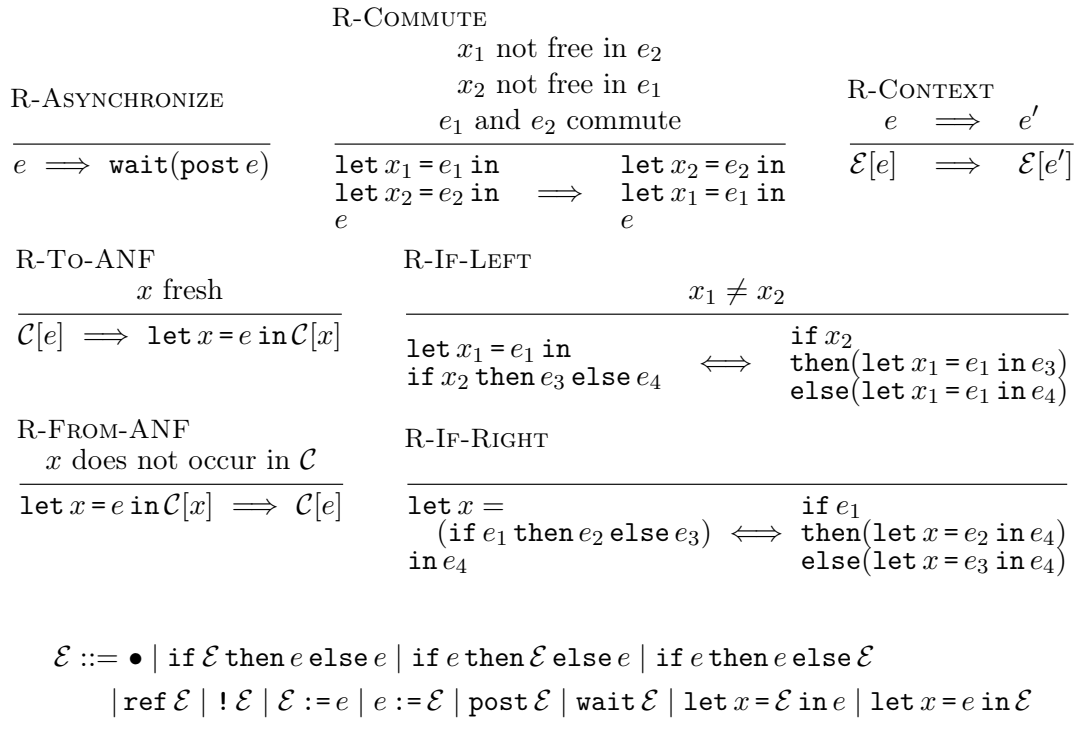


Figure 4.6.: Rewriting rules. We also define expression contexts $\mathcal{E}$. Here, $\mathcal{E}[e]$ stands for the expression generated by replacing $\bullet$ in $\mathcal{E}$ by e.

We assume throughout this section that all programs we consider are well-typed using the type system given at the start of the previous section. In fact, the R-Commute rule is explicitly type-driven, as explained below.

R-Asynchronize is the main rule to introduce asynchronous concurrency. It replaces an expression e by an expression that first creates a task to evaluate e, using $\texttt{post}\, e$, and then immediately waits for that task to return. The only change to program semantics is that it introduces a scheduling point; the soundness proof will show that this does not modify possible behaviors.

R-Commute allows exchanging operations that *commute* [Rinard and Diniz, 1996]. For the commutativity condition, we invoke a standard parallelizability criterion (compare, e.g., Raza et al. [2009]): Suppose we have two expressions e_1, e_2 such that $\Gamma; \eta_i \vdash e_i : \mathit{M} A_i.\, \tau_i \langle \eta_i' \rangle$ for $i = 1, 2$. We say that $\eta * \eta'$ is defined if $\mathsf{rnames}(\eta) \cap \mathsf{rnames}(\eta') = \emptyset$ (compare the side conditions of T-Frame). If $\eta_1 * \eta_2$, $\eta_1' * \eta_2$, $\eta_1 * \eta_2'$, and $\eta_1' * \eta_2'$ are all defined, and additionally, $A_1 \cap A_2 = \emptyset$, $\mathsf{names}(\eta_1) \cap A_2 = \mathsf{names}(\eta_2) \cap A_1 = \emptyset$ and $x_1 \neq x_2$, then e_1 and e_2 are parallelizable, and hence, commute.

The rules R-If-Left and R-If-Right are largely technical and allow moving expressions in and out of if statements. The rules R-To-Anf and R-From-Anf implement ANF translation, using contexts $\mathcal{C}$, to find the next expression to replace while respecting

evaluation order.

Finally, R-CONTEXT allows rewriting deep inside expressions. It does so using *expression contexts* $\mathcal{E}$. An expression context has the structure of an expression, except that at exactly one point of the AST of the expression, a hole $\bullet$ exists. It is more general than evaluation contexts $\mathcal{C}$, since the hole does not have to be at an evaluation position. For instance, we can decompose $\texttt{if}\, x\, \texttt{then}\, 1\, \texttt{else}\, 2$ into $\mathcal{E} = \texttt{if}\, x\, \texttt{then}\, \bullet\, \texttt{else}\, 2$ and $e = 1$, while there is no corresponding decomposition using evaluation contexts $\mathcal{C}$.

As a first step in establishing the correctness of these rewriting rules, we have the following lemma:

Lemma 1 (Rewriting preserves types) *Suppose* $\Gamma; \eta \vdash e : \text{Ⅵ} A.\, \tau\langle\eta'\rangle$ *and* $e \implies e'$. *Then there is some* $A' \supseteq A$ *such that* $\Gamma; \eta \vdash e' : \text{Ⅵ} A'.\, \tau\langle\eta'\rangle$.

One key property of program optimizations, as the one described above, is that we need to prove that the optimization does not enable additional visible program behavior. In particular, any behavior that an optimized program can exhibit must already be exhibited by the original program.

We start by introducing some useful terminology. Let e_1 and e_2 be programs; we wish to show that every behavior of e_1 can be matched by a behavior of e_2. In the following, we use the terms "implementation" and "specification" to refer to expressions e_1 and e_2, respectively, as well as associated objects. For instance, we call e_1 the implementation expression, its precondition the implementation precondition, the heap on which e_1 operates the implementation heap, and so on. Similarly, e_2 is the specification expression and so on.

In the following, we first show why standard approaches to prove the soundness of DontWaitForMe fail. Then, we define a general notion of refinement between programs, *delayed refinement*, and later prove that it implies a standard notion of transformation correctness, namely contextual refinement (see, e.g., Pitts and Stark [1993], Ritter and Pitts [1995]). In particular, we prove that if $e \implies e'$ and e is well-typed, then e' is a delayed refinement of e.

4.2.2. Why existing methods are not sufficient

There are multiple well-established approaches to prove the soundness of program transformations. While all these approaches are quite powerful, it turns out that they don't work very well for the kind of programs we are considering here. In this section, we will explain why an approach along the lines of CaReSL [Turon et al., 2013a], which would demonstrate contextual refinements, fails when used without modification.

As a first step, we extend the program logic. In particular, we add predicates for reasoning about a second, virtual heap and task buffer. We also modify the task data

assertions. Let $x, N, D, e, \Gamma, \eta, A, \sigma$ be variables of the logic.

$$\begin{aligned}\phi ::= {} & \top \mid \bot \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \implies \phi \mid \phi * \phi \mid \phi \mathrel{-\!\!*} \phi \mid t \doteq t \\ & \mid \forall(x : \mathit{type}).\ \phi(x) \mid \exists(x : \mathit{type}).\ \phi(x) \mid t \equiv_I t \\ & \mid t \mapsto_I t \mid \textsc{Wait}(t; x.\ \phi(x)) \mid t \twoheadrightarrow_I D, N \mid t \twoheadrightarrow_I \bot \mid \mathsf{Ag}(t, e, \Gamma, \eta, A, \eta', D) \\ & \mid t \mapsto_S t \mid t \Mapsto \mathsf{running}\colon e \mid t \Mapsto \mathsf{posted}\colon e \mid t \Mapsto \mathsf{done}\colon v \mid \phi \Rrightarrow \phi \mid t \twoheadrightarrow_S N \mid t \twoheadrightarrow_S \bot\end{aligned}$$

We also extend the sorts of variables to also allow expressions (implicit for variables e), typing environments (implicit for variables Γ), resource expressions (implicit for variables η), sets of names (implicit for A), contextx (implicit for variables $\mathcal{C}$), and variable substitutions (implicit for σ).

We extend the configuration from the previous section: we still have the (concrete) heap, known as the *implementation heap*, and the abstract task buffer, know as the *implementation task buffer*. We extend the task data map into the *implementation task data map*, mapping task handles to pairs D, N, where N is a name map (as before), and D is of type $T_D := \mathsf{names} \uplus \mathsf{vars} \uplus \{\boxdot\} \to \mathsf{val}$. The index set of D needs some explanation: we need to look up elements of D, known as connection data, for various interpretations. We can refer to elements of D that are indexed by names ξ (this is used by the interpretation of $\xi \mapsto \tau$), by variables names x (this is used by the interpretation of $x : \tau$ in a typing environment Γ), and $\boxdot$ (read "the return value of the expression"), which looks up connection data for the return value.

Additionally, we have some new ghost state components that model facts about the specification side, and summarize information about tasks. A configuration is given as a separation algebra with the following components:

- An implementation heap H_I, given as a map from locations to values. We have $H_I * H_I' = H_I \uplus H_I'$ whenever $\operatorname{dom} H_I \cap \operatorname{dom} H_I' = \emptyset$.
- A specification heap H_S, given as a map from locations to values. We have $H_S * H_S' = H_S \uplus H_S'$ whenever $\operatorname{dom} H_S \cap \operatorname{dom} H_S' = \emptyset$.
- An implementation task buffer T_I, given as a map from task handles to task states, as above. We have $T_I * T_I' = T_I \cup T_I'$ whenever for all $x \in \operatorname{dom} T_I \cap \operatorname{dom} T_I'$, $T_I(x) = T_I'(x)$.
- A specification task buffer T_S, given as a map from task handles to task states, as above. We have $T_S * T_S' = T_S \uplus T_S'$ whenever $\operatorname{dom} T_S \cap \operatorname{dom} T_S' = \emptyset$.
- An implementation task data map M_I. In this setting, it is given as a map from task handles to pairs consisting of a map D of type T_D and a name map N. We have $M_I * M_I' = M_I \cup M_I'$ whenever for all $x \in \operatorname{dom} M_I \cap \operatorname{dom} M_I'$, $M_I(x) = M_I'(x) \neq \bot$.
- A specification task data map M_S. In this setting, it is given as a map from task handles name maps N. We have $M_S * M_S' = M_S \cup M_S'$ whenever for all $x \in \operatorname{dom} M_S \cap \operatorname{dom} M_S'$, $M_S(x) = M_S'(x) \neq \bot$.

$$\textsc{U-Trans}\ \frac{\phi_1 \Rrightarrow \phi_2 \qquad \phi_2 \Rrightarrow \phi_3}{\phi_1 \Rrightarrow \phi_3} \qquad \textsc{U-Refl}\ \frac{}{\phi \Rrightarrow \phi} \qquad \textsc{U-Frame}\ \frac{\phi \Rrightarrow \phi'}{\phi * \phi_f \Rrightarrow \phi' * \phi_f}$$

$$\textsc{U-TaskI}\ \frac{}{t \rightarrowtriangle_I \bot \Rrightarrow t \rightarrowtriangle_I (D, N)} \qquad \textsc{U-TaskS}\ \frac{}{t \rightarrowtriangle_S \bot \Rrightarrow t \rightarrowtriangle_S N}$$

$$\textsc{H-Post}\ \frac{\{\phi * p \multimap \bot * \mathsf{Ag}(p, e, \Gamma, \eta, \eta', A, D)\}\ \ e\ \ \{x.\phi'(p, x)\}}{\{\phi\}\ \ e\ \ \{x.\textsc{Wait}(p; x.\ \phi'(p, x)) * \mathsf{Ag}(p, e, \Gamma, \eta, \eta', A, D)\}}$$

$$\textsc{E-ReflectSemantics}\ \frac{H, P, p \hookrightarrow H', P', p'}{\lfloor H, P, p \rfloor \Rrightarrow \lfloor H', P', p' \rfloor} \qquad \textsc{E-Post}\ \frac{p' \notin \operatorname{dom} P \qquad p' \neq p}{\lfloor H, P, p \rfloor \Rrightarrow \lfloor H, P[p' \mapsto \mathsf{posted}\colon e], p * p' \rightarrowtriangle_S N}$$

$$\lfloor H, P, p \rfloor := \circledast_{(\ell, v) \in H} \ell \mapsto_S v * \circledast_{(p', t) \in P} p' \Mapsto \mathrm{TS}(p, p', t)$$

$$\mathrm{TS}(p_{run}, p, t) = \begin{cases} p \Mapsto \mathsf{running}\colon e & p = p_{run}, t = \mathsf{run}\colon e \\ p \Mapsto \mathsf{posted}\colon e & p \neq p_{run}, t = \mathsf{run}\colon e \\ p \Mapsto \mathsf{done}\colon v & t = \mathsf{done}\colon v \end{cases}$$

Figure 4.7.: Reasoning rules for the extended program logic. Most rules from before carry over; due to the extended ghost state, we have a more involved H-Post rule that also takes care of the task bookkeeping data. ReflectSemantics reflects the small-step semantics into the specification part of the logic. The function $\lfloor H, P, p \rfloor$ reflects a heap and task buffer into the corresponding ghost state assertions, and the TS function translates task states to the corresponding assertions. E-Post reflects the semantics of `post`, while taking care of associated ghost state. The U-... rules describe additional facts about the update junctor.

- A task bookkeeping map M_B. It is given as a map from task handles to tuples $(e, \Gamma, \eta, A, \eta', D)$, summarizing various information about a task (the initial expression, most of the type information, and an initial connection data map).

  We have $M_B * M'_B = M_B \cup M'_B$ whenever for all $x \in \operatorname{dom} M_B \cap \operatorname{dom} M'_B$, $M_B(x) = M'_B(x)$.

- An optional specification task index J.

  J is either t or $\bot$ (not given); we have $\bot * \bot = \bot$, $t * \bot = \bot * t = t$ and $t * t'$ undefined.

The configuration is then given as the octuple $(H_I, T_I, M_I, M_B, H_S, T_S, M_S, J)$, with the following invariants:

- $\operatorname{dom} M_I \subseteq \operatorname{dom} T_I$ and $\operatorname{dom} M_B \subseteq \operatorname{dom} T_B$;
- $\operatorname{dom} M_S \subseteq \operatorname{dom} T_S$;
- If $J = t$, $t \in \operatorname{dom} M_S$.

Most of the logic is interpreted (with appropriate modifications) as before. Notably, separating conjunction and magic wand are adapted to the new separation algebra. In the following, we will discuss new and significantly modified predicates.

We start with the task data predicates. Since we now have three maps with task data (implementation task data map, specification task data map and task bookkeeping map) in the configuration, we have several corresponding predicates. For the implementation task data map, we have $t \mathrel{-\!\!\!\triangleright}_I D, N$ and $t \mathrel{-\!\!\!\triangleright}_I \bot$; they refer to an entry in the implementation task data map for task t (where t is a handle of a task on the implementation side). Here, $t \mathrel{-\!\!\!\triangleright}_I D, N$ means that the entry contains the connection map D and the name map N, while $t \mathrel{-\!\!\!\triangleright} \bot$ means that the entry contains no data. The equalities and updates given in the previous section carry over with appropriate changes. For the specification task data map, we have $t \mathrel{-\!\!\!\triangleright}_S N$ and $t \mathrel{-\!\!\!\triangleright}_S \bot$; they refer to an entry in the specification task data map for task t (where t is a handle of a task on the specification task). The semantics are the same as for the task data map in the previous section, applied to the specification task data map. Finally, the task bookkeeping predicate $\mathsf{Ag}(t, e, \Gamma, \eta, A, \eta', D)$ states that given an implementation task handle t, the task bookkeeping map contains an entry $(e, \Gamma, \eta, A, \eta', D)$ for t. We have that $\mathsf{Ag}(t_1, e_1, \Gamma_1, \eta_1, A_1, \eta'_1, D_1) * \mathsf{Ag}(t_2, e_2, \Gamma_2, \eta_2, A_2, \eta'_2, D_2) \equiv \mathsf{Ag}(t_1, e_1, \Gamma_1, \eta_1, A_1, \eta'_1, D_1) * (t_1, e_1, \Gamma_1, \eta_1, A_1, \eta'_1, D_1) = (t_2, e_2, \Gamma_2, \eta_2, A_2, \eta'_2, D_2)$.

The predicate $t \mapsto_I t'$ states that in the implementation heap, cell t contains value t'. Similarly, $t \mapsto_S t'$ states that in the specification heap, cell t contains value t'

An entirely new set of predicates is used for reasoning about tasks on the specification side, the *task predicates*. We have three predicates:

1. $t \Mapsto \mathsf{posted}\colon e$ states that in the specification task buffer, there is a posted task with handle t that will execute e. It does not assert anything about whether the task is currently executing.

2. $t \Mapsto \mathsf{running}\colon e$ states that in the specification task buffer, the task with handle t is currently running and will execute e.

3. $t \Mapsto \mathsf{done}\colon v$ states the in the specification task buffer, the task with handle t has terminated with value v.

Variables bound in the expressions e in $t \Mapsto \mathsf{posted}\colon e$ and $t \Mapsto \mathsf{running}\colon e$ are free variables of the corresponding predicates. These predicates reflect a fragment of the task buffer on the implementation side.

Since the configuration now contains many pieces of ghost state, the update operator $\phi \Rrightarrow \phi'$ also needs to be updated. The exact conditions for valid updates are quite technical; we simply give the update rules in Fig. 4.7. The reasoning rules summarize the properties of the $\Rrightarrow$ junctor, connect it with Hoare triples, and reflect the small-step semantics into the logic.

Observe that reasoning about tasks in the logic has a very different flavor for the implementation and the specification side: on the implementation side, we use wait permissions, abstracting away from actual task buffers as much as possible. On the specification side, in contrast, we explicitly modify the low-level task buffer. The reason behind this modeling approach is that we use universal Hoare triples on the implementation side, reasoning about *all possible schedules* of tasks. This is why we want to hide scheduling details and task buffers as much as possible. In contrast, on the specification side, we use existential Hoare triples, reasoning about *the existence of a single schedule*. Here, we need low-level control of the task buffer.

So far, we have been following the approach taking by CaReSL quite closely. If we continued along this path, we would define relational interpretations of types and resource expressions, like this: $[\![\tau]\!](N, x, y)$ would relate values x and y according to τ, with name map N, and $[\![\eta]\!](N)$ would relate the implementation and specification state according to η, using name map N. This approach works well in various cases; for instance, we have $[\![\mathtt{bool}]\!](N, x, y) = x \doteq y \wedge (x \doteq \mathtt{true} \vee x \doteq \mathtt{false})$. Based on this, we could define a proof obligation for contextual simulation as follows: let e, e' be expressions such that $\Gamma; \eta \vdash e : \text{И}A.\, \tau\langle\eta'\rangle$ and $\Gamma; \eta \vdash e' : \text{И}A.\, \tau\langle\eta'\rangle$. Then we say that e is a refinement of e' iff for all contexts $\mathcal{C}$,

$$\begin{array}{l}\big\{[\![\Gamma]\!](N, \sigma, \sigma') * [\![\eta]\!](N) * p \Mapsto \mathsf{running}\colon \mathcal{C}[e'\sigma']\big\}\\ e\sigma\\ \quad\big\{x.\exists N', x'.N \equiv_{\overline{A}} N' * [\![\eta']\!](N') * [\![\tau]\!](N', x, x') * p \Mapsto \mathsf{running}\colon \mathcal{C}[x']\big\}\end{array}$$

In this case, we write $[\![\Gamma; \eta \vdash e \le e' : \text{И}A.\, \tau\langle\eta'\rangle]\!]$.

The problem that we face in this case is that we need to provide a proper interpretation for promises and wait permissions. Let us ignore wait permissions for now and solely focus on promises. In the following, we will propose various strawman definitions, and show where they break.

As a first example, we could try an interpretation along the following lines (we deliber-

ately leave certain parts underspecified to highlight the main problem):

$$\begin{aligned}&[\![\mathsf{promise}_{\xi,A}\,\tau]\!](N,x,x') :=\\ &N.\xi = (x,x') * \textsc{Wait}(x, N', y.(\text{wf conds}) * \exists y'.x' \Mapsto \mathsf{done}\colon y' * [\![\tau]\!](N',y,y'))\end{aligned}$$

For this interpretation, we extend name maps to map to pairs of physical names, so $N.\xi = (x, x')$ asserts that on the implementation side, ξ corresponds to task x, while on the specification side, it corresponds to task x'.

The logic-level wait permission asserts (apart from some unstated wellformedness conditions) that when task x terminates with return value y, there is also some value y' such that task x' terminates with y' (i.e., $x' \Mapsto \mathsf{done}\colon y'$) and x and x' are related according to type τ (i.e., $[\![\tau]\!](N', y, y')$).

In fact, this definition connects nicely with the **wait** operation: It is easy to prove (with this definition) that

$$[\![x : \mathsf{promise}_{\xi,A}\,\tau; \mathsf{emp} \vdash \mathtt{wait}\,x \le \mathtt{wait}\,x : И A.\,\tau\langle\mathsf{emp}\rangle]\!]$$

The problem is that we cannot prove a crucial property about the `post` operation: We expect that if we can prove $[\![\cdot; \mathsf{emp} \vdash e \le e' : И A.\,\tau\langle\mathsf{emp}\rangle]\!]$, this implies that $[\![\cdot; \mathsf{emp} \vdash \mathtt{post}\,e \le \mathtt{post}\,e' : И\xi.\,\mathsf{promise}_{\xi,A}\,\tau\langle\mathsf{Wait}(\xi, A, \mathsf{emp})\rangle]\!]$. Let us attempt such a proof (we drop the environment as well as the pre- and post-condition for simplicity).

Going by the definition of refinement, we know that $\{pre\}\ e\ \{post\}$ holds, and want to show that $\{pre'\}\ \mathtt{post}\,e\ \{post'\}$ holds, for certain $pre, post, pre', post'$. More precisely, we have (after some simplification):

$$\{p \Mapsto \mathsf{running}\colon \mathcal{C}[e']\}\ \ e\ \ \{y.\exists N', y'.[\![\tau]\!](N,y,y') * p \Mapsto \mathsf{running}\colon \mathcal{C}[y']\}$$

and want to prove

$$\{p \Mapsto \mathsf{running}\colon \mathcal{C}[\mathtt{post}\,e']\}\ \mathtt{post}\,e\ \{x.\exists N', x'.[\![\mathsf{promise}_{\xi,A}\,\tau]\!](N',x,x') * p \Mapsto \mathsf{running}\colon \mathcal{C}[y']\}$$

Without loss of generality, we first perform the post on the simulation side, finding that there is a p' such that

$$\begin{aligned}&\{p \Mapsto \mathsf{running}\colon \mathcal{C}[p'] * p' \Mapsto \mathsf{posted}\colon e'\}\\ &\mathtt{post}\,e\\ &\ \{x.\exists N'.[\![\mathsf{promise}_{\xi,A}\,\tau]\!](N',x,p') * p \Mapsto \mathsf{running}\colon \mathcal{C}[p']\}\end{aligned}$$

We can now unfold the definition of the interpretation function and simplify, finding that it is sufficient to show

$$\begin{aligned}&\{p \Mapsto \mathsf{running}\colon \mathcal{C}[p'] * p' \Mapsto \mathsf{posted}\colon e'\}\\ &\mathtt{post}\,e\\ &\ \{x.\textsc{Wait}(x, N', y.(\text{wf conds}) * \exists y'.p' \Mapsto \mathsf{running}\colon y' * [\![\tau]\!](N',y,y'))\}\end{aligned}$$

Considering the rules for `post`, this means we have to find some pre such that

$$\{pre\}\ \ e\ \ \{y.(\text{wf conds}) * \exists y'.p' \Mapsto \mathsf{done}\colon y' * [\![\tau]\!](N',y,y')\}$$

Combining this with what we know about e, this means we will, essentially, have to show that

$$p \Mapsto \mathsf{running}\colon \mathcal{C}[p'] * p' \Mapsto \mathsf{posted}\colon e' \Rrightarrow p' \Mapsto \mathsf{running}\colon e' * \ldots$$

Now, we can *almost* prove this statement: If we knew that we can schedule a task, this would follow immediately. But we have only two rules that allow scheduling a task (as can be seen from the small-step semantics):

$$p \Mapsto \mathsf{running}\colon \mathcal{C}[\texttt{wait}\, p'] * p' \Mapsto \mathsf{posted}\colon e' \Rrightarrow p' \Mapsto \mathsf{running}\colon e' * p \Mapsto \mathsf{posted}\colon \mathcal{C}[\texttt{wait}\, p']$$
$$p \Mapsto \mathsf{running}\colon p' * p' \Mapsto \mathsf{posted}\colon e' \Rrightarrow p' \Mapsto \mathsf{running}\colon e' * p \Mapsto \mathsf{done}\colon p'$$

This means that we can only prove the claim when $\mathcal{C}$ is of the form $\mathcal{C} = \mathcal{C}'[\texttt{wait}\,\bullet]$ or $\mathcal{C} = \bullet$. In the other cases, we cannot prove this claim.

What if we try a different encoding? Clearly, we need to defer the execution of e' to the point where we can schedule a task. So, let's try an alternative approach:

$$[\![\mathsf{promise}_{\xi,A}\,\tau]\!](N, x, x') := N[\xi] = (x, x') *$$
$$\textsc{Wait}\left(x, N', y.\ \begin{array}{l}(\text{wf conds}) * x' \Mapsto \mathsf{posted}\colon e' * \\ (x' \Mapsto \mathsf{running}\colon e' \Rrightarrow \exists y'.x' \Mapsto \mathsf{running}\colon y' * [\![\tau]\!](N', y, y'))\end{array}\right)$$

This approach defers the execution of the simulation-side task.

Let us try to prove the `post` case. Arguing as before, we have to show, for some p':

$$\begin{array}{l}\{p' \Mapsto \mathsf{posted}\colon e'\} \\ e \\ \{y.\exists N'.(\text{wf conds}) * p' \Mapsto \mathsf{posted}\colon e' * (x' \Mapsto \mathsf{running}\colon e' \Rrightarrow \ldots)\}\end{array}$$

Applying the frame rule, deals with the $p' \Mapsto \mathsf{posted}\colon e'$. But remember that the specification of e (see above) has a precondition $p' \Mapsto \mathsf{running}\colon e'$, which we cannot prove here — we have run into a variant of the same problem again!

Yet another approach deals with this problem by changing how we define the relation between expressions. So far, we have used triples of the following form:

$$\{pre * p \Mapsto \mathsf{running}\colon e'\}\ \ e\ \ \{x.\exists x'.p \Mapsto \mathsf{running}\colon x' * post(x, x')\}$$

Our problem was that when we tried to prove facts about `post`, the $p \Mapsto \mathsf{running}\colon e'$ in the precondition got in the way. What if we simply move it to the postcondition?

Let us define *preliminary delayed refinement* as follows: let e, e' be expressions such that $\Gamma; \eta \vdash e : \text{И}A.\,\tau\langle\eta'\rangle$ and $\Gamma; \eta \vdash e' : \text{И}A.\,\tau\langle\eta'\rangle$. Then we say that e is a preliminary delayed refinement of e' iff for all contexts $\mathcal{C}$,

$$\begin{array}{l}\{[\![\Gamma]\!](N, \sigma, \sigma') * [\![\eta]\!](N)\} \\ e\sigma \\ \left\{x.\ p \Mapsto \mathsf{running}\colon \mathcal{C}[e'\sigma'] \Rrightarrow \begin{array}{l}\exists N', x'.N \equiv_{\overline{A}} N' * \\ [\![\eta']\!](N') * [\![\tau]\!](N', x, x') * p \Mapsto \mathsf{running}\colon \mathcal{C}[x']\end{array}\right\}\end{array}$$

Let us write $[\![\Gamma; \eta \vdash e \leq_{\blacklozenge} e' : \text{И}A.\, \tau\langle\eta'\rangle]\!]$ for this. The difference to the original approach is that we move the $p \Mapsto \mathsf{running}\colon \mathcal{C}[e'\sigma']$ — instead of a disjoint conjunct in the precondition, it is now a hypothesis in the postcondition. Apart from that, the triple remains unchanged.

As it turns out, this is almost what we need: For trivial preconditions, everything goes through. In particular, we can prove the closure lemmas for `post` and `wait`. Sadly, things start breaking at another point: suppose we have that

$$[\![\cdot; \mathsf{emp} \vdash \texttt{ref 1} \leq_{\blacklozenge} \texttt{ref 1} : \text{И}\{\xi\}.\, \mathsf{ref}_\xi\, \langle \xi \mapsto \texttt{int}\rangle]\!]$$
$$[\![x : \mathsf{ref}_\xi\, ; \xi \mapsto \texttt{int} \vdash \texttt{!}\, x \leq_{\blacklozenge} \texttt{!}\, x : \text{И}\emptyset.\, \texttt{int}\langle \xi \mapsto \texttt{int}\rangle]\!]$$

Clearly, this should imply that

$$[\![\cdot; \mathsf{emp} \vdash \texttt{! (ref 1)} \leq_{\blacklozenge} \texttt{! (ref 1)} : \text{И}\{\xi\}.\, \texttt{int}\langle \xi \mapsto \texttt{int}\rangle]\!]$$

The trouble is that with preliminary delayed refinement, this is not provable. Roughly, we have to show: Suppose the following holds:

$$\{\phi_1\}\ \ e_1\ \ \{p_1 \Rrightarrow \phi_2 * p_2\}$$
$$\{\phi_2\}\ \ e_2\ \ \{p_2 \Rrightarrow \phi_3 * p_3\}$$

Then we have that

$$\{\phi_1\}\ \ e_1; e_2\ \ \{p_1 \Rrightarrow \phi_3 * p_3\}$$

Here ϕ_i describes a formula that describes a pair of states, while p_i is a formula of the form $p \Mapsto \mathsf{running}\colon e_i$. Note that the specification of e_1 only gives us a conditional instance of ϕ_2 (i.e., we must first show that p_1 holds), while the triple for e_2 needs an unconditional ϕ_2. On a high level, the deeper reason why this fails is that the relational interpretation forces two values or states to be related *at the same time*, while the task predicates are now able to hold *at different points in time*. This means that our approach does not quite work out. To solve this issue, we find that the right solution is to "split the interpretations": Instead of a single relational interpretation $[\![\tau]\!](N, x, y)$, we a pair of interpretations $[\![\tau]\!]_I(N, d, x)$ and $[\![\tau]\!]_S(N, d, y)$ that, together, yield a relational interpretation. We present the details of this construction in the following section.

4.2.3. Delayed refinement

The goal of this section is to define delayed refinement. To understand the idea of delayed refinement, we first need the idea of relatedness. We say that two values/states are "related" when they are indistinguishable using program operations: Two base values are related if they are equal, two tasks are related if they have related outcomes, two references are related if the corresponding heap cells contain related data. This notion will be made precise over the course of this section.

We will define delayed refinement to express the following idea: Let e_1 and e_2 both be typed with the same type, i.e., $\Gamma; \eta \vdash e_1 : \text{И}A.\, \tau\langle\eta'\rangle$ and $\Gamma; \eta \vdash e_2 : \text{И}A.\, \tau\langle\eta'\rangle$. Then we can show:

Given a configuration s_1 be given that satisfies η. For every execution starting from s, reducing to value v_1 in configuration s_1', we have:

- v_1 has type τ;
- s_1' satisfies η';
- for any configuration s_2 that satisfies η and is "related" to s_1, there is an execution of e_2 that reduces to value v_2 in configuration s_2' such that:
  - v_2 has type τ and is "related" to v_1;
  - s_2' satisfies η' and is "related" to s_1'.

We will define delayed refinement step by step. As a first step, we supplement the Hoare triples from the above by *existential Hoare triples* $\langle\phi\rangle\, e\, \langle x.\ \phi'(x)\rangle$, with the following semantics: suppose we are in a configuration satisfying ϕ. Then there *exists* an execution of e that reduces e to a value v such that the final configuration satisfies $\phi'(v)$. To avoid ambiguities, we will call triples of the form $\{\phi\}\ e\ \{x.\ \phi'(x)\}$ *universal Hoare triples*.

We define existential Hoare triples using task predicates. Let $\phi, e, \phi'(x)$ be given, where ϕ and $\phi'(x)$ can be arbitrary formulas, and let p, v be fresh variables. Then an existential triple $\langle\phi\rangle\, e\, \langle x.\ \phi'(x)\rangle$ is a macro for the following:

$$\forall p, \mathcal{C}.\ \phi * p \Mapsto \mathsf{running}{:}\,\mathcal{C}[e] \Rrightarrow \exists (v : \mathsf{val}).\ \phi'(v) * p \Mapsto \mathsf{running}{:}\,\mathcal{C}[v].$$

To define delayed refinement, we again provide an interpretation of types and resource expressions. Since we just saw that the standard approach of defining relational interpretations functions does not work easily in our setting, we instead provide two families of interpretation functions, the *implementation interpretations* and the *specification interpretations*.

Implementation interpretations are given by $[\![\tau]\!]_I(d, N_I, x)$, $[\![\eta]\!]_I(D, N_I)$, $[\![\Gamma]\!]_I(D, N_I, \sigma)$ etc., and specification interpretations by $[\![\tau]\!]_S(D, N_S, x)$ and so on. The parameters N, x and σ have the same meaning as for the unary interpretation above (although we use two separate name maps, N_I and N_S, for the implementation and specification side). The parameters d and D are new: d has type val, and D is a a map of type T_D; they give the *connection data*. The role of the connection data is to connect implementation and specification interpretations together to ensure relatedness.

We again use an implicit typing convention: N, N_I, N_S, N' and other variants have type T_N; D, D', D_i and so on have type T_D; and all variables with an implicit type so far have type val.

The interpretation functions are given in Fig. 4.8. The first thing to note is that $[\![\ldots]\!]_I$ is always an extension of $[\![\ldots]\!]_T$, and the same is true most cases of $[\![\ldots]\!]_S$: The parts of the figure written in black are (almost) identical to the unary case, while the parts in blue correspond changed or new features.

We start again by discussing the interpretation of `bool`. Apart from the already-discussed part, $x \doteq \texttt{true} \lor x \doteq \texttt{false}$, we now also have a conjunct $x \doteq d$. This conjunct allows us to prove relatedness: Two Boolean values are related if they are equal. Note that $(\exists d.\ [\![\texttt{bool}]\!]_I(M, d, N_I, x_I) * [\![\texttt{bool}]\!]_S(M, d, N_S, x_S)) \equiv (x_I \doteq \texttt{true} \lor x_I \doteq \texttt{false}) \land x_I \doteq x_S$. This gives exactly the expected binary interpretation of Boolean types. The interpretation of unit is similar, and the cases emp and $\eta_1 * \eta_2$ are as above.

Example 1 To see how this connects with delayed simulation, consider the following: We wish to prove that `true` is a delayed refinement of `true`. The corresponding logical statement would then be (ignoring some technicalities):

$$\{\top\}\ \texttt{true}\ \{x.\ \exists N_I, d.\ [\![\texttt{bool}]\!]_I(d, N_I, x) * \langle\top\rangle\, \texttt{true}\, \langle y.\ \exists N_S.\ [\![\texttt{bool}]\!]_S(d, N_S, y)\rangle\}$$

Unfolding the interpretation function and writing $\phi(x)$ for $x \doteq \texttt{true} \vee x \doteq \texttt{false}$, we must prove:

$$\{\top\}\ \texttt{true}\ \{x.\ \exists N_I, d.\ \phi(x) \wedge x \doteq d * \langle\top\rangle\, \texttt{true}\, \langle y.\ \exists N_S.\ \phi(y) \wedge y \doteq d\rangle\}\,.$$

As can be seen in the example, we use the connection parameter to connect interpretations that are on different sides of Hoare triples.

The interpretation of ref_ξ and $\xi \mapsto \tau$ is also similar to the unary case: The only difference is that $[\![\xi \mapsto \tau]\!]_{\ldots}(D, N)$ now has to produce a connection parameter for the interpretation of τ. This is done by looking up the correct parameter in the D map, using $D.\xi$.

At this point, it remains to discuss the interpretation of promises and wait permissions. We first define convenience notations for common patterns: $[\![\Gamma; \eta]\!]_I$, $[\![\Gamma; \eta]\!]_S$, $[\![\tau; \eta]\!]_I$ and $[\![\tau; \eta]\!]_S$ describe joint interpretations. For instance, $[\![\tau; \eta]\!]_I(D, N, x)$ is defined as $[\![\tau]\!]_I(D.\boxdot, N, x) * [\![\eta]\!]_I(D, N)$: We interpret τ and η with the same name map N and the appropriate parts of the connection data map D.

One key component of the interpretation of promises is a way to capture the behavior of a task. For this, we define an existential analogue to $[\![\Gamma; \eta \vdash e : \mathsf{M}A.\, \tau\langle\eta'\rangle]\!]_T$, namely $[\![\Gamma; \eta \vdash e : \mathsf{M}A.\, \tau\langle\eta'\rangle]\!]_S$, called the *saved continuation* predicate. It expresses the fact that the expression e has an execution that fits its type. We will in general only try to prove a saved continuation predicate if we know that e is typed as $\Gamma; \eta \vdash e : \mathsf{M}A.\, \tau\langle\eta'\rangle$.

Concretely, the definition says, for any name map N and variable substitution σ: Suppose $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$ and $x_i\sigma$ has type τ_i (semantically, i.e., $[\![\tau_i]\!]_S(D.x_i, N, x_i\sigma)$ for all i). Also suppose that we start from a configuration such that the specification heap and specification task buffer satisfy η (i.e., $[\![\eta]\!]_S(D, N)$). Then there is an execution of e that reduces to a value v such that v has type τ (i.e., $[\![\tau]\!]_S(D'.\boxdot, N', v)$) and the specification heap and task buffer of the final configuration satisfy η' (i.e., $[\![\eta']\!]_S(D', N')$) for some name map N' that satisfies the name persistence condition with N. The D and D' connection data are used to construct relatedness with the specification side.

With this, we come to the interpretation of promises and wait permissions. We start with the two simple cases. The specification interpretation of $\mathsf{promise}_{\xi,A}\,\tau$ consists of two parts. The first part asserts that x coincides with $N.\xi$, just like the interpretations of ref_ξ. The second part ensures the existence of a name map N_{pre} with certain properties. Intuitively, N_{pre} is the name map at the time the task $N.\xi$ was started. The conditions state that N_{pre} is associated with the (simulation-side) task x and overlaps with N on $\mathsf{names}(\tau) \setminus A$, i.e., the names occurring in τ that are not freshly allocated. This is a simulation-side version of the map overlap property that we already saw for the unary interpretation of promises.

$$
\begin{aligned}
[\![\mathsf{bool}]\!]_I(d,N,x) &= [\![\mathsf{bool}]\!]_S(d,N,x) = x \doteq d \wedge (x \doteq \mathtt{true} \vee x \doteq \mathtt{false}) \\
[\![\mathsf{unit}]\!]_I(d,N,x) &= [\![\mathsf{unit}]\!]_S(d,N,x) = x \doteq () \\
[\![\mathsf{ref}_\xi\,]\!]_I(d,N,x) &= [\![\mathsf{ref}_\xi\,]\!]_S(d,N,x) = N.\xi \doteq x \\
[\![\mathsf{promise}_{\xi,A}\,\tau]\!]_I(d,N,x) &= N.\xi \doteq x \wedge \textsc{Wait}\left(x; y.\ \begin{array}{l} \exists e, \Gamma, \eta, \eta', D_{pre}, D', N'. \\ \mathsf{Ag}(x, e, \Gamma, \eta, \eta', A, D_{pre}) * x \rightarrowtriangle_I (D', N') * \\ N' \equiv_{\mathsf{names}(\tau)\setminus A} N * [\![\tau]\!]_I(D'.\boxdot, N', y) * \\ [\![\Gamma; \eta \vdash e : \text{И}A.\, \tau\langle\eta'\rangle]\!]_S(D_{pre}, D') \end{array}\right) \\
[\![\mathsf{promise}_{\xi,A}\,\tau]\!]_S(d,N,x) &= N.\xi \doteq x \wedge \exists N_{pre}.\ x \rightarrowtriangle_S N_{pre} \wedge N \equiv_{\mathsf{names}(\tau)\setminus A} N_{pre}
\end{aligned}
$$

$$
\begin{aligned}
[\![\mathsf{emp}]\!]_I(D,N) &= [\![\mathsf{emp}]\!]_S(D,N) = \top \\
[\![\eta * \eta']\!]_I(D,N) &= [\![\eta]\!]_I(D,N) * [\![\eta']\!]_I(D,N) \\
[\![\eta * \eta']\!]_S(D,N) &= [\![\eta]\!]_S(D,N) * [\![\eta']\!]_S(D,N) \\
[\![\xi \mapsto \tau]\!]_I(D,N) &= \exists v.\ N.\xi \mapsto_I v * [\![\tau]\!]_I(D.\xi, N, v) \\
[\![\xi \mapsto \tau]\!]_S(D,N) &= \exists v.\ N.\xi \mapsto_S v * [\![\tau]\!]_S(D.\xi, N, v) \\
[\![\mathsf{Wait}(\xi, A, \eta)]\!]_I(D,N) &= N.\xi \doteq D.\xi * \textsc{Wait}\left(N.\xi; y.\ \begin{array}{l} \exists D', N'.\ N.\xi \rightarrowtriangle_I (D', N') * \\ [\![\eta]\!]_I(D', N') * N \equiv_{\mathsf{names}(\eta)\setminus A} N' \end{array}\right) \\
[\![\mathsf{Wait}(\xi, A, \eta)]\!]_S(D,N) &= \begin{array}{l} \exists e, \Gamma, \eta_{pre}, D_{pre}, N_{pre}, \sigma. \\ \mathsf{Ag}(D.\xi, e, \Gamma, \eta_{pre}, \eta, A, D_{pre}) * N.\xi \Mapsto \mathsf{posted}: e * \\ [\![\Gamma]\!]_S(D_{pre}, N_{pre}, \sigma) * [\![\eta_{pre}]\!]_S(D_{pre}, N_{pre}) * \\ N.\xi \rightarrowtriangle_S N_{pre} * N_{pre} \equiv_{\mathsf{names}(\eta)\setminus A} N \end{array}
\end{aligned}
$$

$$
\begin{aligned}
[\![x_1 : \tau_1, \ldots, x_n : \tau_n]\!]_I(D,N,\sigma) &= \circledast_{i=1}^{n} [\![\tau_i]\!]_I(D.x_i, N, x_i\sigma) \\
[\![x_1 : \tau_1, \ldots, x_n : \tau_n]\!]_S(D,N,\sigma) &= \circledast_{i=1}^{n} [\![\tau_i]\!]_S(D.x_i, N, x_i\sigma)
\end{aligned}
$$

$$
\begin{aligned}
[\![\tau; \eta]\!]_I(D,N,x) &:= [\![\tau]\!]_I(D.\boxdot, N, x) * [\![\eta]\!]_I(D,N) \\
[\![\tau; \eta]\!]_S(D,N,x) &:= [\![\tau]\!]_S(D.\boxdot, N, x) * [\![\eta]\!]_S(D,N) \\
[\![\Gamma; \eta]\!]_I(D,N,\sigma) &:= [\![\Gamma]\!]_I(D,N,\sigma) * [\![\eta]\!]_I(D,N) \\
[\![\Gamma; \eta]\!]_S(D,N,\sigma) &:= [\![\Gamma]\!]_S(D,N,\sigma) * [\![\eta]\!]_S(D,N)
\end{aligned}
$$

$$
\begin{aligned}
&[\![\Gamma; \eta \vdash e : \text{И}A.\, \tau\langle\eta'\rangle]\!]_S(D,D') = \\
&\quad \forall N, \sigma.\ \langle [\![\Gamma; \eta]\!]_S(D,N,\sigma)\rangle\, e\sigma\, \langle v.\ \exists N'.\ N \equiv_{\overline{A}} N' * [\![\tau; \eta']\!]_S(D', N', v)\rangle
\end{aligned}
$$

Figure 4.8.: Translation of types and resource expressions.

The implementation interpretation of wait permissions is almost exactly the same as $[\![\mathsf{Wait}(\xi, A, \eta)]\!]_T$: The wait permission part only adds connection data where required. Note that since we get D' and N' from task data we know that we will be using the same D' and N' in the implementation interpretation of $\mathsf{promise}_{\xi,A}\,\tau$. The conjunct $N.\xi \doteq D.\xi$ has a technical reason: We need to know the task handle of the implementation-side task in the specification interpretation of the wait permission (the details will be explained below), which is achieved by putting it into the connection data.

The implementation interpretation of $\mathsf{promise}_{\xi,A}\,\tau$ and the specification interpretation of $\mathsf{Wait}(\xi, A, \eta')$ are best considered in tandem. For the sake of discussion, let D, N be fixed. We may assume that $\Gamma; \eta \vdash e : \text{И}A.\,\tau\langle\eta'\rangle$.

First, note that the interpretation of $\mathsf{promise}_{\xi,A}\,\tau$ contains familiar parts from the unary case; they have exactly the same function as before. With the parts that are adapted from before, we find that $[\![\tau]\!]_I$ now needs a connection parameter; to get to this parameter, we use the implementation task data map's first component, the connection map D', and extract the correct connection parameter, $D'.\boxdot$, from it. This also explains the change from $x \twoheadrightarrow N'$ to $x \twoheadrightarrow_I (D', N')$ — we need to agree on the connection data in addition to the name map.

The interpretation of $\mathsf{promise}_{\xi,A}\,\tau$ also contains a completely new part. Suppose A, τ and D' are given. Then the sub-expression

$$\exists e, \Gamma, \eta_{pre}, \eta', D_{pre}.\ \mathsf{Ag}(x, e, \Gamma, \eta_{pre}, \eta', A, D_{pre}) * [\![\Gamma; \eta_{pre} \vdash e : \text{И}A.\,\tau\langle\eta'\rangle]\!]_S(D_{pre}, D')$$

of the wait permission is new.

The existential quantifier and Ag conjunct state that we can find an expression e, which contains the expression that the simulation side task executes, various typing parameters (Γ, η, η') such that $\Gamma; \eta \vdash e : \text{И}A.\,\tau\langle\eta'\rangle$ and a connection data map D that corresponds to the connection data at the time of posting the simulation side task. Note that the agreement also ensures that A is fixed for this task. We use the same agreement predicate in the simulation-side interpretation of wait permissions to ensure that all this data matches up between the two sides.

The other conjunct, $[\![\Gamma; \eta \vdash e : \text{И}A.\,\tau\langle\eta'\rangle]\!]_S(D, D')$, states: Suppose we have a variable instantiation σ that satisfies $[\![\Gamma]\!]_S(N, D_{pre}, \sigma)$ (for some N) and that the precondition $[\![\eta]\!]_S(N, D_{pre})$ holds. Suppose furthermore that a task executing $e\sigma$ is scheduled. Then there is an execution of $e\sigma$ that reduces $e\sigma$ to y, such that for some suitable N', we have that $[\![\tau]\!]_S(N', D', y)$ holds and the final state matches $[\![\eta']\!]_S(N', D')$. In a sense, this conjunct provides a summary of the behavior of e: We know we can always find a suitable execution as long as the preconditions are satisfied.

As we have seen in the previous section, this conjunct only gives us half of the story: We also need a proof that we have a task that is runnable and actually executes e. The specification interpretation of $\mathsf{Wait}(\xi, A, \eta)$ gives us all of that, except for the fact that the task has been scheduled.

Let us first look at the core part of the interpretation:

$$N.\xi \Mapsto \mathsf{posted}\colon e\sigma * [\![\Gamma]\!]_S(D_{pre}, N_{pre}, \sigma) * [\![\eta_{pre}]\!]_S(D_{pre}, N_{pre})$$

What this states is that specification task $N.\xi$ is runnable, executing the expression $e\sigma$, where $[\![\Gamma]\!]_S(N_{pre}, D_{pre}, \sigma)$ and $[\![\eta_{pre}]\!]_S(N_{pre}, D_{pre})$ hold. Comparing this to the implementation interpretation of promises, this gives us all the preconditions that we need to execute $e\sigma$ using the summary we have in the wait permissions, except for the fact that we need to schedule task $N.\xi$. In other words, as soon as we can prove that the implementation side task has terminated and that we can schedule a task on the specification side, we can, in fact, prove that the specification side task executes as expected. We will use an argument along these lines to prove an important case of Lemma 5 below.

The rest of the interpretation provides the technical infrastructure to connect everything together. Notably, the agreement predicate $\mathsf{Ag}(D.\xi, e, \Gamma, \eta_{pre}, \eta, A, D_{pre})$ enforces multiple things:

- e, Γ, η_{pre}, A and D_{pre} are the same as the corresponding values in the implementation interpretation of promises. Also, A is correctly reflected in the wait permission.
- The η' in the implementation interpretation of promises is actually the η from the wait permission.

Also, note that we reference task $D.\xi$; in the implementation interpretation of wait permissions, we enforced that this is $N.\xi$, i.e., the task handle of the implementation task. This gives us a unique key for the task. The role of $N.\xi \twoheadrightarrow_S N_{pre} * N_{pre} \equiv_{\mathsf{names}(\eta)\setminus A} N$ has the same role as the corresponding part of the specification interpretation of promises.

At this point, we have seen the interpretation of types and resource expressions. Similar to the unary case, we can now use these interpretations to relate expressions; we do this by introducing our main new notion, *delayed refinement*.

Definition 1 (Delayed refinement) Suppose $\Gamma; \eta \vdash e_1 : \text{И}A.\, \tau\langle\eta'\rangle$ and $\Gamma; \eta \vdash e_2 : \text{И}A.\, \tau\langle\eta'\rangle$ for some $\Gamma, \eta, A, \tau, \eta'$.

We say that e_1 is a *delayed refinement* of e_2 and write $[\![\Gamma; \eta \vdash e_1 \leq_\diamond e_2 : \text{И}A.\, \tau\langle\eta'\rangle]\!]$ iff, for all D, N_I, σ:

$$\{[\![\Gamma;\eta]\!]_I(D, N_I, \sigma)\}\ \ e_1\sigma\ \left\{ x_1.\exists N_I', D'.\ \begin{array}{l} N_I \equiv_{\overline{A}} N_I' * [\![\tau;\eta']\!]_I(D', N_I', x_1) * \\ [\![\Gamma; \eta \vdash e_2 : \text{И}A.\, \tau\langle\eta'\rangle]\!]_S(D, D') \end{array} \right\}$$

At this point, observe that dropping the saved continuation gives us (almost) the unary interpretation.

Our primary goal is to show the soundness of the DontWaitForMe rewrite rules, as defined above. At this point, we can formalize our primary soundness result using delayed refinement: Given a well-typed program e, applying a DontWaitForMe rewrite rule to it yields another well-typed program e' such that e' is a delayed refinement of e. In particular, this implies that every behavior of e' can be matched by e — we do not introduce new behaviors. Formally, we want to prove:

Theorem 3 (Soundness of DontWaitForMe) *Suppose* $\Gamma; \eta \vdash e : \text{И}A.\, \tau\langle\eta'\rangle$ *and* $e \implies e'$. *Then there is a* $A' \supseteq A$ *such that* $[\![\Gamma; \eta \vdash e' \leq_\diamond e : \text{И}A'.\, \tau\langle\eta'\rangle]\!]$.

Most of the rest of this section is dedicated to the proof of this theorem. We start by building up important properties of the interpretation functions (locality and duplicability) and delayed simulation (program composition and closure properties), culminating in the fundamental lemma: All well-typed programs are related to themselves. As a side result, we also get a closure property for expression contexts. With these tools in hand, we can prove the soundness of the rewriting rules one-by-one; we show the two most interesting cases explicitly, and finally sketch the complete proof of the theorem.

4.2.4. Closure properties and the fundamental lemma

In the following, we only state our results and sketch the proofs. More detailed proof outlines can be found in the appendix, while full proofs are contained in the accompanying Coq development [ESOP-Coq].

The first property to prove is the *locality property* of interpretation functions: Suppose we interpret some type τ using two different name maps. Then the interpretations are equivalent as long as the name maps coincide on the names appearing in τ. A similar result holds for the interpretation of resource expressions and environments, also extending to connection maps.

Lemma 2 (Locality) *Let $\tau, \eta, \Gamma, d, D, N, D', N', x, \sigma$ be given.*

- *Suppose $N \equiv_{\mathsf{names}(\tau)} N'$.*
  Then $[\![\tau]\!]_I(d, N, x) \equiv [\![\tau]\!]_I(d, N', x)$ and $[\![\tau]\!]_S(d, N, x) \equiv [\![\tau]\!]_S(d, N', x)$.
- *Suppose $N \equiv_{\mathsf{names}(\eta)} N'$ and $D \equiv_{\mathsf{rnames}(\eta)} D'$.*
  Then $[\![\eta]\!]_I(D, N) \equiv [\![\eta]\!]_I(D', N')$ and $[\![\eta]\!]_S(D, N) \equiv [\![\eta]\!]_S(D', N')$.
- *Suppose $N \equiv_{\mathsf{names}(\Gamma)} N'$ and $D \equiv_{\mathrm{dom}\,\Gamma} D'$.*
  Then $[\![\Gamma]\!]_I(D, N, \sigma) \equiv [\![\Gamma]\!]_I(D', N', \sigma)$ and $[\![\Gamma]\!]_S(D, N, \sigma) \equiv [\![\Gamma]\!]_S(D', N', \sigma)$.

Proof (Sketch) By structural induction on τ and η.

The second property to prove is the *duplicability property* of some interpretations: The interpretation of types (and environments) gives formulas that do not assert ownership of any resource, and are hence duplicable in the sense that the resulting formula is idempotent for separating conjunction. This models the fact that values can be arbitrarily duplicated without changing their meaning.

Lemma 3 (Duplicability) *We say that an assertion ϕ is* duplicable *iff $\phi \equiv \phi * \phi$. The following are duplicable:*

1. *$\phi_1 \Rrightarrow \phi_2$ for arbitrary ϕ_1, ϕ_2.*
2. *$[\![\tau]\!]_I(d, N, x)$ and $[\![\tau]\!]_S(d, N, x)$.*
3. *$[\![\Gamma]\!]_I(D, N, \sigma)$ and $[\![\Gamma]\!]_S(D, N, \sigma)$.*

Proof (Sketch) The first case follows from the semantics of $\Rrightarrow$; the second case follows by induction on the structure of τ, and the third by reduction to the second case.

Now that we have established two of the main properties of the interpretation functions, we can show facts about delayed refinement. The following lemma shows a compositionality property. It corresponds to a form of the sequence rule in an imperative setting: If two programs c_1 and c_2 are related, and two other programs c_1' and c_2' are related, then $c_1; c_1'$ and $c_2; c_2'$ should be related. Since we are in a functional setting, we formulate it in terms of execution contexts: If e and e' are related, and so are $\mathcal{C}[x]$ and $\mathcal{C}'[x]$ for a variable of the right type, then $\mathcal{C}[e]$ and $\mathcal{C}'[e']$ are related.

Lemma 4 (Binding composition) *Given $\mathcal{C}, \mathcal{C}', e, e', \eta_1, \eta_2, \eta_3, \Gamma, x, \tau_1, \tau_2, A_1, A_2$ such that $[\![\Gamma; \eta_1 \vdash e \leq_\diamond e' : \text{И} A_1.\, \tau_1\langle\eta_2\rangle]\!]$ and $[\![\Gamma, A_1, x : \tau_1; \eta_2 \vdash \mathcal{C}[x] \leq_\diamond \mathcal{C}'[x] : \text{И} A_2.\, \tau_2\langle\eta_3\rangle]\!]$ hold.*
Then $[\![\Gamma; \eta_1 \vdash \mathcal{C}[e] \leq_\diamond \mathcal{C}'[e'] : \text{И} A_1, A_2.\, \tau_2\langle\eta_3\rangle]\!]$ holds.

Proof (Sketch) The key step of the proof is to use the name persistence property and duplicability to strengthen $[\![\Gamma; \eta_1 \vdash e \leq_\diamond e' : \text{И} A_1.\, \tau_1\langle\eta_2\rangle]\!]$ into

$$\{[\![\Gamma;\eta]\!]_I(D, N_I, \sigma)\}\ \ e_1\sigma\ \left\{ x_1.\exists N_I', D'.\ \begin{array}{l} N_I \equiv_{\overline{A}} N_I' * [\![\Gamma, x:\tau; \eta']\!]_I(D', N_I', \sigma[x \mapsto x_1]) * \\ \forall N_S, \sigma'. S(N_S, \sigma', D, D') \end{array} \right\}$$

with

$$S(N_S, \sigma', D, D') := \left\langle [\![\Gamma;\eta]\!]_S(D, N_S, \sigma') \right\rangle e_2\sigma' \left\langle x_2.\ \exists N_S'. N_S \equiv_{\overline{A}} N_S' * [\![\Gamma, x:\tau; \eta']\!]_S(D', N_S', \sigma'[x \mapsto x_2]) \right\rangle$$

A similar result holds for the existential Hoare triples. Using these steps, the claim follows using standard Hoare triple reasoning.

The following lemma states that delayed refinement is closed under the primitives of the core calculus, as given in Fig. 4.9.

Lemma 5 (Closure) *All the delayed refinement in Fig. 4.9 hold.*

This lemma has two interesting cases: `post` and `wait`. We sketch the main ideas:

`post` Observe that the implementation interpretation of promises and wait permissions captures the entire postcondition of the universal Hoare triple occuring in the delayed simulation $[\![\Gamma; \eta \vdash e \leq_\diamond e' : \text{И} A.\, \tau\langle\eta'\rangle]\!]$, while the specification interpretation simply states that a task executing e' has been posted.

Hence, the proof consists of two parts: Using the Hoare rule for `post` and some rewriting, we establish the implementation interpretation. For the specification side, we simply take a single step, posting a task executing e'.

`wait` Using the observations from the previous case, we first perform a wait on the implementation side. This gives us the implementation interpretation of τ and η, as well as a proof that if a task ξ has been scheduled, we can run it to completion.

C-Var

$$\frac{}{[\![x : \tau; \mathsf{emp} \vdash x \leq_\diamond x : И\emptyset.\, \tau\langle \mathsf{emp}\rangle]\!]}$$

C-Const

$$\frac{}{[\![\cdot; \mathsf{emp} \vdash c \leq_\diamond c : И\emptyset.\, ty(c)\langle \mathsf{emp}\rangle]\!]}$$

C-If

$$\frac{[\![\Gamma; \eta \vdash e_1 \leq_\diamond e_1' : И A.\, \tau\langle \eta'\rangle]\!] \qquad [\![\Gamma; \eta \vdash e_2 \leq_\diamond e_2' : И A.\, \tau\langle \eta'\rangle]\!] \qquad \Gamma(x) = \texttt{bool}}{[\![\Gamma; \eta \vdash \texttt{if}\, x\, \texttt{then}\, e_1\, \texttt{else}\, e_2 \leq_\diamond \texttt{if}\, x\, \texttt{then}\, e_1'\, \texttt{else}\, e_2' : И A.\, \tau\langle \eta'\rangle]\!]}$$

C-Let

$$\frac{[\![\Gamma, y : \tau; \eta \vdash e \leq_\diamond e' : И A.\, \tau'\langle \eta'\rangle]\!] \qquad \Gamma(x) = \tau}{[\![\Gamma; \eta \vdash \texttt{let}\, y = x\, \texttt{in}\, e \leq_\diamond \texttt{let}\, y = x\, \texttt{in}\, e' : И A.\, \tau'\langle \eta'\rangle]\!]}$$

C-Alloc

$$\frac{\xi \notin \mathsf{names}(\tau)}{[\![x : \tau; \mathsf{emp} \vdash \texttt{ref}\, x \leq_\diamond \texttt{ref}\, x : И\{\xi\}.\, \mathsf{ref}_\xi\, \langle \xi \mapsto \tau\rangle]\!]}$$

C-Read

$$\frac{}{[\![x : \mathsf{ref}_\xi\, ; \xi \mapsto \tau \vdash \texttt{!}\, x \leq_\diamond \texttt{!}\, x : И\emptyset.\, \tau\langle \xi \mapsto \tau\rangle]\!]}$$

C-Write

$$\frac{x \neq y}{[\![x : \mathsf{ref}_\xi\, , y : \tau; \xi \mapsto \tau \vdash x := y \leq_\diamond x := y : И\emptyset.\, ()\, \langle \xi \mapsto \tau\rangle]\!]}$$

C-Post

$$\frac{[\![\Gamma; \eta \vdash e \leq_\diamond e' : И A.\, \tau\langle \eta'\rangle]\!] \qquad \xi \text{ fresh}}{[\![\Gamma; \eta \vdash \texttt{post}\, e \leq_\diamond \texttt{post}\, e' : И\{\xi\}.\, \mathsf{promise}_{\xi,A}\, \tau\langle \mathsf{Wait}(\xi, A, \eta')\rangle]\!]}$$

C-Wait

$$\frac{\xi \notin \mathsf{names}(\tau) \cup \mathsf{names}(\eta)}{[\![x : \mathsf{promise}_{\xi,A}\, \tau; \mathsf{Wait}(\xi, A, \eta) \vdash \texttt{wait}\, x \leq_\diamond \texttt{wait}\, x : И A.\, \tau\langle \eta\rangle]\!]}$$

C-Frame

$$\frac{[\![\Gamma; \eta \vdash e \leq_\diamond e' : И A.\, \tau\langle \eta'\rangle]\!] \qquad \mathsf{rnames}(\eta) \cap \mathsf{rnames}(\eta_f) = \emptyset \qquad \mathsf{rnames}(\eta') \cap \mathsf{rnames}(\eta_f) = \emptyset \qquad A \cap \mathsf{names}(\eta_f) = \emptyset}{[\![\Gamma; \eta * \eta_f \vdash e \leq_\diamond e' : И A.\, \tau\langle \eta' * \eta_f\rangle]\!]}$$

C-Strengthen

$$\frac{[\![\Gamma; \eta \vdash e \leq_\diamond e' : И A.\, \tau\langle \eta'\rangle]\!] \qquad \Gamma \subseteq \Gamma' \qquad A \subseteq A'}{[\![\Gamma'; \eta \vdash e \leq_\diamond e' : И A'.\, \tau\langle \eta'\rangle]\!]}$$

Figure 4.9.: Closure facts

Now, use three facts (i) ξ has been posted, (ii) we can complete it once scheduled and (iii) the currently executing task wants to perform a wait for ξ: Use the wait to schedule to task ξ, run it, schedule back to the original task, and perform the wait on the now-completed ξ.

While the last two lemmas already tell us that delayed refinement is well-behaved, we can push this even further. In particular, we can prove that well-typed programs and contexts are particularly nicely behaved when it comes to delayed refinement.

The first lemma states a reflexivity property: A well-typed program is a delayed refinement of itself. While this property may seem trivial, note it is not totally obvious even for the simple program $\texttt{wait}\, x$: we really need the fact that x describes tasks with related behavior on the implementation and simulation side. Once higher-order functions are added to the language, this property turns out to be non-trivial, as discussed in the literature on logical relations.

Lemma 6 (Fundamental lemma) *Delayed simulation is reflexive for well-typed expressions:* $\Gamma; \eta \vdash e : \text{И}A.\, \tau\langle\eta'\rangle$ *implies* $[\![\Gamma; \eta \vdash e \leq_\diamond e : \text{И}A.\, \tau\langle\eta'\rangle]\!]$.

Proof (Sketch) By induction on the typing derivation. Use Lemma 4 to reduce to the cases in Lemma 5; each typing rule has a corresponding case in Fig. 4.9.

The second lemma extends Lemma 5 to entire expression contexts $\mathcal{E}$: If e and e' are related, so are $\mathcal{E}[e]$ and $\mathcal{E}[e']$, as long as certain typing constraints are satisfied.

As a first step of this lemma, we need to define how we can ascribe types to expression contexts. For a given expression context $\mathcal{E}$, we assign a "hole type" and a "resultant type". If there is an expression e of hole type, $\mathcal{E}[e]$ will have the resultant type. More precisely, we write $\Gamma; \eta \vdash \bullet : \text{И}A.\, \tau\langle\eta'\rangle \rightsquigarrow \bar{\Gamma}; \bar{\eta} \vdash \mathcal{E} : \text{И}\bar{A}.\, \bar{\tau}\langle\bar{\eta}'\rangle$ to say that the hole type is $\Gamma; \eta \vdash \bullet : \text{И}A.\, \tau\langle\eta'\rangle$ and the resultant type is $\bar{\Gamma}; \bar{\eta} \vdash \mathcal{E} : \text{И}\bar{A}.\, \bar{\tau}\langle\bar{\eta}'\rangle$.

With this, we can make the statement from above precise.

Lemma 7 (Expression contexts) *Suppose* $\Gamma \vdash \{\eta\}\ \ e_1 \leq_\diamond e_2 : \tau\ \ \{\eta'\}$ *and* $\Gamma; \eta \vdash \bullet : \text{И}A.\, \tau\langle\eta'\rangle \rightsquigarrow \bar{\Gamma}; \bar{\eta} \vdash \mathcal{E} : \text{И}\bar{A}.\, \bar{\tau}\langle\bar{\eta}'\rangle$.

Then $\bar{\Gamma} \vdash \{\bar{\eta}\}\ \ \mathcal{E}[e_1] \leq_\diamond \mathcal{E}[e_2] : \bar{\tau}\ \ \{\bar{\eta}'\}$.

Proof (Sketch) By induction on the typing derivation for $\mathcal{E}$. The proof steps are essentially the same as for Lemma 6, except that the $\mathcal{E} = \bullet$ case follows from the assumptions.

4.2.5. Soundness of DontWaitForMe

We now come to the soundness proof of DontWaitForMe. We will show two interesting cases. The R-Asynchronize rule is our main rule for introducing concurrent behavior. Notably, it has an extremely simple proof.

Lemma 8 (R-Asynchronize is sound) *Suppose* $\Gamma; \eta \vdash e : \text{И}A.\, \tau\langle\eta'\rangle$. *Then for some fresh* ξ, $[\![\Gamma; \eta \vdash \texttt{wait (post}\, e\texttt{)} \leq_\diamond e : \text{И}A, \xi.\, \tau\langle\eta'\rangle]\!]$.

$$\text{ET-Hole}\quad \frac{}{\Gamma;\eta \vdash \bullet : И A.\,\tau\langle\eta'\rangle \rightsquigarrow \Gamma;\eta \vdash \mathcal{E} : И A.\,\tau\langle\eta'\rangle}$$

$$\frac{\bar{\Gamma};\bar{\eta} \vdash \bullet : И\bar{A}.\,\bar{\tau}\langle\bar{\eta}'\rangle \rightsquigarrow \Gamma;\eta_1 \vdash e_1 : И A_1.\,\mathtt{bool}\langle\eta_2\rangle \quad \Gamma, A_1;\eta_2 \vdash e_2 : И A_2.\,\tau\langle\eta_3\rangle \quad \Gamma, A_1;\eta_2 \vdash e_3 : И A_2.\,\tau\langle\eta_3\rangle}{\bar{\Gamma};\bar{\eta} \vdash \bullet : И\bar{A}.\,\bar{\tau}\langle\bar{\eta}'\rangle \rightsquigarrow \Gamma;\eta_1 \vdash \mathtt{if}\, e_1\, \mathtt{then}\, e_2\, \mathtt{else}\, e_3 : И A_1, A_2.\,\tau\langle\eta_3\rangle}$$

$$\frac{\Gamma;\eta_1 \vdash e_1 : И A_1.\,\mathtt{bool}\langle\eta_2\rangle \quad \bar{\Gamma};\bar{\eta} \vdash \bullet : И\bar{A}.\,\bar{\tau}\langle\bar{\eta}'\rangle \rightsquigarrow \Gamma, A_1;\eta_2 \vdash e_2 : И A_2.\,\tau\langle\eta_3\rangle \quad \Gamma, A_1;\eta_2 \vdash e_3 : И A_2.\,\tau\langle\eta_3\rangle}{\bar{\Gamma};\bar{\eta} \vdash \bullet : И\bar{A}.\,\bar{\tau}\langle\bar{\eta}'\rangle \rightsquigarrow \Gamma;\eta_1 \vdash \mathtt{if}\, e_1\, \mathtt{then}\, e_2\, \mathtt{else}\, e_3 : И A_1, A_2.\,\tau\langle\eta_3\rangle}$$

$$\frac{\Gamma;\eta_1 \vdash e_1 : И A_1.\,\mathtt{bool}\langle\eta_2\rangle \quad \Gamma, A_1;\eta_2 \vdash e_2 : И A_2.\,\tau\langle\eta_3\rangle \quad \bar{\Gamma};\bar{\eta} \vdash \bullet : И\bar{A}.\,\bar{\tau}\langle\bar{\eta}'\rangle \rightsquigarrow \Gamma, A_1;\eta_2 \vdash e_3 : И A_2.\,\tau\langle\eta_3\rangle}{\bar{\Gamma};\bar{\eta} \vdash \bullet : И\bar{A}.\,\bar{\tau}\langle\bar{\eta}'\rangle \rightsquigarrow \Gamma;\eta_1 \vdash \mathtt{if}\, e_1\, \mathtt{then}\, e_2\, \mathtt{else}\, e_3 : И A_1, A_2.\,\tau\langle\eta_3\rangle}$$

$$\frac{\bar{\Gamma};\bar{\eta} \vdash \bullet : И\bar{A}.\,\bar{\tau}\langle\bar{\eta}'\rangle \rightsquigarrow \Gamma;\eta_1 \vdash e : И A.\,\tau\langle\eta_2\rangle \quad \xi \text{ fresh}}{\bar{\Gamma};\bar{\eta} \vdash \bullet : И\bar{A}.\,\bar{\tau}\langle\bar{\eta}'\rangle \rightsquigarrow \Gamma;\eta_1 \vdash \mathtt{ref}\, e : И A, \xi.\,\mathsf{ref}_\xi\,\langle\eta_2 * \xi \mapsto \tau\rangle}$$

$$\frac{\bar{\Gamma};\bar{\eta} \vdash \bullet : И\bar{A}.\,\bar{\tau}\langle\bar{\eta}'\rangle \rightsquigarrow \Gamma;\eta_1 \vdash e : И A.\,\mathsf{ref}_\xi\,\langle\eta_2 * \xi \mapsto \tau\rangle}{\bar{\Gamma};\bar{\eta} \vdash \bullet : И\bar{A}.\,\bar{\tau}\langle\bar{\eta}'\rangle \rightsquigarrow \Gamma;\eta_1 \vdash \mathtt{!}\, e : И A.\,\tau\langle\eta_2 * \xi \mapsto \tau\rangle}$$

$$\frac{\bar{\Gamma};\bar{\eta} \vdash \bullet : И\bar{A}.\,\bar{\tau}\langle\bar{\eta}'\rangle \rightsquigarrow \Gamma;\eta_1 \vdash e_1 : И A_1.\,\mathsf{ref}_\xi\,\langle\eta_2\rangle \quad \Gamma, A_1;\eta_2 \vdash e_2 : И A_2.\,\tau\langle\eta_3 * \xi \mapsto \tau\rangle}{\bar{\Gamma};\bar{\eta} \vdash \bullet : И\bar{A}.\,\bar{\tau}\langle\bar{\eta}'\rangle \rightsquigarrow \Gamma;\eta_1 \vdash e_1 \mathtt{:=} e_2 : И A_1, A_2.\,\mathsf{unit}\langle\eta_3 * \xi \mapsto \tau\rangle}$$

$$\frac{\Gamma;\eta_1 \vdash e_1 : И A_1.\,\mathsf{ref}_\xi\,\langle\eta_2\rangle \quad \bar{\Gamma};\bar{\eta} \vdash \bullet : И\bar{A}.\,\bar{\tau}\langle\bar{\eta}'\rangle \rightsquigarrow \Gamma, A_1;\eta_2 \vdash e_2 : И A_2.\,\tau\langle\eta_3 * \xi \mapsto \tau\rangle}{\bar{\Gamma};\bar{\eta} \vdash \bullet : И\bar{A}.\,\bar{\tau}\langle\bar{\eta}'\rangle \rightsquigarrow \Gamma;\eta_1 \vdash e_1 \mathtt{:=} e_2 : И A_1, A_2.\,\mathsf{unit}\langle\eta_3 * \xi \mapsto \tau\rangle}$$

$$\frac{\bar{\Gamma};\bar{\eta} \vdash \bullet : И\bar{A}.\,\bar{\tau}\langle\bar{\eta}'\rangle \rightsquigarrow \Gamma;\eta_1 \vdash e_1 : И A_1.\,\tau_1\langle\eta_2\rangle \quad \Gamma, A_1, x : \tau_1;\eta_2 \vdash e_2 : И A_2.\,\tau_2\langle\eta_2\rangle}{\Gamma;\eta_1 \vdash \mathtt{let}\, x \mathtt{=} e_1\, \mathtt{in}\, e_2 : И A_1, A_2.\,\tau_2\langle\eta_3\rangle}$$

$$\frac{\Gamma;\eta_1 \vdash e_1 : И A_1.\,\tau_1\langle\eta_2\rangle \quad \bar{\Gamma};\bar{\eta} \vdash \bullet : И\bar{A}.\,\bar{\tau}\langle\bar{\eta}'\rangle \rightsquigarrow \Gamma, A_1, x : \tau_1;\eta_2 \vdash e_2 : И A_2.\,\tau_2\langle\eta_2\rangle}{\Gamma;\eta_1 \vdash \mathtt{let}\, x \mathtt{=} e_1\, \mathtt{in}\, e_2 : И A_1, A_2.\,\tau_2\langle\eta_3\rangle}$$

$$\frac{\bar{\Gamma};\bar{\eta} \vdash \bullet : И\bar{A}.\,\bar{\tau}\langle\bar{\eta}'\rangle \rightsquigarrow \Gamma;\eta \vdash e : И A.\,\tau\langle\eta'\rangle \quad \xi \text{ fresh}}{\Gamma;\eta \vdash \mathsf{post}\, e : И\{\xi\}.\,\mathsf{promise}_{\xi,A}\,\tau\langle\mathsf{Wait}(\xi, A, \eta')\rangle}$$

$$\frac{\bar{\Gamma};\bar{\eta} \vdash \bullet : И\bar{A}.\,\bar{\tau}\langle\bar{\eta}'\rangle \rightsquigarrow \Gamma;\eta \vdash e : И A_1.\,\mathsf{promise}_{\xi,A_2}\,\tau\langle\eta' * \mathsf{Wait}(\xi, A_2, \eta'')\rangle \quad A_1 \cap A_2 = \emptyset \quad \xi \notin \mathsf{names}(\tau) \cup \mathsf{names}(\eta')}{\Gamma;\eta \vdash \mathtt{wait}\, e : И A_1, A_2.\,\tau\langle\eta' * \eta''\rangle}$$

$$\text{ET-Frame}\quad \frac{\bar{\Gamma};\bar{\eta} \vdash \bullet : И\bar{A}.\,\bar{\tau}\langle\bar{\eta}'\rangle \rightsquigarrow \Gamma;\eta \vdash e : И A.\,\tau\langle\eta'\rangle \quad \mathsf{rnames}(\eta) \cap \mathsf{rnames}(\eta_f) = \emptyset \quad \mathsf{rnames}(\eta') \cap \mathsf{rnames}(\eta_f) = \emptyset \quad A \cap \mathsf{names}(\eta_f) = \emptyset}{\Gamma;\eta * \eta_f \vdash e : И A.\,\tau\langle\eta' * \eta_f\rangle}$$

$$\text{ET-WeakenM}\quad \frac{\bar{\Gamma};\bar{\eta} \vdash \bullet : И\bar{A}.\,\bar{\tau}\langle\bar{\eta}'\rangle \rightsquigarrow \Gamma;\eta \vdash e : И A.\,\tau\langle\eta'\rangle \quad \Gamma \subseteq \Gamma' \quad A \subseteq A'}{\Gamma';\eta \vdash e : И A'.\,\tau\langle\eta'\rangle}$$

Figure 4.10.: Context typing rules, extending the typing rules to contexts.

Proof (Sketch) If we know that $\{\phi\}\ e\ \{x.\phi'(x)\}$, we can immediately derive that $\{\phi\}\ \texttt{wait(post}\, e)\ \{x.\phi'(x)\}$ by the semantics of `wait` and `post`.

Using this fact, it suffices to show $[\![\Gamma; \eta \vdash e \leq_\diamond e : \text{И}A.\, \tau\langle\eta'\rangle]\!]$, which follows from the fundamental lemma (Lemma 6).

The R-Commute rule is interesting because of its side conditions and its powerful statement.

Lemma 9 (R-Commute is sound) *Suppose* $\Gamma; \eta_1 \vdash e_1 : \text{И}A_1.\, \tau_1\langle\eta_1'\rangle$, $\Gamma; \eta_2 \vdash e_2 : \text{И}A_2.\, \tau_2\langle\eta_2'\rangle$ *and* $\Gamma, A_1, A_2, x_1 : \tau_1, x_2 : \tau_2; \eta_1' * \eta_2' \vdash e : \text{И}A.\, \tau\langle\eta'\rangle$ *such that the side conditions of* R-Commute *hold. Then*

$$[\![\Gamma; \eta_1 * \eta_2 \vdash \begin{array}{l}\texttt{let}\, x_1 = e_1\, \texttt{in} \\ \texttt{let}\, x_2 = e_2\, \texttt{in} \\ e_3\end{array} \leq_\diamond \begin{array}{l}\texttt{let}\, x_2 = e_2\, \texttt{in} \\ \texttt{let}\, x_1 = e_1\, \texttt{in} \\ e_3\end{array} : \text{И}A_1, A_2, A.\, \tau\langle\eta'\rangle]\!]$$

Proof (Sketch) Using the specifications of e_2, e_1, and e_3, together with the frame rule, we establish the implementation interpretation of the postcondition of the refinement, with a sequence D_1, D_2, D_3, D_4 of connection data and proofs $S_2(D_1, D_2)$, $S_1(D_2, D_3)$, $S_3(D_3, D_4)$ that show we can execute e_2, e_1 and e_3 using the given connection data. Next, we use locality and the side conditions of R-Commute to prove that there is a D_2' such that $S_1(D_1, D_2')$ and $S_2(D_2', D_3)$; finally, we string together $S_1(D_1, D_2')$, $S_2(D_2', D_3)$ and $S_3(D_3, D_4)$ to provide an execution on the specification side.

With the results so far, it is easy to establish the soundness of DontWaitForMe. We restate the soundness theorem from above:

Theorem 4 (Soundness of DontWaitForMe) *Suppose* $\Gamma; \eta \vdash e : \text{И}A.\, \tau\langle\eta'\rangle$ *and* $e \implies e'$. *Then there is a* $A' \supseteq A$ *such that* $[\![\Gamma; \eta \vdash e' \leq_\diamond e : \text{И}A'.\, \tau\langle\eta'\rangle]\!]$.

Proof (Sketch) By induction on the derivation of $e \implies e'$. The case R-Context follows using Lemma 7 and the induction hypothesis. The cases R-Asynchronize and R-Commute were shown above. The other cases are straightforward, using Lemmas 4, 5, 6 and 7 with routine reasoning.

4.2.6. Connection to standard soundness criteria

So far, everything we have proved about the DontWaitForMe rewriting system has been in the context of our newly-defined delayed refinement. How does it compare to more standard criteria?

In this subsection, we will show that delayed refinement implies a standard refinement relation from the theory of logical relations, *contextual refinement*. To define this criterion properly, suppose we have a new construct `diverge`, which causes divergent behavior. Type $\Gamma; \eta \vdash \texttt{diverge} : \text{И}A.\, \tau\langle\eta'\rangle$ for any $\Gamma, \eta, A, \tau, \eta'$. It is easy to check that $\{\phi\}\ \texttt{diverge}\ \{\bot\}$ and $[\![\Gamma; \eta \vdash \texttt{diverge} \leq_\diamond \texttt{diverge} : \text{И}A.\, \tau\langle\eta'\rangle]\!]$.

Intuitively, e is a contextual refinement of e' if, for all execution contexts $\mathcal{C}$, if $\mathcal{C}[e]$ terminates, so does $\mathcal{C}[e']$. This is a useful criterion since we can use the context $\mathcal{C}$ to probe observable facts about the program (e.g., heap contents, or tasks and their behavior), terminating only when some expected condition is fulfilled. We first give a proper definition of delayed refinement.

Definition 2 Let $e_1, e_2, \Gamma, \eta, \tau, \eta', A$ be given, $\Gamma; \eta \vdash e_1 : \text{И}A.\, \tau\langle\eta'\rangle$ for $i = 1, 2$.

We say that e_1 is a contextual refinement of e_2 if, for any expression context $\mathcal{E}$ with $\Gamma; \eta \vdash \bullet : \text{И}A.\, \tau\langle\eta'\rangle \rightsquigarrow \cdot; \mathsf{emp} \vdash \mathcal{E} : \text{И}A'.\, \tau'\langle\eta''\rangle$, if $\mathcal{E}[e_1]$ has a execution that reduces $\mathcal{E}[e_1]$ to a value, so does $\mathcal{E}[e_2]$.

Note that we slightly deviate from standard practice by using expression contexts instead of evaluation contexts; this allows us to include set-up code in the context. As a consequence, we are able to ensure a trivial pre-condition.

We can prove without much difficulty that delayed refinement implies contextual refinement.

Theorem 5 *Let $e_1, e_2, \Gamma, \eta, A, \tau, \eta'$ be given such that $[\![\Gamma; \eta \vdash e_1 \leq_\diamond e_2 : \text{И}A.\, \tau\langle\eta'\rangle]\!]$ holds.*

Then e_1 is a contextual refinement of e_2.

In particular, suppose that $\Gamma; \eta \vdash e : \text{И}A.\, \tau\langle\eta'\rangle$ and $e \implies^* e'$. Then e' is a contextual refinement of e.

Proof (Sketch) In view of Lemma 7, it is sufficient to consider the case $\mathcal{E} = \bullet$, $\eta = \mathsf{emp}$, $\Gamma = \cdot$. Unfolding $\Gamma \vdash \{\eta\}\ e_1 \leq_\diamond e_2 : \tau\ \{\eta'\}$ in this case gives us

$$\{\top\}\ e_1\ \left\{x.\ \exists D.\ \phi'(D, x) * \langle\top\rangle\, e_2\, \langle y.\ \psi'(D, y)\rangle\right\}.$$

We weaken the postcondition to the following form: $\{\top\}\ e_1\ \{\_.\, \langle\top\rangle\, e_2\, \langle\_.\top\rangle\}$. Using the semantics of universal and existential triples, this reads: For every execution that reduces e_1 to a value, there is an execution of e_2 that reduces it to a value.

The second part follows using Theorem 4, the first part and the fact that contextual refinement is transitive.

5. Loading JavaScript asynchronously — JSDefer

The material in this chapter is largely taken from the paper "Deferrability Analysis for JavaScript" (J. Kloos, R. Majumdar, F. McCabe), HVC 2017 [Kloos et al., 2017].

In this chapter, we apply the ideas from the previous parts of the thesis to a real-world problem: How to improve the loading time of web pages?

Modern web applications use sophisticated client-side JavaScript programs and dynamic HTML to provide a low-latency, feature-rich user experience on the browser. As the scope and complexity of these applications grow, so do the size and complexity of the client-side JavaScript used by these applications. Indeed, web applications download an average of 24 JavaScript files with about 346KB of compressed JavaScript[1]. In network-bound settings, such as the mobile web or some international contexts, optimizing the size and download time of the web page —which is correlated with user satisfaction with the application— is a key challenge.

One particular difficulty is the loading of JavaScript. The browser standards provide a complicated specification for parsing an HTML5 page with scripts [WHATWG, 2016]. In the "normal mode," parsing the page stops while the script is downloaded, and continues again after the downloaded script has been run. With tens of scripts and thousands of lines of code, blocking and waiting on JavaScript can significantly slow down page rendering and time to rendering. In order to speed up parsing, the Web Hypertext Application Technology Working Group (WHATWG), in its HTML5 specification, added "async" and "defer" tags to scripts. A script marked async is loaded in parallel with parsing through an asynchronous background process and run as soon as it is loaded. A script marked defer is also loaded in parallel with parsing, but it is evaluated only when parsing is complete, and in the order in which it was scheduled for download among other deferred scripts. The assumption is that async scripts do not interact with the rest of the page, and deferred scripts do not interact with other (non-deferred) scripts. It is up to the page designer to ensure there is no interaction. If the result of an asynchronously loaded script is required by another script, the programmer must put in explicit synchronization.

The HTML5 specification notes that the exact processing details for script-loading attributes are non-trivial, and involve a number of aspects of HTML5. Indeed, online forums such as Stack Overflow contain many discussions on the use of defer and async tags for page performance, but most end with unchecked rules of thumb ("make sure there are no dependencies") and philosophical comments such as: "[I]t depends upon you and your scripts." At the same time, industrial users are interested in having a simple way to use these attributes.

[1] See `http://httparchive.org/trends.php`, as of June 2017

In this section, we define an automatic *deferring transform*, which takes a page and marks some scripts deferred without changing observable behavior. We start by defining the notion of a *safe deferrable set*, comprising a set of scripts on a given page. If all the scripts in this set are loaded using the `defer` loading mode instead of synchronously, the user visible behavior of the page does not change.

We first discuss how a hypothetical static analysis could be used to calculate a safe deferral set. Since constructing such a static analysis is infeasible, we take a dynamic analysis route.

First, to make the idea of safe deferrable sets usable, we consider the semantics of a web page in terms of event traces, as defined by Raychev et al. [2013], and characterize the safe deferrable set using event traces. In particular, we can use event traces to define a dependency order between scripts, roughly describing the ordering of scripts due to dependencies between memory operations, and the notion of DOM-accessing scripts, which have user-visible behavior. The characterization of a safe deferrable set then (roughly) states that a set of scripts on a page is a safe deferrable set iff it contains no DOM-accessing scripts and is upward-closed under the dependency order.

We further refine the characterization by showing that if the set only contains deterministic scripts, the characterization can be checked by considering a single trace. Based on this refined characterization, we describe a dynamic analysis that conservatively computes a safe deferrable set for a given page.

The dynamic analysis proceeds by tracing the execution of a page using an instrumented browser, logging accesses to mutable state (including the JavaScript heap and the page DOM), non-deterministic behavior, task creation and execution (modeling the event-based execution model of web pages) and happens-before facts between tasks. We augment this information by using an off-the-shelf race detector to detect racing tasks. From the provided information, we can calculate a safe deferral set for the page, using the characterizations. The implementation of the analysis, JSDefer, is built on top of EventRacer [Raychev et al., 2013].

We evaluate our work by applying JSDefer to a corpus of 462 websites of Fortune 500 companies. We find that 295 (64%) of these web pages contain at least one deferrable script, with 65 (14%) containing at least 6 deferrable scripts. The maximum were 38 deferrable scripts, out of 133 scripts on that page. Furthermore, we find that while race conditions and non-determinism are widespread on web pages, we can easily identify a sufficient number of scripts that do not participate in races nor have non-deterministic behavior and are thus candidates for deferral. Finally, actually deferring scripts on these pages shows reasonable improvement in time-to-render (TTR) for 59 pages, where the median improvement of time-to-render was 198.5ms.

5.1. Background: Loading JavaScript

We briefly recall the WHATWG specification for loading HTML5 documents by a browser. A browser parses an HTML5 page into a data structure called the *document object model* (DOM) before rendering it on the user's screen. Parsing the document may require

downloading additional content, such as images or scripts, whose links are provided in the document. The browser downloads images asynchronously, while continuing to parse the document. However, scripts are handled in a different way. The browser downloads scripts synchronously by default, making the parser wait for the download, and evaluates the script before continuing to parse the page. Of course, this puts script download and parsing on the critical path of the rendering pipeline. Since network latency can be quite high (on the order of tens or hundreds of milliseconds) and script execution time may be non-negligible, this may cause noticeable delays in page loading. To allow asynchronous loading of scripts, the WHATWG specification allows two Boolean attributes in a `script` element, *async* and *defer*. In summary, there are three loading strategies for scripts:

- Synchronous loading. When encountering a `script` tag with no special attributes, the browser suspends parsing, downloads the script synchronously, and evaluates it after download is complete. Parsing continues after the evaluation of the script.

- Asynchronous loading. When encountering a `<script src="..."async>` tag, the browser starts an asynchronous download task for the script in the background but continues parsing the page until the script has been loaded. Then, parsing is suspended and the script is evaluated before continuing with parsing.

- Deferred loading. When encountering a `<script src="..."defer>` tag, the browser starts a download task for the script background but continues parsing the page. Once parsing has finished and the script has been downloaded, it is evaluated. Moreover, scripts are evaluated in the order that their corresponding script tags were parsed in the HTML, even though a later script may have finished downloading earlier.

The precise description of the different modes can be found in section 4.12 of the WHATWG specification. It spans over ten pages, and distinguishes between five different loading modes, including the ones listed, inline scripts and special handling for scripts inserted at runtime.

While asynchronous or deferred loading is desirable from a performance perspective, it can lead to *race conditions*, that is, the output of the page may depend on the order in which scripts are executed [Raychev et al., 2013]. Consider the following example:

```
<html><body><script src="http://www.foo.com/script1.js"></script>
<script>if (!script1executed) { alert("Error!"); }</script></body></html>
```

where `script1.js` is simply `script1executed = true;`. As the script is loaded synchronously, the code has no race (yet): the `alert` function will never be called.

If we annotate the first script with the `async` tag, we introduce a race condition. Depending on how fast the script is loaded, it may get executed before or after the inline script. In case it gets executed later, an alert will pop up, noting that the external script has not been loaded yet. Changing the loading mode to defer does not cause a race, per se, but now the alert always pops up; thus deferred loading of the script changes the observable behavior from the original version.

Another kind of race condition is incurred by scripts that perform certain forms of DOM accesses. For instance, consider the following page:

```
<html><body><script src="http://www.foo.com/script2.js"></script>
            <span id="marker">Something</span></body></html>
```

where `script2.js` uses the DOM API to check if a tag with id `marker` exists. Loaded synchronously, the outcome of this check will always be negative. Asynchronous loading would make it non-deterministic, while deferred loading will remain deterministic but the check will always be positive.

Our goal is to analyze a web page and add `defer` tags to scripts, wherever possible. To ensure we can load scripts safely in a deferred way, we need to make certain that deferred loading does not introduce races through program variables or the DOM and does not change the observable behavior. Next, we make this precise.

5.2. Deferrability analysis

Take an HTML5 page that includes one or more JavaScript scripts, some of which are loaded synchronously. When is it safe to load one or more of the synchronously loaded scripts in defer mode instead? To answer this question, we describe criteria in terms of trace semantics. We first give a hypothetical static analysis-based approach using techniques from the previous chapter. Since this approach is known to be unimplementable, we then provide a dynamic analysis that calculates an under-approximation of the set of safely deferrable scripts.

In the following, suppose we are given a web page with scripts $s_1, \ldots, s_n$ (in order of appearance). For this exposition, we assume that all the scripts are loaded synchronously; the extension to pages with mixed loading modes and inline scripts is straightforward.

On a high level, our goal is to produce a modified version of the page where some of the scripts are loaded deferred instead of synchronously, but the visible behavior of the page is the same. Concretely, when loading and displaying the page, the browser constructs a view of the page by way of building a DOM tree, containing both the visible elements of the page and the association of certain event sources (e.g., form fields or `onload` properties of images) with handler functions. The DOM tree is the object graph reachable from `document.root` which consists of objects whose type is a subtype of `Node`; compare WHATWG [2016]. This DOM tree is built in stages, adding nodes to the tree, modifying subtrees and attaching event handlers. This can be due to parsing an element in the HTML document, receiving a resource, user interaction, or script execution.

Definition 3 A *DOM trace* consists of the sequence of DOM trees that are generated during the parsing of a page. The *DOM behavior* of a page is the set of DOM traces that executing this page may generate.

Note that even simple pages may have multiple DOM traces; for instance, if a page contains multiple images, any of these images can be loaded before the others, leading to different intermediate views.

Definition 4 For a page p with scripts $s_1, \ldots, s_n$, and a set $D \subseteq \{s_1, \ldots, s_n\}$ let p' be the page where the members of D are loaded deferred instead of synchronously. We say that D is a *safe deferral set* if the DOM behavior of p' is a subset of the DOM behavior of p.

5.2.1. A hypothetical static approach

Before describing the actual approach, we sketch a hypothetical strategy to find safe deferral sets using a static analysis.

We can consider a web page as a form of asynchronous program: Each HTML tag corresponds to a statement. Most tags are simple updates of the DOM data structure, adding nodes to the DOM tree and, potentially, setting up event handlers. Script tags modify the DOM tree and additionally execute the contained script.

Following the WHATWG specification [WHATWG, 2016], an HTML fragment like

```
<p>Visitor number <script src="visitor.js"></script></p>
```

can be seen as an execution of an asynchronous program along the following lines[2]:

```
/* Add a p node to the current DOM */
node = add_node("p", ...);
/* Add the text node for "visitor number" */
text = add_text("Visitor number ", node);
/* Add the script node */
script = add_node("script", node);
script_body = load("visitor.js");
eval(script_body);
/* ... */
post_event("DocumentContentLoaded");
```

Adding a `defer` attribute to the script changes it to

```
/* Add a p node to the current DOM */
node = add_node("p", ...);
/* Add the text node for "visitor number" */
text = add_text("Visitor number ", node);
/* Add the script node */
script = add_node("script", node);
script_body_p = post(load("visitor.js"));
/* ... */
script_body = wait(script_body_p);
eval(script_body);
post_event("DocumentContentLoaded");
```

This rewriting can be described using the DontWaitForMe rules as follows: We first apply the appropriate variation of the R-Asynchronize rule to get

```
/* Add a p node to the current DOM */
node = add_node("p", ...);
```

[2]We gloss over many details here, including the existence of various scheduling points.

```
/* Add the text node for "visitor number" */
text = add_text("Visitor number ", node);
/* Add the script node */
script = add_node("script", node);
script_body_p = post(load("visitor.js"));
/* start of wait-and-eval block */
script_body = wait(script_body_p);
eval(script_body);
/* env of wait-and-eval block */
/* ... */
post_event("DocumentContentLoaded");
```

By repeated application of R-Commute, we can then move the wait-and-eval block to the end. For this, we need to prove that the script body commutes with all further statements in the program (for the wait statement, it turns out that this is straightforward). Hence, if we had a static analysis that would provide us with sufficient information for commutativity analysis, we could perform this transformation automatically.

Sadly, a sufficiently precise static analysis of Javascript seems far out of reach. Consider the experiences of the TAJS project [Jensen et al., 2009, 2011, 2012], where even simple static analyses of Javascript did not scale well to realistic code examples, as well as their description of the problems encountered. For this reason, we decided to take an alternative, more heuristic approach to deferrability analysis.

5.2.2. Background: Event traces and races in web pages

We recall an event-based semantics of JavaScript [Petrov et al., 2012, Raychev et al., 2013, Adamsen et al., 2017] on which we build our analysis; we follow the simplified presentation of Adamsen et al. [2017]. For a given execution of a web page, fix a set of events E; each event models one parsing action, user interaction event or script execution (compare also the event model of HTML, WHATWG [2016, Section 8.1]). Our semantics will be based on the following operations:

- $rd(e, x)$ and $wr(e, x)$: These operations describe that during the execution of event $e \in E$, some shared object x (which may be a global variable, a JavaScript object, or some browser object, such as a DOM node) is read from or written to.
- $post(e, e')$: This operation states that during the execution of event $e \in E$, a new event $e' \in E$ is created, to be dispatched later (e.g., by setting a timer or directly posting to an event queue).
- $begin(e)$ and $end(e)$: These operations function as brackets, describing that the execution of event $e \in E$ starts or ends.

A *trace* of an event-based program is a sequence of *event executions*. An event execution for an event e is a sequence of *operations* such that the sequence starts with a begin operation $begin(e)$, the sequence ends with an end operation $end(e)$, and otherwise consists of operations of the form $rd(e, x)$, $wr(e, x)$, and $post(e, e')$ for some event $e' \in E$. For

a trace of a program consisting of event executions of events $e_1, e_2, \ldots, e_n$, by abuse of notation, we write $t = e_1 \ldots e_k$.

Furthermore, we define a *happens-before relation*, denoted hb, between the events of a trace. It is a pre-order (i.e., reflexive, transitive, and anti-symmetric) and $e_i \,\mathsf{hb}\, e_j$ holds in two cases: if there is an operation $post(e_i, e_j)$ in the trace, or if e_i and e_j are events created externally by user interaction and the interaction creating e_i happens before that for e_j.

Two events e_i and e_j are *unordered* if neither $e_i \,\mathsf{hb}\, e_j$ nor $e_j \,\mathsf{hb}\, e_i$. They have a race if they are unordered, access the same shared object, and at least one access is a write.

5.2.3. When is a set of scripts deferrable?

To make the deferrability criterion given above more tractable, we give a sufficient condition in terms of events. A sketch of the correctness proof is given in Section 5.4. We first define several notions on events, culminating in the notion of the *dependency order* and the *DOM-modifying script*. We use these two notions to give the sufficient condition.

Consider a page with scripts $s_1, \ldots, s_n$. For each script s_i, there is an event e_{s_i} which corresponds to the execution of s_i. By abuse of notation, we write s_i for e_{s_i}.

We say that e *posts* e' if $post(e, e')$ appears in the event execution of e. We say that e *transitively posts* e' if there is a sequence $e = e_1, \ldots, e_k = e'$ such that s posts e_1 and e_i posts e_{i+1}. This explicitly includes the case $e = e'$ (i.e., we take a reflexive-transitive closure).

Suppose script s transitively posts event e. We call e a *near event* if, for all scripts s', $s \,\mathsf{hb}\, s'$ implies $e \,\mathsf{hb}\, s'$. Otherwise, we call e a *far event*. We say that a script s is *DOM-accessing* iff there is a near event e such that e reads from or writes to the DOM.

Now, consider two events e_i and e_j such that $i < j$. We say that e_i *must come before* e_j iff both e_i and e_j access the same object (including variables, DOM nodes, object fields and other mutable state) and at least one of the accesses is a write. For two scripts s_i and s_j, $i < j$, we say that s_i must come before s_j iff there is a near events $e_{i'}$ of s_i and an event $e_{j'}$ such that $s_j \,\mathsf{hb}\, e_{j'}$ and $e_{i'}$ must come before $e_{j'}$.

The dependency order $s_i \preceq s_j$ is then defined as the reflexive-transitive closure of the must-come-before relation.

The proofs of the following theorems can be found in Section 5.4.

Theorem 6 *Let p be a page with scripts $s_1, \ldots, s_n$ and $D \subseteq \{s_1, \ldots, s_n\}$. If the following two conditions hold:*

1. *If $s_i \in D$, then script s_i is not DOM-accessing in any execution.*
2. *If $s_i \in D$ and $s_i \preceq s_j$ in any execution, then $s_j \in D$.*

then D is a safe deferral set.

The gist of the proof is that all scripts whose behavior is reflected in the DOM trace are not deferred and hence executed in the same order (even with regard to the rest of the document). Due to the second condition, each script starts in a state that it could start

in during an execution of the original page, so its behavior with regard to DOM changes is reflected in the DOM behavior of the original page.

The distinction between near and far events comes from an empirical observation: when analyzing traces produced by web pages in the wild, script-posted events clearly separate in these two classes. Near events are created by the `dispatchEvent` function, or using the `setTimeout` function with a delay of less than 10 milliseconds. On the other hand, far events are event handlers for longer-term operations (e.g., XMLHttpRequest), animation frame handlers, or created using `setTimeout` with a delay of at least 100 milliseconds. There is a noticeable gap in `setTimeout` handlers, with delays between 10 and 100 milliseconds being noticeably absent.

We make use of this observation by treating a script and its near events as an essentially sequential part of the program, checking the validity of this assumption by ensuring that the near events are not involved in any races.

This allows us to formulate a final criterion, which can be checked on a single trace:

Theorem 7 *Let page p and set D be given as above, and consider a single trace executing events $e_1, \ldots, e_n$. Suppose the following holds:*

1. *If e is a near event of s and accesses the DOM, $s \notin D$.*
2. *If e is involved in a race or has non-deterministic control flow, $s \,\mathsf{hb}\, e$ and s' happens before s in program order (including $s = s'$), then $s' \notin D$.*
3. *D is $\preceq$-upward closed.*

Then D is a safe deferral set.

The key idea of this proof is that all scripts in D are "sufficiently deterministic," so the conditions of the previous theorem collapse to checking a unique trace of the script.

In fact, the second condition of the theorem can be weakened a bit, as can be seen in the proof: it is sufficient to consider those s' who access objects that e accesses. Thus, if we have an over-approximation of the objects that e accesses, we can prune the set of scripts that must be marked non-deferrable.

5.2.4. JSDefer: A dynamic analysis for deferrability

The major obstacle in finding a deferrable set of scripts is the handling of actual JavaScript code, which cannot be feasibly analyzed statically. This is because of the dynamic nature of the language and its complex interactions with browsers, including the heavy use of introspection, `eval` and similar constructs, and variations in different browser implementations. For example, we found a script which takes the string representation of a function (which gives its source code) and performs regular expression matching on it; static analysis cannot hope to cope with this. In the following, we present a dynamic analysis for finding a safe deferral set that we call *JSDefer*.

Assumption: For reasons of tractability, we assume in this paper that no user interaction occurs before the page is fully loaded. This is because it is well-known that early user interaction is often not properly handled; in fact, Adamsen et al. [2017] devoted an entire paper to showing how to fix errors due to early user interaction by dropping or postponing events due to user input. Hence, we assume that early user interaction does not occur (or is pre-processed as in the paper cited above before being analyzed by JSDefer).

With this assumption at hand, as reasoned above, we only need to consider scripts themselves and their near events; we call this the *script initialization code*. This part of the code is run during page loading and, empirically is "almost deterministic": it does not run unbounded loops and, for the most part, only exhibits limited use of non-determinism. We provide experimental evidence for this observation below. For this reason, it would be feasible to collect all possible execution traces dynamically (by controlling all sources of non-determinism), and to derive all required information from that finite set of traces. In fact, we only collect a single trace and aggressively mark any scripts that may exhibit non-deterministic control flow.

JSDefer piggybacks on instrumented browsers. In particular, we used a modified version of the instrumented WebKit browser from the EventRacer project [Raychev et al., 2013] to generate a trace, including a happens-before relation. For now, we use a simple, not entirely sound heuristic to detect non-deterministic behavior: We extended the instrumentation to also include coarse information about scripts getting data from non-deterministic and environment-dependent sources. In particular, we mark all calls to the random number generator, functions returning the current time and data, and various properties about the page state. We used the official browser interface descriptions (given in WebIDL) and the JavaScript specification to ensure completeness of marking.

We perform deferrability analysis on the collected trace, using the following steps:

1. Perform a race analysis on the trace; our implementation uses EventRacer.

2. For each event in the trace, check whether it is a near event for some script.

3. Sequentialize the near events for each script into a single event. Call the resulting events *script events*.

4. Calculate the race sets for each script event as the union of race sets for its involved script execution event and near events.

5. Calculate read and write sets for each script event, as well as predicates that check if a script event is DOM-accessing or accesses sources of non-determinism. Additionally compute read-sets for all far events.

   Here, the read set contains all the objects that have been read by the event without a preceding write by the event, and the write set contains all objects on which a write has been performed.

6. Using the read and write sets, calculate the dependency relation between scripts. This uses the read and write sets from the previous step.

7. Compute the set of deferrable scripts using the criterion of Theorem 7.

This calculation computes a safe deferrable set, although we make no claims with regard to maximality. In a final step, we rewrite the top-level HTML file of the page to add defer annotations to all scripts in the deferrable set.

5.3. Evaluation

We evaluated JSDefer on the websites of the Fortune 500 companies [Fortune 500] as a corpus. To gather deferrability information, we used an instrumented WebKit browser to generate event traces. For each website, we ran the browser with a timeout of 30s, with a 1s grace period after page loading to let outstanding events settle. In this way, we could collect traces for 462 web pages; in the other cases, either the browser could not handle the page or the page would not load. We then run JSDefer on each of the collected traces. The analysis was successful on all traces.

Rewriting the HTML files based on the safe deferral set worked on 460 pages; the other two pages contained invalid HTML that could not be handled by the tool. For 11 of the pages, we did not find JavaScript on the page; later analysis showed that at the time of trace collection, we only received an error page from these sites. Due to the large amount of data already collected, we dropped these pages from the corpus. All in all, the main part of the analysis comprises 451 web pages.

In the evaluation, we want to answer five main questions:

1. How much is deferred and asynchronous loading already used? How is it used?
2. Are our assumptions about determinism justified?
3. How many deferrable scripts can we infer?
4. What kind of scripts are deferrable?
5. Does deferring these scripts gain performance?

5.3.1. Tools and environment

For the experiments, we need several tools.

Instrumented browser. The instrumented browser is used to collect the trace information for the dynamic analysis. To achieve this, we extended the instrumented WebKit browser from the EventRacer project and added a simple analysis for non-determinism. The log file produced by this browser contains all the information that EventRacer uses, plus additional information on the generation of non-deterministic and environment-dependent values.

Deferrability analysis tool. This tool is the core component of JSDefer and performs actual deferrability analysis. It reads the event log from above, and produces a human-readable report detailing the following:

1. Which scripts are loaded on the page, from where, and in which way (inline synchronous, async, deferred, inserted at runtime)?
2. For each script, does it use non-deterministic or environment-dependent values? For this analysis, we take both the execution of the main body of the script and its near events into account; we call the execution of this the *script initialization*.
3. Which scripts initializations are involved in race conditions? We list the races identified by EventRacer that contain at least one script initialization event.
4. The list of scripts that can be deferred, given by location in the HTML file and script path. This list is also output in a machine-readable format.

Web page instrumentation and performance measurement. For the performance measurement, we created modified versions of the the web pages in our corpus which included the additional defer annotations. This was performed using a simple tool that reads the list of deferred scripts and modifies the top-level HTML file. We then used a proxy to (a) intercept the downloads of the main HTML files, giving the deferred and non-deferred version depending on a HTTP header, and (b) serve all resource files from the origin server.

The measurements were performed using a version of the WebPageTest tool [Viscomi et al., 2015]. We used shaping to simulate a connection over an LTE network, as to simulate a realistic usage scenario of a mobile user accessing the pages in our corpus.

Environment The trace collection and page analysis steps were performed on a machine with 48 Xeon E7-8857 CPU and 1536 GiB of memory, running Debian Linux 8 (kernel 4.4.41). The proxy set-up was run on a smaller machine (24 Xeon X5650 CPUs, 48 GiB memory) with the same OS; the WebPageTest instance was run by our industrial collaborator Instart Logic in their performance-test environment.

5.3.2. How are async and defer used so far?

As a first analysis step, we analyzed if pages were using async and defer annotations already, and in which situations this was the case. The numbers are given in Table 5.1.

The first observation from the numbers is that defer is very rarely used, while there is a significant numbers of users of async. Further analysis shows many of these asynchronous scripts come from advertising, tracking, web analytics, and social media integration. For instance, Google Analytics is included in this way on at least 222 websites[3]. Another common source is standard frameworks that include some of their scripts this way. In these cases, the publishers provide standard HTML snippets to load their scripts, and

[3]Many common scripts are available under multiple aliases, so we performed a best-effort hand count.

Async or defer	#pages
Neither	32
Defer only	0
Async only	404
Only scripts included using standard snippets	256
Others	148
Both	15

Table 5.1.: Number of pages in the corpus that use async or defer. The sub-classification of async scripts was done manually, with unclear cases put into "others".

the standard snippets include an async annotation. On the other hand, 254 pages include some of their own scripts using async. In some pages, explicit dependency handling is used to make scripts capable of asynchronous loading, simulating a defer-style loading process.

5.3.3. Are our assumptions justified?

The second question is if our assumptions about non-determinism are justified. We answer it in two parts, first considering the use of non-deterministic functions, and then looking at race conditions.

Non-determinism: To perform non-determinism analysis, we used a browser that was instrumented for information flow control. This allowed us to identify scripts that actually use non-deterministic data in a way that may influence other scripts, by leaking non-deterministic data or influencing the control flow. We considered three classes of non-determinism sources:

1. `Math.random`. For most part, this function is used to generate unique identifiers, but we found a significant amount of scripts that actually use this function to simulate stochastic choice.

2. `Date.now` and related functions. These functions are included since their result depends on the environment. We found that usually, these functions are called to generate unique identifiers or time stamps, and to calculate time-outs.

   Nevertheless, we found examples for which it would not be feasible to automatically detect safety automatically. For instance, we found one page that had a busy-wait loop in the following style:

   ```
   var end = Date.now() + timeout;
   while (Date.now() < end) {}
   ```

   While it is easy to see manually that this code would not influence deferrability decisions, an automatic analysis would have to perform quite some work.

3. Functions and properties about the current browser state, including window size, window position and document name. While we treat these as a source of non-determinism, it would be better to classify them as environment dependent values; we find that in the samples we analyzed, they are not used in way that would engender non-determinism. Rather, they are used to calculate positions of windows and the like.

As it turns out, many standard libraries make at least some use of non-determinism. For instance, jQuery and Google's analytics and advertising libraries generate unique identifiers this way.

Additionally, many scripts and libraries have non-deterministic control flow. We found 1704 cases of scripts with non-deterministic control flow over all the pages we analyzed. That being said, this list contains a number of duplicates: In total, at least 546 of these scripts were used more than once[4]. They form 100 easily-identified groups, the largest of which are Google Analytics (with two script names), accounting for 187 of these scripts, various versions of jQuery from various providers (40 instances) and YouTube (20 instances).

More importantly, we analyzed how many of the scripts we identified as deferrable have non-deterministic control flow. As it turns out, there was no overlap between the two sets: Our simple heuristic of scripts calling a source of non-determinism was sufficient to rule out all non-deterministic scripts.

Race conditions: We additionally analyzed whether non-determinism due to race conditions played a role. In this case, the findings were, in fact, simple: While there are numerous race conditions, they all occur between far events. We did not encounter any race conditions that involved a script (i.e., the execution of the body of the script) or its near events.

One further aspect is that tracing pages does not exercise code in event handlers for user inputs. This may hide additional dependencies and race conditions. As reasoned above, we assume that no user interaction occurs before the page is loaded (in particular, after deferred scripts have run). The reasoning for this was given above; we plan to address this limitation in further work.

5.3.4. Can we derive deferrability annotations for scripts?

To evaluate the potential of inferring deferrability annotations, we used the analysis described above to classify the scripts on a given page into five broad classes:

- The script is loaded synchronously and can be deferred,
- The script is already loaded with defer or async (no annotation needs to be inferred here);

[4]We clustered by URL (dropping all but the last two components of the domain name and all query parameters), which misses some duplicates

Table 5.2.: Number of deferrable scripts. This includes pages with no scripts.

Number of deferrable scripts	On how many pages?
0	167 (156 excluding pages without scripts)
1	86
2	55
3–5	89
6–10	47
more than 10	18

- The script is an inline script; in this case, deferring would require to make the script external, with questionable performance impact;
- The script is not deferrable since it performs DOM writes;
- The script is not deferrable because it is succeeded by a non-deferrable script in the dependency order.

The general picture is that the number of deferrable scripts highly depends on the page being analyzed. 295 of all pages contain deferrable scripts, and 209 of all pages permit deferring multiple scripts. Moreover, on 18 of the pages considered, at least 11 scripts can be deferred. Among these top pages, most have between 11 and 15 deferrable scripts (4 with 11, 2 with 12, 4 with 13, 5 with 15), while the top three pages have 16, 17 and 38 deferrable scripts on them.

Further analysis shows that some pages have been hand-optimized quite heavily, so that everything that could conceivably be deferred is already loaded with defer or async. Conversely, some pages have many scripts that can be deferred. We also find that it is quite common that some scripts will not be deferred because non-deferrable scripts depend on them. In many cases, these dependencies are hard ordering constraints: For instance, jQuery is almost never deferrable since later non-deferrable scripts will use the functionality it provides.

We also analyzed what percentage of scripts are deferrable on a given page. Discarding the pages that had no deferrable scripts on them (corresponding to a percentage of 0), we get the following picture:

Percentage of deferrable scripts	On how many pages?
$< 10\%$	180
$10 - 20\%$	56
$20 - 30\%$	37
$30 - 40\%$	14
$40 - 50\%$	6
$50 - 60\%$	1
$60 - 70\%$	1

That being said, we observe some spurious dependencies between scripts; this indicates room for improvement of the analysis. As an example, consider the jQuery library again. Among other things, it has a function for adding event handlers to events. Each of these

event handlers is assigned a unique identifier by jQuery. For this, it uses a global variable `guid` that is incremented each time an event handler is added; clients treat the ID as an opaque handle. Nevertheless, if multiple scripts attach event handlers in this way, there is a an ordering constraint between them due to the reads and writes to `guid`, event though the scripts may commute with each other.

Looking at the pages with a high number of deferrable scripts, we find that there are two broad classes that cover many deferrable scripts: "Class definitions", which create or extend an existing JavaScript object with additional attributes (this would correspond to class definitions in languages such as Java), and "poor man's deferred scripts", which install an event handler for one of the events triggered at page load time (load, DOMContentLoaded and jQuery variants thereof) and only then execute their code.

5.3.5. Does deferring actually gain performance?

Since we found a significant number of scripts that can actually be deferred, we also measure how performance and behavior is affected by adding defer annotations. As described above under "tools and environment", we used a proxy-based setup to present versions of each web page with and without the additional defer annotations from deferrability analysis to WebPageTest. We then measured the time-to-render (i.e., the time from starting the page load to the first drawing command of the page) for each version of each page, We choose time-to-render as the relevant metric because the content delivery industry uses it as the best indicator of the user's perception of page speed. This belief is supported by studies, e.g. Gao et al. [2017].

We took between 38 and 50 measurements for each case, with a median of 40. The measurements were taken for each page that had at least one deferrable script and could successfully be rewritten.

One issue that we encountered is that many pages force a connection upgrade to SSL. In a separate analysis, we polled all 500 pages in the corpus to see how many of them force SSL upgrades; in total, 475 pages were reachable, and out of these, 209 forced an upgrade. Since our measurement setup could not deal with interposing on SSL connections, the data from measurements on these sites should be considered suspect, since some resources on these pages may not have been loaded correctly. For this reason, we threw out the data corresponding to pages that force SSL connections, or use embedded SSL links that we would have to interpose. In the end, we considered 169 pages that had deferrable scripts on them and did not force SSL upgrades or contain explicit SSL links.

The first observation to make is that the load time distribution tends to be highly non-normal and multi-modal. This can be seen in a few samples of load time distribution, as shown in Fig. 5.1. The violin plots in these graphs visualize an approximation of the probability distribution of the loading time for each case.

For this reason, we quantify the changes in performance by considering the *median change in time-to-render* for each page, meaning we calculate the median of all the pairwise differences in time-to-render between the modified and the unmodified version of the page. This statistic is used as a proxy for the likely change in loading time by applying JSDefer. In the following, we abbreviate the median change in time-to-render as

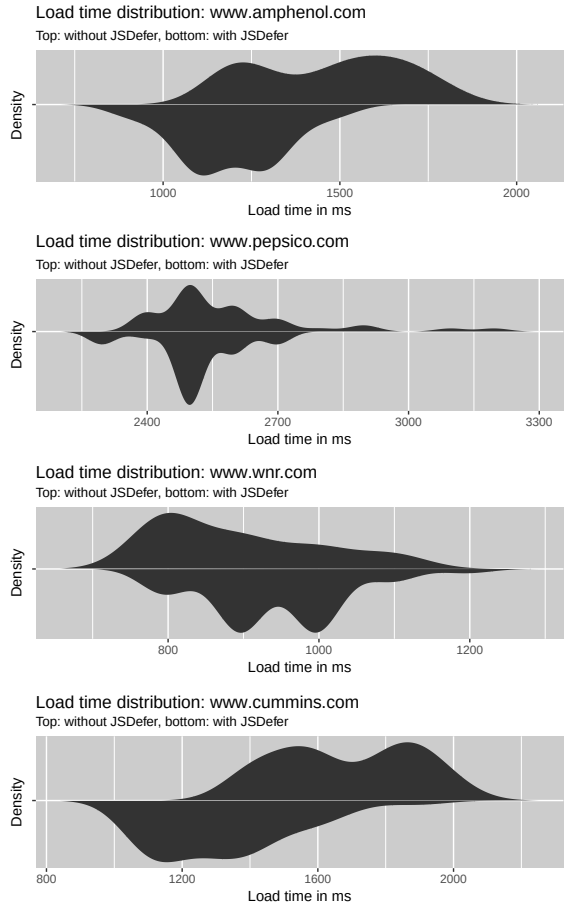


Figure 5.1.: Violin plots of load time distributions for some pages, before and after applying JSDefer. The distribution is a smoothed representation of the (discrete) distribution of the sample.
The plots can be read as follows: Each plot represent two probability distributions, given as density functions. One density function grows upwards from the origin line, giving the density of the TTR distribution of the page before applying JSDefer. The other density function grows downwards from the origin line, giving the density after applying JSDefer. Loading time increases to the right, so a distribution can be considered better if it has more of its probability mass on the left.

MCTTR. We additionally use the Mann-Whitney U test to ensure that we only consider those cases where MCTTR gives us statistically significant results.

Out of the 169 considered pages, 66 had a statistically significant MCTTR.

The actual median changes are shown in Fig. 5.2, together with confidence intervals. The data is also given in Table 5.3. This table also contains the median TTR of the original page. Several things are of note here:

1. As promised in the introduction, the median improvement in TTR is 198.5ms in the examples provided, while their median load time is 3097ms.

2. Most of the pages that pass the significance test have positive MCTTR, meaning that applying JSDefer provides benefits to time-to-render: For 59 pages, JSDefer had a positive effect, versus 7 pages where it had a negative effect. (85 versus 14 including SSL pages).

3. 49 of the pages in our sample have an estimated MCTTR of at least 100ms=0.1s. This difference corresponds to clearly perceptible differences in time-to-render.

   Even when taking the lower bound of the 95% confidence interval, 32 of the pages still have this property.

4. For 7 pages, we get a negative MCTTR, corresponding to worse loading time. This indicates that JSDefer should not be applied blindly.

We tried to analyze the root causes for the worsening of load times. For this, we used Chrome Developer Tools to generate a time-line of the page load, as well as a waterfall diagram of resource loading times. The results were mostly inconclusive; we could observe that the request for loading some scripts on two of these pages was delayed, and conjecture that we are hitting edge cases in the browser's I/O scheduler.

Another observation that can be made by analyzing the violin plots is that JSDefer sometimes drastically changes the loading time distribution of pages, but there is no clear pattern. The interested reader may want to see for themselves by looking at the complete set of plots in the supplementary material [5].

An interesting factor in the analysis was the influence of *pre-loading*: For each resource (including scripts) that is encountered on a page, as soon as the reference to the script is read (which may well be quite some time before "officially" parsing the reference), a download task for that resource is started[6], so that many download tasks occur in parallel. This manifests itself in many parallel downloads, often reducing latency for downloads of scripts and resources. This eats up most of the performance we could possibly win; preliminary experiments with pre-loading switched off showed much bigger improvements. Nevertheless, even in the presence of such pre-loading, we were able to gain performance. We also performed some timing analysis of page downloads to understand how performance is gained or lost, and found that the most important factor is, indeed,

[5] Available at `http://www.mpi-sws.org/~jkloos/JSDefer-results`

[6] Glossing over the issue of connection limits

Table 5.3.: MCTTR values for pages with significant MCTTR, sorted by ascending MCTTR. All times are given in milliseconds.

Page	MCTTR	MCTTR (95% confidence interval)	Median TTR of original page
www.williams.com	-452.0	[-698.0,-201.0]	2300.0
www.visteon.com	-401.0	[-899.0,-99.0]	6996.0
www.mattel.com	-401.0	[-900.0,-1.0]	3995.0
www.statestreet.com	-299.0	[-400.0,-100.0]	2596.0
www.fnf.com	-201.6	[-500.0,-1.0]	3896.0
www.cbscorporation.com	-99.0	[-100.0,0.0]	1296.0
www.wnr.com	-98.0	[-100.0,0.0]	895.0
www.lansingtradegroup.com	98.6	[1.0,118.0]	2597.0
www.kiewit.com	99.0	[0.0,101.0]	1096.0
www.emcorgroup.com	99.0	[0.0,201.0]	1696.0
www.dovercorporation.com	99.0	[0.0,100.0]	1896.0
www.domtar.com	99.0	[1.0,100.0]	1896.0
www.eogresources.com	99.0	[0.0,100.0]	1896.0
www.johnsoncontrols.com	99.0	[0.0,101.0]	3296.0
www.altria.com	99.0	[0.0,101.0]	499.0
www.jmsmucker.com	99.0	[0.0,199.0]	996.0
www.itw.com	99.0	[1.0,100.0]	1295.0
www.walgreensbootsalliance.com	100.0	[1.0,101.0]	1096.0
www.bostonscientific.com	100.0	[1.0,101.0]	1297.0
www.apachecorp.com	100.0	[0.0,199.0]	1396.0
www.lifepointhealth.net	100.0	[99.0,100.0]	1396.0
www.marathonoil.com	100.0	[99.0,101.0]	1097.0
www.cstbrands.com	100.0	[99.0,199.0]	1897.0
www.mohawkind.com	101.0	[100.0,200.0]	1496.0
www.delekus.com	101.0	[98.0,200.0]	1795.0
www.stanleyblackanddecker.com	103.0	[100.0,199.0]	1196.0
www.fanniemae.com	112.3	[1.0,296.0]	2999.0
www.citigroup.com	114.0	[99.0,201.0]	1296.0
www.microsoft.com	130.0	[14.0,206.0]	1455.0
www.pultegroupinc.com	139.0	[93.0,219.0]	1120.0
www.mosaicco.com	196.0	[100.0,200.0]	1496.0
www.tysonfoods.com	198.0	[100.0,280.0]	1796.0
www.iheartmedia.com	198.0	[1.0,300.0]	1696.0
www.rrdonnelley.com	199.0	[104.0,201.0]	2097.0
www.raytheon.com	199.0	[0.0,401.0]	1697.0
www.navistar.com	199.6	[53.0,318.0]	2740.0
www.genesishcc.com	200.0	[1.0,399.0]	4497.0
www.chs.net	200.0	[100.0,298.0]	1796.0
www.newellbrands.com	200.0	[100.0,299.0]	1197.0
www.navient.com	200.0	[0.0,304.0]	2597.0
www.ncr.com	200.0	[96.0,300.0]	2096.0
www.sempra.com	200.0	[100.0,300.0]	1696.0
www.univar.com	200.0	[101.0,300.0]	1496.0
www.avoncompany.com	200.0	[100.0,300.0]	1596.0
www.pricelinegroup.com	200.0	[199.0,201.0]	1596.0
www.pacificlife.com	201.0	[100.0,399.0]	3296.0
www.weyerhaeuser.com	242.2	[200.0,300.0]	2497.0
www.techdata.com	298.0	[100.0,303.0]	2296.0
www.tenneco.com	299.0	[200.0,300.0]	1896.0
www.dana.com	299.0	[200.0,300.0]	1496.0
www.cablevision.com	299.0	[298.0,300.0]	2196.0
www.amphenol.com	300.0	[200.0,400.0]	1496.0
www.calpine.com	300.0	[201.0,302.0]	2098.0
www.nov.com	300.0	[103.0,498.0]	3396.0
www.harman.com	303.0	[300.0,400.0]	2195.0
www.burlingtonstores.com	395.0	[200.0,501.0]	4179.0
www.centene.com	398.0	[308.0,412.0]	2306.0
www.cummins.com	398.9	[299.0,496.0]	1695.0
www.markelcorp.com	500.0	[498.0,501.0]	1596.0
www.spectraenergy.com	501.0	[499.0,600.0]	2395.0
www.spiritaero.com	598.0	[499.0,601.0]	1797.0
www.wholefoodsmarket.com	611.7	[412.0,790.0]	2138.0
www.deanfoods.com	700.0	[401.0,3900.0]	3796.0
www.mutualofomaha.com	702.0	[700.0,800.0]	2396.0
www.lkqcorp.com	800.0	[700.0,900.0]	3301.0
www.ppg.com	891.4	[514.0,1299.0]	5096.0

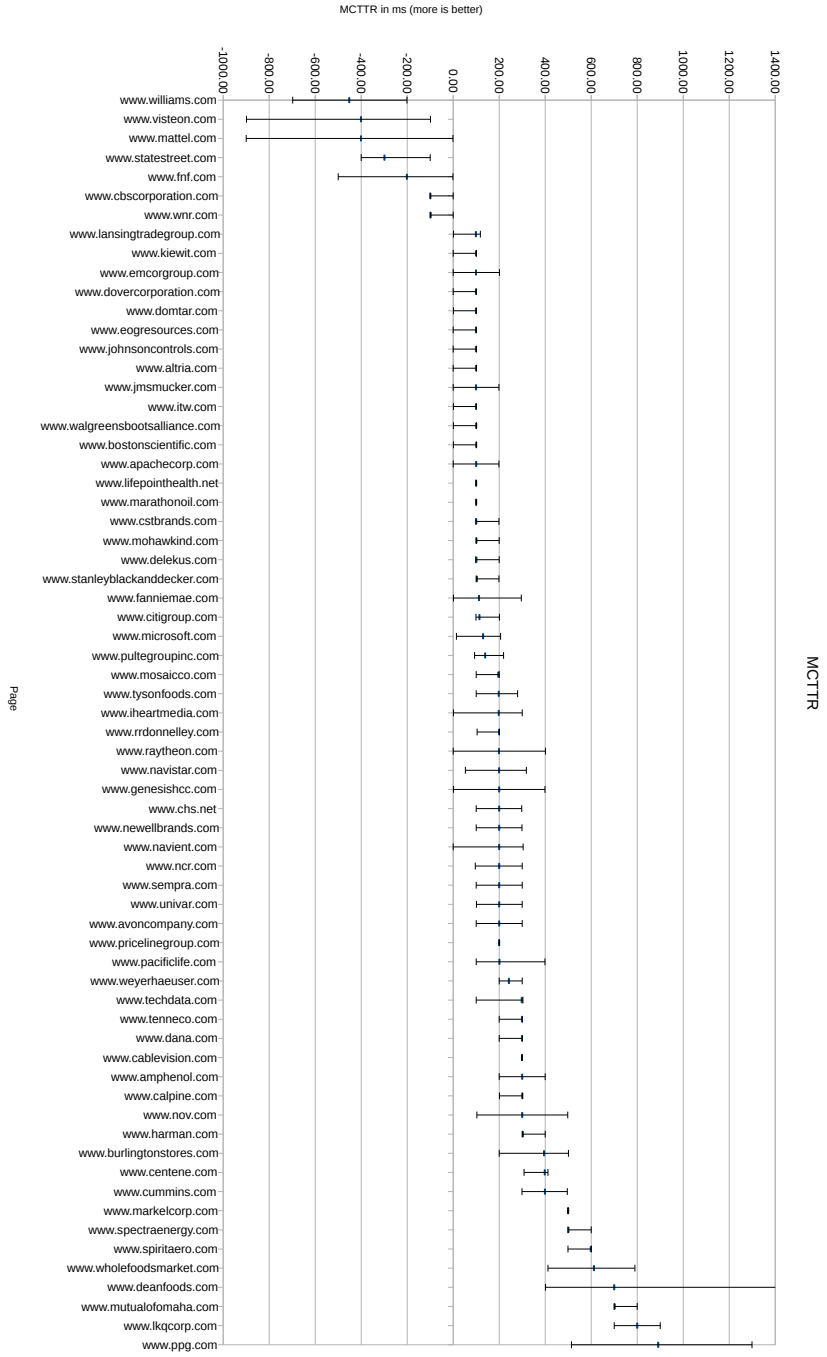


Figure 5.2.: MCTTR values for pages with significant MCTTR. This is a visualization of Table 5.3.

the time spent waiting for scripts to become available. The time saved by executing scripts later was only a minor factor.

Finally, to judge the impact of the improvements we achieved, we discussed the results with our industrial collaborator. Instead of considering the MCTTR, they analyzed the violin plots directly, and they indicated that they consider the improvement that JSDefer can achieve to be significant.

5.3.6. Threats to validity

There are some threats to validity due to the set-up of the experiments.

1. External validity, questions 2–5: Websites often provide different versions of their website for different browsers, or have browser-dependent behavior. Since we use a specific version of WebKit for our instrumented browser, results for different browsers may differ.

   In practice, one would address this by providing different versions of the website as well. An efficient way of doing this is part of further work.

2. External validity, all questions: The selection of the corpus is not entirely random. That being said, the pages in the corpus were not chosen by technical criteria, and manual examination of some samples indicate that the structure and code quality is quite diverse.

3. Internal validity, question 5: We could not completely control for network delays in the testing set-up.

4. Internal validity, question 2: Due to the set-up of the analysis, we could not ensure that the pages did not change between analysis steps. Thus, in the non-determinism matching step, we may have missed cases. We did cross-check on a few samples, but could not do so exhaustively.

5. Internal validity, question 5: The SSL issue already described above. To ensure validity, we reported data from a known-good subset as well as the complete data.

5.4. Soundness of the analysis

In this section, we sketch the proofs of theorems 6 and 7. We deliberately leave out details of the semantics of web pages, deferring to browser implementations. It can be made more precise by giving a formal specification of the semantics of JavaScript and HTML, as executed by a specific browser. We omit such a specification since it is out of scope for this paper, and would engender a research project in its own right. For some progress towards such semantics, compare Bodin et al. [2014], Bohannon and Pierce [2010], Petrov et al. [2012], Guha et al. [2010] et cetera.

The main ingredient will be two notions. One is commutativity, as defined in Rinard and Diniz [1996]. The other is the notion of observational equivalence.

Definition 5 (Ingredients of the soundness proof) Let a web page be given.

1. The *DOM state* is defined as above.
2. The *page state* is the part of the browser state at a given point in time that contains all the state accessible by a JavaScript script on the page. The page state includes the DOM state as well as the JavaScript environment of the page.
3. A *DOM trace* is a sequence of DOM states that can be constructed by executing the page and appending the current DOM state to the sequence whenever it differs from the last state in the sequence (compare Definition 3).

   The DOM trace of an event is defined similarly by considering only the execution of the event; the DOM trace of a sequence of events is defined in the same way.
4. The *DOM behavior* of a page is the collection of all possible DOM traces for that page. The DOM behavior of events and sequences of events is defined analogously.
5. A page p_1 *observationally refines* a page p_2 iff the DOM behavior from p_1 is included in the DOM behavior of p_2. The same holds for events sequences.

   p_1 and p_2 are *observationally equivalent* iff they observationally refine each other.
6. Two events e_1 and e_2 *commute* if the following two conditions holds starting from any page state s: First, starting from state s, $e_1; e_2$ and $e_2; e_1$ have the same DOM behavior. Second, starting from s, executing $e_1; e_2$ can produce state s' iff executing $e_2; e_1$ can produce state s', and vice versa.

Also, let p be a page and D a set of scripts on p. Let p_D be a version of the page with the scripts in D loaded `defer`. Then D is a safe deferral set if p_D observationally refines p.

With these definitions, we can sketch the proofs of these theorems.

Theorem 8 (Restatement of 6) *Let p and D be given such that*

1. *If $s_i \in D$, then s_i is not DOM-accessing in any execution.*
2. *D is $\preceq$-upward closed.*

Then p_D observationally refines p.

Proof (Proof sketch) By induction over $|D|$. The case $|D| = 0$ is trivial.

Let $e_1, \ldots, e_k$ be a trace of p_D, $s \in D$ the first script in page order that is contained in D, and e_i the event corresponding to the execution of s.

By the induction hypothesis, we known that $p_{D\setminus\{s\}}$ observationally refines p, so it is sufficient to show that p_D observationally refines $p_{D\setminus\{s\}}$. Without loss of generality, assume that $D = \{s\}$. Then $p_{D\setminus\{s\}} = p$.

Let $e_1, \ldots, e_m$ be the maximal prefix of $e_1, \ldots, e_k$ such that $e_1, \ldots, e_m, e'$ would be a valid execution prefix of p, and e' correspond to the execution of s. We have to show

that $e_i\ e_1, \ldots, e_m, e_i, e_{m+1}, \ldots, e_{i-1}$ is a valid execution prefix of p. We again proceed by induction over the length of this sub-sequence. The case where $m + 1 > i - 1$ is trivial. Otherwise, it suffices to show that e_i and e_{i-1} can be exchanged, since the induction hypothesis implies the rest.

Suppose first of all that e_i and e_{i-1} race. Then they can be exchanged, since (by the race condition) $e_1, \ldots, e_{i-2}, e_i, e_{i-1}$ must be a valid execution of p.

Thus, we may assume that e_i and e_{i-1} do not race. We show that they commute, so two conditions must be satisfied:

1. $e_{i-1}; e_i$ and $e_i; e_{i-1}$ must have the same DOM behavior. But since e_i does not access the DOM, its DOM behavior is empty, and the DOM behaviors of both sequences reduce to the DOM behavior of e_{i-1}.

2. Starting from a state s, $e_{i-1}; e_i$ and $e_i; e_{i-1}$ must achieve the same states.

   Suppose e_i and e_{i-1} access the same object, and at least one access is a write. We consider the possible reasons e_{i-1} has been posted:

   - e_{i-1} is an event due to user interaction.

     By the assumptions above, user interaction only takes place after the page has been fully loaded, i.e., `DocumentContentLoaded` handler has been executed. But by the semantics of `defer`, e_i is executed before that handler. So e_{i-1} cannot come before e_i.

   - There is some script s' such that e_{i-1} is transitively posted by s'.

     Suppose first that s' comes after s in the page. Then, by definition of the must-come-before relation, $s \preceq s'$. Thus, $s' \in D$, and by the semantics of `defer`, $e_i \,\mathsf{hb}\, e_{i-1}$ — contradiction.

     Clearly, $s' \neq s$ (because $s = s'$ implies $e_i \,\mathsf{hb}\, e_{i-1}$), so s' comes before s in the page. If e_{i-1} was a near event of s', then s would have to come before s' by the definition of near events, contradiction. Thus, e_{i-1} is a far event of s'. But this implies that e_{i-1} and e_i are racing — contradiction.

   - e_{i-1} is not posted due to user interaction or transitively from one script. The only kind of event left at this point are events corresponding to browser-internal behavior; all of these events only access the DOM and browser-internal state that is not accessible to JavaScript.

     Since e_i and e_{i-1} must access the same object, this implies that they both access a DOM object — contradiction, e_i is not DOM-accessing.

   Thus, e_i and e_{i-1} do not access any shared object, where one of the accesses is a write. At this point, we can apply the Bernstein criteria [Bernstein, 1966] to show commutativity.

In a full proof, one would have to also account for the near events of all scripts; this doesn't pose any major challenges, but complicates the argument with technicalities.

Theorem 9 (Restatement of 7) *Let p and D be given, and $e_1, \ldots, e_n$ be a trace of p.*

1. *If e is a near event of s and accesses the DOM, $s \notin D$.*
2. *If e is involved in a race or has non-deterministic control flow, $s \,\mathsf{hb}\, e$ and s' happens before s in program order (including $s = s'$), $s' \notin D$.*
3. *D is $\preceq$-upward closed.*

Then p_D observationally refines p.

Proof (Proof sketch) We reduce to Theorem 6. It suffices to show that that if $s \in D$, then it is not DOM-accessing in any execution. For this, we show that if a near event e of s is DOM-accessing, then $e = e_i$ for some i with $s \,\mathsf{hb}\, e$ in the given trace.

Using condition (2), we find that all events that are transitively posted by s are deterministic, in the sense that they have only one trace. Since they are not involved in race conditions, we can treat them independently of other events. Furthermore, the "no non-deterministic control flow" condition also implies that their execution is independent of any earlier non-determinism in the execution. Thus, we can assume without loss of generality that there is exactly one execution trace for the near events of s on p. So, if e is DOM-accessing in any trace, it is DOM-accessing in all traces, in particular in $e_1, \ldots, e_n$, and hence $e = e_i$ for some i as above.

The second half of (2) is used to ensure that D is $\preceq$-upward closed for any execution, using a similar determinism argument.

6. Conclusion

In this thesis, I have presented methods for heap-based reasoning about asynchronous programs. The key idea was to introduce a model for program state based on a heap (for handling mutable state) and promises (for handling asynchronous concurrency). This is formalized by a program logic that involves separation logic constructs together with wait permissions, the latter being responsible for modeling state that will become available on task completion.

This logic can be automated using the ALST type system, which allows push-button reasoning about a reasonable fragment of OCaml programs using the Lwt library. The type system can still be improved in substantial ways; we will discuss this below under further work.

Switching from reasoning about single programs to reasoning about program transformations, we find that proving properties such as observational refinement is much trickier for asynchronous programs, since we have to deal with the constrained opportunities for scheduling. Delayed refinement deals with this head-on, by splitting the refinement proof into two phases, which makes the coupling between the two programs being compared much weaker. It builds directly one wait permissions and ALST. In particular, it provides two connected semantic interpretations of the ALST type system. As a trade-off, the increased reasoning power requires us to ensure that the programs being compared must have the same type under ALST. This is unsurprising considering the complex properties we are able to establish.

After setting up a significant amount of theory, we can reap the benefits. The first immediate result is that standard rewriting rules for introducing asynchrony into programs (as given by the DontWaitForMe rewriting system) are sound.

Using all the previous results, we designed an asynchronous loading transformation for web pages. Since static analysis of real-world JavaScript code is infeasible, we designed a dynamic analysis that works well enough in practice. As an important side result, we gained some interesting insight into the behavior of real-world JavaScript code; in particular, we find that non-deterministic behavior of scripts is clearly localized. On the experimental side, we found that optimization is possible for a decent number of scripts, but the performance impact was somewhat disappointing — it seems that the pre-loading features of modern browsers negate most of the gains that could be made with this optimization in place.

After discussing what has been done in this line of work, let me list a number of future directions. I will classify the work into three broad categories: Extensions of ALST, expanding the scope of delayed simulation, and refining the underlying logic to deal with more interesting settings.

Extending ALST As mentioned in the section on ALST limitations, the type system has some limitations that should be removed.

One possible direction is that the current ALST model does not allow for truly higher-order functions in the presence of side effects. This is because we fix the shape of resource expressions in function types. But if we wanted to type functions such as `List.iter`, we would need a way to provide polymorphism in pre- and postconditions. The main issue in making functions polymorphic in resource expressions is that the resource expressions can be interdependent in non-trivial ways (consider the aforementioned `List.iter`); a possible approach to deal with this problem would be to use Bounded Refinement Types [Vazou et al., 2015].

Another direction is to support the full power of wait permissions: Recall that for logic-level wait permissions, we have rules such as SplitWait, that allows us to split a wait permission into two.

Adding rules to split and merge wait permissions requires significant technical work: The current handling of names allocated by tasks is somewhat brittle (using the A parameter of promises and wait permissions), and it is not clear how to scale it to multiple waits for the same tasks. This needs to be replaced with some other way of ensuring that all waits for a given task produce the same names while satisfying allocated name invariants.

Another extension to wait permissions is to allow for *prepare permissions*: Suppose we introduce a new form of resource expression, $\mathsf{Prepare}(\xi, A, \eta, \eta')$, with the following meaning: It describes that a task ξ has been posted, with precondition η and postcondition η' (where the names in A are fresh). We could then specify `post` and `wait` using the following rules (using ALST-style notation):

$$\frac{\Gamma; \eta \vdash e : \text{И} A.\, \tau\langle \eta' \rangle \qquad \xi \text{ fresh}}{\Gamma; \mathsf{emp} \vdash \mathtt{post}\, e : \text{И}\{x\}.\, \mathsf{promise}_{\xi,A}\, \tau\langle \mathsf{Prepare}(\xi, A, \eta, \eta') \rangle}$$

$$\frac{\begin{array}{c}\Gamma; \eta \vdash e : \text{И} A.\, \mathsf{promise}_{\xi,A'}\, \tau\langle \mathsf{Wait}(\xi, A', \eta') * \eta'' \rangle \\ \eta'' \text{ does not contain prepare permissions} \qquad \text{(wf conds)}\end{array}}{\Gamma; \eta \vdash \mathtt{wait}\, e : \text{И} A, A'.\, \tau\langle \eta' \rangle}$$

$$\frac{\Gamma \vdash \eta * \mathsf{Prepare}(\xi, A, \eta, \eta') \text{ wf}}{\Gamma \vdash \eta * \mathsf{Prepare}(\xi, A, \eta, \eta') \preceq \mathsf{Wait}(\xi, A, \eta')}$$

A third direction is to support reasoning about unlimited task posting. The difficulty here is to find a good way to summarize a potentially unbounded number of tasks in such a way that no or little information is lost.

More about delayed simulation While we have presented delayed simulation as a reasoning framework for asynchronous programs, it seems that it is applicable to other problems as well. We conjecture that it would allow reasoning about other non-trivial forms of control flow, such as exception-based control flow, delimited continuations, and coroutines.

Another possible direction would be to reason about pairs of programs that differ in synchronization: For instance, one could use delayed simulation to show that removing locks from a given multi-threaded program does not introduce new behaviors, even in the presence of parallel threads.

Finally, there is a conjectured connection with Lipton reduction: It seems likely that one can prove that `post` statements are right movers. Proving this property could likely be done using a careful application of the R-Commute rule from DontWaitForMe.

Refining the model and the program logic Currently, we assume that the resource transfer between tasks is done in a relatively simple fashion: Resources flow along post and wait lines from one task to the next. But what about, say, shared resources that are used by unrelated tasks? Would an invariant-based approach be strong enough to allow useful reasoning for such cases?

Another assumption is that tasks are only posted by the regular execution of other tasks. But taking JavaScript as an example, we find that often, task are posted when a certain environment event happens. In this case, each event may post multiple tasks, and an event may happen multiple times, posting the same task over and over again. Is the current model sufficient to reason about this? If not, how does it need to be extended?

Finally, in many cases, scheduling of tasks is not completely non-deterministic. Does it help to take scheduling constraints into account?

Bibliography

M. Abadi and G. D. Plotkin. A model of cooperative threads. *Logical Methods in Computer Science*, 6(4), 2010. doi: 10.2168/LMCS-6(4:2)2010. URL `https://doi.org/10.2168/LMCS-6(4:2)2010`.

C. Q. Adamsen, A. Møller, R. Karim, M. Sridharan, F. Tip, and K. Sen. Repairing event race errors by controlling nondeterminism. In *ICSE 2017*, 2017.

S. Agarwal, R. Barik, V. Sarkar, and R. K. Shyamasundar. May-happen-in-parallel analysis of X10 programs. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2007, San Jose, California, USA, March 14-17, 2007*, pages 183–193, 2007. doi: 10.1145/1229428.1229471. URL `http://doi.acm.org/10.1145/1229428.1229471`.

A. J. Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In *ESOP 2006*, pages 69–83, 2006. doi: 10.1007/11693024_6. URL `http://dx.doi.org/10.1007/11693024_6`.

F. Aleen and N. Clark. Commutativity analysis for software parallelization: letting program transformations see the big picture. In *ASPLOS 2009*, pages 241–252, 2009. doi: 10.1145/1508244.1508273. URL `http://doi.acm.org/10.1145/1508244.1508273`.

J. Archibald. JavaScript promises: An introduction. `https://developers.google.com/web/fundamentals/getting-started/primers/promises`, 2017.

async (OCaml). Jane street capital's asynchronous execution library. `https://github.com/janestreet/async`, 2013.

L. Augustsson. Cayenne—a language with dependent types. In *Advanced Functional Programming*, pages 240–267. Springer, 1999.

A. J. Bernstein. Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers*, (5):757–763, 1966.

A. Bichhawat, V. Rajani, D. Garg, and C. Hammer. Information flow control in webkit's javascript bytecode. In *Principles of Security and Trust - Third International Conference, POST 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, pages 159–178, 2014. doi: 10.1007/978-3-642-54792-8_9. URL `http://dx.doi.org/10.1007/978-3-642-54792-8_9`.

G. M. Bierman, M. Abadi, and M. Torgersen. Understanding typescript. In *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings*, pages 257–281, 2014. doi: 10.1007/978-3-662-44202-9_11. URL http://dx.doi.org/10.1007/978-3-662-44202-9_11.

BlockingJS. Remove Render-Blocking JavaScript, Apr 2015. URL https://developers.google.com/speed/docs/insights/BlockingJS.

R. L. Bocchino Jr. Deterministic parallel java. In *Encyclopedia of Parallel Computing*, pages 566–573. Springer, 2011. doi: 10.1007/978-0-387-09766-4_119. URL http://dx.doi.org/10.1007/978-0-387-09766-4_119.

M. Bodin, A. Charguéraud, D. Filaretti, P. Gardner, S. Maffeis, D. Naudziuniene, A. Schmitt, and G. Smith. A trusted mechanised javasript specification. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 87–100, 2014. doi: 10.1145/2535838.2535876. URL http://doi.acm.org/10.1145/2535838.2535876.

A. Bohannon and B. C. Pierce. Featherweight firefox: Formalizing the core of a web browser. In *USENIX Conference on Web Application Development, WebApps'10, Boston, Massachusetts, USA, June 23-24, 2010*, 2010. URL https://www.usenix.org/conference/webapps-10/featherweight-firefox-formalizing-core-web-browser.

A. Bouajjani, M. Emmi, C. Enea, B. K. Ozkan, and S. Tasiran. Verifying robustness of event-driven asynchronous programs against concurrency. In *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, pages 170–200, 2017. doi: 10.1007/978-3-662-54434-1_7. URL https://doi.org/10.1007/978-3-662-54434-1_7.

J. Boyland. Fractional permissions. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, pages 270–288. Springer, 2013. doi: 10.1007/978-3-642-36946-9_10. URL http://dx.doi.org/10.1007/978-3-642-36946-9_10.

S. Brookes. A semantics for concurrent separation logic. *Theor. Comput. Sci.*, 375(1-3): 227–270, 2007. doi: 10.1016/j.tcs.2006.12.034. URL https://doi.org/10.1016/j.tcs.2006.12.034.

S. Brookes and P. W. O'Hearn. Concurrent separation logic. *SIGLOG News*, 3(3):47–65, 2016. doi: 10.1145/2984450.2984457. URL http://doi.acm.org/10.1145/2984450.2984457.

C. Calcagno, P. W. O'Hearn, and H. Yang. Local action and abstract separation logic. In *22nd IEEE Symposium on Logic in Computer Science (LICS 2007), 10-12 July 2007, Wroclaw, Poland, Proceedings*, 2007.

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. doi: 10.1145/2049697.2049700. URL `http://doi.acm.org/10.1145/2049697.2049700`.

P. Cerný, T. A. Henzinger, A. Radhakrishna, L. Ryzhyk, and T. Tarrach. Efficient synthesis for concurrency by semantics-preserving transformations. In *CAV 2013*, pages 951–967, 2013.

P. Charles, C. Grothoff, V. A. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA 2005*, pages 519–538, 2005. doi: 10.1145/1094811.1094852. URL `http://doi.acm.org/10.1145/1094811.1094852`.

R. Chugh and R. Jhala. Dependent types for javascript. *CoRR*, abs/1112.4106, 2011. URL `http://arxiv.org/abs/1112.4106`.

K. E. Coons, M. Musuvathi, and K. S. McKinley. Bounded partial-order reduction. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, pages 833–848, 2013. doi: 10.1145/2509136.2509556. URL `http://doi.acm.org/10.1145/2509136.2509556`.

K. Crary, D. Walker, and J. G. Morrisett. Typed memory management in a calculus of capabilities. In *POPL 1999*, pages 262–275, 1999.

T. Dinsdale-Young, M. Dodds, P. Gardner, M. J. Parkinson, and V. Vafeiadis. Concurrent abstract predicates. In *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings*, pages 504–528, 2010. doi: 10.1007/978-3-642-14107-2_24. URL `https://doi.org/10.1007/978-3-642-14107-2_24`.

T. Dinsdale-Young, L. Birkedal, P. Gardner, M. J. Parkinson, and H. Yang. Views: compositional reasoning for concurrent programs. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 287–300, 2013. doi: 10.1145/2429069.2429104. URL `http://doi.acm.org/10.1145/2429069.2429104`.

D. Distefano, P. W. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *TACAS 2006*, pages 287–302, 2006. doi: 10.1007/11691372_19.

M. Dodds, X. Feng, M. J. Parkinson, and V. Vafeiadis. Deny-guarantee reasoning. In *ESOP 2009*, pages 363–377, 2009. URL `http://dx.doi.org/10.1007/978-3-642-00590-9_26`.

J. E. Donahue and A. J. Demers. Data types are values. *ACM Trans. Program. Lang. Syst.*, 7(3):426–445, 1985. doi: 10.1145/3916.3987. URL `http://doi.acm.org/10.1145/3916.3987`.

D. Dreyer, G. Neis, and L. Birkedal. The impact of higher-order state and control effects on local relational reasoning. *Journal of Functional Programming*, 22(4-5):477–528, 2012. doi: 10.1017/S095679681200024X.

ES6. *ECMAScript 2015 Language Specification – ECMA-262 6th Edition*. ECMA International, Rue du Rhone 114, CH-1204 Geneva, 2015.

A. El-Hassany, J. Miserez, P. Bielik, L. Vanbever, and M. T. Vechev. Sdnracer: concurrency analysis for software-defined networks. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 402–415, 2016. doi: 10.1145/2908080.2908124. URL `http://doi.acm.org/10.1145/2908080.2908124`.

M. Emmi, S. Qadeer, and Z. Rakamaric. Delay-bounded scheduling. In *POPL 2011*, pages 411–422, 2011.

ESOP-Coq. Coq development for dontwaitforme. `https://github.com/johanneskloos/esop`, 2017.

I. Facebook. flow – a static type checker for javascript, 2016. URL `https://flowtype.org`.

L. FAINBERG, O. Ehrlich, G. Shai, O. Gadish, A. DOBO, and O. Berger. Systems and methods for acceleration and optimization of web pages access by changing the order of resource loading, Feb. 3 2011. URL `https://www.google.com/patents/US20110029899`. US Patent App. 12/848,559.

J. Fischer, R. Majumdar, and T. D. Millstein. Tasks: language support for event-driven programming. In *PEPM 2007*, pages 134–143, 2007. doi: 10.1145/1244381.1244403. URL `http://doi.acm.org/10.1145/1244381.1244403`.

C. Flanagan and S. Qadeer. Thread-modular model checking. In *SPIN 2003*, pages 213–224, 2003.

A. Flores-Montoya, E. Albert, and S. Genaim. May-happen-in-parallel based deadlock analysis for concurrent objects. In *Formal Techniques for Distributed Systems - Joint IFIP WG 6.1 International Conference, FMOODS/FORTE 2013, Held as Part of the 8th International Federated Conference on Distributed Computing Techniques, DisCoTec 2013, Florence, Italy, June 3-5, 2013. Proceedings*, pages 273–288, 2013. doi: 10.1007/978-3-642-38592-6_19. URL `https://doi.org/10.1007/978-3-642-38592-6_19`.

Fortune 500. Fortune 500, 2016. URL `http://beta.fortune.com/fortune500/`.

T. Freeman and F. Pfenning. Refinement types for ML. In *PLDI*, pages 268–277, 1991.

R. Fuhrer and V. Saraswat. Concurrency refactoring for x10. In *3rd ACM Workshop on Refactoring Tools*, pages 1–4, 2009.

P. Ganty, R. Majumdar, and A. Rybalchenko. Verifying liveness for asynchronous programs. In *POPL 2009*, pages 102–113, 2009.

Q. Gao, P. Dey, and P. Ahammad. Perceived performance of webpages in the wild: Insights from large-scale crowdsourcing of above-the-fold QoE, 2017. arXiv:1704.01220.

I. Gavran, F. Niksic, A. Kanade, R. Majumdar, and V. Vafeiadis. Rely/guarantee reasoning for asynchronous programs. In *26th International Conference on Concurrency Theory, CONCUR 2015, Madrid, Spain, September 1.4, 2015*, pages 483–496, 2015. doi: 10.4230/LIPIcs.CONCUR.2015.483. URL `https://doi.org/10.4230/LIPIcs.CONCUR.2015.483`.

J.-Y. Girard. Une extension de l'interpretation de gödel a l'analyse, et son application a l'elimination des coupures dans l'analyse et la theorie des types. *Studies in Logic and the Foundations of Mathematics*, 63:63–92, 1971.

go. The Go programming language. `http://golang.org/`, 2012—2017.

P. Godefroid. Using partial orders to improve automatic verification methods. In *Computer Aided Verification, 2nd International Workshop, CAV '90, New Brunswick, NJ, USA, June 18-21, 1990, Proceedings*, pages 176–185, 1990. doi: 10.1007/BFb0023731. URL `https://doi.org/10.1007/BFb0023731`.

I. Google. Closure tools, 2016. URL `https://developers.google.com/closure/`.

GTK. The GTK+ project. `http://www.gtk.org/`, 2007—2017.

A. Guha, C. Saftoiu, and S. Krishnamurthi. The essence of javascript. In *ECOOP 2010*, pages 126–150, 2010.

A. Hejlsberg, S. Wiltamuth, and P. Golde. *C# language specification*. Addison-Wesley Longman Publishing Co., Inc., 2003.

T. A. Henzinger, R. Jhala, R. Majumdar, and S. Qadeer. Thread-modular abstraction refinement. In *CAV 2003*, pages 262–274, 2003.

C. Hsiao, C. Pereira, J. Yu, G. Pokam, S. Narayanasamy, P. M. Chen, Z. Kong, and J. Flinn. Race detection for event-driven mobile applications. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pages 326–336, 2014. doi: 10.1145/2594291.2594330. URL `http://doi.acm.org/10.1145/2594291.2594330`.

Infer. Infer static analyzer. `https://fbinfer.com`, 2017.

Intents. Intents and intent filters | android developers. `https://developer.android.com/guide/components/intents-filters.html`, 2017.

S. S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK, January 17-19, 2001*, pages 14–26, 2001.

B. Jacobs, J. Smans, and F. Piessens. A quick tour of the verifast program verifier. In *APLAS 2010*, pages 304–311, 2010.

C. S. Jensen, A. Møller, V. Raychev, D. Dimitrov, and M. T. Vechev. Stateless model checking of event-driven applications. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, pages 57–73, 2015. doi: 10.1145/2814270.2814282. URL `http://doi.acm.org/10.1145/2814270.2814282`.

S. H. Jensen, A. Møller, and P. Thiemann. Type analysis for javascript. In *Static Analysis, 16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009. Proceedings*, pages 238–255, 2009.

S. H. Jensen, M. Madsen, and A. Møller. Modeling the HTML DOM and browser API in static analysis of javascript web applications. In *FSE'11 and ESEC'11*, pages 59–69, 2011.

S. H. Jensen, P. A. Jonsson, and A. Møller. Remedying the eval that men do. In *ISSTA 2012*, pages 34–44, 2012.

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

A. Jung and J. Tiuryn. A new characterization of lambda definability. In *TLCA '93*, pages 245–257, 1993. doi: 10.1007/BFb0037110. URL `http://dx.doi.org/10.1007/BFb0037110`.

R. Jung, D. Swasey, F. Sieczkowski, K. Svendsen, A. Turon, L. Birkedal, and D. Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *POPL '15*, pages 637–650, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3300-9. doi: 10.1145/2676726.2676980. URL `http://doi.acm.org/10.1145/2676726.2676980`.

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

A. Kaiser, D. Kroening, and T. Wahl. Efficient coverability analysis by proof minimization. In *CONCUR 2012 - Concurrency Theory - 23rd International Conference, CONCUR 2012, Newcastle upon Tyne, UK, September 4-7, 2012. Proceedings*, pages 500–515, 2012. doi: 10.1007/978-3-642-32940-1_35. URL `https://doi.org/10.1007/978-3-642-32940-1_35`.

V. Kashyap, K. Dewey, E. A. Kuefner, J. Wagner, K. Gibbons, J. Sarracino, B. Wiedermann, and B. Hardekopf. JSAI: a static analysis platform for javascript. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, pages 121–132, 2014. doi: 10.1145/2635868.2635904. URL `http://doi.acm.org/10.1145/2635868.2635904`.

M. Kawaguchi, P. M. Rondon, A. Bakst, and R. Jhala. Deterministic parallelism via liquid effects. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 45–54, 2012. doi: 10.1145/2254064.2254071. URL `http://doi.acm.org/10.1145/2254064.2254071`.

J. Kloos and R. Majumdar. Proving the soundness of asynchronous program transformations. Technical report, MPI-SWS, 2018.

J. Kloos, R. Majumdar, and V. Vafeiadis. Asynchronous Liquid Separation Types. In *ECOOP 2015*, volume 37 of *LIPIcs*, pages 396–420. Dagstuhl, 2015. URL `http://dx.doi.org/10.4230/LIPIcs.ECOOP.2015.396`. full version at `http://www.mpi-sws.org/~jkloos/ALST-full.pdf`.

J. Kloos, R. Majumdar, and F. McCabe. Deferrability analysis for javascript. In *Hardware and Software: Verification and Testing - 13th International Haifa Verification Conference, HVC 2017, Haifa, Israel, November 13-15, 2017, Proceedings*, pages 35–50, 2017. doi: 10.1007/978-3-319-70389-3_3. URL `https://doi.org/10.1007/978-3-319-70389-3_3`.

R. Krebbers, R. Jung, A. Bizjak, J. Jourdan, D. Dreyer, and L. Birkedal. The essence of higher-order concurrent separation logic. In *ESOP 2017*, pages 696–723, 2017. doi: 10.1007/978-3-662-54434-1_26. URL `https://doi.org/10.1007/978-3-662-54434-1_26`.

M. Krogh-Jespersen, K. Svendsen, and L. Birkedal. A relational model of types-and-effects in higher-order concurrent separation logic. In *POPL 2017*, POPL 2017, pages 218–231, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4660-3. doi: 10.1145/3009837.3009877. URL `http://doi.acm.org/10.1145/3009837.3009877`.

M. N. Krohn, E. Kohler, and M. F. Kaashoek. Events can make sense. In *Proceedings of the 2007 USENIX Annual Technical Conference, Santa Clara, CA, USA, June 17-22, 2007*, pages 87–100, 2007. URL `http://www.usenix.org/events/usenix07/tech/krohn.html`.

B. Kuhn, K. Marifet, and J. Wogulis. Asynchronous loading of scripts in web pages, Apr. 29 2014. URL `https://www.google.com/patents/US8713424`. US Patent 8,713,424.

J. K. Lee, J. Palsberg, R. Majumdar, and H. Hong. Efficient may happen in parallel analysis for async-finish parallelism. In *Static Analysis - 19th International Symposium, SAS 2012, Deauville, France, September 11-13, 2012. Proceedings*, pages 5–23, 2012. doi: 10.1007/978-3-642-33125-1_4. URL `https://doi.org/10.1007/978-3-642-33125-1_4`.

K. R. M. Leino. Verifying concurrent programs with chalice. In *Verification, Model Checking, and Abstract Interpretation, 11th International Conference, VMCAI 2010, Madrid, Spain, January 17-19, 2010. Proceedings*, page 2, 2010. doi: 10.1007/978-3-642-11319-2_2. URL `https://doi.org/10.1007/978-3-642-11319-2_2`.

X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009. doi: 10.1145/1538788.1538814. URL `http://doi.acm.org/10.1145/1538788.1538814`.

H. Liang, X. Feng, and Z. Shao. Compositional verification of termination-preserving refinement of concurrent programs. In *CSL-LICS '14*, pages 65:1–65:10, 2014. doi: 10.1145/2603088.2603123. URL `http://doi.acm.org/10.1145/2603088.2603123`.

E. Lipton, B. Roy, S. Calvert, M. Gibbs, N. Kothari, M. Harder, and D. Reed. Dynamically loading scripts, Mar. 30 2010. URL `https://www.google.com/patents/US7689665`. US Patent 7,689,665.

R. J. Lipton. Reduction: A new method of proving properties of systems of processes. In *Conference Record of the Second ACM Symposium on Principles of Programming Languages, Palo Alto, California, USA, January 1975*, pages 78–86, 1975. doi: 10.1145/512976.512985. URL `http://doi.acm.org/10.1145/512976.512985`.

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

V. B. Livshits and E. Kiciman. Doloto: code splitting for network-bound web 2.0 applications. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2008, Atlanta, Georgia, USA, November 9-14, 2008*, pages 350–360, 2008. doi: 10.1145/1453101.1453151. URL `http://doi.acm.org/10.1145/1453101.1453151`.

N. A. Lynch and F. W. Vaandrager. Forward and backward simulations: I. untimed systems. *Inf. Comput.*, 121(2):214–233, 1995. doi: 10.1006/inco.1995.1134. URL `https://doi.org/10.1006/inco.1995.1134`.

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

S. Maffeis, J. C. Mitchell, and A. Taly. An operational semantics for javascript. In *Programming Languages and Systems, 6th Asian Symposium, APLAS 2008, Bangalore, India, December 9-11, 2008. Proceedings*, pages 307–325, 2008. doi: 10.1007/978-3-540-89330-1_22. URL `http://dx.doi.org/10.1007/978-3-540-89330-1_22`.

P. Maiya, A. Kanade, and R. Majumdar. Race detection for android applications. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pages 316–325, 2014. doi: 10.1145/2594291.2594311. URL `http://doi.acm.org/10.1145/2594291.2594311`.

R. Majumdar and Z. Wang. Bbs: A phase-bounded model checker for asynchronous programs. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, pages 496–503, 2015.

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

S. Markstrum, R. M. Fuhrer, and T. D. Millstein. Towards concurrency refactoring for x10. In *PPOPP 2009*, pages 303–304, 2009. doi: 10.1145/1504176.1504226. URL `http://doi.acm.org/10.1145/1504176.1504226`.

S. Marlow. The async package. `https://hackage.haskell.org/package/async`, 2012.

N. D. Matsakis and F. S. K. II. The rust language. In *Proceedings of the 2014 ACM SIGAda annual conference on High integrity language technology, HILT 2014, Portland, Oregon, USA, October 18-21, 2014*, pages 103–104, 2014. doi: 10.1145/2663171.2663188. URL `http://doi.acm.org/10.1145/2663171.2663188`.

The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2012. URL `http://coq.inria.fr`. Version 8.4.

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

K. Molitorisz, J. Schimmel, and F. Otto. Automatic parallelization using autofutures. In *MSEPT 2012*, pages 78–81, 2012. doi: 10.1007/978-3-642-31202-1_8. URL `https://doi.org/10.1007/978-3-642-31202-1_8`.

M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, pages 446–455, 2007. doi: 10.1145/1250734.1250785. URL `http://doi.acm.org/10.1145/1250734.1250785`.

E. Mutlu, S. Tasiran, and B. Livshits. Detecting javascript races that matter. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, pages 381–392, 2015. doi: 10.1145/2786805.2786820. URL `http://doi.acm.org/10.1145/2786805.2786820`.

A. Nanevski, G. Morrisett, and L. Birkedal. Polymorphism and separation in hoare type theory. In *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming, ICFP 2006, Portland, Oregon, USA, September 16-21, 2006*, pages 62–73, 2006. doi: 10.1145/1159803.1159812. URL `http://doi.acm.org/10.1145/1159803.1159812`.

G. Naumovich and G. S. Avrunin. A conservative data flow algorithm for detecting all pairs of statement that may happen in parallel. In *SIGSOFT '98, Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering, Lake Buena Vista, Florida, USA, November 3-5, 1998*, pages 24–34, 1998. doi: 10.1145/288195.288213. URL `http://doi.acm.org/10.1145/288195.288213`.

G. Naumovich, G. S. Avrunin, and L. A. Clarke. An efficient algorithm for computing *MHP* information for concurrent java programs. In *Software Engineering - ESEC/FSE'99, 7th European Software Engineering Conference, Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Toulouse, France, September 1999, Proceedings*, pages 338–354, 1999a. doi: 10.1007/3-540-48166-4_21. URL `https://doi.org/10.1007/3-540-48166-4_21`.

G. Naumovich, G. S. Avrunin, and L. A. Clarke. Data flow analysis for checking properties of concurrent java programs. In *Proceedings of the 1999 International Conference on Software Engineering, ICSE' 99, Los Angeles, CA, USA, May 16-22, 1999.*, pages 399–410, 1999b. URL `http://portal.acm.org/citation.cfm?id=302405.302663`.

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

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

OKWS. The OK web server. `https://github.com/okws/okws`, 2011.

S. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Inf.*, 6:319–340, 1976.

B. Petrov, M. T. Vechev, M. Sridharan, and J. Dolby. Race detection for web applications. In *PLDI 2012*, pages 251–262, 2012. doi: 10.1145/2254064.2254095. URL `http://doi.acm.org/10.1145/2254064.2254095`.

B. C. Pierce. *Types and programming languages*. MIT Press, 2002. ISBN 978-0-262-16209-8.

A. M. Pitts and I. D. B. Stark. Observable properties of higher order functions that dynamically create local names, or what's new? In *MFCS'93*, pages 122–141, 1993. doi: 10.1007/3-540-57182-5_8. URL `https://doi.org/10.1007/3-540-57182-5_8`.

G. D. Plotkin. Lambda-definability and logical relations. Technical report, SAI-RM Memorandum 4, 1973.

A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *TACAS '98*, pages 151–166, 1998. doi: 10.1007/BFb0054170. URL `http://dx.doi.org/10.1007/BFb0054170`.

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

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

S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *TACAS 2005*, pages 93–107, 2005.

A. Raad, J. F. Santos, and P. Gardner. DOM: specification and client reasoning. In *Programming Languages and Systems - 14th Asian Symposium, APLAS 2016, Hanoi, Vietnam, November 21-23, 2016, Proceedings*, pages 401–422, 2016. doi: 10.1007/978-3-319-47958-3_21. URL http://dx.doi.org/10.1007/978-3-319-47958-3_21.

V. Raychev, M. T. Vechev, and M. Sridharan. Effective race detection for event-driven programs. In *OOPSLA 2013*, pages 151–166, 2013. doi: 10.1145/2509136.2509538. URL http://doi.acm.org/10.1145/2509136.2509538.

M. Raza, C. Calcagno, and P. Gardner. Automatic parallelization with separation logic. In *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, pages 348–362, 2009. doi: 10.1007/978-3-642-00590-9_25. URL http://dx.doi.org/10.1007/978-3-642-00590-9_25.

J. C. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP Congress*, pages 513–523, 1983.

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

M. C. Rinard and P. C. Diniz. Commutativity analysis: A new analysis framework for parallelizing compilers. In *PLDI 1996*, pages 54–67, 1996. doi: 10.1145/231379.231390. URL http://doi.acm.org/10.1145/231379.231390.

E. Ritter and A. M. Pitts. A fully abstract translation between a lambda-calculus with reference types and standard ML. In *TLCA '95*, pages 397–413, 1995. doi: 10.1007/BFb0014067. URL https://doi.org/10.1007/BFb0014067.

P. M. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *PLDI 2008*, pages 159–169, 2008.

P. M. Rondon, M. Kawaguchi, and R. Jhala. Low-level liquid types. In *POPL 2010*, pages 131–144, 2010.

J. F. Santos, P. Maksimović, D. Naudžiūnienė, T. Wood, and P. Gardner. Javert: Javascript verification toolchain. In *POPL 2018*, 2018. To be published.

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

K. Sen, S. Kalasapur, T. G. Brutch, and S. Gibbs. Jalangi: a selective record-replay and dynamic analysis framework for javascript. In *ESEC/FSE'13*, pages 488–498, 2013.

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

S. Souders. High-performance web sites. *Commun. ACM*, 51(12):36–41, Dec. 2008. ISSN 0001-0782. doi: 10.1145/1409360.1409374. URL `http://doi.acm.org/10.1145/1409360.1409374`.

R. Surendran and V. Sarkar. Automatic parallelization of pure method calls via conditional future synthesis. In *OOPSLA 2016*, pages 20–38, 2016. doi: 10.1145/2983990.2984035. URL `http://doi.acm.org/10.1145/2983990.2984035`.

Swing. Creating a GUI with jfc/swing. `http://docs.oracle.com/javase/tutorial/uiswing/index.html`, 1995—2017.

P. Thiemann. Towards a type system for analyzing javascript programs. In *Programming Languages and Systems, 14th European Symposium on Programming,ESOP 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, pages 408–422, 2005. doi: 10.1007/978-3-540-31987-0_28. URL `http://dx.doi.org/10.1007/978-3-540-31987-0_28`.

A. Turon, D. Dreyer, and L. Birkedal. Unifying refinement and hoare-style reasoning in a logic for higher-order concurrency. In *ICFP 2013*, ICFP '13, pages 377–390, New York, NY, USA, 2013a. ACM. ISBN 978-1-4503-2326-0. doi: 10.1145/2500365.2500600. URL `http://doi.acm.org/10.1145/2500365.2500600`.

A. Turon, J. Thamsborg, A. Ahmed, L. Birkedal, and D. Dreyer. Logical relations for fine-grained concurrency. In *POPL 2013*, POPL '13, pages 343–356, New York, NY, USA, 2013b. ACM. ISBN 978-1-4503-1832-7. doi: 10.1145/2429069.2429111. URL `http://doi.acm.org/10.1145/2429069.2429111`.

V. Vafeiadis and M. J. Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR 2007 - Concurrency Theory, 18th International Conference, CONCUR 2007, Lisbon, Portugal, September 3-8, 2007, Proceedings*, pages 256–271, 2007. doi: 10.1007/978-3-540-74407-8_18. URL `https://doi.org/10.1007/978-3-540-74407-8_18`.

N. Vazou, P. M. Rondon, and R. Jhala. Abstract refinement types. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, pages 209–228, 2013. doi: 10.1007/978-3-642-37036-6_13. URL `https://doi.org/10.1007/978-3-642-37036-6_13`.

N. Vazou, A. Bakst, and R. Jhala. Bounded refinement types. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*, pages 48–61, 2015. doi: 10.1145/2784731.2784745. URL `http://doi.acm.org/10.1145/2784731.2784745`.

R. Viscomi, A. Davies, and M. Duran. *Using WebPageTest: Web Performance Testing for Novices and Power Users*. O'Reilly Media, Inc., 1st edition, 2015. ISBN 1491902590, 9781491902592.

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

WHATWG. HTML – Living Standard, Sep 2016. `https://html.spec.whatwg.org/multipage/`.

T. Witkowski, N. Blanc, D. Kroening, and G. Weissenbacher. Model checking concurrent linux device drivers. In *(ASE 2007)*, pages 501–504, 2007.

A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, 1994. doi: 10.1006/inco.1994.1093. URL `https://doi.org/10.1006/inco.1994.1093`.

X. Wu, Y. Wen, L. Chen, W. Dong, and J. Wang. Data race detection for interrupt-driven programs via bounded model checking. In *SERE 2012 – Companion volume*, pages 204–210, 2013.

X10. SatX10: A scalable plug & play parallel solver. http://x10-lang.org/satx10, 2012.

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

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

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

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

XMLHttpRequest. Using xmlhttprequest. `https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest/Using_XMLHttpRequest`, 2017.

Appendix

A. Type safety for ALST

This section sketches the type safety proof for the core calculus.

A.1. Adapting the type system

For the type safety proof, we slightly extend the notion of names to include *committed names*: Define

$$A ::= \cdot \mid \xi, \hat{A} \mid c : \xi, \hat{A}$$

Here, A contains two types of name bindings: ξ binds a name that can be freely chosen as long as it is fresh with regard to an environment Γ and a name environment χ (i.e., ξ is not bound in Γ and $\xi \notin \operatorname{im} \chi$). Conversely, $c : \xi$ denotes a *committed name*: This name has been chosen at some other point, but the corresponding resource has not yet been allocated. This situation occurs when an asynchronous task allocates resources. Consider the following example expression e:

```
let h = post (ref 1) in wait h
```

In this example, **post (ref 1)** types as

$$\cdot \mid \varnothing \mid \varnothing; \mathsf{emp} \vdash \textbf{post (ref 1)} : И\pi, \mu.\, \mathsf{promise}_\pi\, \mathsf{ref}_\mu\, \mathtt{int}\langle \mathsf{Wait}(\pi, \mu \mapsto \mathtt{int}_{=1})\rangle$$

Consider now the global configuration $(\varnothing, \{p_0 \mapsto e\}, p_0)$. Taking one step produces the following configuration: $(\varnothing, \{p_0 \mapsto e_0, p_1 \mapsto e_1\}, p_0)$ where $e_0 :=$ **let h =**p_1**in wait h** and $e_1 :=$**ref 1**.

Let $\omega := \{p_1 \mapsto \mathsf{ref}_\mu\, \mathtt{int}\}$ and $\chi := \{p_1 \mapsto \pi\}$. By the existing typing rule for **post**, we have that

$$\mu \mid \omega \mid \chi; e_0 \vdash \mathsf{ref}_\mu\, \mathtt{int}\langle \mu \mapsto \mathtt{int}_{=1}\rangle : .$$

Also, we have that $\cdot \mid \omega \mid \chi; \mathsf{emp} \vdash e_1 : И\mu.\, \mathsf{ref}_\mu\, \mathtt{int}\langle \mu \mapsto \mathtt{int}_{=1}\rangle$.

To be able to prove type preservation, some way is needed to ensure that the μ in the typing of e_0 and the μ in the typing of e_1 coincide, so that the references to μ in both refer to the same resource. This is where committed names come in: By making the resource names committed, the choice of μ in the example is limited so that the names stay coherent. This is achieved by modifying the definition of the typing rule for **post** as in Figure A.1. Using these rules, we can type e_1 as follows: $\cdot \mid \omega \mid (\chi_1, \{\mu\}); \mathsf{emp} \vdash e_1 : И c : \mu.\, \mathsf{ref}_\mu\, \mathtt{int}\langle \mu \mapsto \mathtt{int}_{=1}\rangle$.

$$\text{T-Post}\quad \frac{\Gamma \mid \omega \mid \chi; \eta \vdash e : \text{И}A.\, \tau\langle\eta'\rangle \qquad \pi \text{ fresh resource name variable}}{\Gamma \mid \omega \mid \chi; \eta \vdash \mathtt{post}\, e : \text{И}A^c, \pi.\, \mathsf{promise}_\pi\, \tau\langle\mathsf{Wait}(\pi, \eta')\rangle}$$

where

$$A^c = \begin{cases} \cdot & A = \cdot \\ c : \xi, A'^c & A = \xi, A' \\ c : \xi, A'^c & A = c : \xi, A' \end{cases}$$

Figure A.1.: Typing of `post`, using committed names. In this rule and T-Ref, freshness is defined as follows: ξ is a fresh resource name variable with regard to Γ and χ if ξ is not bound in Γ and $\xi \notin \mathrm{Alloc}(\chi)$, and $c : \xi$ is always fresh.

A.2. The statement of type preservation

For the type preservation proof, as sketched in section 3.2.3, typing is extended to *configuration typing*, as described by the following elaboration of the above-mentioned section:

Let three functions be given: 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. In particular, it is a map $\gamma : \mathsf{Locs} \cup \mathsf{Tasks} \rightharpoonup_{\text{fin}} \tau \cup \varphi$, where $\operatorname{im} \gamma|_{\mathsf{Locs}} \subseteq \tau$ and $\operatorname{im} \gamma|_{\mathsf{Tasks}} \subseteq \varphi$.

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 the same that was introduced in the previous section.

Configuration typing is then defined as follows:

$$\frac{\begin{array}{c}\text{For all } \ell \in \operatorname{dom} H : \cdot \mid \omega(\gamma) \mid \chi \vdash H(\ell) : \psi(\ell) \\ \text{For all } p \text{ such that } P(p) = \mathsf{run}\colon e, \cdot \mid \omega(\gamma) \mid \chi; \psi(p) \vdash e : \gamma(p) \\ \text{For all } p \text{ such that } P(p) = \mathsf{done}\colon v, \cdot \mid \omega(\gamma) \mid \chi; \psi(p) \vdash v : \gamma(p)\end{array}}{\psi, \chi \vdash (H, P, p) : \gamma}$$

where $\omega(\gamma)(\ell) := \gamma(\ell)$

$\omega(\gamma)(p) := \tau$ when $p \neq p_0$ and $\psi(p) = \text{И}A.\, \tau\langle\eta\rangle$

Note that for $\omega(\gamma)$ to be well-defined, we need to ensure that the τ in the definition is independent from the choice of names in A. This is achieved by using committed names: One invariant that will be shown in the type preservation proof will be that A only contains committed names, which implies that τ is actually fully determined.

The intuition behind configuration typing is that 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.

Wellformedness is rather complex: Let γ, ψ and χ. Then the following conditions must hold for γ, ψ and χ to be wellformed:

1. γ, ψ and χ describe the same sets of resources:

   $\operatorname{dom}\gamma = \operatorname{dom}\psi = \operatorname{dom}\chi.1$.

2. The strong types of heap cells match their weak (reference) types:

   For all $\ell \in \operatorname{dom}\gamma$, $\cdot \vdash \psi(\ell) \preceq \gamma(\ell)$.

3. Resources in preconditions exist:

   Define $\text{toplocs}(p) := \{\xi \mid \psi(p) = \xi \mapsto \_ * \_\}$ and $\text{toptasks}(p) := \{\xi \mid \psi(p) = \mathsf{Wait}(\xi, \_) * \_\}$.

   Then for all $p \in \operatorname{dom}\psi$: If $\xi \in \text{toplocs}(p)$, then there is some ℓ such that $\chi(\ell) = \xi$, and if $\xi \in \text{toptasks}(p)$, there is some p' such that $\chi(p') = \xi$.

4. Names are unique: $\chi.1$ is injective.

5. Resources are owned by exactly one active task:

   Define the set of *statically active tasks* A inductively as follows: The initial tasks p_0 is active, $p_0 \in A$. Furthermore, for any task $p \in A$, if $\xi \in \text{toptasks}(p)$ and $\chi(p') = \xi$, then $p' \in A$.

6. Post conditions have only committed names:

   For all tasks $p \neq p_0$, if $\gamma(p) = \text{И}A.\,\tau\langle\eta\rangle$, then A contains only committed names.

7. Wait permissions fit with actual postconditions:

   Let $p_1, p_2 \in \operatorname{dom}\gamma \cap \mathsf{Tasks}$ be two tasks such that $\xi := \chi(p_2) \in \text{toptasks}(p_1)$, and $\eta := \psi(p_1)(\xi)$. Then $\gamma(p_2) = \text{И}A.\,\tau\langle\eta\rangle$ for some A and τ.

Finally, specialization of full types can be defined. 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 types γ and γ', and two name maps χ and χ', we say that (γ, χ) *specialize 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 $\varphi = \text{И}A.\,\tau\langle\eta\rangle$ and $\varphi' = \text{И}A'.\,\tau'\langle\eta'\rangle$. Then there are substitution σ, σ' such that $\tau\sigma = \tau'\sigma'$, $\eta\sigma = \eta'\sigma'$, and σ, σ' only map non-committed names.

The type safety theorem is then given as:

Theorem 10 (Type safety, Theorem 1) *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.

The proof is performed using a standard preservation/progress argument. The key theorems can be stated as follows:

Theorem 11 (Preservation) *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 $(\gamma, \psi) \rhd (\gamma', \psi')$.

Theorem 12 (Progress) *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 there are two possibilities: Either all processes in P' have reduced to values (i.e., for all $p \in \operatorname{dom} P$, $P(p) = \mathsf{done}: : v$ or $P(p) = \mathsf{run}: : v$), or there are (H', P', p') such that $(H, P, p) \hookrightarrow (H', P', p')$.

A further important observation is the following:

Theorem 13 (Initialization) *Suppose an expression can be typed as $\cdot; \mathsf{emp} \vdash e : \varphi$ (this can generally be assumed at the start of the program). Define $\gamma := \{p_0 \mapsto \varphi\}$, $\psi := \{p_0 \mapsto \mathsf{emp}\}$, $\chi := (\varnothing, \varnothing)$. Then $\psi, \chi \vdash (\varnothing, \{p_0 \mapsto \mathsf{run}: : e\}, p_0) : \gamma$ and ψ, χ, γ wf.*

The proofs of the progress and initialization theorems are entirely routine. The rest of this appendix gives the main points of the preservation proof.

A.3. The type preservation proof

Type preservation is shown in two steps: First, type preservation is shown for local steps, and this result is then used to prove global type preservation.

The following lemma is one of the main results needed for local type preservation:

Lemma 10 (Value typing) *Suppose $\Gamma \mid \omega \mid \chi; \eta \vdash v : И A. \tau\langle\eta'\rangle$. Then: $\Gamma, \chi \vdash \eta \preceq \eta'$ and $\Gamma \mid \omega \mid \chi \vdash v : \tau$.*

This lemma can be shown by induction over the derivation of $\Gamma \mid \omega \mid \chi; \eta \vdash v : И A. \tau\langle\eta'\rangle$.

Furthermore, the following proof-theoretic results can be easily shown:

Lemma 11 (Weakening) *If $\Gamma \mid \omega \mid \chi; \eta \vdash e : \varphi$ and $\Gamma' \supseteq \Gamma$, $\omega' \supseteq \omega$, $\chi' \supseteq \chi a$, and $\Gamma | \omega, \chi$ is wellformed, then $\Gamma' \mid \omega' \mid \chi'; \eta \vdash e : \varphi$.*

Lemma 12 (Comitting names) *If $\Gamma \mid \omega \mid \chi; \eta \vdash e : И A. \tau\langle\eta'\rangle$, $\Gamma \mid \omega \mid \chi \cup A; \eta \vdash e : И c : A. \tau\langle\eta'\rangle$. Here, $\chi \cup A := (\chi_1, \chi_2 \cup A)$.*

Lemma 13 (Substitutions) *Suppose $\Gamma, x : \tau \mid \omega \mid \chi; \eta \vdash e : \varphi$. If $x \notin \operatorname{freenames} \eta$, $x \notin \operatorname{freenames} \varphi$ and $\Gamma' \mid \omega \mid \chi \vdash v : \tau$, then $\Gamma \mid \omega \mid \chi; \eta \vdash e[v/x] : \varphi$.*

To prove a sufficiently strong version of local type preservation, the following definition is needed:

Definition 6 Let (e, H, P) be a local configuration, p a task identifier, $\Gamma, \gamma, \chi, \psi$ be given.

Then ψ, χ and γ *type the local configuration* (e, H, P), written as $\Gamma, \psi, \chi \vdash_p (e, H, P) : \gamma$, if

$$\frac{\begin{array}{c} p \in \operatorname{dom}\gamma \qquad \text{For all } \ell \in \operatorname{dom} H : \Gamma \mid \omega(\gamma) \mid \chi \vdash H(\ell) : \psi(\ell) \\ \text{For all } p' \neq p \text{ such that } P(p') = \mathsf{run}\colon : e : \Gamma \mid \omega(\gamma) \mid \chi; \psi(p') \vdash e : \gamma(p') \\ \text{For all } p' \neq p \text{ such that } P(p') = \mathsf{done}\colon : v : \Gamma \mid \omega(\gamma) \mid \chi; \psi(p') \vdash v : \gamma(p') \end{array}}{\Gamma, \psi, \chi \vdash_p (e, H, P) : \gamma}$$

Furthermore, $\Gamma \vdash \psi, \chi, \gamma$ wf if conditions 1, 3, 4, 5, 6, 7 from ψ, χ, γ wf hold and for all $\ell \in \operatorname{dom}\gamma$, $\Gamma \vdash \chi(\ell) \preceq \gamma(\ell)$.

Theorem 14 (Local type preservation) *Let (e, H, P) be a local configuration, p a task identifier such that $p \notin \operatorname{dom} P$, $\Gamma, \gamma, \psi, \chi, \eta, \varphi$ be given. Suppose*

1. $\Gamma \mid \omega\gamma \mid \chi; \eta \vdash e : \varphi$,
2. $\Gamma, \psi, \chi \vdash_p (e, H, P) : \gamma$,
3. $\Gamma \vdash \psi, \chi, \gamma$ *wf.*

Suppose furthermore that there is a local configuration (e', H', P') such that $(e, H, P) \hookrightarrow_l (e', H', P')$.

Then there are $\gamma', \psi', \chi', \eta', \varphi'$ such that:

1. φ' *specializes* φ,
2. $(\gamma, \chi) \rhd (\gamma', \chi')$,
3. $\Gamma \mid \omega\gamma' \mid \chi'; \eta' \vdash e' : \varphi'$,
4. $\Gamma, \psi', \chi' \vdash_p (e', H', P') : \gamma'$,
5. $\Gamma \vdash \psi', \chi', \gamma'$ *wf.*
6. $\psi' = \psi[p \leftarrow \eta'] \cup \psi''$, *where* $\operatorname{dom}\psi \cap \operatorname{dom}\psi'' = \varnothing$.
   Furthermore, for all $p'' \in \operatorname{dom}\psi''$, $\operatorname{names}(\psi''(p'')) \subseteq \operatorname{names}(\psi(p))$.
7. *All names in* $\operatorname{names}(\eta') \setminus \operatorname{names}(\eta)$ *are fresh.*

Proof The proof is by a somewhat lengthy induction over the derivation of $\Gamma \mid \chi \mid \omega(\gamma); \eta \vdash e : \varphi$, keeping ψ general. Four cases are given explicitly.

T-Post: In this case, $\varphi = \text{И}A, \pi.\, \mathsf{promise}_\pi\, \tau\langle \mathsf{Wait}(\pi, \bar{\eta})\rangle$, $e = \mathtt{post}\, e_b$, $e' = p'$ for a $p' \notin \operatorname{dom} P$ with $p' \neq p$, and π fresh. Furthermore, $\Gamma \mid \omega\gamma \mid \chi; \eta \vdash e_b : \text{И}A.\, \tau\langle\bar{\eta}\rangle$.

By wellformedness, we may assume that all non-committed names in A are fresh as well.

Define:

- $\varphi' := \mathsf{promise}_\pi\, \tau\langle \mathsf{Wait}(\pi, \bar{\eta})\rangle$.
- $\gamma' := \gamma \cup \{p' \mapsto \text{И}c : A.\, \tau\langle\bar{\eta}\rangle\}$,
- $\psi' := \psi[p \leftarrow \mathsf{Wait}(\pi, \bar{\eta})] \cup \{p' \mapsto \eta\}$.
- $\chi' := (\chi.1 \cup \{p' \mapsto \pi\}, \chi.2 \setminus \{\pi\})$ (note that both $\pi \in \chi.2$ and $\pi \notin \chi.2$ are permissible, depending on whether π is committed),
- $\eta' := \mathsf{Wait}(\pi, \bar{\eta})$

Clearly, φ' specializes φ, and $(\gamma, \chi) \triangleright (\gamma', \chi')$. $\Gamma \mid \omega\gamma' \mid \chi'; \eta' \vdash p' : \varphi'$ is straightforward. Also, $\psi' = \psi[p \leftarrow \eta'] \cup \{p_1 \mapsto \eta\}$, and $\mathrm{names}(\eta) \subseteq \mathrm{names}(\eta)$.

To check whether $\Gamma, \psi', \chi' \vdash_p (e', H', P') : \gamma'$, it is sufficient to check whether $\Gamma \mid \omega\gamma' \mid \chi'; \psi'(p') \vdash P'(p') : \gamma'(p'$. Unfolding the definitions, this reduces to $\Gamma \mid \omega\gamma' \mid \chi'; \eta \vdash e_b : \text{И}c : A.\, \tau\langle e_b\rangle\bar{\eta}$. But this follows by Lemmas 11 and 12.

Finally, $\Gamma \vdash \psi', \chi', \gamma'$ wf holds:

1. Since $\mathrm{dom}\,\psi = \mathrm{dom}\,\chi.1 = \mathrm{dom}\,\gamma$ and $p \in \mathrm{dom}\,\psi$, $\mathrm{dom}\,\psi' = \mathrm{dom}\,\chi'.1 = \mathrm{dom}\,\gamma' = \mathrm{dom}\,\psi \cup \{p'\}$.
2. For all $\ell \in \mathrm{dom}\,\gamma'$, $\ell \in \mathrm{dom}\,\gamma$, so $\Gamma \vdash \psi'(\ell) \preceq \gamma'(\ell)$ follows from $\Gamma \vdash \psi, \chi, \gamma$ wf.
3. Let $p'' \in \mathrm{dom}\,\psi'$. To show: For all $\xi \in \mathrm{toplocs}_{\psi'}(p'')$, there is an ℓ such that $\chi(\ell) = \xi$, and for $\xi \in \mathrm{toptasks}_{\psi'}(p'')$, there is a p''' such that $\chi(p''') = \xi$.

   If $p'' \neq p, p'$, this follows from $\Gamma \vdash \psi, \chi, \omega$ wf.

   For $p'' = p$, the claim is trivial, since $\mathrm{toplocs}_{\psi'}(p) = \varnothing$ and $\mathrm{toptasks}_{\psi'}(p) = \{\pi\}$, and $\chi'(p') = \pi$.

   For $p'' = p'$, the claim follows since $\mathrm{toplocs}_{\chi'}(p') = \mathrm{toplocs}_{\chi}(p)$, $\mathrm{toptasks}_{\chi'}(p') = \mathrm{toptasks}_{\chi}(p)$ and $\chi'_1 \supseteq \chi_1$.
4. Since π is fresh, χ'_1 is injective.
5. Let A be the set of statically active tasks for ψ, and A' that for ψ'. It is easy to check that $A' = A \cup \{p'\}$. For $p_1, p_2 \notin \{p, p'\}$, $\mathrm{topnames}_{\psi'}(p_1) \cap \mathrm{topnames}_{\psi'}(p_2) = \varnothing$ follows from $\mathrm{topnames}_{\psi}(p_1) \cap \mathrm{topnames}_{\psi}(p_2) = \varnothing$, and a similar argument works for $p_1 = p'$ or $p_2 = p'$.

   Now, w.l.o.g., suppose $p_1 = p$. Then $\mathrm{topnames}_{\psi'}(p_1) = \{p'\}$, and since p' is fresh, $p' \notin \mathrm{topnames}_{\psi'}(p_2)$, since $p_1 \neq p_2$.
6. Checking that all postconditions have only committed names is straightforward.
7. Let $p_1, p_2 \in \mathrm{dom}\,\gamma' \cap \mathsf{Tasks}$ such that $\xi := \chi'(p_2) \in \mathrm{toptasks}_{\psi'}(p_1)$, and $\eta := \psi'(p_1)(\xi)$. If $p_1 \neq p, p'$, it turns out that $\xi = \chi(p_2) \in \mathrm{toptasks}_{\psi}(p)1$ and $\eta = \psi(p_1)(\xi)$, Then $\gamma'(p_2) = \gamma(p_2) = \text{И}A.\, \tau\langle\eta\rangle$ for some A and τ. If $p_1 = p'$, a similar argument gives the required result. If $p_1 = p$, $\xi = \pi$, and the claim is straightforward to check.
8. The only name in $\mathrm{names}(\eta') \setminus \mathrm{names}(\eta)$ is π, which is fresh.

T-WaitTransfer: Suppose first that $e = \mathtt{wait}\, p'$, $P(p') = \mathsf{done}\colon : v$, $e' = v$, $\bar{\varphi} = \mathit{И}A.\, \mathsf{promise}_\pi\, \tau\langle \eta_1 * \mathsf{Wait}(\pi, \eta_2)\rangle$ and $\varphi = \mathit{И}A.\, \tau\langle \eta_1 * \eta_2\rangle$. Furthermore, $\Gamma \mid \omega(\gamma) \mid \chi; \eta \vdash p' : \bar{\varphi}$, $H' = H$ and $P' = P$.

By Lemma 10, we get that $\Gamma, \chi \vdash \eta \preceq \eta_1 * \mathsf{Wait}(\pi, \eta_2)$ and $\Gamma \mid \omega(\gamma) \mid \chi \vdash p' : \mathsf{promise}_\pi\, \tau$. In particular, this implies that if $\omega p' = (\tau', \eta')$, then $\Gamma, \chi \vdash \tau' \preceq \tau$ and $\Gamma, \chi \vdash \eta' \preceq \eta_2$.

Since $\Gamma, \psi, \chi \vdash_p (e, H, P) : \gamma$, we also get that $\Gamma \mid \omega(\gamma) \mid \chi; \psi(p') \vdash v : \omega(p')$. Applying Lemma 10 again, we get that

$$\Gamma \mid \omega(\gamma) \mid \chi \vdash v : \tau'. \tag{A.1}$$

Set $\varphi' := \varphi$, $\gamma' := \gamma$, $\chi' := \chi$, $\eta' := \eta_1 * \eta_2$ and $\psi' := \psi[p \leftarrow \eta']$, where $f[x \leftarrow v](x') := \begin{cases} f(x') & x \neq x' \\ v & x = x' \end{cases}$.

It straightforward to check that φ' specializes φ and $(\gamma, \chi) \rhd (\gamma', \chi')$, that $\psi' = \psi[p \leftarrow \eta']$ and that $\Gamma, \psi', \chi' \vdash_p (e', H', P') : \gamma'$. $\Gamma \mid \omega\gamma' \mid \chi'; \eta' \vdash e' : \varphi'$ easily from (A.1).

To check that $\Gamma \vdash \psi', \chi', \gamma'$ wf, note that points 1, 2, 3, 4, 6 and 8 are straightforward. It remains to show 5 and 7; since the arguments are very similar, we only show 5.

Let A be the set of statically active tasks for ψ, and A' the same set for ψ'. Then it turns out that $A = A' \uplus \{p'\}$. By arguments similar to the above case, it is sufficient to show $\mathrm{topnames}_{\psi'}(p) \cap \mathrm{topnames}_{\psi'}(p'') = \varnothing$ for $p'' \neq p, p'$.

Because $\mathrm{topnames}_{\psi'}(p) = \mathrm{topnames}_{\psi}(p')$ and $\mathrm{topnames}_{\psi'}(p'') = \mathrm{topnames}_{\psi}(p'')$, this follows from $\Gamma \vdash \psi, \chi, \gamma$ wf.

Suppose now that $e = \mathtt{wait}\, \bar{e}$, $e' = \mathtt{wait}\, \bar{e}'$, $(\bar{e}, H, P) \hookrightarrow_l (\bar{e}', H', P')$. We have that $\Gamma \mid \omega\gamma \mid \chi; \eta \vdash \bar{e} : \bar{\varphi}$ with $\bar{\varphi} := \mathit{И}A.\, \mathsf{promise}_\pi\, \tau\langle \eta_1 * \mathsf{Wait}(\pi, \eta_2)\rangle$.

By the induction hypothesis (with ψ), there are $\bar{\varphi}', \psi', \chi', \gamma', \eta'$ such that $\bar{\varphi}'$ specializes $\bar{\varphi}$, $(\gamma, \chi) \rhd (\gamma', \chi')$, $\Gamma \mid \omega\gamma' \mid \chi'; \eta' \vdash \bar{e}' : \varphi'$, $\Gamma, \psi', \chi' \vdash_p (\bar{e}', H', P') : \gamma'$ and ψ', χ', γ' wf.

Since $\bar{\varphi}'$ specializes $\bar{\varphi}$, w.l.o.g. $\bar{\varphi}' = \mathit{И}A'.\, \mathsf{promise}_\pi\, \tau\langle \eta_1 * \mathsf{Wait}(\pi, \eta_2)\rangle$. Set $\varphi' := \mathit{И}A'.\, \tau\langle \eta_1 * \eta_2\rangle$.

It is then easy to check that φ' specializes φ and $\Gamma \mid \omega\gamma' \mid \chi'; \eta' \vdash e' : \varphi'$, and all the other conditions carry over from above.

T-Write: We only consider the case $e = \ell := v$; the other cases are similar to the case $e = \mathtt{wait}\, e$, $e \neq p$, above.

We have that $e' = \mathbf{unit}$, $P' = P$, $H' = H[\ell \leftarrow v]$, $\varphi = \mathit{И}A.\, \mathsf{unit}\langle \eta_2 * \mu \mapsto \tau_2\rangle$, $\Gamma \mid \omega(\gamma) \mid \chi; \eta \vdash v : \mathit{И}A_1.\, \tau_2\langle \eta_1\rangle$, $\Gamma, A_1 \mid \omega(\gamma) \mid \chi; \eta_1 \vdash \ell : \mathit{И}A_2.\, \mathsf{ref}_\mu\, \tau\langle \eta_2 * \mu \mapsto \tau_1\rangle$ and $\Gamma, A_1, A_2 \vdash \tau_2 \preceq \tau$.

By two applications of Lemma 10, transitivity of subtyping and strengthening, we get that $\Gamma \vdash \eta \preceq \eta_2 * \mu \mapsto \tau_1$, $\Gamma \mid \omega(\gamma) \mid \chi \vdash v : \tau_2$ and $\Gamma \mid \omega(\gamma) \mid \chi \vdash \ell : \mathsf{ref}_\mu\, \tau$.

Set $\gamma' := \gamma$, $\varphi' := \varphi$, $\chi' := \chi$, $\eta' := \eta_2 * \mu \mapsto \tau_2$ and $\psi' := \psi[\ell \leftarrow \eta']$.

Again, trivially $(\gamma, \chi) \rhd (\gamma', \chi')$ and φ' specializes φ, and $\psi' = \psi[\ell \leftarrow \eta']$. Also, $\Gamma \mid \omega(\gamma') \mid \chi'; \eta' \vdash e' : \varphi'$ reduces to $\Gamma \mid \omega(\gamma') \mid \chi'; \eta' \vdash \mathbf{unit} : \mathit{И}A'.\, \mathsf{unit}\langle\eta'\rangle$ is straightforward, and checking $\Gamma, \psi', \chi' \vdash_p (e, H, P) : \gamma'$ reduces to $\Gamma \mid \omega(\gamma') \mid \chi' \vdash v : \tau_2$, which follows easily by weakening.

It remains to check that ψ', χ', γ' wf. It is easy to see that points 1, 3, 4, 5, 6, 7, 8 are satisfied. For point 2, it remains to show that $\Gamma \vdash \psi'(\ell) \preceq \gamma'(\ell)$. But since $\psi'(\ell) = \tau_2$ and $\gamma'(\ell) = \tau$, this follows immediately from $\Gamma, A_1, A_2 \vdash \tau_2 \preceq \tau$ by strengthening.

T-Frame: We have $\eta = \eta_1 * \eta_2$, $\varphi = \mathit{И}A.\, \tau\langle\eta_1' * \eta_2\rangle$ and $\Gamma \mid \omega(\gamma) \mid \chi; \eta_1 \vdash e : \mathit{И}A.\, \tau\langle\eta_1'\rangle$.

Applying the induction hypothesis with $\psi[p \leftarrow \eta_1]$ instead of ψ, we get $\varphi', \eta', \psi', \gamma', \omega'$ such that

1. φ' specializes $\mathit{И}A.\, \tau\langle\eta_1'\rangle$,
2. $(\gamma, \chi) \rhd (\gamma', \chi')$,
3. $\Gamma \mid \omega(\gamma') \mid \chi'; \eta' \vdash e : \varphi'$,
4. $\Gamma, \psi', \chi' \vdash_p (e', H', P') : \gamma'$,
5. $\Gamma \vdash \psi', \chi', \gamma'$ wf.
6. $\psi' = \psi[p \leftarrow \eta'] \cup \psi''$, where $\operatorname{dom}\psi \cap \operatorname{dom}\psi'' = \varnothing$.

   Furthermore, for all $p'' \in \operatorname{dom}\psi''$, $\operatorname{names}(\psi''(p'')) \subseteq \operatorname{names}(\psi(p))$.
7. All names in eta' $\setminus\, \eta$ are fresh.

From this, we get that, w.l.o.g. with regard to name choices, $\varphi' = \mathit{И}A'.\, \tau\langle\eta_1'\rangle$. Define $\varphi'' := \mathit{И}A'.\, \tau\langle\eta_1' * \eta_2\rangle$.

We also want to define $\eta'' := \eta' * \eta_2$, and $\psi'' := \psi'[p \leftarrow \eta'']$. $\Gamma, \psi', \chi' \vdash_p (e', H', P') : \gamma'$ still holds.

To show $\Gamma \mid \omega(\gamma') \mid \chi'; \eta'' \vdash e : \varphi''$, we apply T-Frame. This requires us to show that $\Gamma \vdash \eta''$ wf, and more precisely, $\operatorname{names}(\eta') \cap \operatorname{names}(\eta_2) = \varnothing$.

But if $\xi \in \operatorname{names}(\eta')$, either $\xi \in \operatorname{names}(\eta)$, or ξ is fresh. In the first case, since $\Gamma \mid \omega(\gamma) \mid \chi; \eta \vdash e : \varphi$, we have $\Gamma, \chi \vdash \eta$ wf and hence, $\operatorname{names}(\eta_1) \cap \operatorname{names}(\eta_2) = \varnothing$. Thus, $\xi \notin \operatorname{names}(\eta_2)$. If ξ is fresh, clearly $\xi \notin \operatorname{names}(\eta_2)$.

It remains to show that $\Gamma \vdash \psi'', \chi', \gamma'$ wf. Points 1, 2, 4, 6 and 8 are straightforward. For point 3, it suffices to show: For $\xi \in \operatorname{toplocs}_{\psi''}(p)$, there is some ℓ such that $\chi'(\ell) = \xi$ and for $\xi \in \operatorname{toptasks}(p')$, there is some p'' such that $\chi'(p'') = \xi$. If $\xi \in \operatorname{names}(\eta')$, this follows from $\Gamma \vdash \psi', \chi', \gamma'$ wf. If $\xi \in \operatorname{names}(\eta_2)$, this follows from $\Gamma \vdash \psi, \chi, \gamma$ wf and $(\gamma, \chi) \rhd (\gamma', \chi')$.

For point 5, it suffices to show: For all $p'' \neq p$, $\operatorname{topnames}_{\psi''}(p) \cap \operatorname{topnames}_{\psi''}(p'') = \varnothing$. Let $\xi \in \operatorname{topnames}_{\psi''}(p)$. Then either $\xi \in \operatorname{names}(\eta')$ or $\xi \in \operatorname{names}(\eta_2)$. In the first case, the claim follows from $\Gamma \vdash \psi', \chi', \gamma'$ wf. In the second case, suppose first $p'' \neq p'$. Then if $\xi \in \operatorname{topnames}_{\psi''}(p'')$, we also have $\xi \in \operatorname{topnames}_{\psi}(p'')$, using the structure

invariant of ψ' ($\psi' = \psi[p \leftarrow \eta'] \cup \psi'''$). But this contradicts $\Gamma \vdash \psi, \chi, \gamma$ wf. Thus, suppose $p'' = p'$. Then by the second part of the structure invariant, $\xi \in \text{names}(\eta_1)$, so $\xi \in \text{names}(\eta_1) \cap \text{names}(\eta_2)$ – contradiction.

For point 7, a similar argument can be used.

The other cases are similar to existing cases (T-Ref, T-Read) or entirely standard (T-App reduces to Lemma 13, T-ForallElim and T-Subtype are straightforward along the lines of T-Frame, the rules for weak memory typing are standard).

Using Theorem 14, global type preservation as in Theorem 11 can be proved.

Theorem 15 (Preservation, Theorem 11) *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 $(\gamma, \psi) \rhd (\gamma', \psi')$.

Proof By case analysis on $(H, P, p) \hookrightarrow (H', P', p')$. There are three cases:

EG-Local: In this case, $p = p'$. Let $(e, H, \bar{P})$ and $(e', H', \bar{P}')$ the corresponding local configurations (i.e., $(H, P, p) = (H, P \uplus \{p \mapsto \mathsf{run}: : e\}, p)$) and similar for $(e', H', \bar{P}')$. Then $\cdot \mid \omega(\gamma) \mid \chi; \psi(p) \vdash e : \gamma(p)$ and $\cdot, \psi, \chi \vdash_p (e, H, P) : \gamma$ (this is easy to check by unfolding the definition), and ψ, χ, γ wf. Therefore, by Theorem 14, there are $\gamma', \chi', \varphi', \eta', \psi'$ such that

1. φ' specializes $\omega(p)$,
2. $(\gamma, \chi) \rhd (\gamma', \rhd')$,
3. $\cdot \mid \omega(\gamma') \mid \chi'; \eta' \vdash e' : \varphi'$,
4. $\cdot, \psi, \chi \vdash_p (H', P', p') : \gamma'$, using $p = p'$,
5. ψ', χ', γ' wf.
6. $\psi'(p) = \eta'$.

Set $\gamma'' := \gamma'[p \leftarrow \varphi']$. Then by definition of specialization, $(\gamma, \chi) \rhd (\gamma'', \chi')$. Furthermore, it is easy to check that $\psi', \chi' \vdash (H', P', p') : \gamma''$, and that ψ', χ', γ' wf.

EG-Finish: In this case, $P[p] = \mathsf{run}: : v$, $P' = P[p \leftarrow \mathsf{done}: : v]$, $H = H'$, $P'[p'] = \mathsf{run}: : \_$. Set $\psi' := \psi$, $\gamma' := \gamma$ and $\chi' := \chi$. Trivially, ψ', χ', γ' wf, and $(\gamma, \chi) \rhd (\gamma', \rhd')$. To show that $\psi', \chi' \vdash (H', P', p') : \gamma'$, it suffices to show $\cdot \mid \omega(\gamma) \mid \chi; \psi(p) \vdash v : \gamma(p)$, but this follows from $\psi, \chi \vdash (H, P, p) : \gamma$.

EG-WaitRun: Similar to the previous case.

B. Delayed refinement and soundness of DWFM

We will not give complete proofs of all lemmas and theorems. For everything proved using Coq, we just sketch the main arguments and link to the corresponding mechanized proof.

We start this chapter with an overview of the Coq development; we assume some knowledge of Iris in this chapter (compare Jung et al. [2015], Krebbers et al. [2017]). Based on this, we sketch the proofs of the more interesting lemmas.

B.1. Overview of the development

The Coq development formalizes most of Chapter 4; it leaves out some proofs or simplifies the lemma:

1. Lemma 1: Omitted. This is tedious to do in Coq because of the need to invert typing derivations.
2. Theorem 4: We have proved all the cases, but the complete proof would again require typing inversions.
3. Theorem 5: Omitted.

We give paper proofs for these in this appendix.

The development consists of various modules, listed below. Their dependencies are given in Figure B.1. We have the following modules (given in a topological order):

corecalculus: This defines the core language, including small-step semantics.

types: This provides the type system, including wellformedness conditions and typing rules.

specification: This provides Iris reasoning support for the core calculus. It contains:

- The Iris infrastructure for setting up universal and existential Hoare triples;
- The definition of the various predicates used in the program logic for the binary interpretation of types;
- An axiomatization of a weakest precondition operator for the core calculus. Note that it is possible, with significant additional work, to actually turn this into a proper definition and all the axioms into theorems.

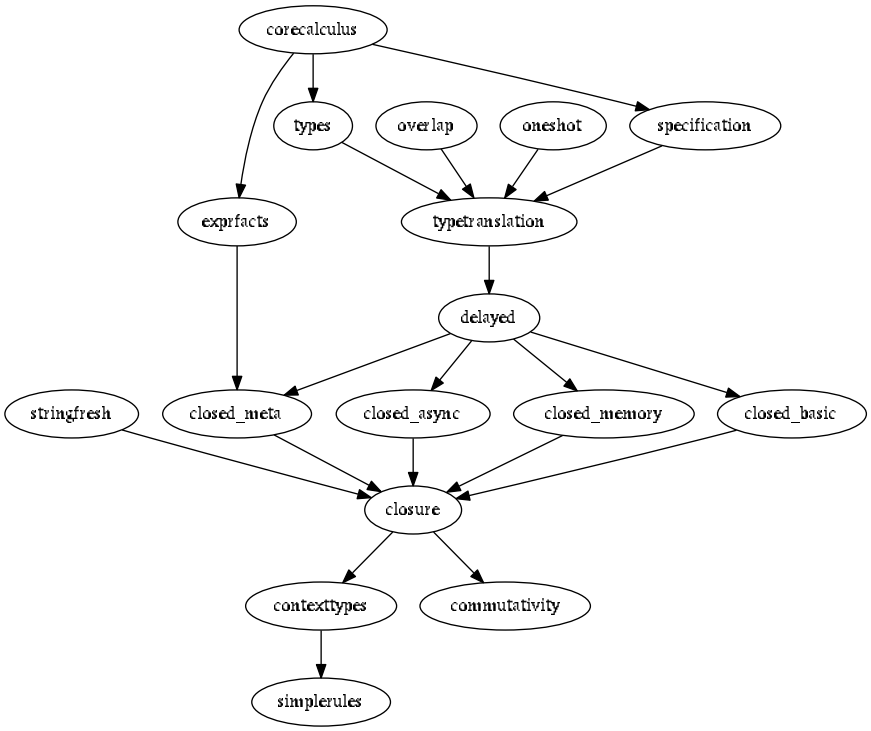


Figure B.1.: Dependencies between the Coq modules. An arrow from A to B means that A is used by B.

- The definition of existential Hoare triples, and a number of lemmas that prove the most common cases of E-ReflectSemantics.

exprfacts: Various helper lemmas about syntactic helper functions such as substitutions;

overlap: The map overlap predicate $\equiv_X$ and various lemmas about it.

oneshot: A generic definition of predicates in the style of $\twoheadrightarrow_I$ and $\twoheadrightarrow_S$.

typetranslation: The interpretation functions. This module also contains the locality and duplicability lemmas.

delayed: The definition of delayed simulation;

closed_meta: Proofs of Lemma 4 and the cases C-Frame and C-Strengthen of Lemma 5;

closed_async: Proofs of the cases C-Post and C-Wait of Lemma 5;

closed_memory: Proofs of the cases C-Alloc, C-Read and C-Write of Lemma 5;

closed_basic: Proofs of the remaining cases of Lemma 5;

stringfresh: Fresh name generator with correctness proof;

closure: Proof of Lemma 6;

contexttypes: An alternative definition of expression context typing, and a proof of Lemma 7;

commutativity: Proof of Lemma 9

simplerules: Proof of Lemma 8 and the cases of Theorem 4.

We will give a quick tour of the main features of the proof.

B.1.1. corecalculus

This file describes the core language. Expressions e are encoded using the algebraic datatype `expr`, and values v using the algebraic datatype `val`. We have a total function `of_val : val → expr`, and a partial function `to_val : expr → option val`, converting between expressions and values. Auxiliary definitions provide `loc` (heap locations), `tid` (task handles), `const` (constants; this includes `Ctrue`, `Cfalse` and `Cunit` as constructors for `true`, `false` and `()`, and `Cloc` and `Ctid` as injections for heap locations and task handles). Heaps `heap` are finite maps from `loc` to `val`, while task buffers `task_buffer` are finite maps `tid` to `task_state`, and algebraic data type with two constructors `running` (taking an expression) and `done` (taking a value).

Furthermore, we provide evaluation contexts $\mathcal{C}$ as lists of context items `ctx_item`, and expression contexts $\mathcal{E}$ as lists of expression context items `ectx_item`, with instantiation functions `fill_ctx` : `expr` → `list ctx_item` → `expr` and `fill_ectx` : `expr` → `list ectx_item` → `expr`.

Substitutions are performed using the functions `subst` : `stringmap expr` → `expr` → `expr` (substituting variables in an expression) and `subst_ctx` : `stringmap expr` → `list ctx_item` → `list ctx_item` (substituting variables in an evaluation context).

The predicate `Closed` : `stringset` → `expr` → `Prop` describes that an expression is closed up to a given set of variables: `Closed` $X\, e$ means that $\mathrm{free}(e) \subseteq X$.

The small-step semantics are encoded in the `head_step`, `local_step` and `global_step` relations, directly reflecting the ALST semantics.

B.1.2. types

This file describes the type system. Types are encoded by the algebraic data type `type`, and resource expressions by `hexpr`. Logical names are given by a type `lname`, and sets of logical names by `lnames`. Environments are given by `env`, a map from strings to `type`.

The names function is implemented using a typeclass: An instance `Names`X provides a function `names` : X → `lnames`. Instances exist for `type`, `hexpr` and `env`. The rnames function is available as `res_names`.

The monoid structure on resource expressions is described by `hexpr_equiv` (instantiating the `Equiv` typeclass); definitions involving resource expressions also provide a proof of invariance under the monoid laws (see, e.g., `he_names_proper`).

In the development, we also include wellformedness conditions. The `heap_wf` predicate checks that a resource expression is wellformed: `heap_wf` $A\, \eta$ ensures that η is well-formed, and all names appearing in it are contained in A.

The typing relation is given by `typing` — it gives the typing relation from the paper, augmented with wellformedness conditions.

The module also contains some lemmas relating typing and wellformedness.

B.1.3. specification

Universal Hoare triples are defined using a weakest precondition operator, which has been axiomatized to model Iris' standard `WP` operator. The reason why we don't use the built-in operator is that this operator has multi-threading baked in as concurrency model; since our execution model is different, we cannot reuse it. An obvious item of future work is to actually define the WP operator for asynchronous programs properly, and proving all the axioms as lemmas.

The WP operator is given as `WP`$e\{\{x, \phi(x)\}\}$, and gives the weakest precondition that guarantees that for every execution of e, if it reduces to x, $\phi(x)$ holds on the final state. Hoare triples can then be easily be defined as $\{\phi\}\ \ e\ \ \{x.\ \phi'(x)\} := \phi \mathrel{-\!\!*} \texttt{WP}e\{\{x, \phi'(x)\}\}$ (compare the Iris standard library for this).

Instead of the H-Post rule from Chapter 4, we use a more general approach: We parameterize on a type `impl_task_dataT`. On posting, we get an instance $\mathsf{Ag}(t, d)$ for a

value d of type `impl_task_dataT`, implementing the following rule (cf. `wp_post`):

$$\frac{\forall t, \{\mathsf{Ag}(t,d)\}\ \ e\ \ \{x.\ \phi(t,x)\}}{\mathtt{WP}\,\mathtt{post}\,e\{\{v, \exists t.\ulcorner v = \mathtt{Ctid}t\urcorner * \mathtt{wait}\,t\,\phi * \mathsf{Ag}(t,d)\}\}}$$

Another change is that we use a more strict version of SplitWait: In anticipation of a reasonable definition of `split`, we axiomatize this rule as a set of four fancy updates:

$$\mathtt{wait}(t, x.\ \phi(t,x) * \phi'(t,x)) \Rrightarrow^{\uparrow\mathtt{waitN}} \mathtt{wait}(t, x.\ \phi(t,x)) * \mathtt{wait}(t, x.\ \phi'(t,x))$$
$$\triangleright\,\mathtt{wait}(t, x.\ \phi(t,x) * \phi'(t,x)) \Rrightarrow^{\uparrow\mathtt{waitN}} \triangleright\,\mathtt{wait}(t, x.\ \phi(t,x)) * \triangleright\,\mathtt{wait}(t, x.\ \phi'(t,x))$$
$$\mathtt{wait}(t, x.\ \phi(t,x)) * \mathtt{wait}(t, x.\ \phi'(t,x)) \Rrightarrow^{\uparrow\mathtt{waitN}} \mathtt{wait}(t, x.\ \phi(t,x) * \phi'(t,x))$$
$$\triangleright\,\mathtt{wait}(t, x.\ \phi(t,x)) * \triangleright\,\mathtt{wait}(t, x.\ \phi'(t,x)) \Rrightarrow^{\uparrow\mathtt{waitN}} \triangleright\,\mathtt{wait}(t, x.\ \phi(t,x) * \phi'(t,x))$$

A preliminary development of wait permissions as transition systems (discussed below) shows that these rules can be proven, and it turns out that this is sufficient for the DWFM soundness proof.

The set-up of task properties and existential triples closely follows the approach of Krogh-Jespersen et al. [2017]. Notably, the $\Rrightarrow$ operator of the program logic of Chapter 4 is mapped to a fancy update, as follows: $\phi \Rrightarrow \phi'$ in the program logic corresponds to $\Box\phi \Rrightarrow^{E} \phi'$ in Iris, for an appropriate set E of names.

B.1.4. typetranslation

While this module largely follows the definitions in Fig. 4.8, some technical changes are needed. All the changes are related to the interpretation of promises and wait permissions. The first change is that we use the changed Hoare triple for `post` from above; we instantiate `impl_task_dataT` to the following record type:

```
Record task_data :=
  TaskData {
      td_expr: expr;
      td_env: env;
      td_pre: hexpr;
      td_post: hexpr;
      td_alloc: gset gname;
      td_D_pre: conn_map;
      td_D_name: gname;
      td_N_name: gname;
    }.
```

The first six components correspond to $e, \Gamma, \eta_{pre}, \eta', A, D_{pre}$ in $\mathsf{Ag}(t, e, \Gamma, \eta_{pre}, \eta', A, D_{pre})$. The other two provide logical names that allow us to implement $t \twoheadrightarrow_I D', N'$: This predicate is implemented as $\mathtt{own}(td\_D\_name, D') * \mathtt{own}(td\_N\_name, N')$.

The second, more complex change is to do with the recursive structure of $[\![\eta]\!]_S$. Note that in the $\mathsf{Wait}(\xi, A, \eta)$ case, we need to include $[\![\eta_{pre}]\!]_S$. But there is no guarantee that η_{pre} is "smaller" than η (or even $\mathsf{Wait}(\xi, A, \eta)$), so we cannot guarantee that the inherent recursion terminates!

In fact, as stated, we can construct an example where there is divergent behavior: Let $\eta_i = \mathsf{Wait}(\xi_i, \emptyset, \mathsf{emp})$, $N(\xi_i) = t_i$, and $\mathsf{Ag}(t_i, e, \Gamma, \eta_{i+1}, \emptyset, \mathsf{emp}, \ldots)$. Then the interpretation of η_0 would engender an infinite tower of $\mathsf{Wait}(\xi, \emptyset, \eta_i)$ interpretations. Interestingly enough, this also implies that there was an infinite chain of posts, with task ξ_{i+1} posting task ξ_i. Since we assume that every state can be realized by executing a program a finite number of steps, this is clearly nonsensical.

For this reason, we introduce the idea of *posting depth*: A resource expression η has posting depth n if, in the current state, we can find a simulation interpretation of η that has at most n stacked Wait subterms in it. We define the interpretation function in three steps:

1. Interpretation for a fixed posting depth. Most cases are exactly as in Fig. 4.8. In the Wait case, we refer to another interpretation function (which will have lower posting depth) to interpret η_{pre}. See `int_s_heap_approx`.

2. Interpretation with given posting depth, see `int_s_heap_rec`. This is the interpretation function with an additional posting depth parameter n. For $n = 0$, it interprets η_{pre} as false; for $n > 0$, it interprets η_{pre} with $n - 1$ instead of n. Thus, if the posting depth of η is at most n, `int_s_heap_rec` gives exactly the interpretation we expect.

3. The final interpretation function. This simply gives the interpretation with an existentially quantified posting depth. See `int_s_heap`.

A somewhat minor third change is that we use a stronger condition than duplicability, namely *persistence*. This property implies duplicability, but requires us to change the implementation interpretation of promises a bit: We need to enclose the wait permission inside an invariant, as to make sure it is actually persistent.

With this, the duplicability lemmas follow trivially from the persistence lemmas: `int_s_type_persistent`, `int_i_type_persistent`, `int_s_env_persistent` and `int_i_env_persistent`.

The locality lemmas, `int_s_type_local`, `int_i_type_local`, `int_s_heap_local`, `int_i_heap_local`, `int_s_env_local` and `int_i_env_local`, follow using easy induction proofs.

B.1.5. delayed

The main content of this file are the three definitions, `simulation` (the saved continuation predicate), `delayed_typed` (the main definition of delayed simulation) and `delayed_simulation` (which packs delayed simulation with the well-typedness preconditions). It also contains lemmas showing the invariance under the resource expression monoid laws.

With these definitions in place, we can discuss some of the proofs.

B.2. Interesting proofs

We will discuss the following proofs:

- Lemma 1: Paper proof.
- Lemma 4: outline of the Coq proof.
- Lemma 5: outline of the Coq proof for C-Post, C-Wait, C-Frame, C-Strengthen. The other cases are easy.
- The lemmas covering the inductive cases of Lemma 6 and Lemma 7.
- Lemma 7: outline of the Coq proof.
- Lemma 8: outline of the Coq proof.
- Lemma 9: outline of the Coq proof.
- Theorem 4: statement of the lemmas proved in Coq, and paper proof of the theorem.
- Theorem 5: Paper proof.

Lemma 14 (Rewriting preserves types, Lemma 1) *Suppose* $\Gamma; \eta \vdash e : \text{И}A.\, \tau\langle\eta'\rangle$ *and* $e \implies e'$. *Then there is some* $A' \supseteq A$ *such that* $\Gamma; \eta \vdash e : \text{И}A'.\, \tau\langle\eta'\rangle$.

Proof This Lemma is shown as a side result in the proof of Theorem 4, below.

Lemma 15 (Binding composition, Lemma 4) *Given* $\mathcal{C}$, $\mathcal{C}'$, e, e', η_1, η_2, η_3, Γ, x, τ_1, τ_2, A_1, A_2 *such that*

$$[\![\Gamma; \eta_1 \vdash e \leq_\diamond e' : \text{И}A_1.\, \tau_1\langle\eta_2\rangle]\!] \text{ and}$$
$$[\![\Gamma, A_1, x : \tau_1; \eta_2 \vdash \mathcal{C}[x] \leq_\diamond \mathcal{C}'[x] : \text{И}A_2.\, \tau_2\langle\eta_3\rangle]\!]$$

hold. Then $[\![\Gamma; \eta_1 \vdash \mathcal{C}[e] \leq_\diamond \mathcal{C}'[e'] : \text{И}A_1, A_2.\, \tau_2\langle\eta_3\rangle]\!]$ *holds.*

This lemma is proved as `delayed_bind` in `closed_meta`.

Proof This is a summary of the proof of `delayed_bind` in `closed_meta`.

Most of the interesting work is done in three lemmas, `strengthen_env_generic`, `bind_existential_part` and `bind_universal_part`. We sketch the proof of each of them as separate claim.

Claim 1: Let $[\![\tau]\!]_X(d, N, x)$ be an interpretation function for types, and $\phi(D, N)$ a formula, where $[\![\tau]\!]_X$ is local in N and ϕ is local in D for some set of names L. Define $[\![\Gamma]\!]_X$ as for $[\![\Gamma]\!]_I$ and $[\![\Gamma]\!]_S$ in terms of $[\![\tau]\!]_X$.

Let $\Gamma, \tau, x, D, D', v, \sigma, N, N'$ be given, and suppose $\operatorname{dom}\Gamma \cap L = \emptyset$, $x \notin L$ and $O \supseteq \mathsf{names}(\Gamma)$.

If $[\![\Gamma]\!]_X(D, N, \sigma)$, $N \equiv_O N'$, $[\![\tau]\!]_X(d, N', v)$ and $\phi(D', N')$ hold, it also holds that $[\![\Gamma, x : \tau]\!]_X(D''.N', \sigma[x \mapsto v])$ and $\phi(D'', N')$ hold, where $D''.x = d$, $D''.i = D.i$ for $i \in \operatorname{dom}\Gamma$ and $D''.i = D'.i$ otherwise.

Proof: This lemma is an exercise in using locality properties.

Claim 2: Suppose that the following two hold:

$$[\![\Gamma; \eta_1 \vdash e : \text{И}A_1.\, \tau_1\langle\eta_2\rangle]\!]_S(D_1, D_2)$$
$$[\![\Gamma, A_1, x : \tau_1; \eta_2 \vdash \mathcal{E}[x] : \text{И}A_2.\, \tau_2\langle\eta_3\rangle]\!]_S(D_2', D_3)$$

where $D_2'.x = D_2.\boxdot$, $D_2'.x = D_1.x$ for $x \in \operatorname{dom}\Gamma$ and $D_2'.x = D_2.x$ otherwise. Suppose furthermore that $\mathsf{names}(\Gamma) \cap A_1 = \emptyset$, x does not occur in $\mathcal{E}$, and after substituting all variables in $\mathcal{E}$ with values, one gets an evaluation context $\mathcal{C}$.

Then $[\![\Gamma; \eta_1 \vdash \mathcal{E}[e] : \text{И}A_1, A_2.\, \tau_2\langle\eta_3\rangle]\!]_S(D_1, D_3)$.

Proof: Using Claim 1, strengthen the first saved continuation such that the postcondition of the existential Hoare triple coincides with the precondition of the second saved continuation. Then, combine the executions into a single execution using the properties of the small-step semantics.

The properties of the context are used to ensure that $\mathcal{E}[e]$ can take steps lifted from the steps of e; we go for this more general form of the lemma since it allows us to prove stronger versions of the closure lemmas easily.

Claim 3: $[\![\Gamma; \eta_1 \vdash e \leq_\diamond : \text{И}A_1.\, \tau_1\langle\eta_2\rangle]\!]$ and $[\![\Gamma, A_1, x : \tau_1; \eta_2 \vdash \mathcal{E}[x] \leq_\diamond \mathcal{E}'[x] : \text{И}A_2.\, \tau_2\langle\eta_3\rangle]\!]$. Suppose furthermore that $\mathsf{names}(\Gamma) \cap A_1 = \emptyset$, x does not occur in $\mathcal{E}$ nor $\mathcal{E}'$, and after substituting all variables in $\mathcal{E}$ respectively $\mathcal{E}'$ with values, one gets an evaluation context $\mathcal{C}$ respectively $\mathcal{C}'$.

Then $[\![\Gamma; \eta_1 \vdash \mathcal{E}[e] \leq_\diamond \mathcal{E}'[e'] : \text{И}A_1, A_2.\, \tau_2\langle\eta_3\rangle]\!]$.

Proof: Define D_2' as in the previous proof. One shows first that one can, again, strengthen the implementation interpretation of the postcondition for the first delayed simulation to match the precondition of the second delayed simulation, using Claim 1. Then, one combines the two specifications using the program logic bind rule to give a universal Hoare triple with two existential Hoare triples, for the execution of e' and for $\mathcal{E}'[x]$. Finally, one uses Claim 2 to finish the proof.

The lemma is then an easy corollary.

Lemma 16 (Closure for `post`: Lemma 5, C-Post) *Suppose that we have that* $[\![\Gamma; \eta \vdash e \leq_\diamond e' : \text{И}A.\, \tau\langle\eta'\rangle]\!]$, *and* $\xi \notin \mathsf{names}(\Gamma), \mathsf{names}(\eta), \mathsf{names}(\tau), \mathsf{names}(\eta'), A$. *Then* $[\![\Gamma; \eta \vdash \texttt{post}\, e \leq_\diamond \texttt{post}\, e' : \text{И}\{\xi\}.\, \mathsf{promise}_{\xi,A}\, \tau\langle\mathsf{Wait}(\xi, A, \eta')\rangle]\!]$.

This lemma is proved as `closed_post` in `closed_async`.

Proof The proof starts by demonstrating:

Claim: $\{\mathsf{Ag}(t, e', \Gamma, \eta, \eta', A, D) * t \rightarrowtriangle_I \bot * [\![\Gamma; \eta]\!]_I(D, N, \sigma)\}\ \ e\sigma\ \ \{x.\ \phi_{\text{promise}}(x) * \phi_{\text{wait}}\}$, where ϕ_{promise} and ϕ_{wait} are the bodies of the wait permissions occurring in the implementation interpretation of $\mathsf{promise}_{\xi,A}\,\tau$ and $\mathsf{Wait}(\xi, A, \eta')$, respectively. In other words, we have $[\![\mathsf{promise}_{\xi,A}\,\tau]\!]_I(d, N, x) = N.\xi \doteq x \wedge \textsc{Wait}(x; y.\phi_{\text{promise}}(y))$ and $[\![\mathsf{Wait}(\xi, A, \eta')]\!]_I(D, N) = N.\xi \doteq D.\xi * \textsc{Wait}(N.\xi; y.\ \phi_{\text{wait}})$.

Proof: Using the assumptions, we find that it is sufficient to demonstrate:

$$\begin{aligned}&\mathsf{Ag}(t, e', \Gamma, \eta, \eta', A, D) * t \rightarrowtriangle_I \bot *\\ &\exists N', D'.\ N \equiv_{\overline{A}} N' * [\![\tau; \eta']\!]_I(D', N', x) * [\![\Gamma; \eta \vdash e' : \mathsf{M}A.\,\tau\langle\eta'\rangle]\!]_S(D, D')\\ \Rrightarrow &\phi_{\text{promise}}(x) * \phi_{\text{wait}}.\end{aligned}$$

This turns out to be entirely straightforward.

Main proof: At this point, apply the H-Post rule and SplitWait; this gives the wait permissions from the interpretations of $\mathsf{promise}_{\xi,A}\,\tau$ and $\mathsf{Wait}(\xi, A, \eta')$.

Wrap the wait permission for $\mathsf{promise}_{\xi,A}\,\tau$ inside an invariant; then, it is easy to prove that $[\![\mathsf{promise}_{\xi,A}\,\tau]\!]_I((), N[\xi \mapsto t], t)$ holds (using a locality argument to change the name map).

Similarly, one may prove that $[\![\mathsf{Wait}(\xi, A, \eta')]\!]_I(D[\xi \mapsto t, \boxdot \mapsto ()], N[\xi \mapsto t])$ holds.

Choosing $N' := N[\xi \mapsto t]$ and $D' := D[\xi \mapsto t, \boxdot \mapsto ()]$, it suffices to demonstrate:

- $N \equiv_{\overline{\{\xi\}}} N'$ and $D'.\boxdot = ()$ — these are trivial.
- $[\![\Gamma; \eta \vdash \mathtt{post}\ e' : \mathsf{M}\{\xi\}.\,\mathsf{promise}_{\xi,A}\,\tau\langle\mathsf{Wait}(\xi, A, \eta')\rangle]\!]_S D, D')$.

  To prove this, note that we can take one E-Post step to post a task running e', getting a task identifier p'. Update $p' \rightarrowtriangle_S \bot$ to $p' \rightarrowtriangle_S N$ for the name map used to interpret the precondition.

  Proving $[\![\mathsf{promise}_{\xi,A}\,\tau]\!]_S((), N[\xi := p'], p')$ and $[\![\mathsf{Wait}(\xi, A, \eta')]\!]_S(D', N[\xi := p'])$ is then easy; for the latter, use the posting depth of $[\![\eta]\!]_S(D, N)$ plus 1 as the new posting depth.

Lemma 17 (Closure for `wait`: Lemma 5, C-Wait) *Suppose* $\xi \notin \mathsf{names}(\tau), \mathsf{names}(\eta)$. *Then* $[\![x : \mathsf{promise}_{\xi,A}\,\tau; \mathsf{Wait}(\xi, A, \eta) \vdash \mathtt{wait}\ x \leq_\diamond \mathtt{wait}\ x : \mathsf{M}A.\,\tau\langle\eta\rangle]\!]$.

This lemma is proved as `closed_wait` in `closed_async`.

Proof Let ϕ_{promise} and ϕ_{wait} as above. Fix D, N and σ.

First, extract $\triangleright\textsc{Wait}(t, x.\ \phi_{\text{promise}}(x))$ from $[\![\mathsf{promise}_{\xi,A}\,\tau]\!]_I(D.\boxdot, N, x)$ by opening the invariant and cloning the wait permission using SplitWait. One clone is used to close the invariant.

Then, combine $\triangleright\textsc{Wait}(t, x.\ \phi_{\text{promise}}(x))$ and $\textsc{Wait}(t, \_.\ \phi_{\text{wait}})$ using the $\triangleright$-guarded version of SplitWait.

Apply H-Wait to get rid of the `wait` and open the wait permission.

Using the various agreement predicates, find the D' and N', and define $N''.\xi := N'.\xi$ for $\xi \in A$, $N''.\xi := N.\xi$ otherwise. Use N'' and D' as the resulting name and connection data map. $N \equiv_{\overline{A}} N''$ is trivial, and $[\![\tau]\!]_I(D'.\boxdot, N'', v)$ (where v is the return value) and $[\![\eta']\!]_I(D', N'')$ follow using locality.

The interesting part is to show the simulation side. We find that $[\![\mathsf{Wait}(\xi, A, \eta)]\!]_S(D', N')$ cannot have a posting depth of zero, since it equivalent to false in that case. By unpacking all preconditions, we have (after applying equalities and agreements):

- $[\![\Gamma_0; \eta_0 \vdash e_0 : \text{И}A.\, \tau\langle\eta\rangle]\!]_S(D_0, D')$, where Γ_0, η_0 et cetera come from the Ag agreement predicate.
- $t' \Mapsto \mathsf{posted}\colon U_0\sigma'$ for some σ',
- $[\![\Gamma_0, \eta_0]\!]_S(N_0, D_0, \sigma')$,
- $t \Mapsto \mathsf{running}\colon \mathcal{C}[\texttt{wait}\, t']$.

By performing a EG-WaitSchedule step scheduling t', we can apply the saved continuation, executing t' to completion. One further EG-Finish step makes t the running task again, and a final E-Wait step finishes off the task.

At this point, it is sufficient to collect all postconditions; this is done along the same lines as for the implementation side.

Lemma 18 (Closure under the frame rule: Lemma 5, C-Frame) *Suppose* $[\![\Gamma; \eta \vdash e \leq_\diamond e' : \text{И}A.\, \tau\langle\eta'\rangle]\!]$, $\mathit{rnames}(\eta') \cap \mathit{rnames}(\eta_f) = \emptyset$, $\mathit{rnames}(\eta) \cap \mathit{rnames}(\eta_f) = \emptyset$ *and* $\mathit{names}(\eta_f) \cap A = \emptyset$.

Then $[\![\Gamma; \eta * \eta_f \vdash e \leq_\diamond e' : \text{И}A.\, \tau\langle\eta' * \eta_f\rangle]\!]$.

This lemma is shown as `closed_frame` in `closed_meta`.

Proof The proof is mostly straightforward; it reduces to the following lemma, applied both to the universal and the existential side:

Claim: Let $X = I$ or $X = S$, $\tau, \eta, \eta_f, D, D', d, v, N, N'$ be given such that:

- $N \equiv_{\mathsf{rnames}(\eta_f)} N'$;
- $[\![\eta * \eta_f]\!]_X(D'', N'') \equiv [\![\eta]\!]_X(D'', N'') * [\![\eta_f]\!]_X(D'', N'')$;
- $[\![\eta]\!]_X$ has the expected locality properties;
- $\mathsf{rnames}(\eta) \cap \mathsf{rnames}(\eta_f) = \emptyset$;
- $[\![\eta]\!]_X(D, N)$ holds;
- $[\![\tau; \eta]\!]_X(D, N', v)$ holds.

then $[\![\tau; \eta * \eta_f]\!]_X(D'', N', v)$ holds, where $D''.\xi = D.\xi$ for $D \in \mathsf{rnames}(\eta_f)$, and $D''.\xi = D'.\xi$ otherwise.

Proof: By locality.

Lemma 19 (Closure under strengthening: Lemma 5, C-Strengthen) *Suppose* $[\![\Gamma; \eta \vdash e \leq_\diamond e' : И A.\, \tau\langle\eta'\rangle]\!]$, $\Gamma \subseteq \Gamma'$ *and* $A \subseteq A'$. *Then* $[\![\Gamma'; \eta \vdash e \leq_\diamond e' : И A'.\, \tau\langle\eta'\rangle]\!]$.

This lemma is shown as `closed_strengthen` in `closed_meta`.

Proof It is sufficient to show that $N \equiv_{\overline{A}} N'$ implies $N \equiv_{\overline{A}'} N'$, and that $[\![\Gamma']\!]_X(D, N, \sigma)$ implies $[\![\Gamma]\!]_X(D, N, \sigma)$ for $X = I, S$. Both are straightforward.

Lemma 20 (Strengthened form of 5) *Suppose, where appropriate, that* $\Gamma \vdash \eta_1$ *wf. The following hold:*

- $[\![\Gamma; \mathit{emp} \vdash c \leq_\diamond c : И\emptyset.\, \mathrm{ty}(c)\langle\mathit{emp}\rangle]\!]$;
- $[\![\Gamma; \mathit{emp} \vdash x \leq_\diamond x : И\emptyset.\, \tau\langle\mathit{emp}\rangle]\!]$, *where* $\Gamma(x) = \tau$;
- $[\![\Gamma; \eta_1 \vdash e_1 \leq_\diamond e_1' : И A_1.\, \tau_1\langle\eta_2\rangle]\!]$ *and* $[\![\Gamma, A_1, x : \tau_1; \eta_2 \vdash e_2 \leq_\diamond e_2' : И A_2.\, \tau_2\langle\eta_3\rangle]\!]$ *imply* $[\![\Gamma; \eta_1 \vdash \mathtt{let}\, x = e_1\, \mathtt{in}\, e_2 \leq_\diamond \mathtt{let}\, x = e_1'\, \mathtt{in}\, e_2' : И A_1, A_2.\, \tau_2\langle\eta_3\rangle]\!]$.
- $[\![\Gamma; \eta_1 \vdash e_1 \leq_\diamond e_1' : И A_1.\, \mathtt{bool}\langle\eta_2\rangle]\!]$, $[\![\Gamma; \eta_2 \vdash e_2 \leq_\diamond e_2' : И A_2.\, \tau\langle\eta_3\rangle]\!]$ *and* $[\![\Gamma; \eta_2 \vdash e_3 \leq_\diamond e_3' : И A_2.\, \tau\langle\eta_3\rangle]\!]$ *imply* $[\![\Gamma; \eta_2 \vdash \mathtt{if}\, e_1\, \mathtt{then}\, e_2\, \mathtt{else}\, e_3 \leq_\diamond \mathtt{if}\, e_1'\, \mathtt{then}\, e_2'\, \mathtt{else}\, e_3' : И A_1, A_2.\, \tau\langle\eta_3\rangle]\!]$.
- $[\![\Gamma; \eta_1 \vdash e \leq_\diamond e' : И A.\, \tau\langle\eta_2\rangle]\!]$, $\xi \notin \Gamma$, $\xi \notin A$ *and* $\Gamma, A, \{x\} \vdash \eta_2 * \xi \mapsto \tau$ *wf imply* $[\![\Gamma; \eta_1 \vdash \mathtt{ref}\, e \leq_\diamond \mathtt{ref}\, e' : И A, \xi.\, \mathit{ref}_\xi\, \langle\eta_2 * \xi \mapsto \tau\rangle]\!]$.
- $[\![\Gamma; \eta_1 \vdash e \leq_\diamond e' : И A.\, \mathit{ref}_\xi\, \langle\eta_2 * \xi \mapsto \tau\rangle]\!]$ *implies* $[\![\Gamma; \eta_1 \vdash \mathtt{!}\, e \leq_\diamond \mathtt{!}\, e' : И A.\, \tau\langle\eta_2 * \xi \mapsto \tau\rangle]\!]$.
- $[\![\Gamma; \eta_1 \vdash e_1 \leq_\diamond e_1' : И A_1.\, \mathit{ref}_\xi\, \langle\eta_2\rangle]\!]$ *and* $[\![\Gamma, A_1; \eta_2 \vdash e_2 \leq_\diamond e_2' : И A.\, \tau\langle\eta_3 * \xi \mapsto \tau\rangle]\!]$ *implies* $[\![\Gamma; \eta_1 \vdash e_1 := e_2 \leq_\diamond e_1' := e_2' : И A_1, A_2.\, \mathit{unit}\langle\eta_3 * \xi \mapsto \tau\rangle]\!]$.
- *Suppose* $[\![\Gamma; \eta \vdash e \leq_\diamond e' : И A.\, \tau\langle\eta'\rangle]\!]$, $\xi \notin \Gamma$, $\xi \notin A$, *and* $\Gamma, A, \xi \vdash \mathit{Wait}(\xi, A, \eta')$ *wf. Then* $[\![\Gamma; \eta \vdash \mathtt{post}\, e \leq_\diamond \mathtt{post}\, e' : И\{\xi\}.\, \mathit{promise}_{\xi,A}\, \tau\langle\mathit{Wait}(\xi, A, \eta')\rangle]\!]$.
- $[\![\Gamma; \eta_1 \vdash e \leq_\diamond e' : И A_1.\, \mathit{promise}_{\xi,A_2}\, \tau\langle\eta_2 * \mathit{Wait}(\xi, A_2, \eta_3)\rangle]\!]$ *together with* $A_1 \cap A_2 = \mathrm{dom}\,\Gamma \cap A_2 = \emptyset$, $\xi \notin \mathit{rnames}(\tau)$, $\xi \notin \mathit{rnames}(\eta_3)$ *and* $\Gamma, A_1, A_2 \vdash \eta_2 * \eta_3$ *wf imply* $[\![\Gamma; \eta_1 \vdash \mathtt{wait}\, e \leq_\diamond \mathtt{wait}\, e' : И A_1 \cup A_2.\, \tau\langle\eta_2 * \eta_3\rangle]\!]$.
- $[\![\Gamma; \eta_1 \vdash e \leq_\diamond e' : И A.\, \tau\langle\eta_2\rangle]\!]$, *together with* $\mathit{rnames}(\eta_1) \cap \mathit{rnames}(\eta_f) = \mathit{rnames}(\eta_2) \cap \mathit{rnames}(\eta_f) = \emptyset$, $\mathrm{Names}(\eta_f) \cap A = \emptyset$, $\Gamma \vdash \eta_f$ *wf and* $\Gamma, A \vdash \eta_f$ *wf imply*

$[\![\Gamma; \eta_1 * \eta_f \vdash e \leq_\diamond e' : \text{И}A.\, \tau\langle \eta_2 * \eta_f \rangle]\!]$.

- $[\![\Gamma'; \eta \vdash e \leq_\diamond e' : \text{И}A'.\, \tau\langle \eta' \rangle]\!]$, *together with* $\Gamma' \subseteq \Gamma$, $A' \subseteq A$, $\Gamma' \vdash \eta_1$ *wf and* $\Gamma, A \vdash \eta_2$ *wf imply* $[\![\Gamma; \eta \vdash e \leq_\diamond e' : \text{И}A.\, \tau\langle \eta' \rangle]\!]$.

The cases of this lemma as shown in `closure`.

Proof By reduction to Lemmas 5 and 4.

Lemma 21 (Closure under well-typed $\mathcal{E}$, Lemma 7) *Suppose* $\Gamma \vdash \eta$ *wf,* $\bar{\Gamma}; \bar{\eta} \vdash \bullet : \text{И}\bar{A}.\, \bar{\tau}\langle \bar{\eta}' \rangle \rightsquigarrow \Gamma; \eta \vdash \mathcal{E} : \text{И}A.\, \tau\langle \eta' \rangle$ *and* $[\![\bar{\Gamma}; \bar{\eta} \vdash e \leq_\diamond e' : \text{И}\bar{A}.\, \bar{\tau}\langle \bar{\eta}' \rangle]\!]$.
Then $[\![\Gamma; \eta \vdash \mathcal{E}[e] \leq_\diamond \mathcal{E}[e'] : \text{И}A.\, \tau\langle \eta' \rangle]\!]$.

This lemma is shown as `ectx_closed` in `contexttypes`.

Proof By a simple induction on the derivation of $\bar{\Gamma}; \bar{\eta} \vdash \bullet : \text{И}\bar{A}.\, \bar{\tau}\langle \bar{\eta}' \rangle \rightsquigarrow \Gamma; \eta \vdash \mathcal{E} : \text{И}A.\, \tau\langle \eta' \rangle$, using Lemma 20 for all cases case but the trivial case ET-Hole.

Lemma 22 (Soundness of R-Asynchronize, Lemma 8) *Suppose* $\Gamma; \eta \vdash e : \text{И}A.\, \tau\langle \eta' \rangle$, $\xi \notin \operatorname{dom}\Gamma \cup A$ *and* $\Gamma, \xi \vdash \mathit{Wait}(\xi, A, \eta')$ *wf. Then* $[\![\Gamma; \eta \vdash \texttt{wait}(\texttt{post}\, e) \leq_\diamond e : \text{И}A, \xi.\, \tau\langle \eta' \rangle]\!]$.

This lemma is shown as `sound_asynchronise` in `simplerules`.

Proof Using Lemma 6, we get $[\![\Gamma; \eta \vdash e \leq_\diamond e : \text{И}A, \xi.\, \tau\langle \eta' \rangle]\!]$.
From this, we show that $[\![\Gamma, \xi; \eta \vdash e \leq_\diamond e : \text{И}A, \xi.\, \tau\langle \eta' \rangle]\!]$, and applying H-Post and H-Wait in turn proves the claim.

Lemma 23 (Soundness of R-Commute, Lemma 9) *Suppose*

- $\Gamma; \eta_1 \vdash e_1 : \text{И}A_1.\, \tau_1\langle \eta_1' \rangle$,
- $\Gamma; \eta_2 \vdash e_2 : \text{И}A_2.\, \tau_2\langle \eta_2' \rangle$,
- $\Gamma, A_1, A_2, x : \tau_1, y : \tau_2; \eta_1' * \eta_2' \vdash e : \text{И}A.\, \tau\langle \eta' \rangle$,
- $x \neq y$,
- $\mathit{rnames}(\eta_1) \cap \mathit{rnames}(\eta_2) = \emptyset$,
- $\mathit{rnames}(\eta_1') \cap \mathit{rnames}(\eta_2) = \emptyset$,
- $\mathit{rnames}(\eta_1) \cap \mathit{rnames}(\eta_2') = \emptyset$,
- $\Gamma, A_2 \vdash \eta_1$ *wf,* $\Gamma, A_1 \vdash \eta_2$ *wf,*
- $\Gamma, A_1, A_2 \vdash \eta_1' \otimes \eta_2'$ *wf,*
- $x, y \notin \operatorname{dom}\Gamma$.

Then

$$[\![\Gamma; \eta_1 * \eta_2 \vdash \begin{array}{l}\texttt{let}\, x_1 = e_1\, \texttt{in} \\ \texttt{let}\, x_2 = e_2\, \texttt{in} \\ e_3\end{array} \leq_\diamond \begin{array}{l}\texttt{let}\, x_2 = e_2\, \texttt{in} \\ \texttt{let}\, x_1 = e_1\, \texttt{in} \\ e_3\end{array} : \text{И} A_1, A_2, A.\, \tau\langle\eta'\rangle]\!]$$

This lemma is shown as `sound_commute` in `commutativity`.

Proof We just sketch the main idea of the proof; apart from this idea, everything else is just tedious routine reasoning about types and substitutions and heavy use of locality lemmas.

It is easy to show the following:

- $\Gamma; \eta_1 * \eta_2 \vdash e_1 : \text{И} A_1.\, \tau_1\langle\eta_1' * \eta_2\rangle$,
- $\Gamma; \eta_1 * \eta_2 \vdash e_2 : \text{И} A_1.\, \tau_1\langle\eta_1 * \eta_2'\rangle$,
- $\Gamma, A_2; \eta_1 * \eta_2' \vdash e_1 : \text{И} A_1.\, \tau_1\langle\eta_1' * \eta_2'\rangle$,
- $\Gamma, A_1; \eta_1' * \eta_2 \vdash e_2 : \text{И} A_1.\, \tau_1\langle\eta_1' * \eta_2'\rangle$,
- $\Gamma, A_2, y : \tau_2; \eta_1 * \eta_2' \vdash e_1 : \text{И} A_1.\, \tau_1\langle\eta_1' * \eta_2'\rangle$,
- $\Gamma, A_1, x : \tau_1; \eta_1' * \eta_2 \vdash e_2 : \text{И} A_1.\, \tau_1\langle\eta_1' * \eta_2'\rangle$,
- $[\![\Gamma; \eta_1 \vdash e_1 \leq_\diamond e_1 : \text{И} A_1.\, \tau_1\langle\eta_1'\rangle]\!]$,
- $[\![\Gamma; \eta_2 \vdash e_2 \leq_\diamond e_2 : \text{И} A_2.\, \tau_2\langle\eta_2'\rangle]\!]$,
- $[\![\Gamma, A_1, A_2, x : \tau_1, y : \tau_2; \eta_1' * \eta_2' \vdash e \leq_\diamond e : \text{И} A.\, \tau\langle\eta'\rangle]\!]$.

The last three cases use Lemma 6.

For the actual proof, we unfold the definition of delayed refinement. We have to prove:

$$\begin{array}{l}\{[\![\Gamma; \eta_1 * \eta_2]\!]_I(D, N, \sigma)\} \\ \texttt{let}\, y = e_2\, \texttt{in}\, \texttt{let}\, x = e_1\, \texttt{in}\, e \\ \left\{ v.\ \begin{array}{l}\exists D', N'.\ N \equiv_{\overline{A_1, A_2, A}} N' * \\ [\![\tau; \eta]\!]_(D', N', v) * \\ \underbrace{[\![\Gamma; \eta_1 * \eta_2 \vdash \texttt{let}\, x = e_1\, \texttt{in}\, \texttt{let}\, y = e_2\, \texttt{in}\, e : \text{И} A_1, A_2, A.\, \tau\langle\eta'\rangle]\!]_S(D, D')}_{\phi_{\text{post}}}\end{array} \right\}\end{array}$$

By using the simulation triple for e_2, we find that it is sufficient to prove, for some fixed v_2, N_2 and D_2:

$$\left\{[\![\Gamma; \eta_1]\!]_I(D, N, \sigma) * [\![\eta_2']\!]_I(D_2, N_2) * S_2\right\}\ \texttt{let}\, x = e_1\, \texttt{in}\, e[y/v_2]\ \{v.\ \phi_{\text{post}}\}$$

where

$$S_2 := [\![\Gamma; \eta_2 \vdash e_2 : \text{И} A_2.\, \tau_2\langle\eta_2'\rangle]\!]_S(D, D_2).$$

Set $D_2'.\xi := D_2.\xi$ for $\xi \in \mathrm{Names}(\eta_2')$, and $D_2.x := D.x$ otherwise; Similarly, set $N_2'.\xi := N_2.\xi$ for $\xi \in A_2$ and $N_2'.\xi := N.\xi$ otherwise.

By locality, we reduce to showing that

$$\left\{[\![\Gamma; \eta_1]\!]_I(D_2', N_2', \sigma) * [\![\eta_2']\!]_I(D_2, N_2) * S_2\right\}\ \texttt{let}\, x = e_1\, \texttt{in}\, e[y/v_2]\ \{v.\ \phi_{\mathrm{post}}\}$$

We can then apply the simulation triple for e_1 and find it is sufficient to prove, for some fixed v_1, N_1 and D_1:

$$\begin{array}{l}\left\{[\![\Gamma]\!]_I(N, D, \sigma) * [\![\tau_1; \eta_1']\!]_I(D_1, N_1, v_1) * [\![\tau_2; \eta_2']\!]_I(D_2, N_2, v_2) * S_2 * S_1\right\} \\ e[x/v_1, y/v_2] \\ \{v.\phi_{\mathrm{post}}\}.\end{array}$$

where

$$S_1 := [\![\Gamma; \eta_1 \vdash e_1 : \text{И} A_1.\, \tau_1\langle\eta_1'\rangle]\!]_S(D, D_1).$$

Define $D''.x := D_1.\boxdot$, $D''.y := D_2.\boxdot$, $D''.\xi := D_1.\xi$ for $\xi \in \mathrm{Names}(\eta_1')$, $D''.\xi := D_2.\xi$ for $\xi \in \mathrm{Names}(\eta_2')$ and $D''.\xi := D.\xi$ otherwise. Define $N''.\xi := N_1.\xi$ for $\xi \in A_1$, $N''.\xi := N_2.\xi$ for $\xi \in A_2$ and $N''.\xi := N.\xi$ otherwise. By locality, we find it is sufficient to prove:

$$\left\{[\![\Gamma, x : \tau_1, y : \tau_2; \eta_1' * \eta_2']\!]_I(N'', D'', \sigma[x \mapsto v_1, y \mapsto v_2]) * S_2 * S_1\right\}\ e[x/v_1, y/v_2]\ \{v.\phi_{\mathrm{post}}\}.$$

By applying the specification of e, we find it suffices to prove, for some fixed v, N', D', $\mathcal{C}$:

$$\begin{array}{r}S_1 * S_2 * S \vdash [\![\Gamma; \eta_1 * \eta_2]\!]_S(D, N, \sigma) * p \Rightarrow \mathsf{running}\colon \mathcal{C}[(\texttt{let}\, x = e_1\, \texttt{in}\, \texttt{let}\, y = e_2\, \texttt{in}\, e)\sigma] \\ \Rrightarrow \underbrace{\exists N', v.\ p \Rightarrow \mathsf{running}\colon \mathcal{C}[v] * [\![\tau; \eta]\!]_S(D', N', v)}_{\phi_{\mathrm{all}}}\end{array}$$

where

$$S := [\![\Gamma, A_1, A_2, x : \tau_1, y : \tau_2; \eta_1' * \eta_2' \vdash e : \text{И} A.\, \tau\langle\eta'\rangle]\!]_S(D'', D')$$

Using locality, we can change $[\![\Gamma; \eta_1 * \eta_2]\!]_S(D, N)$ into $[\![\Gamma; \eta_1]\!]_S(D_2', N) * [\![\eta_2]\!]_S(D, N)$. This allows us to apply S_1; we find there are v' and N_1 such that it suffices to show:

$$\begin{array}{r}S_2 * S \vdash [\![\Gamma; \eta_2]\!]_S(D, N, \sigma) * [\![\tau_1; \eta_1']\!]_S(D_1, N_1, v_1) * \\ p \Rightarrow \mathsf{running}\colon \mathcal{C}[(\texttt{let}\, y = e_2\, \texttt{in}\, e[x/v_1])\sigma] \Rrightarrow \phi_{\mathrm{all}}\end{array}$$

At this point, we can apply S_2 and find v_2, N_2 such that it is sufficient to prove:

$$\begin{array}{r}S \vdash [\![\Gamma]\!]_S(D, N, \sigma) * [\![\tau_1; \eta_1']\!]_S(D_1, N_1, v_1) * [\![\tau_2; \eta_2']\!]_S(D_2, N_2, v_2) * \\ p \Rightarrow \mathsf{running}\colon \mathcal{C}[e[x/v_1, y/v_2]]\sigma \Rrightarrow \phi_{\mathrm{all}}\end{array}$$

Define $N''.\xi := N_1.\xi$ for $\xi \in A_1$, $N''.\xi := N_2.\xi$ for $\xi \in A_2$ and $N''.\xi := N.\xi$ otherwise. By locality, we find that it is sufficient to prove:

$$S \vdash [\![\Gamma, A_1, A_2, x : \tau_1, y : \tau_2; \eta_1' * \eta_2']\!]_S(D'', N'', \sigma) * p \Rightarrow \mathsf{running}\colon \mathcal{C}[e[x/v_1, y/v_2]\sigma] \Rrightarrow \phi_{\mathrm{all}}$$

At this point, apply S to finish the proof.

Theorem 16 (Soundness of DWFM, Theorem 4) *Suppose* $\Gamma; \eta \vdash e : \mathsf{И} A.\, \tau\langle\eta'\rangle$ *and* $e \implies e'$. *Then* $[\![\Gamma; \eta \vdash e : \mathsf{И} A'.\, \tau\langle\eta'\rangle]\!]$ *for some* $A' \supseteq A$.

Most of the work for this theorem is done in `simplerules`.

Proof We prove a slight stronger version: Instead of the existence of a single A', we show that, for a given finite set B with $A \cap B = \emptyset$, there are infinitely many $A' \supseteq A$ such that $A' \cap B = \emptyset$.

By induction on the derivation of $e \implies e'$ and inversion of the typing derivation.

First, consider R-Context. In this case, we have to show two lemmas:

Claim: Suppose $\Gamma; \eta \vdash \mathcal{E}[e] : \mathsf{И} A.\, \tau\langle\eta'\rangle$. Then there are $\bar\Gamma, \bar\eta, \bar A, \bar\tau, \bar\eta'$ such that $\bar\Gamma; \bar\eta \vdash \bullet : \mathsf{И}\bar A.\, \bar\tau\langle\bar\eta'\rangle \rightsquigarrow \Gamma; \eta \vdash \mathcal{E} : \mathsf{И} A.\, \tau\langle\eta'\rangle$ and $\bar\Gamma; \bar\eta \vdash e : \mathsf{И}\bar A.\, \bar\tau\langle\bar\eta'\rangle$.

Proof: Induction on the size of the derivation of $\Gamma; \eta \vdash \mathcal{E}[e] : \mathsf{И} A.\, \tau\langle\eta'\rangle$. First suppose $\mathcal{E} \neq \bullet$ and the last rule is T-X. Then invert the typing derivation, apply ET-X and use the induction hypothesis.

Otherwise, choose $\bar\Gamma = \Gamma$, $\bar\eta = \eta$ and so on and apply ET-Hole.

Claim: Suppose $\bar\Gamma; \bar\eta \vdash \bullet : \mathsf{И}\bar A.\, \bar\tau\langle\bar\eta'\rangle \rightsquigarrow \Gamma; \eta \vdash \mathcal{E}[e] : \mathsf{И} A.\, \tau\langle\eta'\rangle$ and $\bar A \subseteq \bar A'$ such that $\bar A' \cap A \subseteq \bar A$. Then $\bar\Gamma; \bar\eta \vdash \bullet : \mathsf{И}\bar A'.\, \bar\tau\langle\bar\eta'\rangle \rightsquigarrow \Gamma; \eta \vdash \mathcal{E}[e] : \mathsf{И}\bar A' \cup A.\, \tau\langle\eta'\rangle$.

Proof: By induction over the derivation. The side condition is used to ensure that fresh names stay fresh.

Now, since we are in the R-Context case, we find that there are $\bar e$, $\bar e'$ such that $e = \mathcal{E}[\bar e]$ and $e' = \mathcal{E}[\bar e']$. Using the first claim, we find $\bar\Gamma$, $\bar\eta$, $\bar A$, $\bar\tau$, $\bar\eta'$ such that $\bar\Gamma; \bar\eta \vdash \bullet : \mathsf{И}\bar A.\, \bar\tau\langle\bar\eta'\rangle \rightsquigarrow \Gamma; \eta \vdash \mathcal{E} : \mathsf{И} A.\, \tau\langle\eta'\rangle$ and $\bar\Gamma; \bar\eta \vdash \bar e : \mathsf{И}\bar A.\, \bar\tau\langle\bar\eta'\rangle$.

By the induction hypothesis, we find that there are infinitely many $\bar A' \supseteq \bar A$ with $\bar A' \cap A \subseteq \bar A$ and $A' \cap B = \emptyset$ such that $[\![\bar\Gamma; \bar\eta \vdash \bar e' \leq_\diamond \bar e : \mathsf{И}\bar A.\, \bar\tau\langle\bar\eta'\rangle]\!]$. By the second claim, we have $\bar\Gamma; \bar\eta \vdash \bullet : \mathsf{И}\bar A'.\, \bar\tau\langle\bar\eta'\rangle \rightsquigarrow \Gamma; \eta \vdash \mathcal{E} : \mathsf{И}\bar A' \cup A.\, \tau\langle\eta'\rangle$. By Lemma 7, $[\![\Gamma; \eta \vdash e' \leq_\diamond e : \mathsf{И} A \cup \bar A'.\, \tau\langle\eta'\rangle]\!]$.

Next, consider R-Asynchronize. It is easy to check that there an infinite set of possible ξ such that all the preconditions of Lemma 8 are fulfilled.

For all other cases, choose any $A' \supseteq A$. It is easy to check, by inversion and pushing subtyping and framing as far down as possible, that the preconditions of the corresponding case lemma are satisfied.

We omit the (already proved) second part of the following theorem:

Theorem 17 (Context closure, Theorem 5) *Let* $e_1, e_2, \Gamma, \eta, A, \tau, \eta'$ *be given such that* $[\![\Gamma; \eta \vdash e_1 \leq_\diamond e_2 : \mathsf{И} A.\, \tau\langle\eta'\rangle]\!]$ *holds. Then* e_1 *is a contextual refinement of* e_2.

Proof In view of Lemma 7, it is sufficient to consider the case $\mathcal{E} = \bullet$, $\eta = \mathsf{emp}$, $\Gamma = \cdot$.

Unfold the definitions. We have:

$$\{\top\}\ e_1\ \left\{ x_1.\exists N_I', D'.\ \begin{array}{l} N_I \equiv_{\overline{A}} N_I' * [\![\tau; \eta']\!]_I(D', N_I', x_1) * \\ [\![\cdot; \mathsf{emp} \vdash e_2 : \mathsf{И} A.\, \tau\langle\eta'\rangle]\!]_S(D, D') \end{array} \right\}$$

Unfolding $[\![\cdot; \mathsf{emp} \vdash e_2 : \mathit{M}A.\, \tau\langle\eta'\rangle]\!]_S(D, D')$ gives, for all N, σ, p, $\mathcal{C}$:

$$\top * p \Mapsto \mathsf{running}{:}\,\mathcal{C}[e_2] \Rrightarrow \exists v.\ \exists N'.\ N \equiv_{\overline{A}} N' * [\![\tau; \eta']\!]_S(D', N', v) * p \Mapsto \mathsf{running}{:}\,\mathcal{C}[v]$$

By weakening both triples to remove $[\![\tau; \eta']\!]_{\ldots}$ and now-useless quantification, we get (fixing $\mathcal{C} = \bullet$):

$$\{\top\}\ \ e_1\ \ \{x_1.\forall p, \mathcal{C}.\ p \Mapsto \mathsf{running}{:}\,\mathcal{C}[e_2] \Rrightarrow \exists v.\ p \Mapsto \mathsf{running}{:}\,\mathcal{C}[v]\}$$

Applying the soundness theorem for universal Hoare triples, this reduces to the following statement: For every execution of e_1 that reduces e_1 to a value, it holds that $\forall p, \mathcal{C}.\ p \Mapsto \mathsf{running}{:}\,\mathcal{C}[e] \Rrightarrow \exists v.\ p \Mapsto \mathsf{running}{:}\,\mathcal{C}[v]$. But by the semantics of $\Rrightarrow$, this implies that there is an execution that reduces e_2 to a value.

C. Curriculum Vitae

Research Interests

Concurrency (in particular, asynchronous, task-based and event-based concurrency), dynamic and static program analysis, program logics and type systems.

Employment

May 2018 — to date Reseach Engineer, Diffblue Ltd, Oxford

October 2011 — March 2018 Doctoral student, MPI-SWS, Kaiserslautern

During this time, I worked on various projects:

- Cyber-physical systems: I designed a strategy for safely switching between different controller implementations without losing stability.
- IIC: I worked on an efficient algorithm for coverability analysis in a specific class of infinite-state systems, namely well-structured transition systems, and I worked on a tool that implements this algorithm for Petri nets. The algorithm is an extension of Bradley's IC3 algorithm.

  The tool implementation is available on request; experimentally, we found that our tool is competitive with other state-of-the-art analysis tools for Petri nets.
- ALST: I extended Rondon et al.'s Liquid Types to handle programs that incorporate asynchronous function calls.

  This project consisted of two parts: Extending a rich type system with features to track the structure of the heap and outstanding asynchronous calls ("tasks"), and an analysis algorithm that derives the heap and task structure for OCaml programs augmented with asynchronous call primitives.
- JSDefer: A performance optimization for web pages, based on an analysis that identifies scripts which can be loaded in the background using the HTML5 "defer" attribute.

  The tool can be found at my Github page under `https://github.com/johanneskloos/JSDefer-eventracer`; an experimental evaluation can be found in my HVC 2017 paper (see `https://www.mpi-sws.org/~jkloos/hvc.pdf`).
- DWFM: Proving the correctness of a family of optimization schemes for programs using asynchronous/event-based concurrency.

This work was largely theoretical, combining methods from program logics and logical relations to show that a number of common rewriting rules preserve program behavior.

From June 15th to October 31st 2015, I was on leave from my PhD program for an internship in industry.

June 2015 — October 2015 Research intern, Instart Logic, Palo Alto

During this time, I worked on a dynamic analysis tool to help check the correctness of a program transformation scheme used internally at Instart Logic, and performed initial work on JSDefer.

April 2008 — September 2011 Research engineer, Fraunhofer IESE, Kaiserslautern

I worked on various projects (industrial and publicly funded research) in connection with model-based testing, with a focus on test case derivation from to usage models and safety analyses.

Education

October 2011 — December 2017 PhD program, Max Planck Institute for Software Systems (compare employment)

October 2001 — April 2008 Diplom in Informatik,
Technische Universität Kaiserslautern, Kaiserslautern

I studied computer science, with a focus on theoretical computer science.

Publications

1. Johannes Kloos and Rupak Majumdar, "Supervisor Synthesis for Controller Upgrades", DATE 2013
2. Johannes Kloos, Filip Niksic, Ruzica Piskac and Rupak Majumdar, "Incremental Inductive Coverability", CAV 2013
3. Johannes Kloos, Rupak Majumdar and Viktor Vafeiadis, "Asynchronous Liquid Separation Types", ECOOP 2015
4. Johannes Kloos, Rupak Majumdar and Frank McCabe, "Deferrability Analysis for JavaScript", HVC 2017.