
# Skipping the Binder Bureaucracy with Mixed Embeddings in a Semantics Course (Functional Pearl)

ADAM CHLIPALA, MIT, USA

Rigorous reasoning about programs calls for some amount of bureaucracy in managing details like variable binding, but, in guiding students through big ideas in semantics, we might hope to minimize the overhead. We describe our experiment introducing a range of such ideas, using the Coq proof assistant, without any explicit representation of variables, instead using a higher-order syntax encoding that we dub "mixed embedding": it is neither the fully explicit syntax of deep embeddings nor the syntax-free programming of shallow embeddings. Marquee examples include different takes on concurrency reasoning, including in the traditions of model checking (partial-order reduction), program logics (concurrent separation logic), and type checking (session types) – all presented without any side conditions on variables.

CCS Concepts: • **Theory of computation** → **Program semantics**; **Program reasoning**.

Additional Key Words and Phrases: formal verification, proof assistants, mechanized semantics, binder encoding

**ACM Reference Format:**
Adam Chlipala. 2021. Skipping the Binder Bureaucracy with Mixed Embeddings in a Semantics Course (Functional Pearl). *Proc. ACM Program. Lang.* 5, ICFP, Article 94 (August 2021), 28 pages. https://doi.org/10.1145/3473599

## 1 INTRODUCTION

The seminal paper on separation logic [Reynolds 2002] presents its intellectual core, the frame rule, as

$$\frac{\{p\}c\{q\}}{\{p * r\}c\{q * r\}}$$

with a side condition, "where no variable occurring free in $r$ is modified by $c$." Evidently we must formalize concepts like free and modified variables before grappling with the meaty questions of modular reasoning about programs. Students new to semantics might prefer to get to the punchline.

The same bureaucracy shows up around many of the biggest ideas in program proof, from model checking (to understand commutativity of concurrent threads, must we track carefully which variables are local to which threads?) to type systems (must a type-soundness proof necessarily involve structural lemmas about typing contexts?). In a university course based on the Coq proof assistant, we set out to introduce these big ideas to students without the bureaucracy. Our online book, freely available with both LaTeX source and Coq examples, is called *Formal Reasoning About Programs*[1]. The author has used it in five course offerings so far.

We do follow *Software Foundations* [Pierce et al. 2018] in presenting traditional type-soundness proofs with explicit variables and typing contexts, taking advantage of the pleasant coincidence that simply typed languages can be formalized relying on capture-avoiding substitution only for

[1] http://adam.chlipala.net/frap/

Author's address: Adam Chlipala, MIT, Cambridge, MA, USA, adamc@csail.mit.edu.


2475-1421/2021/8-ART94
https://doi.org/10.1145/3473599

closed terms. However, after that glimpse inside the proverbial sausage factory of binder encoding, we change gears and formalize languages with no explicit representation of variables.

Our gamble is that functional programming has gone mainstream enough (e.g., uses of continuations in callback-driven libraries in many languages) that students are prepared to deal with *higher-order* encodings, where bodies of binding constructs are represented with functions of the metalanguage. Higher-order abstract syntax [Harper et al. 1987; Pfenning and Elliot 1988] is the best-known technique in the family, but it is difficult to integrate with general-purpose proof assistants, since it requires relatively weak function spaces that rule out inspection of syntax trees simply by omitting programming features for pattern matching. Instead, we can use functions over *semantic values*, rather than over syntax. This encoding is not suitable for explicit introspection into binding structure, but, as we will show, many of the big ideas in semantics are quite orthogonal to such introspection (e.g., when do two threads commute with each other? when is it permissible to combine two verified data structures into a new verified procedure?). Furthermore, this style allows for very short formal definitions of languages that are nonetheless quite flexible and pleasant to program in, thanks to the implicit inclusion of all programming features from the metalanguage, much as e.g. monads are used to add side effects to Haskell [Peyton Jones and Wadler 1993].

This style, which is associated with techniques like interaction trees [Xia et al. 2020], we dub *mixed embedding*, in contrast to deep embedding and shallow embedding (all to be reviewed shortly). The new terminology seemed called for, as here there is a whole family of interesting encoding variations, somewhat divorced from semantics variations. The style has been known at least as folklore for quite some time, and our goal in this paper is to illustrate how it is a good fit for smooth presentation of several classic ideas at the highest standard of rigor but without muddling the picture with binder bookkeeping.

Higher-order encodings and nitpicking over binder details both impose conceptual overhead on students. Computer-science pedagogy is a thriving research field, with its empirical methods for evaluating how different interventions help students learn. Like most developers of new graduate-level university courses, the author was not nearly so systematic about the effort, instead just working with his teaching assistants to gauge what students were able to accomplish with reasonable effort, adjusting book content, lectures, and assignments accordingly. The most recent offering (Spring 2021) turned out pretty well in that respect, in terms of how long students reported they spent completing the assignments, which are designed to provide a gentle ramp-up through the material and to take an average of eight hours each, across students. About 25 students stuck with the assignments all the way through the semester; perhaps as many as 10 of them were already specializing in programming-languages research at the start of the term, but most did not have background fine-tuned to the subject matter. Out of twelve assignments, the average time required was between 7 and 9 hours for four of them, between 9 and 11 hours for six, and under 7 hours for the remaining two.

This summary is hardly the result of careful analysis of a controlled experiment, and indeed such a principled study could be worth doing, but we are sufficiently convinced that the recipe works to guide students to a point of carrying out examples significantly more advanced than in competing courses. For instance, instead of hitting a high-water mark of proving some simple "while" programs with Hoare logic, we have students doing mostly automated proofs of concurrent programs with heap-allocated data structures, using separation logic. Every exercise in program verification is backed by first-principles soundness proof of all ingredients. Readers who have worked extensively with mechanized metatheory of expressive languages will know well the pain of dealing with binder bookkeeping, and we consider it likely that the encoding style demonstrated in this paper makes a significant difference by dodging such bookkeeping.

Many introductions to semantics will, e.g., present the syntactic approach to type soundness [Wright and Felleisen 1994] as though its progress and preservation conditions are constructed from whole cloth as part of a problem-specific proof method. We prefer to emphasize commonalities across approaches, e.g. revealing said syntactic approach as just another case of finding a clever strengthened invariant for a transition system. Therefore, we rely on a number of common definitions, not belabored here because the book and associated Coq code lay them out in detail. A *transition system* encompasses a set of states, a subset of them marked initial, and a binary transition relation. An *invariant* of a transition system is a property that is always true, in all of the system's reachable states. Some invariants are *inductive*, in the sense that we can prove them fairly directly by induction. More commonly, we need to do the equivalent of *strengthening the induction hypothesis* to prove an invariant, switching to proving a stronger invariant. These terms will recur through all our examples.

This paper is written for an audience mostly already familiar with the semantic techniques in question, though it demonstrates an explanatory style designed to work for readers encountering these techniques for the first time, only occasionally adding remarks that put the encoding decisions in context. (Think of the latter as moments when the lecturer in front of class freezes the students to make a remark to a colleague in the audience, justifying a pedagogical choice.) The reading experience is designed to be a pleasant ride through familiar material, at the end of which we hope the reader agrees things worked out more smoothly than we are used to in related proof-assistant developments.

## 2 DEEP, SHALLOW, AND MIXED EMBEDDINGS

### 2.1 The Basics

Every formal proof is carried out in some *metalanguage*, which, in our case, is Coq's logic and programming language called Gallina. A syntactic language that we formalize is called an *object language*. Often it is convenient to do reasoning without any particular object language, as in this simple arithmetic function that can be defined directly in Gallina.

```
Definition foo (x y : nat) : nat :=
  let u := x + y in
  let v := u * y in
  u + v.
```

However, it is difficult to prove some important facts about terms encoded directly in the metalanguage. For instance, we can't easily do induction over the syntax of all such terms. To allow that kind of induction, we can define an object language inductively.

```
Inductive exp :=
| Const (n : nat)
| Var (x : var)
| Plus (e1 : exp) (e2 : exp)
| Times (e1 : exp) (e2 : exp)
| Let (x : var) (e1 : exp) (e2 : exp).
```

That last example program, with implicit free variables $x$ and $y$, may now be redefined in the `exp` type.

```
Definition foo' : exp :=
  Let "u" (Plus (Var "x") (Var "y"))
      (Let "v" (Times (Var "u") (Var "y"))
           (Plus (Var "u") (Var "v"))).
```

We say that `foo` uses a *shallow embedding*, because it is coded directly in the metalanguage, with no extra layer of syntax. Conversely, `foo'` uses a *deep embedding*, since it goes via the inductively defined `exp` type. (This terminology was introduced by Boulton et al. [1992] and is ubiquitous among users of proof assistants today.)

These extremes are not our only options. In higher-order logics like Coq's, we may also choose what might be called *mixed embeddings*, which define syntax-tree types that allow some use of general functions from the metalanguage. Here's an example, as an alternative definition of `exp`.

```
Inductive exp :=
| Const (n : nat)
| Var (x : string)
| Plus (e1 : exp) (e2 : exp)
| Times (e1 : exp) (e2 : exp)
| Let (e1 : exp) (e2 : nat -> exp).
```

The one change is in the type of the `Let` constructor, where now no variable name is given, and instead *the body of the "let" is represented as a Gallina function from numbers to expressions.* The intent is that the body is called on the number that results from evaluating the first expression. We dub this style mixed embedding because it includes both explicit syntax trees (as in deep embeddings) and appeals to the metalanguage's function spaces (as in shallow embeddings).

As an illustration of the technique in action, here's our third encoding of the simple example program.

```
Definition foo'' : exp :=
  Let (Plus (Var "x") (Var "y"))
      (fun u => Let (Times (Const u) (Var "y"))
                    (fun v => Plus (Const u) (Const v))).
```

With a bit of subtlety, we can define an interpreter for this language.

```
Fixpoint interp (e : exp) (v : fmap var nat) : nat :=
  match e with
  | Var x => v $! x (* operation for lookup in a finite map *)
  | Const n => n
  | Plus e1 e2 => interp e1 v + interp e2 v
  | Times e1 e2 => interp e1 v * interp e2 v
  | Let e1 e2 => interp (e2 (interp e1 v)) v
  end.
```

Note how, in the `Let` case, since the body `e2` is a function, before evaluating it, we call it on the result of evaluating `e1`. This language would actually be sufficient even if we removed the `Var` constructor and the `v` argument of the interpreter. Coq's normal variable binding is enough to let us model interesting programs and prove things about them by induction on syntax.

It is important here that Coq's induction principles give us useful induction hypotheses, for constructors whose recursive arguments are functions. The second argument of `Let` above is an example. When we do induction on expression syntax to establish `forall e, P(e)` (e.g., `P(e)` could express that if `e` passes a certain static analysis, it cannot evaluate to zero), the case for `Let e1 e2` includes two induction hypotheses. The first one is standard: `P(e1)`. The second one is more interesting: `forall n : nat. P(e2(n))`. That is, the theorem holds on all results of applying body `e2` to arguments.

What we have seen so far is very similar to the well-established technique of higher-order abstract syntax (HOAS) [Harper et al. 1987; Pfenning and Elliot 1988]. The key difference is that HOAS represents binders with functions over *the syntax to be substituted for bound variables*, rather than here the *semantics* to be substituted. It is also important that HOAS is usually applied with much weaker function spaces than we use here, for instance forbidding the function from doing syntactic case analysis on its argument. Otherwise, it is possible to write *exotic terms* that do not correspond to programs in the object language, as it is usually formalized on paper. For instance, in any sort of object language using type polymorphism to do information hiding, we worry that exotic terms could allow violation of encapsulation guarantees, say by direct introspection into the syntax of a value that we should consider to belong to an abstract type. Many kinds of classic metatheorems of interest would be falsified. However, with a mixed embedding where we embrace the chance to write exotic terms for programming convenience, many other classic theoretical exercises become easier to carry out on more interesting object-language programs, as we demonstrate next.

### 2.2 A Mixed Embedding for Hoare Logic

This general strategy also applies to modeling imperative languages, say with a simple global heap indexed by naturals and storing a natural per cell. We can define a polymorphic type family `cmd` of commands, indexed by the type of value that a command is meant to return.

```
Inductive cmd : Set -> Type :=
| Return {result : Set} (r : result) : cmd result
| Bind {result result'} (c1 : cmd result') (c2 : result' -> cmd result)
  : cmd result
| Read (a : nat) : cmd nat
| Write (a v : nat) : cmd unit.
```

It is possible to follow along with the details in the associated Coq code, so we switch to favoring more traditional LaTeX syntax, now that we have laid out the big ideas of the encoding. The above definition restated becomes:

$$\begin{aligned}
\mathsf{Return} &: \forall \alpha.\ \alpha \to \mathsf{cmd}\ \alpha \\
\mathsf{Bind} &: \forall \alpha, \beta.\ \mathsf{cmd}\ \beta \to (\beta \to \mathsf{cmd}\ \alpha) \to \mathsf{cmd}\ \alpha \\
\mathsf{Read} &: \mathbb{N} \to \mathsf{cmd}\ \mathbb{N} \\
\mathsf{Write} &: \mathbb{N} \to \mathbb{N} \to \mathsf{cmd}\ \mathsf{unit}
\end{aligned}$$

We use notation $x \leftarrow c_1; c_2$ as shorthand for Bind $c_1$ $(\lambda x.\ c_2)$, making it possible to write some very natural-looking programs in this type. Here are two examples.

$$\begin{aligned}
\text{array_max}(0, a) &= \text{Return } a \\
\text{array_max}(i+1, a) &= v \leftarrow \text{Read } i; \text{array_max}(i, \max(v, a))
\end{aligned}$$

$$\begin{aligned}
\text{increment_all}(0) &= \text{Return } () \\
\text{increment_all}(i+1) &= v \leftarrow \text{Read } i; _ \leftarrow \text{Write } i\ (v+1); \text{increment_all}(i)
\end{aligned}$$

Function array_max computes the highest value found in the first $i$ slots of memory, using an accumulator $a$. Function increment_all adds 1 to every one of the first $i$ memory slots.

These examples include a subtlety in the distinction between metalanguage and object language. Reverting briefly to Coq syntax is instructive:

```
Fixpoint array_max (i acc : nat) : cmd nat :=
  match i with
  | O => Return acc
  | S i' =>
    h_i' <- Read i';
    array_max i' (max h_i' acc)
  end.
```

Note that we are not writing programs directly as syntax trees but rather working with recursive functions that *compute syntax trees*. We are able to do so despite the fact that we built no support for recursion into the cmd type family. Likewise, we didn't need to build in any support for max, addition, pattern matching, or any of the other operations that are easy to code up in Gallina. Thus, students can start programming and proving with these features immediately, despite not investing in explicit object-language features.

It is straightforward to implement an interpreter for this object language, where each command's interpretation maps input heaps to pairs of output heaps and results. Note that we have no need for an explicit variable environment. Also, all notations from this definition not explicitly defined above are standard Coq notations, representing native operations of the metalanguage.

$$\begin{aligned}
[\![\text{Return } v]\!]h &= (h, v) \\
[\![\text{Bind } c_1\ c_2]\!]h &= \text{let } (h', v) = [\![c_1]\!]h \text{ in } [\![c_2(v)]\!]h' \\
[\![\text{Read } a]\!]h &= (h, h(a)) \\
[\![\text{Write } a\ v]\!]h &= (h[a \mapsto v], ())
\end{aligned}$$

We can also define a syntactic Hoare-logic [Hoare 1969] relation for this type, where preconditions are predicates are over initial heaps, and postconditions are predicates over *result values* and final heaps. Here our predicates are native predicates of the metalanguage, e.g. a precondition has type `heap -> Prop`. Thus, we use native notations of the metalanguage in these predicates, with no need to enumerate a grammar of predicate constructs (since we will not need to do induction on predicate structure, the way we need to do so for command structure).

$$\frac{}{\{P\}\text{Return } v\{\lambda r, h.\ P(h) \wedge r = v\}} \qquad \frac{\{P\}c_1\{Q\} \quad \forall r.\ \{Q(r)\}c_2(r)\{R\}}{\{P\}\text{Bind } c_1\ c_2\{R\}}$$

$$\frac{}{\{P\}\mathsf{Read}\ a\{\lambda r, h.\ P(h) \wedge r = h(a)\}} \quad \frac{}{\{P\}\mathsf{Write}\ a\ v\{\lambda r, h.\ \exists h'.\ P(h') \wedge h = h'[a \mapsto v]\}}$$

$$\frac{\{P\}c\{Q\} \quad (\forall h.\ P'(h) \Rightarrow P(h)) \quad (\forall r, h.\ Q(r, h) \Rightarrow Q'(r, h))}{\{P'\}c\{Q'\}}$$

The most interesting wrinkle is in the rule for Bind, where the premise about the body command $c_2$ starts with universal quantification over all possible results $r$ of executing $c_1$. That result is passed off, via function application, both to the body $c_2$ and to $Q$, which serves as the postcondition of $c_1$ and the precondition of $c_2$.

This Hoare logic can be used to verify the two example programs from earlier in this section; see the accompanying Coq code for details. We also have a standard soundness theorem, which follows by induction on Hoare-logic derivations (in a seven-line proof script that mostly delegates to standard automation tactics).

Theorem 2.1. *If $\{P\}c\{Q\}$ and $P(h)$ for some heap $h$, then let $(h', r) = [\![c]\!]h$. It follows that $Q(r, h')$.*

Pausing briefly to reflect: all these ideas have been developed without any explicit representation of variables or their values. The same can be said of HOAS, but we see another key advantage in these examples: we are able to use the full power of the metalanguage Gallina in programming with effects. A metaphor might help explain the source of the advantage. IO features could very well have been added to Haskell by introducing a deep embedding of syntax trees for imperative programs. We would need to add a constructor to this tree type for every useful operation, such as recursion or pattern matching. Instead, Peyton Jones and Wadler [1993] chose the more elegant way of introducing a monad, such that arbitrary pure functions can be used to chain together side effects (following on from earlier work that represented such computations as explicit syntax trees much like ours, as studied by Gordon [1993] in the context of formal program-equivalence reasoning). Another perspective is that exotic terms are a major worry for some kinds of metatheory, but for many exercises in program verification, *exotic terms are very handy to allow coding of interesting programs*! With no explicit effort at language-definition time, we can allow students to write and verify impure programs using the full suite of functional-programming constructs from the metalanguage.

It is worth also drawing a connection to work using weak HOAS and the theory of contexts [Honsell et al. 2001a] within general-purpose proof assistants, for instance to formalize $\pi$-calculus [Honsell et al. 2001b]. In these formalizations, a binder body is a function from a distinguished type of *names*. To avoid exotic terms, that special type is intentionally impoverished, supporting at most equality comparison. As a result, examples as we just went through are impossible, as for instance we cannot use these names to drive interesting patterns of recursion or pattern matching. Instead, we are back to needing to code all interesting programming constructs explicitly in our syntax-tree type, which makes sense for many metatheoretical studies but not for getting up-to-speed quickly with interesting program verification.

### 2.3 Adding More Effects

We can continue to enhance our object language with different kinds of side effects that are not supported natively by Gallina. First, we add *nontermination*, in the form of unbounded loops. For a type $\alpha$, we define $\mathbb{O}(\alpha)$ as the type of *loop-body outcomes*, either $\mathsf{Done}(a)$ to indicate that the loop should terminate or $\mathsf{Again}(a)$ to indicate that the loop should keep running. Our loops are functional, maintaining accumulators as they run, and the $a$ argument gives the latest accumulator

value in each case. So we add this constructor:

$$\mathsf{Loop} \quad : \quad \forall \alpha.\ \alpha \to (\alpha \to \mathsf{cmd}\ (\mathbb{O}(\alpha))) \to \mathsf{cmd}\ \alpha$$

Here's an example of looping in action, in a program that returns the address of the first occurrence of a value in memory or loops forever if that value is not found in the whole infinite memory.

$$\begin{aligned}\mathsf{index\_of}(n) \quad = \quad & \mathsf{Loop}\ 0\ (\lambda i.\ v \leftarrow \mathsf{Read}\ i; \mathsf{if}\ v = n\ \mathsf{then}\ \mathsf{Return}\ (\mathsf{Done}(i)) \\ & \mathsf{else}\ \mathsf{Return}\ (\mathsf{Again}(i+1)))\end{aligned}$$

With the addition of nontermination, it's no longer straightforward to write an interpreter in Coq, which enforces termination for all recursive functions. Instead, we implement a small-step operational semantics $\longrightarrow$; see the accompanying Coq code for details, or look ahead to subsection 3.1. We build an extended Hoare logic, keeping all the rules from last section and adding the following new one, and we have implicitly switched from a *total-correctness* logic (enforces termination) to *partial correctness* (postcondition characterizes what is true *if* the program terminates).

Like before, the loop proof rule is parameterized on a loop invariant, but now the loop invariant takes a loop-body outcome as parameter.

$$\frac{\forall a.\ \{I(\mathsf{Again}(a))\}c(a)\{I\}}{\{I(\mathsf{Again}(i))\}\mathsf{Loop}\ i\ c\{\lambda r.\ I(\mathsf{Done}(r))\}}$$

This new Hoare logic is usable to verify the example program from above and many more, and we can also prove a soundness theorem. The operational semantics gives us the standard way of interpreting our programs as transition systems, with states $(c, h)$.

Theorem 2.2. *If $\{P\}c\{Q\}$ and $P(h)$ for some heap $h$, then if $c$ terminates when run in $h$, the final heap and return value satisfy $Q$. (We formalize this statement in our common language of invariants, saying it is an invariant that any reachable state is either not finished or satisfies $Q$.)*

We can add a further side effect to the language: *exceptions*. Actually, we stick to a simple variant of this classic side effect, where there is just one exception, and it cannot be caught. We associate this exception with *program failure*, and the Hoare logic will ensure that programs never actually fail.

The extension to program syntax is easy:

$$\mathsf{Fail} \quad : \quad \forall \alpha.\ \mathsf{cmd}\ \alpha$$

That is, a failing program can be considered to return any result type, since it will never actually return normally, instead throwing an uncatchable exception.

The operational semantics is also easily extended to signal failures, with a new special system state called Failed. We also add this Hoare-logic rule.

$$\frac{}{\{\lambda\_.\ \bot\}\mathsf{Fail}\{\lambda\_, \_.\ \bot\}}$$

That is, failure can only be verified against an unsatisfiable precondition, so that we know that the failure is unreachable.

With this extension, we can prove a soundness-theorem variant, capturing the impossibility of failure.

Theorem 2.3. *If $\{P\}c\{Q\}$ and $P(h)$ for some heap $h$, then it is an invariant of $(c, h)$ that the state never becomes* Failed.

```
let interp c =
  let h : (nat, nat) Hashtbl.t = Hashtbl.create 0 in

  let rec interp' (c : 'a cmd) : 'a =
    match c with
    | Return v -> v
    | Bind (c1, c2) -> interp' (c2 (interp' c1))
    | Read a -> Obj.magic (try Hashtbl.find h a with Not_found -> 0)
    | Write (a, v) -> Obj.magic (Hashtbl.replace h a v)
    | Loop (i, b) ->
      begin match Obj.magic (interp' (Obj.magic (b i))) with
      | Done r -> r
      | Again r -> interp' (Loop (r, b))
      end
    | Fail -> failwith "Fail"

  in h, interp' c
```

Fig. 1. An OCaml interpreter for our latest object language

Note that this version of the theorem still tells us interesting things about programs that run forever. It is easy to implement runtime assertion checking with code that performs some test and runs Fail if the test does not pass. An infinite-looping program may perform such tests infinitely often, and we learn that none of the tests ever fail.

Figure 1 demonstrates another advantage of this mixed-embedding style: we can extract our programs to OCaml and run them efficiently, via a simple interpreter (which must regrettably include some unchecked `Obj.magic` casts, to adjust for the type-system-expressiveness gap between Coq and OCaml[2]). That is, rather than using functional programming to implement our three kinds of side effects, we implement them directly with OCaml's mutable heap, unbounded recursion, and exceptions, respectively. As a result, our extracted programs achieve the asymptotic performance that we would expect, thinking of them as C-like code, where interpreters in a pure functional language like Gallina would necessarily add at least an extra logarithmic factor in the modeling of unboundedly growing heaps.

## 3 SEPARATION LOGIC

Separation logic [Reynolds 2002] extends Hoare logic with clever support for reasoning about pointer aliasing and lack thereof, though its presentation has traditionally been complicated by variable side conditions.

Before proceeding, let us take stock of some of the traditional bureaucratic challenges of Hoare logics in general, beginning with issues already avoided in the prior section. We write the general category descriptions in boldface to draw attention to them when skimming the paper. Usually these logics **distinguish between program variables and logical variables**, so for instance a variable literally assigned in a program is quite different from a variable bound by a quantifier in a

[2]Technically, the example here *could* be encoded using GADTs in OCaml or Haskell (which Coq extraction does not take advantage of today), though, in general, Coq programs may use true dependent types and thus face lack of viable translations in those other languages.

specification. After a side-effecting command, we must update the precondition to take the side effect into account, e.g. we may need to rewrite uses of *program variables*, while *logical variables* are timeless, and their uses need not be rewritten. We have seen how mixed embeddings allow use of the metalanguage's normal variable namespace for all of the above.

Standard operational semantics depend on substituting values for variables, a syntactic operation, which often necessitates **proof of administrative lemmas about substitution**. As in HOAS, we have avoided explicit substitution so far and will continue to do so through our remaining examples.

Typical Hoare logics require **explicit proof rules (let alone explicit object-language grammar productions) for all programming constructs**, from different primitive types and operators to loops and recursion. Mixed embeddings allow most of these (essentially those that can be written as functional programs in the logics of proof assistants, which enforce termination) to be inherited from the metalanguage.

Another classic inconvenience of Hoare logics is the need for **variable-related side conditions of proof rules**. We opened the paper with a brief example, which we expand on below.

### 3.1 An Object Language with Dynamic Memory Allocation

Before we get into proofs, let's fix a mixed-embedding object language.

$$\begin{array}{rrcl} \text{Commands} & c & ::= & \mathsf{Return}\ v \mid x \leftarrow c; c \mid \mathsf{Loop}\ i\ f \mid \mathsf{Fail} \\ & & & \mid \mathsf{Read}\ n \mid \mathsf{Write}\ n\ n \mid \mathsf{Alloc}\ n \mid \mathsf{Free}\ n\ n \end{array}$$

A small-step operational semantics explains what these commands mean.

$$\frac{(h, c_1) \longrightarrow (h', c_1')}{(h, x \leftarrow c_1; c_2(x)) \longrightarrow (h', x \leftarrow c_1'; c_2(x))} \quad \frac{}{(h, x \leftarrow \mathsf{Return}\ v; c(x)) \longrightarrow (h, c(v))}$$

$$\frac{}{(h, \mathsf{Loop}\ i\ f) \longrightarrow (h, x \leftarrow f(i); \mathsf{match}\ x\ \mathsf{with}\ \mathsf{Done}(a) \Rightarrow \mathsf{Return}\ a \mid \mathsf{Again}(a) \Rightarrow \mathsf{Loop}\ a\ f)}$$

$$\frac{h(a) = v}{(h, \mathsf{Read}\ a) \longrightarrow (h, \mathsf{Return}\ v)} \quad \frac{h(a) = v}{(h, \mathsf{Write}\ a\ v') \longrightarrow (h[a \mapsto v'], \mathsf{Return}\ ())}$$

$$\frac{\mathrm{dom}(h) \cap [a, a+n) = \emptyset}{(h, \mathsf{Alloc}\ n) \longrightarrow (h[a \mapsto 0^n], \mathsf{Return}\ a)} \quad \frac{}{(h, \mathsf{Free}\ a\ n) \longrightarrow (h - [a, a+n), \mathsf{Return}\ ())}$$

Note that the loop rule doesn't manage to keep the different language constructs separate from each other, instead using Bind to explain the meaning of Loop. The reason is that we need some way to focus evaluation attention on one loop iteration $f(i)$, but there is not a natural place to stage that evaluation within the original term, e.g. we can't take a step that replaces $f$ itself because we will probably need to reuse $f$ for later iterations. We could invent a new intermediate syntax for running loops, but it is easier to use Bind as already defined. Note that we take advantage of the metalanguage's pattern matching to give a (relatively) concise characterization of one iteration worth of loop execution.

A few remarks about the last four rules: The basic Read and Write operations now get *stuck* when accessing unmapped addresses, with heap $h$ as a partial function. The premise of the rule for Alloc enforces that address $a$ denotes a currently unmapped memory region of size $n$. We write $0^n$ for a sequence of $n$ zeroes to write into memory. Similarly, the conclusion of the Free rule unmaps a whole size-$n$ region, starting at $a$. We could also have chosen to enforce in this rule that the region starts out as mapped into $h$.

### 3.2 Assertion Logic

Separation logic is based on two big ideas. The first one has to do with the *assertion logic*, which we use to write invariants; while the second one has to do with the *program logic*, which we use to prove that programs satisfy specifications. The assertion logic is based on predicates over *partial memories*, or finite maps from addresses to stored values. Because they are finite, they omit infinitely many addresses, and it is crucial that we are able to describe heaps that intentionally leave addresses out of their domains. Informally, a predicate *claims ownership* of addresses in the domains of matching heaps.

We can describe the connectives of separation logic in terms of the sets of partial heaps that they accept, where $\bullet$ is the empty heap and $h[p \mapsto v]$ denotes the updating of address $p$ to value $v$ in $h$.

$$\begin{array}{rcl} \mathsf{emp} & = & \{\bullet\} \\ p \mapsto v & = & \{\bullet[p \mapsto v]\} \\ [\phi] & = & \{h \mid \phi \wedge h = \bullet\} \\ \exists x.\, P(x) & = & \{h \mid \exists x.\, h \in P(x)\} \\ P * Q & = & \{h_1 \uplus h_2 \mid h_1 \in P \wedge h_2 \in Q\} \end{array}$$

The formula emp accepts only the empty heap, while formula $p \mapsto v$ accepts only the heap whose only address is $p$, mapped to value $v$. We overload the $\mapsto$ operator in that second line above, to denote "points-to" on the lefthand side of the equality and finite-map overriding on the righthand side. Notation $[\phi]$ is *lifting* a *pure* (i.e., regular old mathematical) proposition $\phi$ into an assertion, enforcing both that the heap is empty and that $\phi$ is true. We also adapt the normal existential quantifier to this setting.

The essential definition is the last one, of the *separating conjunction* $*$. We use the notation $h_1 \uplus h_2$ for *disjoint union* of heaps $h_1$ and $h_2$, implicitly enforcing $\mathsf{dom}(h_1) \cap \mathsf{dom}(h_2) = \emptyset$. The intuition of separating conjunction is that we *partition* the overall heap into two subheaps, each of which matches one of the respective conjuncts $P$ and $Q$. This connective implicitly enforces *lack of aliasing*, leading to separation logic's famous conciseness of specifications that combine data structures.

We can also define natural comparison operators between assertions, overloading the usual notations for equivalence and implication of propositions.

$$\begin{array}{rcl} P \Leftrightarrow Q & = & \forall h.\, h \in P \Leftrightarrow h \in Q \\ P \Rightarrow Q & = & \forall h.\, h \in P \Rightarrow h \in Q \end{array}$$

The core connectives satisfy a number of handy algebraic laws and admit handy reasoning with cancellation of terms on the two sides of an implication, but we will not go into detail here, which would take us astray from our main message about lack of variable side conditions.

### 3.3 Program Logic

We use an automatic cancellation procedure to discharge some of the premises from the rules of the program logic, which we present now. First, here are the rules that are (almost) exactly the same as from last section.

$$\frac{}{\{P\}\mathsf{Return}\ v\{\lambda r.\, P * [r = v]\}} \qquad \frac{\{P\}c_1\{Q\} \quad (\forall r.\, \{Q(r)\}c_2(r)\{R\})}{\{P\}x \leftarrow c_1; c_2(x)\{R\}}$$

$$\frac{\forall a.\, \{I(\mathsf{Again}(a))\}f(a)\{I\}}{\{I(\mathsf{Again}(i))\}\mathsf{Loop}\ i\ f\{\lambda r.\, I(\mathsf{Done}(r))\}} \qquad \frac{}{\{[\bot]\}\mathsf{Fail}\{\lambda\_.\, [\bot]\}}$$

$$\frac{\{P\}c\{Q\} \quad P' \Rightarrow P \quad \forall r.\ Q(r) \Rightarrow Q'(r)}{\{P'\}c\{Q'\}}$$

Note a subtle victory in the Bind rule: we do not need to worry about whether assigned variable $x$ appears free in precondition or postcondition! Rather, **logical and program variables** are kept completely distinct, as an automatic consequence of the mixed-embedding style.

More interesting are the rules for primitive memory operations. First, we have the rule for Read.

$$\overline{\{\exists v.\ a \mapsto v * R(v)\}\mathsf{Read}\ a\{\lambda r.\ a \mapsto r * R(r)\}}$$

In words: before reading from address $a$, it must be the case that $a$ points to some value $v$, and predicate $R(v)$ records what else we know about the memory at that point. Afterward, we know that $a$ points to the result $r$ of the read operation, and $R$ is still present. We call $R$ a *frame predicate*, recording what we know about parts of memory that the command does not touch directly. We might also say that the *footprint* of this command is the singleton set $\{a\}$. In general, frame predicates record preserved facts about addresses outside a command's footprint. The next few rules don't have frame predicates baked in; we finish with a rule that adds them back, in a generic way for arbitrary Hoare triples.

$$\overline{\{\exists v.\ a \mapsto v\}\mathsf{Write}\ a\ v'\{\lambda\_.\ a \mapsto v'\}}$$

This last rule, for Write, is even simpler. We see a straightforward illustration of overwriting $a$'s old value $v$ with the new value $v$'.

$$\overline{\{\mathsf{emp}\}\mathsf{Alloc}\ n\{\lambda r.\ r \mapsto 0^n\}} \quad \overline{\{a \mapsto ?^n\}\mathsf{Free}\ a\ n\{\lambda\_.\ \mathsf{emp}\}}$$

The rules for allocation and deallocation deploy a few more notations: $0^n$ for sequences of $n$ zeroes and $?^n$ for sequences of $n$ arbitrary values.

The next rule, the *frame rule*, gives the second key idea of separation logic, supporting the *small-footprint* reasoning style.

$$\frac{\{P\}c\{Q\}}{\{P * R\}c\{\lambda r.\ Q(r) * R\}}$$

In other words, any Hoare triple can be extended by conjoining an arbitrary predicate $R$ in both precondition and postcondition. Even more intuitively, when a program satisfies a spec, it also satisfies an extended spec that records the state of some other part of memory that is untouched (i.e., is outside the command's footprint).

Note the conspicuous absence of **side conditions about variables**. As we build the object language's variables atop the metalanguage's variables, all appropriate bookkeeping is handled silently for us.

The accompanying Coq code shows examples of defining classic data-representation predicates and proving classic programs correct (mostly automatically). However, for modular proofs, the frame rule has such an important role that we want to emphasize it here. It is possible to define (recursively) a predicate $\mathsf{llist}(\ell, p)$, capturing the idea that the heap contains exactly an imperative linked list, rooted at pointer $p$, representing functional linked list $\ell$. We can also prove a general specification for a list-reversal function:

$$\forall \ell, p.\ \{\mathsf{llist}(\ell, p)\}\mathtt{reverse}(p)\{\lambda r.\ \mathsf{llist}(\mathsf{rev}(\ell), r)\}$$

Now consider that we have the roots $p_1$ and $p_2$ of two disjoint lists, respectively representing $\ell_1$ and $\ell_2$. It is easy to instantiate the general theorem and get $\{\mathsf{llist}(\ell_1, p_1)\}\mathtt{reverse}(p_1)\{\lambda r.\ \mathsf{llist}(\mathsf{rev}(\ell_1), r)\}$

and $\{\mathsf{llist}(\ell_2, p_2)\}\texttt{reverse}(p_2)\{\lambda r.\ \mathsf{llist}(\mathsf{rev}(\ell_2), r)\}$. Applying the frame rule to the former theorem, with $R = \mathsf{llist}(\ell_2, p_2)$, we get:

$$\{\mathsf{llist}(\ell_1, p_1) * \mathsf{llist}(\ell_2, p_2)\}\texttt{reverse}(p_1)\{\lambda r.\ \mathsf{llist}(\mathsf{rev}(\ell_1), r) * \mathsf{llist}(\ell_2, p_2)\}$$

Similarly, applying the frame rule to the latter, with $R = \mathsf{llist}(\mathsf{rev}(\ell_1), r)$, we get:

$$\{\mathsf{llist}(\ell_2, p_2) * \mathsf{llist}(\mathsf{rev}(\ell_1), r)\}\texttt{reverse}(p_2)\{\lambda r'.\ \mathsf{llist}(\mathsf{rev}(\ell_2), r') * \mathsf{llist}(\mathsf{rev}(\ell_1), r)\}$$

Now it is routine to derive the following spec for a larger program:

$$\begin{array}{l}\{\mathsf{llist}(\ell_1, p_1) * \mathsf{llist}(\ell_2, p_2)\}\\ \quad r \leftarrow \texttt{reverse}(p_1); r' \leftarrow \texttt{reverse}(p_2);\ \mathsf{Return}(r, r')\\ \{\lambda(r, r').\ \mathsf{llist}(\mathsf{rev}(\ell_1), r) * \mathsf{llist}(\mathsf{rev}(\ell_2), r')\}\end{array}$$

Note how there was no fuss with **making sure that program variable $r$ doesn't appear anywhere in the Hoare triple for `reverse`$(p_2)$**, or vice versa for $r'$ and $\texttt{reverse}(p_1)$. It is quite simply syntactically impossible to introduce the kind of confounding dependency that motivated the original frame rule's side condition.

### 3.4 Soundness Proof

Our Hoare logic is sound with respect to the object language's operational semantics.

Theorem 3.1. *If $\{P\}c\{Q\}$ and $P(\bullet)$ (i.e., $P$ holds on an empty heap), then it is an invariant of the transition system starting from $(\bullet, c)$ that either the command has become a* Return *or another execution step is possible.*

As usual, the key to the proof is to find a stronger invariant that can be proved by invariant induction. In this case, we use the invariant $\lambda(h, c).\ \{\{h\}\}c\{Q\}$. That is, assert a Hoare triple where the precondition enforces exact heap equality with the current heap. The postcondition can remain the same throughout execution.

A few key lemmas are interesting enough to mention here; we leave other details to the Coq code.

First, we need to prove that this fancier invariant implies the one from the theorem statement, and the most direct statement needs to be strengthened, to get the induction to go through.

Lemma 3.2 (Progress). *If $\{P\}c\{Q\}$ and $P(h_1)$, then either $c$ is a* Return *or it is possible to take a step from $(h_1 \uplus h_2, c)$, for any disjoint $h_2$.*

Proof. By induction on the derivation of $\{P\}c\{Q\}$. □

Note the essential inclusion of a disjoint union with the auxiliary heap $h_2$. Without this strengthening of the obvious property, we would get stuck in the case of the proof for the frame rule. Also note how important it is that we retain the ability to do induction on derivations, despite the use of a higher-order encoding within a general-purpose logic.

Lemma 3.3 (Preservation). *If $(h, c) \longrightarrow (h', c')$ and $\{\{h\}\}c\{Q\}$, then $\{\{h'\}\}c'\{Q\}$.*

Proof. By induction on the derivation of $(h, c) \longrightarrow (h', c')$. □

The different cases of the proof depend on some not-entirely-obvious inversion lemmas. For instance, here is the one we prove for Write.

Lemma 3.4. *If $\{P\}$Write $a\ v'\{Q\}$, then there exists $R$ such that:*

- $P \Rightarrow \exists v.\ a \mapsto v * R$
- $a \mapsto v' * R \Rightarrow Q(())$

Proof. By induction on the derivation of $\{P\}\text{Write } a\ v'\{Q\}$. □

Again, without the introduction of the $R$ variable, we would get stuck proving the case for the frame rule.

This lemma involves enough hassle with dependent typing that it is worth discussing the Coq code briefly.

```
Lemma invert_Write : forall a v' P Q,
  hoare_triple P (Write a v') Q
  -> exists R, (P ===> (exists v, a |-> v) * R)%sep
               /\ a |-> v' * R ===> Q tt.
```

Some cases of our induction on `hoare_triple` derivations are hard to prove contradictory. For instance, consider the case for `Read`. It is not trivial to show that a `Write` can't possibly be a `Read`. With the automatically generated induction principle, we find such a case remaining, with hypotheses `nat = unit` and `JMeq.JMeq (Read a0) (Write a v)`. The first one equates the result types of the two operations, while the second one equates the operations themselves, using *heterogeneous equality*. Luckily the first hypothesis is contradictory by itself, as the two types have different cardinalities. The root cause of the problem is a heuristic for choosing proper induction hypotheses (mostly delegated to Coq's `dependent induction` tactic) in the presence of richly typed syntax trees, and the problem could be sidestepped with a smarter choice of induction hypothesis, at the expense of more complex lemma statements. Admittedly, this issue is the biggest wart we have encountered with this mixed-embedding style.

## 4 PARTIAL-ORDER REDUCTION

After that warmup with sequential programs, our first of three examples in concurrency reasoning is *partial-order reduction* [Godefroid et al. 1996], a technique typically applied to shrink state spaces for model checking. The Coq code includes a simple implementation of model checking in Coq's Ltac tactic language, for arbitrary transition systems (though the process is doomed to fail when used on infinite transition systems).

One usual complication of this kind of program analysis is **distinguishing between global and local variables**. For instance, it becomes important to note that any program operation touching only thread-local variables *commutes* with any operation of another thread. We will continue to avoid needing to formalize local variables or local state very explicitly.

Partial-order reduction also traditionally depends on **static analysis of code with variable manipulation and complex control flow**, to approximate how a group of threads might manipulate shared state. For instance, we might develop an abstract interpretation that assigns abstract descriptions to variables and updates them during an iterative process, and we might need to formulate rules for how to flow abstract descriptions through split and join points in control flow. Instead, our version here will work only with concrete variable values and reuse the metalanguage's variable contexts to keep track of them, and the core formulation and soundness proof will include no explicit consideration of complex control flow. (We will not get a completely free lunch, as model-checking of individual programs still requires explicit proof reasoning about control flow.)

### 4.1 An Object Language with Shared-Memory Concurrency

We work with this object language.

$$\text{Commands}\quad c\quad ::=\quad \text{Fail} \mid \text{Return } v \mid x \leftarrow c; c \mid \text{Read } a \mid \text{Write } a\ v \mid \text{Lock } a \mid \text{Unlock } a \mid c||c$$

In addition to the basic structure of the languages from the last two sections, we have three features specific to concurrency. We follow the common "threads and locks" style of synchronization, with commands Lock $a$ and Unlock $a$ for acquiring and releasing locks, respectively. We also have $c_1||c_2$ for running commands $c_1$ and $c_2$ in parallel, giving a scheduler free reign to interleave their atomic steps.

The Coq version adopts a few simplifications. Motivated by the wart mentioned at the end of section 3, we force all commands to yield result type `nat`. We also switch to modeling the heap as total, i.e. all addresses are considered mapped into memory.

The operational semantics is small-step, especially because big-step semantics are notoriously awkward for concurrency. Each state of the system is a triple $(h, l, c)$, with $h$ and $c$ the heap and current command from our usual semantics. New component $l$ is a *lockset*, recording which locks are currently held, without distinguishing between different threads that might have taken them.

$$\frac{(h, l, c_1) \to (h', l', c_1')}{(h, l, x \leftarrow c_1; c_2(x)) \to (h', l', x \leftarrow c_1'; c_2(x))} \quad \frac{}{(h, l, x \leftarrow \mathsf{Return}\ v; c_2(x)) \to (h, l, c_2(v))}$$

$$\frac{}{(h, l, \mathsf{Read}\ a) \to (h, l, \mathsf{Return}\ h(a))} \quad \frac{}{(h, l, \mathsf{Write}\ a\ v) \to (h[a \mapsto v], l, \mathsf{Return}\ 0)}$$

$$\frac{a \notin l}{(h, l, \mathsf{Lock}\ a) \to (h, l \cup \{a\}, \mathsf{Return}\ 0)} \quad \frac{a \in l}{(h, l, \mathsf{Unlock}\ a) \to (h, l \setminus \{a\}, \mathsf{Return}\ 0)}$$

$$\frac{(h, l, c_1) \to (h', l', c_1')}{(h, l, c_1||c_2) \to (h', l', c_1'||c_2)} \quad \frac{(h, l, c_2) \to (h', l', c_2')}{(h, l, c_1||c_2) \to (h', l', c_1||c_2')}$$

Note that the last two rules are the only source of *nondeterminism* in this semantics, where a single state can step to multiple different next states. This nondeterminism corresponds to the freedom we give to a scheduler that may pick which thread runs next. Though this kind of concurrent programming is very expressive and often achieves very high performance, it comes at a cost in reasoning, as there may be *exponentially many different schedules* for a single program, measured with respect to the textual length of the program. A popular name for this pitfall is *the state-explosion problem*.

Note also that we have omitted any looping constructs from this object language, so all programs terminate. The Coq formalization uses the mixed-embedding style, making it not entirely obvious that all programs really do terminate. In any case, if we must tame the state-explosion problem, we already have our work cut out for us, even when the state space rooted at any concrete state is finite!

### 4.2 Shrinking the State Space via Local Actions

We are interested in using model checking to explore reachable states of a transition system explicitly, checking that a decidable invariant predicate holds for each one. For our object language in this section, a good invariant is that commands are *not about to fail*, formalized as:

$$\begin{aligned} \mathsf{natf}(\mathsf{Fail}) &= \bot \\ \mathsf{natf}(x \leftarrow c_1; c_2(x)) &= \mathsf{natf}(c_1) \\ \mathsf{natf}(c_1||c_2) &= \mathsf{natf}(c_1) \wedge \mathsf{natf}(c_2) \\ \mathsf{natf}(\_) &= \top \end{aligned}$$

Here is an example of a program execution that avoids failures.

$$\begin{array}{rcl}(\bullet[0 \mapsto 1], \emptyset, n \leftarrow \mathsf{Read}\ 0; \mathsf{Write}\ 0\ (n+1)) & \rightarrow & (\bullet[0 \mapsto 1], \emptyset, n \leftarrow \mathsf{Return}\ 1; \mathsf{Write}\ 0\ (n+1)) \\ & \rightarrow & (\bullet[0 \mapsto 1], \emptyset, \mathsf{Write}\ 0\ (1+1)) \\ & \rightarrow & (\bullet[0 \mapsto 2], \emptyset, \mathsf{Return}\ 0)\end{array}$$

When exploring the state space of this program, a näive model checker will generate each of these states explicitly, even the "silly" second one that reduces to the third without reading or writing the shared state. We can short-circuit those extra states by writing a simple function that makes all appropriate purely local reductions, everywhere within a command.

$$\begin{array}{rcl}\lfloor x \leftarrow c_1; c_2(x) \rfloor & = & \lfloor c_2(v) \rfloor, \text{ when } \lfloor c_1 \rfloor = \mathsf{Return}\ v \\ \lfloor x \leftarrow c_1; c_2(x) \rfloor & = & x \leftarrow \lfloor c_1 \rfloor; \lfloor c_2(x) \rfloor, \text{ when } \lfloor c_1 \rfloor \text{ is not Return} \\ \lfloor c_1 || c_2 \rfloor & = & \lfloor c_1 \rfloor || \lfloor c_2 \rfloor \\ \lfloor c \rfloor & = & c\end{array}$$

Using this relation, we can define an alternative step relation that short-circuits local steps.

$$\frac{(h, l, c) \rightarrow (h', l', c')}{(h, l, c) \rightarrow_L (h', l', \lfloor c' \rfloor)}$$

The base semantics can be used to define transition systems in the usual way, with $\mathbb{T}(h, l, c) = \langle \{(h, l, c)\}, \rightarrow \rangle$ (giving the transition system's initial states and transition relation). We can also define short-circuiting transition systems with $\mathbb{T}_L(h, l, c) = \langle \{(h, l, \lfloor c \rfloor)\}, \rightarrow_L \rangle$. A theorem shows that the latter overapproximates the former.

Theorem 4.1. *If* natf *is an invariant of* $\mathbb{T}_L(h, l, c)$*, then it is also an invariant of* $\mathbb{T}(h, l, c)$*.*

Proof. By induction on a trace $(h, l, c) \rightarrow^* (h', l', c')$, matching each original step with zero or one alternative steps. □

### 4.3 Partial-Order Reduction

What made the reduction in Theorem 4.1 sound? It was that local actions *commute* with all actions in other threads. A particular run of a system in the base semantics might indeed choose to run a nonlocal action before a local action that is enabled. However, we can *reorder* any such action to instead come after every enabled local action, without affecting the final state. This reordering is an example of commutativity in action.

By recognizing and exploiting other varieties of commutativity, we can shrink state spaces even further, even reducing the spaces of certain interesting program families from exponential size to linear size. A popular technique of this kind is *partial-order reduction* [Godefroid et al. 1996].

To check commutativity more flexibly, we must use more than just the fact that a local action commutes with any action in another thread. For instance, we should take advantage of the fact that any two Read actions commute. We will do some *static analysis* of programs to overapproximate which kinds of atomic actions they might perform. Such an analysis is designed to be trivially computable. Here's an example of one analysis, formulated as a relation $\mathsf{summarize}(c, (r, w, \ell))$, which asserts that the only globally visible actions that could be performed by thread $c$ are reads to addresses in $r$, writes to addresses in $w$, and acquires or releases of locks in $\ell$.

$$\frac{}{\mathsf{summarize}(\mathsf{Return}\ r, s)} \qquad \frac{}{\mathsf{summarize}(\mathsf{Fail}, s)} \qquad \frac{\mathsf{summarize}(c_1, s) \quad \forall r.\ \mathsf{summarize}(c_2(r), s)}{\mathsf{summarize}(x \leftarrow c_1; c_2(x), s)}$$

$$\frac{a \in r}{\mathsf{summarize}(\mathsf{Read}\ a, (r, w, \ell))} \qquad \frac{a \in w}{\mathsf{summarize}(\mathsf{Write}\ a\ v, (r, w, \ell))}$$

$$\frac{a \in \ell}{\mathsf{summarize}(\mathsf{Lock}\ a, (r, w, \ell))} \qquad \frac{a \in \ell}{\mathsf{summarize}(\mathsf{Unlock}\ a, (r, w, \ell))}$$

$$\frac{\mathsf{summarize}(c_1, s) \quad \mathsf{summarize}(c_2, s)}{\mathsf{summarize}(c_1 || c_2, s)}$$

All that was fairly easy to define as an inductive judgment, but note that we have defined a logic program for **a simple abstract interpretation, without having to deal explicitly with program variables or complex control flow**. Rather, the second premise of the rule for Bind just quantifies over all possible results for the first subcommand $c_1$. Different results may lead to quite different control flow in the function encoding the second subcommand $c_2$, and we need some other heuristics to execute this logic program effectively. For instance, when we find an if at the head of $c_2$'s body, we should do case analysis on the if's test expression. It is convenient, though, that we will be able to prove soundness without confronting those details directly.

Those relations do all we need to record which actions a thread might not commute with. The other key ingredient is an extractor for the next atomic action in a thread, written as a partial function.

$$\begin{aligned}
\mathsf{nextAction}(\mathsf{Return}\ r) &= \mathsf{Return}\ r \\
\mathsf{nextAction}(\mathsf{Fail}) &= \mathsf{Fail} \\
\mathsf{nextAction}(\mathsf{Read}\ a) &= \mathsf{Read}\ a \\
\mathsf{nextAction}(\mathsf{Write}\ a\ v) &= \mathsf{Write}\ a\ v \\
\mathsf{nextAction}(\mathsf{Lock}\ a) &= \mathsf{Lock}\ a \\
\mathsf{nextAction}(\mathsf{Unlock}\ a) &= \mathsf{Unlock}\ a \\
\mathsf{nextAction}(x \leftarrow c_1; c_2(x)) &= \mathsf{nextAction}(c_1)
\end{aligned}$$

Given a next atomic action and a summary of another thread, it is now easy to define commutativity of the two.

$$\begin{aligned}
\mathsf{commutes}(\mathsf{Return}\ _, _) &= \top \\
\mathsf{commutes}(\mathsf{Fail}, _) &= \top \\
\mathsf{commutes}(\mathsf{Read}\ a, (_, w, _)) &= a \notin w \\
\mathsf{commutes}(\mathsf{Write}\ a\ _, (r, w, _)) &= a \notin r \cup w \\
\mathsf{commutes}(\mathsf{Lock}\ a, (_, _, \ell)) &= a \notin \ell \\
\mathsf{commutes}(\mathsf{Unlock}\ a, (_, _, \ell)) &= a \notin \ell \\
\mathsf{commutes}(_, _) &= \bot
\end{aligned}$$

With these ingredients, we can define a predicate porSafe that figures out when a state is eligible for the partial-order-reduction optimization, which is to force the first thread to run next, ignoring the other threads for now. In working out the formal details, we will confine ourselves to commands $c_1 || c_2$ with distinguished "first threads" $c_1$, though everything can be generalized to other settings (and doing that generalization could be a worthwhile exercise for the reader, though it requires a lot of logical bookkeeping). This optimization is only safe when the first thread can take a step and when that step commutes with any action that other threads (combined into $c_2$) might perform. Formally, we define $\mathsf{porSafe}(h, l, c_1, c_2, s)$ as follows, where $s$ should be a valid summary of $c_2$.

- There is some $c_0$ where $\mathsf{nextAction}(c_1) = c_0$. That is, thread $c_1$ has some uniquely determined atomic action lined up to run next.
- There exist $h'$, $l'$, and $c_1'$ such that $(h, l, c_1) \rightarrow (h', l', c_1')$. That is, thread $c_1$ is actually able to take a step, which might not be possible if e.g. trying to take a lock that is already held.
- And the crucial compatibility condition: $\mathsf{commutes}(c_0, s)$. That is, all actions that other threads might perform commute with $c_0$, the first action of $c_1$.

With the applicability condition defined, it is now straightforward to define an optimized step relation, parameterized on an accurate summary $s$ for $c_2$.

$$\frac{(h, l, c_1) \rightarrow (h', l', c_1')}{(h, l, c_1 || c_2) \rightarrow_C^s (h', l', c_1' || c_2)} \quad \frac{\neg \mathsf{porSafe}(h, l, c_1, c_2, s) \quad (h, l, c_2) \rightarrow (h', l', c_2')}{(h, l, c_1 || c_2) \rightarrow_C^s (h', l', c_1 || c_2')}$$

The whole thing is wrapped up into transition systems as $\mathbb{T}_C(h, l, c_1, c_2, s) = \left\langle \{(h, l, c_1 || c_2)\}, \rightarrow_C^s \right\rangle$.

Our proof of soundness for this reduction will depend on having some constant upper bound on program execution time. This relation computes a conservative overapproximation (again showing off convenient abstract interpretation via logic programming).

$$\frac{}{\mathsf{timeOf}(\mathsf{Return}\ r, n)} \quad \frac{}{\mathsf{timeOf}(\mathsf{Fail}, n)} \quad \frac{}{\mathsf{timeOf}(\mathsf{Read}\ a, n+1)} \quad \frac{}{\mathsf{timeOf}(\mathsf{Write}\ a\ v, n+1)}$$

$$\frac{}{\mathsf{timeOf}(\mathsf{Lock}\ a, n+1)} \quad \frac{}{\mathsf{timeOf}(\mathsf{Unlock}\ a, n+1)}$$

$$\frac{\mathsf{timeOf}(c_1, n_1) \quad \forall r.\ \mathsf{timeOf}(c_2(r), n_2)}{\mathsf{timeOf}(x \leftarrow c_1; c_2(x), n_1 + n_2 + 1)} \quad \frac{\mathsf{timeOf}(c_1, n_1) \quad \mathsf{timeOf}(c_2, n_2)}{\mathsf{timeOf}(c_1 || c_2, n_1 + n_2 + 1)}$$

It may be surprising that, in our formal mixed embedding, there exist commands with no provable upper bounds, according to this relation. We leave it as an exercise to the reader to find a concrete example. (Actually, the Coq code includes an example and its proof of unboundedness.)

With these ingredients, we can state the reduction theorem.

Theorem 4.2. *If* $\mathsf{summarize}(c_2, s)$ *and* $\mathsf{timeOf}(c_1 || c_2, n)$*, then to prove* $\mathsf{natf}$ *as an invariant of* $\mathbb{T}(h, l, c_1 || c_2)$*, it suffices to prove* $\mathsf{natf}$ *as an invariant of* $\mathbb{T}_C(h, l, c_1, c_2, s)$*.*

Proof. There is quite a lot of fiddly detail in the proof, but here is the essence. Assume for the sake of contradiction that there exists some base-semantics execution to a state that is about to fail. Since $c$ runs in bounded time, we can *complete* that execution to continue running to some stuck state, which is also about to fail. By induction on the length of the execution, we repeatedly commute "out-of-order" actions that go against the partial-order-reduction rules, until we arrive at a fully compliant execution. Within this proof, the fact that the execution has run to stuckness is important to guarantee that, if the partial-order-reduction optimization is enabled in some state, we always find some future point when the first thread is selected to run. Our corrected execution must itself be about to fail, contradicting our assumption about an invariant of the system with partial-order reduction. □

And there ends the proof, notable, in comparison to the literature, for what it doesn't include. There was no formalization of **local variables** and how assignments to them commute with everything. The reason is that we bootstrap off of Coq's local variables and thus avoid any explicit modeling of local-variable environments. We also managed to define and reason about two simple static analyses, *without* any treatment of branching control flow, because we inherit all control flow from the metalanguage. Nonetheless, example programs may include quite expressive (terminating) control flow. Besides, all the above are rather orthogonal to the main point of partial-order reduction, as this proof without reference to them reveals.

## 5 CONCURRENT SEPARATION LOGIC

Sections 3 and 4 respectively introduced techniques for reasoning about two tricky aspects of programs: heap-allocated linked data structures and shared-memory concurrency. When we add concurrency to the mix for a program-reasoning problem, we are often surprised at how much more complex it becomes. This section introduces a pleasant exception to the rule, *concurrent separation logic* [O'Hearn 2007], a rather small addition to separation logic that supports invariant-based reasoning about threads-and-locks shared-memory programs.

This example brings together the mixed-embedding advantages from the prior two sections, so we need not introduce any new flavors of avoided bureaucracy.

### 5.1 Object Language: Loops and Locks

Here's the object language we adopt, which should be old hat by now, just mixing together features of the object languages from Sections 3 and 4.

$$\begin{array}{lll} \text{Commands} & c & ::= \ \mathsf{Fail} \mid \mathsf{Return}\ v \mid x \leftarrow c; c \mid \mathsf{Loop}\ i\ f \\ & & \quad \mid \mathsf{Read}\ a \mid \mathsf{Write}\ a\ v \mid \mathsf{Alloc}\ n \mid \mathsf{Free}\ a\ n \mid \mathsf{Lock}\ a \mid \mathsf{Unlock}\ a \mid c||c \end{array}$$

$$\frac{(h, l, c_1) \rightarrow (h', l', c_1')}{(h, l, x \leftarrow c_1; c_2(x)) \rightarrow (h', l', x \leftarrow c_1'; c_2(x))} \quad \frac{}{(h, l, x \leftarrow \mathsf{Return}\ v; c_2(x)) \rightarrow (h, l, c_2(v))}$$

$$\frac{}{(h, l, \mathsf{Loop}\ i\ f) \rightarrow (h, l, x \leftarrow f(i); \mathsf{match}\ x\ \mathsf{with}\ \mathsf{Done}(a) \Rightarrow \mathsf{Return}\ a \mid \mathsf{Again}(a) \Rightarrow \mathsf{Loop}\ a\ f)}$$

$$\frac{h(a) = v}{(h, l, \mathsf{Read}\ a) \rightarrow (h, l, \mathsf{Return}\ v)} \quad \frac{h(a) = v}{(h, l, \mathsf{Write}\ a\ v') \rightarrow (h[a \mapsto v'], l, \mathsf{Return}\ ())}$$

$$\frac{\mathrm{dom}(h) \cap [a, a + n) = \emptyset}{(h, l, \mathsf{Alloc}\ n) \rightarrow (h[a \mapsto 0^n], l, \mathsf{Return}\ a)} \quad \frac{}{(h, l, \mathsf{Free}\ a\ n) \rightarrow (h - [a, a + n), l, \mathsf{Return}\ ())}$$

$$\frac{a \notin l}{(h, l, \mathsf{Lock}\ a) \rightarrow (h, l \cup \{a\}, \mathsf{Return}\ ())} \quad \frac{a \in l}{(h, l, \mathsf{Unlock}\ a) \rightarrow (h, l \setminus \{a\}, \mathsf{Return}\ ())}$$

$$\frac{(h, l, c_1) \rightarrow (h', l', c_1')}{(h, l, c_1||c_2) \rightarrow (h', l', c_1'||c_2)} \quad \frac{(h, l, c_2) \rightarrow (h', l', c_2')}{(h, l, c_1||c_2) \rightarrow (h', l', c_1||c_2')}$$

### 5.2 The Program Logic

We will build on basic separation logic, using the same kind of assertions and even adopting all of the original rules unchanged.

As for new rules: when two threads use disjoint regions of memory, it is trivial to apply this rule of concurrent separation logic to verify the threads independently.

$$\frac{\{P_1\}c_1\{Q_1\} \quad \{P_2\}c_2\{Q_2\}}{\{P_1 * P_2\}c_1||c_2\{\lambda\_.\ [\bot]]\}}$$

The separating conjunction $*$ turned out to be just the right way to express the idea of "splitting the heap into a part for the first thread and a part for the second thread." Because $c_1$ and $c_2$ touch disjoint memory regions, all of their memory operations commute, so that we need not worry about the state-explosion problem, in all the ways that the scheduler might interleave their steps.

Note that, since for simplicity our running example family of concurrent object languages includes no way for parallel compositions to terminate, it makes sense to assign a contradictory overall postcondition.

However, with realistic shared-memory programs, we don't get off as easy as the parallel-composition rule suggests. Threads *do* share memory regions, using *synchronization* to tame the state-explosion problem. Our object language includes locks as its example of synchronization, and concurrent separation logic is specialized to locks. We may keep the simplistic-seeming rule for parallel composition and implicitly enrich its power by adding a twist, in the form of some other rules.

The big twist from O'Hearn's logic is that we parameterize everything over some finite set $L$ of locks that may be used.

Furthermore, another parameter is a function $\mathcal{I}$ that maps locks to invariants, which have the same type as preconditions. The idea is this: when no one holds a lock, *the lock owns a chunk of memory that satisfies its invariant*. When a thread holds the lock, the lock doesn't own any memory; it is waiting for the thread to unlock it and *donate back* a chunk of memory satisfying the invariant. We now think of the precondition of a Hoare triple as only describing the *local memory* of a thread, which no other thread may access, while locks and their invariants coordinate the *shared memory* regions of an application. The proof rules will coordinate dynamic motion of memory regions between the shared regions and local regions. This motion is only part of a proof technique; it has no runtime content reflected in the operational semantics!

With all of that set-up, the final two rules may seem surprisingly simple.

$$\frac{a \in L}{\{\mathsf{emp}\}\mathsf{Lock}\ a\{\lambda\_.\ \mathcal{I}(a)\}} \qquad \frac{a \in L}{\{\mathcal{I}(a)\}\mathsf{Unlock}\ a\{\lambda\_.\ \mathsf{emp}\}}$$

When a thread takes a lock, it appears as if *a memory chunk satisfying that lock's invariant materializes in the local memory space*. Conversely, when a thread releases a lock, it appears as if *the lock grabs a memory chunk satisfying the invariant out of the local memory space*. The rules are coordinating conceptual ownership transfers between local memory and the global lock memory.

The accompanying Coq code shows a few example verifications of interesting programs.

### 5.3 Soundness Proof

We can adapt the separation-logic soundness proof to concurrency, with just a few new ideas. We will appreciate a derived connective for writing assertions, the "big star," *iterated separating conjunction*, with quantification over finite sets, written like $\otimes_{x \in S} P(x)$. The definition is:

$$\otimes_{x \in \{v_1, \ldots, v_n\}} P(x) \quad = \quad P(v_1) * \ldots * P(v_n)$$

The reader may be worried about the inherently unordered nature of sets. For each ordering of a set, we get a syntactically distinct formula on the righthand side of the defining equation. Luckily, separating conjunction $*$ is associative and commutative, so all orders lead to logically equivalent formulas.

With those preliminaries out of the way, we can state the soundness theorem, referring again to the *not-about-to-fail* predicate natf from last section, extended appropriately to say that loops are not about to fail.

Theorem 5.1 (Soundness). *If* $\{P\}c\{Q\}$, *and if a heap* $h$ *satisfies the predicate* $(P * \otimes_{a \in L} \mathcal{I}(a))$, *then* natf *is an invariant of the system starting at state* $(h, \emptyset, c)$.

The theorem lays out restrictions on the starting heap. It must have a segment to serve as the root thread's local heap, matching precondition $P$. Then, for each lock $a \in L$, there must be an

associated memory region satisfying $\mathcal{I}(a)$. Our use of separating conjunction forces each of these regions to occupy disjoint memory from all the others.

The progression of lemmas for this theorem is most interesting for its careful quantification over frame predicates, where reasoning about one thread "underneath" a thread "fork" operation often prompts us to extend the frame with the private memory of the other thread. As usual, the overall soundness proof proceeds by induction to prove a strengthened invariant, which uses the current lockset $l$ to limit the domain of the iterated separating conjunction.

As with partial-order reduction, we manage to wrap up this proof with no consideration of how local variables really are local, since variables are handled completely implicitly through variable binding in the metalanguage.

## 6 PROCESS ALGEBRA AND SESSION TYPES

The last two sections dealt with the most popular sort of concurrent programming, the threads-and-locks shared-memory style. It's a fundamentally imperative style, with side effects coordinating synchronization across threads. Another well-established (and increasingly popular) style is *message passing*, which is closer in spirit to functional programming. In that world, there is, in fact, no memory at all, let alone shared memory. Instead, state is incorporated into the text of thread code, and information passes from thread to thread by sending *messages* over *channels*. There are two main kinds of message passing. In the *asynchronous* or *mailbox* style, a thread can deposit a message in a channel, even when no one is ready to receive the message immediately. Later, a thread can come along and effectively dequeue the message from the channel. In the *synchronous* or *rendezvous* style, a message send only executes when a matching receive, on the same channel, is available immediately. The threads of the two complementary operations *rendezvous* and pass the message in one atomic step.

Packages of semantics and proof techniques for such languages are often called *process algebras*, as they support an algebraic style of reasoning about the source code of message-passing programs. That is, we prove laws very similar to the familiar equations of algebra, and we use those laws to "rewrite" inside larger processes, by replacing their subprocesses with others we have shown suitably equivalent. Well-known process algebras include the $\pi$-calculus [Milner 1999] and the Calculus of Communicating Systems [Milner 1982]; the one we focus on is idiosyncratic and designed partly to make the Coq proofs manageable.

We will develop type-based reasoning techniques for this style of program, and we are used to type-soundness theorems leaning on **typing contexts and crucial lemmas like weakening and substitution**. Instead, in mixed-embedding style, we define type systems rigorously but skip typing contexts and focus in on the interesting concurrency-specific details.

Taking a traditional type system and adding dependent typing tends to require **modifying the type signatures of key judgments**. However, with mixed embedding, we can add dependent typing in a rather local way. The resulting system again shows off **implicit support for complex control flow**, now within types.

### 6.1 An Object Language with Synchronous Message Passing

$$\begin{array}{lll} \text{Channels} & c & \\ \text{Processes} & p & ::= \quad \nu[\vec{c}](x); p(x) \mid \mathsf{block}(c); p \mid !c(v); p \mid ?c(x); p(x) \mid p||p \mid \mathsf{dup}(p) \mid \mathsf{done} \end{array}$$

The binding behavior of the "$\nu$" and "?" constructs is encoded in our usual mixed-embedding way, such that the body $p$ is a function of the metalanguage.

Here's the intuitive explanation of each syntax construction.

- **Fresh channel generation** $\nu[\vec{c}](x); p(x)$ creates a new *private* channel to be used by the body process $p(x)$, where we replace $x$ with the channel that is chosen. Following tradition, we use the Greek letter $\nu$ (nu) for this purpose. Each generation operation takes a parameter $\vec{c}$, which we call the *support* of the operation. It gives a list of channels already in use for other purposes, so that the fresh channel must not equal any of them. (We assume an infinite domain of channels, so that, for any specific list, it is always possible to find a channel not in that list.) The introduction of the support parameter is the key compromise to help us avoid fiddly variable-binding reasoning as in the scope-extrusion rule of $\pi$-calculus.
- **Abstraction boundaries** $\mathsf{block}(c); p$ prevent "the outside world" from sending $p$ any messages on channel $c$ or receiving any messages from $p$ via $c$. That is, $c$ is treated as a local channel for $p$. Here experts will see that these first two constructs collude to play the full role of the usual $\nu$ of $\pi$-calculus (though we accept expressiveness limitations when it comes to sending channels over channels).
- **Sends** $!c(v); p$ and **receives** $?c(x); p(x)$, where we use an exclamation mark to suggest "telling something" and a question mark to suggest "asking something." Processes of these kinds can rendezvous when they agree on the channel. When $!c(v); p_1$ and $?c(x); p_2(x)$ rendezvous, they respectively evolve to $p_1$ and $p_2(v)$.
- **Parallel compositions** $p_1||p_2$ work as we're used to by now.
- **Duplications** $\mathsf{dup}(p)$ act just like infinitely many copies of $p$ composed in parallel. We use them to implement nonterminating "server" processes that are prepared to respond to many requests over particular channels. In traditional process algebra, duplication fills the role that loops and recursion fill in conventional programming.
- **The inert process** done is incapable of doing anything at all. It stands for a finished program.

We give an operational semantics in the form of a *labeled transition system*. That is, we not only express how a step takes us from one state to another, but we also associate each step with a *label* that summarizes what happened. Our labels will include the *silent* label $\epsilon$, read labels $?c(v)$, and write labels $!c(v)$. The latter two indicate that a thread has read a value from or written a value to channel $c$, respectively, and the parameter $v$ indicates which value was read or written. We write $p_1 \xrightarrow{l} p_2$ to say that process $p_1$ steps to $p_2$ by performing label $l$. We use $p_1 \longrightarrow p_2$ as an abbreviation for $p_1 \xrightarrow{\epsilon} p_2$.

We start with the rules for sends and receives.

$$\frac{}{!c(v); p \xrightarrow{!c(v)} p} \qquad \frac{}{?c(x); p(x) \xrightarrow{?c(v)} p(v)}$$

They record the action in the obvious way, but there is already an interesting wrinkle: the rule for receives *picks a value $v$ nondeterministically*. This nondeterminism is resolved by the next two rules, the rendezvous rules, which force a read label to match a write label precisely.

$$\frac{p_1 \xrightarrow{!c(v)} p_1' \quad p_2 \xrightarrow{?c(v)} p_2'}{p_1||p_2 \longrightarrow p_1'||p_2'} \qquad \frac{p_1 \xrightarrow{?c(v)} p_1' \quad p_2 \xrightarrow{!c(v)} p_2'}{p_1||p_2 \longrightarrow p_1'||p_2'}$$

A fresh channel generation can step according to any valid choice of channel.

$$\frac{c \notin \vec{c}}{\nu[\vec{c}](x); p(x) \longrightarrow \mathsf{block}(c); p(c)}$$

An abstraction boundary prevents steps with labels that mention the protected channel. (We overload notation $c \in l$ to indicate that channel $c$ appears in the send/receive position of label $l$.)

$$\frac{p \xrightarrow{l} p' \quad c \notin l}{\mathsf{block}(c); p \xrightarrow{l} \mathsf{block}(c); p'}$$

Any step can be lifted up out of a parallel composition.

$$\frac{p_1 \xrightarrow{l} p_1'}{p_1 || p_2 \xrightarrow{l} p_1' || p_2} \quad \frac{p_2 \xrightarrow{l} p_2'}{p_1 || p_2 \xrightarrow{l} p_1 || p_2'}$$

Finally, a duplication can spawn a new copy ("thread") at any time.

$$\frac{}{\mathsf{dup}(p) \longrightarrow \mathsf{dup}(p) || p}$$

The labeled-transition-system approach may seem a bit unwieldy for just explaining the behavior of programs. Where it really pays off is in supporting a modular, algebraic reasoning style about processes. In class, we go into the algebraic theory of refinement, allowing us to do equational-style proofs relating different terms in process algebra. Though we omit examples here for space reasons, we promise they involve no explicit bookkeeping with variables. Instead, as an example of reasoning about process calculus that shows essential unity with the invariant-based theorems of the prior sections, we turn to session types.

Process algebra can be helpful for modeling network protocols. Here, multiple *parties* step through a script of exchanging messages and making decisions based on message contents. A buggy party might introduce a *deadlock*, where, say, party A is blocked waiting for a message from party B, while B is also waiting for A. *Session types* [Honda 1993] are a style of static type system that rule out deadlock while allowing convenient separate checking of each party, given a shared protocol type.

There is an almost unlimited variation of different versions of session types. We still step through a progression of three variants here, and even by the end there will be obvious protocols that don't fit the framework. Still, we aim to convey the core ideas of the approach.

### 6.2 Basic Two-Party Session Types

Each of our type systems will apply to the object language from the prior subsection. Assume for now that a protocol involves exactly two parties. Here is a simple type system, explaining a protocol's "script" from the perspective of one party.

$$\begin{array}{rrcl} \text{Base types} & \sigma & & \\ \text{Session types} & \tau & ::= & !c(\sigma); \tau \mid ?c(\sigma); \tau \mid \mathsf{done} \end{array}$$

We model simple parties with no internal duplication or parallelism. A session type looks like an abstracted version of a process, remembering only the *types* of messages exchanged on channels, rather than their *values*. A simple set of typing rules makes the connection.

$$\frac{v : \sigma \quad p : \tau}{!c(v); p : !c(\sigma); \tau} \quad \frac{\forall v : \sigma.\ p(v) : \tau}{?c(x); p(x) : ?c(\sigma); \tau} \quad \frac{}{\mathsf{done} : \mathsf{done}}$$

The only wrinkle in these rules is the use of universal quantification for the receive rule, to force the body to type-check under any well-typed value read from the channel. Actually, such proof obligations may be nontrivial when we encode this object language in the mixed-embedding style, where the body $p$ in the rule could include arbitrary metalanguage computation, to choose a body

based on the value $v$ read from the channel. Note that this higher-order approach allows us to avoid all explicit modeling of variables and typing contexts.

The associated Coq code demonstrates tactics to deal with the complications of higher-order encoding, for automatic type-checking of concrete programs. That code is also where we keep all of our concrete examples of object-language programs. As in prior examples, though, many metatheorem proofs go through without **explicit accounting (e.g., type-system weakening or substitution lemmas)** for the complexities of using the metalanguage's function space in representing session types.

For the rest of this section, we will interpret last subsection's object language as a transition system with one small change: we only allow silent steps. That is, we only model whole programs, with no communication with "the environment." As a result, we consider self-contained protocols.

A satisfying soundness theorem applies to our type system. To state it, we first need the crucial operation of *complementing* a session type.

$$\begin{array}{rcl} \overline{!c(\sigma); \tau} & = & ?c(\sigma); \overline{\tau} \\ \overline{?c(\sigma); \tau} & = & !c(\sigma); \overline{\tau} \\ \overline{\mathsf{done}} & = & \mathsf{done} \end{array}$$

It is apparent that complementation just swaps the sends and receives. When the original session type tells one party what to do, the complement type tells the other party what to do. The power of this approach is that we can write one global protocol description (the session type) and then check two parties' code against it separately. A new version of one party can be dropped in without rechecking the other party's code.

Using complementation, we can give succinct conditions for deadlock freedom of a pair of parties.

Theorem 6.1. *If $p_1 : \tau$ and $p_2 : \overline{\tau}$, then it is an invariant of $p_1 || p_2$ that an intermediate process is either* done||done *or can take a step.*

Proof. By invariant induction, after strengthening the invariant to say that any intermediate process takes the form $p_1' || p_2'$, where, for some type $\tau'$, we have $p_1' : \tau'$ and $p_2' : \overline{\tau'}$. The inductive case of the proof proceeds by simple inversion on the derivation of $p_1' : \tau'$, where by the definition of complement it is apparent that any communication $p_1'$ performs has a matching action at the start of $p_2'$. The choice of $\tau'$ changes during such a step, to the "tail" of the old $\tau'$. □

### 6.3 Dependent Two-Party Session Types

It is a boring protocol that follows such a regular communication pattern as our first type system accepts. Rather, it tends to be crucial to change up the expected protocol steps, based on *values* sent over channels. It is natural to switch to a *dependent* type system to strengthen our expressiveness. That is, a communication type will allow its body type to depend on the value sent or received.

$$\text{Session types} \quad \tau \quad ::= \quad !c(x : \sigma); \tau(x) \mid ?c(x : \sigma); \tau(x) \mid \mathsf{done}$$

Each nontrivial construct does more than give the base type that should be sent or received on or from a channel. We also bind a variable $x$, to stand for the value sent or received. It may be unintuitive that we must introduce a binder even for sends, when the sender is in control of which value will be sent. The reason is that we must allow the sender to then continue with different subprotocols for different values that might be sent. We should not force the sender's hand by fixing a value in advance, when that value might depend on arbitrary program logic. Indeed, it is handy to use the mixed-embedding style here, so we have the full power of the metalanguage to compute which session type should apply after a given send or receive. That is, the benefit of **"complex control flow for free"** applies at the type level, too.

Table 1. Different kinds of bureaucracy avoided in our different examples

| | Sep. Log. | P.-O. Red. | Conc. Sep. Log. | Sess. Types |
|---|---|---|---|---|
| **Encoding of pure features** | ✓ | ✓ | ✓ | ✓ |
| **Substitution in semantics** | ✓ | ✓ | ✓ | ✓ |
| **Program vs. logical variables** | ✓ | | ✓ | |
| **Variable-related side conditions** | ✓ | | ✓ | |
| **Thread-local vs. shared vars.** | | ✓ | ✓ | |
| **Details of abstract interp.** | | ✓ | | |
| **Typing contexts and weakening** | | | | ✓ |
| **Adding dependent types** | | | | ✓ |

Very little change is needed in the typing rules.

$$\frac{v : \sigma \quad p : \tau(v)}{!c(v); p : !c(x : \sigma); \tau(x)} \qquad \frac{\forall v : \sigma.\ p(v) : \tau(v)}{?c(x); p(x) : ?c(x : \sigma); \tau(x)} \qquad \frac{}{\mathsf{done} : \mathsf{done}}$$

Our deadlock-freedom property is easy to reestablish.

Theorem 6.2. *If $p_1 : \tau$ and $p_2 : \overline{\tau}$, then it is an invariant of $p_1 || p_2$ that an intermediate process is either* done||done *or can take a step.*

Proof. Literally the same Coq proof script as for Theorem 6.1! Dependent types aren't usually that easy to introduce. □

### 6.4 Multiparty Session Types

New complications arise when more than two parties are communicating in a protocol. Now it is no longer possible to start from one party's view of a protocol and compute any other party's view. The reason is that each message only involves two parties. Any other party will not see that message in its own session type, making it impossible to preserve that message in a complement-like operation.

Instead, we define one global session type that includes only "send" operations. However, we name the parties and parameterize on a mapping from channels to unique parties that own their send and receive ends. That is, for any given channel and operation on it (send or receive), precisely one party is given permission to perform the operation – and indeed, when the time comes, that party is *obligated* to perform the operation, to avoid deadlock.

The style of the previous two type systems and their soundness theorems adapts well to this variant, though we leave details to the book.

## 7 DISCUSSION AND RELATED WORK

Table 1 reviews the different categories of bureaucratic overhead that are common in program proof, noting which of the main examples from this paper benefit from avoiding each nuisance.

Encoding of terms with interesting variable-binding structure has been a thorn in the side of mechanized proof for decades. Hygiene rules with names like "the Barendregt convention" [Barendregt 1985] make sense on paper but then provoke significant bureaucracy in full rigor. For that reason, most developments use something other than concrete strings as variable names. A proposal by de Bruijn [1972] introduced de Bruijn indices and levels, which represent bound variables as natural numbers, counting how many inner scopes one must pass through to find the binders they refer to. Nominal logic [Pitts 2003], with its explicit reasoning about finite support and renaming, has formed another foundation for implementations in several proof assistants. Among popular choices, the locally nameless style [Aydemir et al. 2008] was popularized most recently; it adopts a

pragmatic mix of concrete variable names for free variables and de-Bruijn-style indices for bound variables. All of these *first-order* encodings introduce significant bureaucracy, for aspects that we avoided in this paper.

Harper et al. [1987] introduced *higher-order abstract syntax (HOAS)*, which encodes binders using function spaces of appropriate metalanguages. In fact, the appropriateness conditions are difficult to square with general-purpose logics, e.g. Hoare-logic proofs of significant code bases seem not to have been developed in any HOAS-compatible logic. To bridge that gap, variations have been proposed, including weak HOAS [Honsell et al. 2001a] and parametric HOAS [Chlipala 2008]. These encodings are still awkward to apply to results as basic as syntactic type soundness, as they do not support substitution in a particularly natural way. With mixed embeddings, we give up on the requirement to rule out *exotic terms* (which get too clever in their use of the metalanguage's function space), instead embracing the chance to write down terms that would not fit in a typical core calculus, because they use arbitrary features of Coq's Gallina language.

Mixed embeddings have been part of the community folklore and have showed up in a few projects. The FSCQ verified file system [Chen et al. 2015] uses a mixed embedding to add disk-access operations atop Gallina. Interaction trees [Xia et al. 2020] are a more general framework, parameterized over kinds of side effects, supporting interesting composition operators.

Another educational presentation of Coq, *Programs and Proofs: Mechanizing Mathematics with Dependent Types* [Sergey 2014], presents a separation logic with very similar program constructors. However, those constructors are not from an inductive definition but rather they just return values in a semantic domain from a library for Hoare Type Theory [Nanevski et al. 2008], which indeed already presented similar benefits in binder bookkeeping (and served as the present author's own introduction to the style in related work [Chlipala et al. 2009]). By not giving programs syntax in some kind of algebraic datatype, we lose the ability do induction over the syntax of programs, so for instance we cannot hope to take a single object language and show off how to approach it with both partial-order reduction and concurrent separation logic, unless both results are built into the semantic domain from the start.

*Software Foundations* [Pierce et al. 2018] is the most popular proof-assistant-backed introduction to semantics. It sticks to concrete encoding of variables, facilitating presentation of both syntactic type-soundness proofs and Hoare logic. *Concrete Semantics* [Nipkow and Klein 2014], based on Isabelle/HOL, also sticks to concrete encoding of variables for a fixed imperative language, working through compiler correctness, type soundness, Hoare logic, and abstract interpretation. Our own course covers all the above topics and more. One theory, not completely justified by the anecdotes reported here, is that avoidance of variable bureaucracy helps us not just make it through more distinct examples of celebrated metatheorems but also decreases the friction of applying any one deduction system to interesting programs. For instance, by the end of the term, we have students doing mostly automated proofs of functional correctness for concurrent programs that use linked data structures, without having skipped any steps in justifying the reasoning principles they employ. It seems plausible that dealing constantly with variable binding would slow presentation of every topic and force the course to wind up at a less complete point.

## ACKNOWLEDGMENTS

The author would like to thank the teaching assistants for the five offerings so far of courses he taught based on this material: Joonwon Choi, Andres Erbsen, Samuel Gruetter, Clément Pit-Claudel, Benjamin Sherman, and Peng Wang. Numerous students in class and miscellaneous other users of the materials have contributed suggestions for *Formal Reasoning About Programs*, which can

be reconstructed from history on GitHub[3]. Thanks are also due to Tej Chajed for feedback on a draft and to Tobias Nipkow and Ilya Sergey for checking the last section's characterization of their books.

## REFERENCES

Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. 2008. Engineering formal metatheory. In *Proc. POPL* (San Francisco, California, USA). 3–15. https://doi.org/10.1145/1328438.1328443

Hendrik Pieter Barendregt. 1985. *The lambda calculus - its syntax and semantics*. Studies in logic and the foundations of mathematics, Vol. 103. North-Holland.

Richard J. Boulton, Andrew Gordon, Michael J. C. Gordon, John Harrison, John Herbert, and John Van Tassel. 1992. Experience with Embedding Hardware Description Languages in HOL. In *Proceedings of the IFIP TC10/WG 10.2 International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience*. North-Holland Publishing Co., NLD, 129–156.

H. Chen, D. Ziegler, T. Chajed, A. Chlipala, M. F. Kaashoek, and N. Zeldovich. 2015. Using Crash Hoare Logic for Certifying the FSCQ File System. In *Proc. SOSP*. https://doi.org/10.1145/2815400.2815402

Adam Chlipala. 2008. Parametric Higher-Order Abstract Syntax for Mechanized Semantics. In *Proc. ICFP*. https://doi.org/10.1145/1411204.1411226

Adam Chlipala, Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. 2009. Effective Interactive Proofs for Higher-Order Imperative Programs. In *Proc. ICFP*. https://doi.org/10.1145/1596550.1596565

Nicolas G. de Bruijn. 1972. Lambda-calculus notation with nameless dummies: a tool for automatic formal manipulation with application to the Church-Rosser theorem. *Indag. Math.* 34(5) (1972), 381–392.

Patrice Godefroid, J. van Leeuwen, J. Hartmanis, G. Goos, and Pierre Wolper. 1996. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Springer-Verlag, Berlin, Heidelberg.

Andrew D. Gordon. 1993. An Operational Semantics for I/O in a Lazy Functional Language. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture* (Copenhagen, Denmark) *(FPCA '93)*. Association for Computing Machinery, New York, NY, USA, 136–145. https://doi.org/10.1145/165180.165199

Robert Harper, Furio Honsell, and Gordon D. Plotkin. 1987. A Framework for Defining Logics. In *Proceedings of the Symposium on Logic in Computer Science (LICS '87), Ithaca, New York, USA, June 22-25, 1987*. IEEE Computer Society, 194–204.

C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (Oct. 1969), 576–580. https://doi.org/10.1145/363235.363259

Kohei Honda. 1993. Types for dyadic interaction. In *CONCUR'93*, Eike Best (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 509–523. https://doi.org/10.1007/3-540-57208-2_35

Furio Honsell, Marino Miculan, and Ivan Scagnetto. 2001a. An Axiomatic Approach to Metareasoning on Nominal Algebras in HOAS. In *Proc. ICALP*. 963–978. https://doi.org/10.1007/3-540-48224-5_78

Furio Honsell, Marino Miculan, and Ivan Scagnetto. 2001b. $\pi$-Calculus in (Co)Inductive-Type Theory. *Theor. Comput. Sci.* 253, 2 (Feb. 2001), 239–285. https://doi.org/10.1016/S0304-3975(00)00095-5

R. Milner. 1982. *A Calculus of Communicating Systems*. Springer-Verlag, Berlin, Heidelberg.

Robin Milner. 1999. *Communicating and Mobile Systems: The $\pi$-Calculus*. Cambridge University Press, USA.

Aleksandar Nanevski, Greg Morrisett, Avraham Shinnar, Paul Govereau, and Lars Birkedal. 2008. Ynot: Dependent Types for Imperative Programs. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming* (Victoria, BC, Canada) *(ICFP '08)*. Association for Computing Machinery, New York, NY, USA, 229–240. https://doi.org/10.1145/1411204.1411237

Tobias Nipkow and Gerwin Klein. 2014. *Concrete Semantics: With Isabelle/HOL*. Springer Publishing Company, Incorporated. https://doi.org/10.1007/978-3-319-10542-0

Peter W. O'Hearn. 2007. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.* 375, 1-3 (2007), 271–307. https://doi.org/10.1016/j.tcs.2006.12.035

Simon L. Peyton Jones and Philip Wadler. 1993. Imperative Functional Programming. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Charleston, South Carolina, USA) *(POPL '93)*. Association for Computing Machinery, New York, NY, USA, 71–84. https://doi.org/10.1145/158511.158524

F. Pfenning and C. Elliot. 1988. Higher-order abstract syntax. In *Proc. PLDI* (Atlanta, Georgia, United States). 199–208. https://doi.org/10.1145/53990.54010

Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, Andrew Tolmach, and Brent Yorgey. 2018. *Programming Language Foundations*. Electronic textbook.

[3] https://github.com/achlipala/frap

Andrew M. Pitts. 2003. Nominal logic, a first order theory of names and binding. *Information and Computation* 186, 2 (2003), 165–193. https://doi.org/10.1016/S0890-5401(03)00138-X Theoretical Aspects of Computer Software (TACS 2001).

John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proc. LICS*. https://doi.org/10.1109/LICS.2002.1029817

Ilya Sergey. 2014. *Programs and Proofs: Mechanizing Mathematics with Dependent Types*. https://doi.org/10.5281/zenodo.4996239 Lecture notes with exercises.

A.K. Wright and M. Felleisen. 1994. A Syntactic Approach to Type Soundness. *Inf. Comput.* 115, 1 (Nov. 1994), 38–94. https://doi.org/10.1006/inco.1994.1093

Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. 2020. Interaction trees: representing recursive and impure programs in Coq. *Proc. ACM Program. Lang.* 4, POPL (2020), 51:1–51:32. https://doi.org/10.1145/3371119