Verification Condition Generation with the Dijkstra State Monad

Cole Schlesinger
Princeton University

Nikhil Swamy
Microsoft Research

Abstract

The Hoare state monad provides a powerful means of structuring the verification of higher-order, stateful programs. This paper defines a new variant of the Hoare state monad, which, rather than being a triple of a pre-condition, a return type, and a post-condition, is a pair of a return type and a predicate transformer. We dub this monad the *Dijkstra state monad*.

Using the Dijkstra state monad, we define a new unification-based type inference algorithm, which succeeds in computing verification conditions for higher-order stateful programs. We prove our algorithm sound. We also prove it complete with respect to a simple surface-level typing judgment, resembling ML type inference. In other words, we show that any recursion-free program typeable in our surface system can also be typed in the Dijkstra monad. Thus, programmers may use our algorithm to type their programs in the Dijkstra monad and obtain more precise types, knowing that when our algorithm fails to infer a type, the failure is due to a typing error that can be detected by our simple surface type system. Recursive functions can be typed as usual if they are annotated with their loop invariants. We also show how to structure specifications so that despite the use of higher-order logic in the types of higher-order functions, we can generate first-order verification conditions for many programs. The result is a light-weight, yet powerful system for specification and verification of deep properties of stateful functional programs.

We have implemented our inference algorithm as a front-end to the F* compiler and report on a preliminary evaluation of our tool on a collection of benchmarks.

1. Introduction

Functional programmers often extol the gains in productivity, modularity and elegance provided by higher-order functions. Such functions are routinely combined with effects, whether directly, as in ML, or monadically, as in Haskell. While the type systems of these languages rule out many common errors, and also guarantee some properties beyond basic type safety (*e.g.*, parametricity), these properties remain relatively simple. Proving functional correctness requires a more advanced analysis. But, as is well known, the very combination of higher-order functions and state that programmers love poses difficulties for program verification tools.

A promising way to structure the verification of higher-order stateful programs is via the Hoare state monad (Nanevski et al. 2008), or Hoare monad, for short. Conceptually, the Hoare monad is a simple idea: it is a refinement of the normal state monad, recording pre- and post-conditions on the input and output state. Informally, `ST Pre a Post` is the type of a computation from input heaps `h` satisfying the pre-condition `Pre h` to a pair containing the result of the computation `x:a` and an output heap `h'` satisfying `Post x h'`. We show its definition below.[1]

```
type ST Pre a Post = h:heap{Pre h} → (x:a * h':heap{Post x h'})
```

While the Hoare monad has been put to use successfully by Nanevski et al. in tools like Ynot, this work has primarily been in the context of interactive proof assistants like Coq. There are at least two difficulties. First, type inference for the Hoare monad (i.e., verification condition generation) is a bidirectional type inference algorithm. Second, the inferred verification conditions (VCs), which may be in higher-order logic, have to be solved interactively using tactic-based proving in Coq.

To get a sense of the difficulties involved simply in computing VCs (let alone solving them, for the moment), consider the program `inv` below, which updates the contents of an integer reference with its inverse.

```
let inv x = x := 1 / !x
```

What is the weakest pre-condition for this program in the Hoare monad? Typically, a weakest pre-condition calculus works by starting with a programmer-supplied post-condition and "pushing" it backwards through the computation to compute a pre-condition. Here, we have no post-condition to get us started. So, are we stuck?

Of course not. Our first insight, borrowed from Dijkstra, is that weakest pre-condition computations are designed to be parametric in the post-condition formula over which the weakest pre-condition is computed. That is, a weakest pre-condition computation can be viewed as a *predicate transformer*, transforming an arbitrary post-condition predicate into a pre-condition.

Based on this insight, we can write a precise specification for `inv`, shown below. We quantify over all post-conditions (predicates relating a `unit` value and an output heap), and define the weakest pre-condition of `inv` over this symbolic post-condition. In the type below, the functions *sel* and *upd* are the usual heap select and update functions from McCarthy's theory of functional arrays (Mccarthy 1962).

```
∀ψ. x:ref int → ST (λ h. sel h x ≠ 0 && ψ () (upd h x (1/sel h x))) unit ψ
```

As we will see shortly, writing specifications in this style (with polymorphic post-conditions) is nice for several reasons. Two of them are already apparent. First, a symbolically computed pre-condition does not require a programmer-provided post-condition. Second, notice that the pre-condition of `inv` is able to describe the output heap (*upd* `h x` (1/*sel* `h x`)) function of the input heap `h`. Prior approaches (e.g., Nanevski et al. (2008)) have relied on two-state post-conditions to relate the input and output-heap—this is unnecessary if specifications are always post-condition parametric.

To appreciate the power of post-condition parametricity, let us look at another aspect of the Hoare state monad. We sketch the signature of the bind operator of the `ST` monad below.

```
ST Pre a Post1 → (x:a → ST (Post1 x) b Post) → ST Pre b Post
```

This combinator allows two monadic computations to be composed, but it requires the post-condition of the first computation `f` to be syntactically identical to the pre-condition of the second computation `g`. Thus, when inferring types for the Hoare monad, we need

[1] We write dependent function arrows as `x:t → t'`, where the formal parameter `x:t` is in scope in the codomain `t'`; this is sometimes written Π`x:t.t'`.

a subsumption rule in the type system that captures the *rule of consequence*—i.e., we need to be able to strengthen the pre-condition of g and/or weaken the post-condition of f in order to infer a type for bindST f g. However, type inference in a setting that includes subtyping or subsumption is challenging.

Once again, post-condition polymorphism comes to the rescue. If we were to uniformly write all our specification in the style of Dijkstra's predicate transformers, instead of the Hoare monad, the result would be what we call the *Dijkstra state monad*, which we write DST and sketch below.

```
type DST a Tx = ∀ψ. ST (Tx ψ) a ψ
val bindDST: DST a Tx1
          → (x:a → DST b (Tx2 x))
          → DST b (Λψ. Tx1 (λ x. Tx2 x ψ))
```

The DST monad takes just two parameters: the type a is the type of the value returned by the computation, while Tx is a predicate transformer that computes a pre-condition for the computation for any post-condition. The bind combinator of the DST monad, unlike the ST monad, allows two computations to be composed regardless of the relationship between their pre- and post-conditions. We no longer need a subsumption rule in order to infer types for programs in the DST monad.

At this point, the reader might wonder if there is another design path we might follow—one that avoids higher order and polymorphic predicates. Unfortunately, such a route to success seems impossible in a higher-order language. In essence, a function f that abstracts over a function g must have a specification that abstracts over the specification of g. The predicate polymorphism of DST makes this easy to express. We show the type of apply below, but leave a detailed discussion until later.

```
val apply:∀α,β,η f:(y:α → DST β (η y)) →x:α → DST β (η x)
let apply f x = f x
```

In summary, in order to write specifications for higher-order functions, one needs to use predicate polymorphism anyway. We advocate embracing this style of specification wholeheartedly by using the DST monad throughout.

1.1 The shape of our development and contributions

We present our ideas in the setting of the F⋆ programming language (Swamy et al. 2011a), a dependently typed dialect of ML. We strive to develop an effective verification condition generator targeting F⋆ programs that make heavy use of state. Ideally, we would like to produce proof obligations that can be automatically discharged by Z3 (de Moura and Bjørner 2008), the SMT solver used by F⋆'s typechecker. To simplify the presentation, rather than work with the entire F⋆ language, we define a small calculus λDST, intended to be a tiny, ML-like subset of F⋆.

Our goal is to define a sound and complete weakest pre-condition calculus for recursion-free λDST programs. Unfortunately, in its entirely unrestricted form, this proves intractable. The main source of difficulty is that to type higher-order functions we need to infer higher-rank types, which is, in general, undecidable (Wells 1994). However, with some restrictions familiar from work that aims to infer higher-rank types, (e.g., as in HMF (Leijen 2008), we require higher-order arguments to be annotated), we make progress. Our development and contributions proceeds as follows:

A simple specification of the set of typeable programs. We present a simple, surface-level type system that defines the set of λDST programs for which we can infer precise verification conditions. This type system computes basic ML types for λDST programs. However, by design, not all ML programs can be typed in our surface type system, e.g., those programs that contain functions with unannotated higher-order arguments are rejected.

A core type inference algorithm. We then present our main type inference algorithm for λDST programs, interpreting them in the DST monad, and inferring precise specifications for them.

Type inference is sound (Theorem 1). When a well-typed program λDST is executed in a context that satisfies the inferred VCs, then all assertions in the program are guaranteed to succeed. For wrong programs, the inferred constraints are unsatisfiable. We prove this result by elaborating well-typed λDST programs into F⋆, and relying on the soundness of F⋆.

Type inference is complete (Theorem 2). We prove that all λDST programs that can be typed in the surface level type system can also be typed using our core inference algorithm, yielding a more informative type. As such, programmers who are accustomed to the type system of ML, need only understand the restrictions of our simple surface level type system. They may then use our algorithm and obtain more precise types, knowing that when our algorithm fails to infer a type, the failure is due to a typing error that can be detected by the simple surface type system.

First-order VCs (Theorem 3). We prescribe a specific form in which to annotate types for higher-order libraries, making carfully structured use of predicate transformers and other constructs from higher-order logic. For programs that adhere to this prescription, we prove that our core inference algorithm will yield only first-order verification conditions. These VCs can then be fed to an SMT solver. This is in contrast to prior work on verifying higher-order stateful code using the Hoare state monad, *e.g.*, HTT/Ynot (Nanevski et al. 2008), which, despite some automation, requires interactive proofs in the Coq proof assistant (Bertot and Castéran 2004).

Implementation. We have implemented our algorithm as a front-end to the F⋆ compiler (Swamy et al. 2011a). Following inference, we elaborate λDST programs into F⋆. By relying on the self-certified F⋆ core typechecker (Strub et al. 2012) for verification, we can obtain a highly reliable path for certification of higher-order stateful programs—comparable in reliability to certification of, say, Ynot programs using Coq.

Evaluation. We report on a preliminary experimental evaluation of our tool on a suite of thirty small benchmarks, totaling approximately 500 lines of code. More substantially, the Dijkstra monad and the verification condition generator described here has been applied to the automatic verification of a suite of JavaScript programs, programs that make heavy use of a dynamically typed higher-order store. The topic of JavaScript verification, orthogonal to this paper, is described in another technical report (Swamy et al. 2012a).

Outline. We start in Section 2 by discussing our solution informally, using several examples. Along the way, we review F⋆ and the Hoare state monad. Section 3 presents λDST and its surface typing relation formally. Section 4 presents our core inference algorithm, as well as the metatheoretical properties of our system. We discuss further examples and our implementation in Sections 5 and 6. Section 7 discusses related work and concludes. Full statements and proofs of the theorems, including a complete definition of the λDST calculus, can be found in the appendix.

2. Overview

This paper defines λDST, an ML-like surface language; a type inference algorithm; and an elaboration of λDST into F⋆. λDST is not a separate language design in its own right—it is simply the subset of F⋆ we use in this paper. F⋆ is itself a variant of ML with a similar syntax and dynamic semantics but with a type system based on dependent types. To date, F⋆ has been used to program and verify more than 30,000 lines of code, including security protocols, web browser extensions, and distributed applications. Its

main typechecker, compiler, and runtime support are coded in F#, although its core typechecker is written in F* itself and has been certified for correctness.

2.1 Basics of F*

Three elements in the type system of F* play a significant role in this paper: value-indexed types, dependent functions, and ghost refinement types. Using these features, it is possible to write, say, n:nat → array int n, the type of a function that allocates an array of n integers. Formally, the type is a dependent function from values n in its domain nat to array int n, the latter being an example of a type that is indexed both by a type (int) as well as a value (n). The *kind* given to the type constructor array is $\star \Rightarrow$ nat $\Rightarrow \star$, indicating that it constructs a $\star$-kinded type (the kind given to the types of values and computations) from a $\star$-kinded type and a nat-typed value. Types are ascribed to terms using a single colon ($x{:}t$) whereas kinds are ascribed to types using two colons (int::$\star$).

The type nat itself can be defined using the ghost refinement, x:int{x ≥ 0}, a refinement of the primitive type int. Here, the formula x ≥ 0 is itself a type, built using the type constructor ≥ ::int ⇒ int ⇒ E. The kind E is a base kind in F*, and stands for the F* propositions that have non-constructive (say, SMT-supplied) proofs. Importantly, the representation of nat is the same as int, and indeed nat is a subtype of int.

The formulas that appear in ghost refinements are decided using an SMT solver. As such, decidability of type checking in F* is modulo decidability of the logic used in the solver. We generally use a first-order logic extended with some common theories, including equality, linear arithmetic, and functional arrays.

2.2 The Hoare state monad

The state monad is a convenient way of structuring stateful computations in a functional language. Computations in the state monad are functions from an initial heap to a pair containing the computed value and resulting heap, *i.e.*, heap → (α * heap). Nanevski et al. (2008) integrate a Hoare logic (Hoare 1969) with the state monad using dependent types. The resulting monad is called the Hoare state monad, or Hoare monad.

Encoding the Hoare state monad in F*

```
type heap :: ⋆
type ST (φ::heap ⇒ E) (α::⋆) (ψ::α ⇒ heap ⇒ E) =
      h:heap{φ h} →(x:α * h':heap{ψ x h'})

val returnST : ∀ α::⋆,
                 ψ::α ⇒ heap ⇒ E.
                 x:α
               →ST (ψ x) α ψ
let returnST α ψ x h = (x,h)

val bindST : ∀ α::⋆, β::⋆,
               φ::heap ⇒ E,
               ψ1::α ⇒ heap ⇒ E,
               ψ::β ⇒ heap ⇒ E.
               ST φ α ψ1
             →(x:α →ST (ψ1 x) β ψ)
             →ST φ β ψ
let bindST α β φ ψ1 ψ f g h = let x, h = f h in g x h
```

Informally, the Hoare state monad is simply the state monad (heap →(α* heap)) augmented with predicates to track the pre- and post-conditions of a stateful computation that may read or write a heap and produce an α-typed result. In the listing above heap is an abstract type, and the ST$\phi\alpha\psi$ is the type of computation which when run in a heap h satisfying the pre-condition ϕ h produces a result x:α and an output heap h' satisfying the relation ψ x h', unless it diverges.

As usual, the ST monad comes with with two operations, bindST and returnST, for sequencing monadic computations and lifting pure values into the monad, respectively (Wadler 1992). Lifting values into the Hoare state monad is simple. Values are pure, by definition having no effect on the heap, and so any properties of the initial heap are preserved in the resulting heap. Working backwards in a weakest pre-condition style, returnST takes the form of a polymorphic function on types α and values x of type α, yielding a stateful computation where the final heap is unchanged and returned along with x.

The bindST operator allows a stateful computation f with a result type α to be composed with a stateful function g that expects an α and returns a β. bindST is also polymorphic in the pre- and post-conditions of f and g. The bindST function allows two stateful computations to be composed, so long as the post-condition of the first matches the pre-condition of the second. The type, in effect, captures the classic Hoare rule for sequencing.

2.3 The Dijkstra state monad

Notice that writing specifications for the ST-monad that are polymorphic in the post-condition predicate is quite convenient, e.g., one does not have to directly state that returnST leaves the input heap h unchanged. Without this polymorphic style one would need state extra conditions to relate the output heap to the input heap, for example, by making use of two-state predicates as post-conditions. To avoid the additional complexity of two-state predicates (and to simplify type inference), we embrace post-condition polymorphism wholeheartedly. We identify a variant of the Hoare monad, the Dijkstra state monad, and define it below.

Definition of the Dijkstra state monad in F*

```
type DST (α::⋆) φ::(α ⇒ heap ⇒ E) ⇒ heap ⇒ E=
      ∀ψ::(α ⇒ heap ⇒ E). ST (φ ψ) α ψ
val return : ∀α. x:α →DST α(Λψ::α⇒ heap⇒ E.ψ x)
let return α x ψ h = (x, h)

val bind : ∀α::⋆, β::⋆,
           φ1::(α ⇒ heap ⇒ E) ⇒ heap ⇒ E,
           φ2::α ⇒ (β ⇒ heap ⇒ E) ⇒ heap ⇒ E.
           DST α φ1
         →(x:α →DST β (φ2 x))
         →DST β (Λψ. φ1 (λy. φ2 y ψ))
let bind α β φ1 φ2 f g ψ h =
    let y, h' = f (λy.φ2 y ψ) h in
    g y ψ h'
```

Rather than work directly with pre- and post-conditions, we abstract the semantics of each computation using a predicate transformer ϕ (Dijkstra 1975) that characterizes the effects of the computation in a weakest pre-condition style. In essence, DST t ϕ is a refinement of the state monad, quantifying over all post-conditions ψ and generating a pre-condition (ϕ ψ) on the initial heap h and a similar refinement, ψ, for the post-condition, relating the return value x to the final heap h'.

The displays above show the kinds of all the type variables and explicit type abstraction and application. Henceforth, we omit explicitly mentioning these when they can be inferred from the context.

2.4 Polymorphic specifications for higher-order functions

In the previous section, predicate polymorphism allowed us to give precise types to our monadic combinators. However, programs that use these combinators also need predicate-polymorphic specifications. The type of apply mentioned in the Introduction is one example: we revisit it in more detail here.

A function f that abstracts over another function g must have a specification that abstracts over the specification of g. Indeed, we

insist such a function to be maximally general in the specification of g, i.e., f must be parametric in the predicate transformer that represents the semantics of g. In Section 4.5, we impose a well-formedness condition on the types of higher-order functions to enforce this condition.

To illustrate, we start with the simplest higher-order function, the apply combinator, which abstracts function application. We show its type and definition in our system below.

The type and implementation of apply

```
val apply:∀α,β. η.(y:α →DST β (η y)) →x:α →DST β (η x)
let apply f x = f x
```

The specification above shows that the predicate transformer of apply itself is just the predicate transformer of the abstracted function f.

The apply combinator, like all higher-order functions in our system, has a type of the following shape: $\eta.(x{:}t_1 \to \mathsf{DST}\ t_2(\eta y)) \to \mathsf{DST}\ t\ \phi$. Each function-typed argument in a type is accompanied by a predicate transformer η which captures the specification of the abstracted function. The predicate transformer variable η is in scope to the right of the dot that follows it, i.e., it may appear free in t_1, t_2, t or ϕ. Requiring every higher-order function to be given a type with this syntactic shape imposes a uniformity in the shape of function types, while facilitating a significantly simpler algorithm for comparing and unifying types, without loss of expressiveness.

The apply function is simple. But, more complex higher-order functions require more elaborate specifications. The display below shows the type and definition of twice, the combinator that applies its argument twice.

Annotated higher-order function: twice

```
val twice: ∀α. η.(y:α →DST α(η y))
                →x:α
                →DST α(Λψ. η x (λ z. η z ψ))
let twice f x = let y = f x in f y
```

In the body of the function twice (above), we apply the abstracted function f twice, each time in a context that requires a different post-condition. In general, we may apply an abstracted function in arbitrarily many contexts. Indexing the DST monad with predicate transformers captures this precisely. The above type is of the form $\eta.(y{:}\tau \to \mathsf{DST}\ t\ (\eta\ y)) \to t'$, where η is a predicate transformer variable that models the behavior of the abstracted function $y{:}\tau \to \mathsf{DST}\ t\ (\eta\ y)$. With predicate transformers, the type of twice is exactly as one would expect: the predicate transformer for the entire function is simply the predicate transformer variable composed with itself. In Section 5, we show how predicate transformers can be combined with loop invariants to give precise specifications to higher-order recursive functions.

Now, rather than attempting to automatically infer such rich specifications for functions like twice, our algorithm requires all functions that abstract over function parameters to be annotated with their specifications. Given such an annotation, our algorithm can infer instantiations of predicate transformer variables at every call site of these functions.

The rationale behind this design is that, typically, higher-order functions (*e.g.*, map, fold) abstract over some form of complex (and often recursive) control. Since recursive functions require programmer-provided invariants anyway, an inference algorithm over-engineered to infer precise pre-conditions for higher-order functions is a poor trade-off. Instead, since the number of call sites outweighs the number of definitions of a higher-order function, a simpler algorithm that can infer annotations at call sites using annotations at definitions seems a better choice.

A client of twice

```
let inc x = x := !x + 1
let addTwo r = twice (λ(). inc r) ()
(* Inferred type *)
val addTwo : r:ref int →DST unit (Λψ PreInc r (λx. PreInc x ψ))
  where PreInc = λx.Λψ.λs0. ψ () (upd s0 x ((sel s0 x) + 1))
(* Elaborated to F* *)
let addTwo = λr. twice unit PreInc (Λγ.λ(). inc r) ()
```

Given the annotation on twice, a programmer may freely use the inc function, and write the code shown above. We can infer its type and elaborate it to F*—passing closures that capture references, generalizing its type, and inferring the instantiations is all computed automatically.

2.5 Polymorphic specifications for first-order code

Polymorphic specifications are not only useful for higher-order code. In first-order programs with no annotations, pre-conditions can be computed symbolically using polymorphic post-conditions, as long as such programs do not employ recursion. As first-order functions often appear frequently in higher-order code as top-level, utility or even anonymous functions, complete type inference over a first-order, recursion-free fragment of λ_{DST} relieves a substantial burden on the programmer.

Implementation of next_fragment in λ_{DST}

```
let next_fragment l =
  if is_empty_stream !out
  then None
  else let f, rem = stream_read !out l in
       out := rem;
       Some f
```

As a concrete example, we show a program and its inferred specification (adapted from an ongoing development, independent of this paper, that aims to certify an implementation of the TLS-1.2 protocol[2]). The details of next_fragment and its specification are of secondary importance—the main point is that the specification is arguably more complex than the function itself! Code like this is often written by F* programmers, manually threading state in a monadic style, and painstakingly writing detailed specifications even for small functions. With the λ_{DST} type inference and elaboration algorithm, we can now write imperative programs in direct style, and automatically compute a specification as precise as the manual specification written previously.

Inferred specification of next_fragment

```
val next_fragment: l:int →
 DST (option fragment)
      (Λψ. λs. (IsEmpty s.out ⟹ ψ None s) ∧
       (not(IsEmpty s.out) ⟹
         (∀f. StreamRead s.out f ⟹
           ψ(Some f) (Upd s out (App s.out f)))))
```

We conclude our overview with a technical remark. Hoare-style program logics are sometimes formulated with post-conditions that can mention both the pre- and the post-state. However, with polymorphic specifications, the additional complexity is unnecessary. The types inferred for inc, addTwo, next_fragment, *etc.*, illustrate how we can speak about the post-state as a function of the pre-state using predicate transformers. In a sense, polymorphic predicates give us a lightweight notion of framing—specifications need only mention the locations that are modified.

[2] http://tools.ietf.org/html/rfc5246

$$
\begin{array}{lcll}
v & ::= & x \mid () \mid \mathsf{true} \mid \mathsf{false} \mid \lambda x.e & \text{value} \\
e & ::= & v \mid v_1\ v_2 \mid \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 & \\
 & \mid & \mathbf{if}\ v\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 \mid \mathsf{ref}\ v \mid !v \mid v_1 := v_2 & \text{expr.} \\
\tau & ::= & \alpha \mid \mathsf{unit} \mid \mathsf{bool} \mid \mathsf{heap} \mid \mathsf{set} \mid \mathsf{ref}\ \tau & \text{small type} \\
\mathsf{t} & ::= & \tau \mid \mathsf{t} \to \mathsf{t}' & \text{surf. ty.} \\
s & ::= & \mathsf{t} \mid \forall \bar{\alpha}.\mathsf{t} & \text{surf. ty. scheme} \\
\Gamma & ::= & \cdot \mid x{:}s \mid \Gamma, \Gamma' &
\end{array}
$$

Figure 1. Surface syntax of λ_{DST} terms and types

3. A surface-level type system for λ_{DST}

We start our formal presentation by defining λ_{DST} and a simple, surface-level type system for this calculus. In Section 4 we develop our main type inference algorithm that interprets λ_{DST} programs in the DST monad. Our objective in this part of the paper is to provide a simple intuitive specification of the subset of λ_{DST} for which our main inference algorithm is guaranteed to succeed. Theorem 2 (Completeness) establishes that every program typeable in our surface type system can also be typed by our main algorithm.

3.1 Syntax

The syntax of λ_{DST} terms is shown in Figure 1. Values v and expressions e are standard for a λ calculus with references (reference values also being names x). Later, we show how to elaborate λ_{DST} programs into $\text{F}^\star$. For this purpose, λ_{DST} inherits some of the idiosyncrasies of $\text{F}^\star$. Notably, we restrict function arguments to be values, as it simplifies the treatment of value dependency in $\text{F}^\star$. Non-value function arguments must be hoisted using a **let**-binding.

We also show the syntax of surface-level types in λ_{DST}. This resembles the type language of ML, where polymorphic type schemes s are distinguished from mono-types t. However, λ_{DST} goes a step further and separates out a subset of the mono-types, i.e., the non-function, or small types, τ. As we will see, the key restriction in λ_{DST} is that in order for inference to succeed, function parameters and references must always have small types.

3.2 Typing

The surface typing judgment is written $\Gamma \vdash e : \mathsf{t}$, shown in Figure 2. The judgment is mostly standard for an ML-like language. For example, we have the value restriction, embodied in the rules (S-Bind) and (S-Gen), where only the types of let-bound values are generalized. The main interesting elements to pay attention to are the three highlighted rules.

The rule (S-x) states that a type scheme $\forall \bar{\alpha}.\mathsf{t}$ can only be instantiated by substituting small types for the type variables. Recall from Section 2.4 that we require function types to have a specific syntactic shape—higher-order functions must also abstract over predicate transformers corresponding to their function-typed arguments. Restricting (S-x) in this manner ensures that this invariant is preserved even through type instantiation.

The rule (S-Lam) shows that we only infer types for λ_{DST} functions that abstract over non-function-typed values. As already mentioned in Section 2.4, we view the inference of types for higher-order arguments as a poor trade-off—the complexity required in the inference algorithm outweighs the likely benefit, particularly as most higher-order functions involve some form of recursive or iterative control and must be annotated with an invariant anyway. Of course, λ_{DST} programs can still be higher-order—the context Γ is free to bind names at types of an arbitrary order. Thus, in a sense, we infer types for λ_{DST} programs that are clients of fully annotated higher-order libraries.

Finally, the rule (S-Ref) requires that references hold only non-function values. To appreciate the need for this restriction, consider a program that allocates a ref-cell, storing a value of type $t_1 = x{:}\mathsf{int} \to \mathsf{DST}\ \mathsf{int}\ \phi$ in it. Later, it updates the same ref-cell storing a value of type $t_2 = x{:}\mathsf{int} \to \mathsf{DST}\ \mathsf{int}\ \phi'$ in it. This program would be well-typed in ML (since the reference can just be given the type $\mathsf{ref}\ (\mathsf{int} \to \mathsf{int})$. However, typing it in the DST monad requires giving it a type that is a common supertype of t_1 and t_2. Inferring such a common supertype is non-trivial.

The last restriction may make it seem that λ_{DST} programs are restricted to use a first-order store. This is not true. So long as a reference can be given a small type, there is no problem. For example, in concurrent work on using λ_{DST} to verify JavaScript programs, we routinely employ a higher-order store, but, every reference has a small type, since each value in the store, even if it contains a function closure, has type dynamic.

4. Verification condition generation for λ_{DST}

We now present our main type inference algorithm. To simplify the presentation, we assume that a source λ_{DST} program has already been typed using the surface-level judgment, and converted into an annotated form e, where every λ-bound variable is annotated with its type, and every use of a variable with a polymorphic type is annotated with its type instantiations. This annotation of source terms is entirely straightforward—we relegate to the appendix the definition of a judgment $\Gamma \vdash e : \mathsf{t} \rightsquigarrow e$, which surface types a source term e and emits an annotated term e. Although, formally, we present our type inference procedure as a two-pass algorithm, in practice (as in our implementation), surface typing and verification condition generation can be done in a single pass.

4.1 Syntax and interpretation of types

The syntax of annotated λ_{DST} terms and the full language of types is shown in Figure 3. Annotated terms include λ-abstractions with annotations on the bound variable and variables $x_{\bar{\tau}}$ subscripted with type instantiations, if any. The full type language includes the small types τ of the previous section. But, function types t are dependent and have a monadic co-domain, $\mathsf{DST}\ t\ \phi$, where ϕ is a predicate transformer characterizing the function's effect on the heap. Note that monadic types may only appear to the right of an arrow, as is standard in a monadic interpretation of ML (Swamy et al. 2011b). Type schemes σ are as in the surface type system, except the generalized type is now a full type t instead of a surface type t.

Function types. Our restrictions on the shape of function types are manifest in the syntax. First, note that all function types have a monadic co-domain, i.e., a co-domain of the form $\mathsf{DST}\ t\ \phi$. This means that all functions are considered impure. This is clearly sub-optimal. For example, the function $f = \lambda x.x+1$ could be given a precise, pure type using only the language of refinements in $\text{F}^\star$, *e.g.*, $x{:}\mathsf{int} \to y{:}\mathsf{int}\{y = x+1\}$. However, for simplicity in λ_{DST}, we do not have refinement types on their own. Pre- and post-conditions on the Hoare monad are the only way to state refinement properties. So, we write the type of f as $x{:}\mathsf{int} \to \mathsf{DST}\ \mathsf{int}\ (\Lambda\psi.\lambda h.\psi\ (x+1)\ h)$. From this type we can conclude that f has no write effects on the heap, recovering some information about its purity, albeit in a roundabout way. In the long run, we expect to integrate λ_{DST} with recent work on inferring precise monadic types for ML programs (Swamy et al. 2011b) to alleviate this limitation. In the meantime, we limit ourselves to functions with monadic co-domains.

Next, note that function types come in two flavors. The first takes the form $x{:}\tau \to \mathsf{DST}\ t\ \phi$. This is a dependent function, with a small type τ in its domain. Higher-order functions, however, take a more stylized form, and are written $\overline{\eta_\kappa}.t_1 \to \mathsf{DST}\ t_2\ \phi$. Note, the formal parameter t_1 is not directly named. Instead, we bind a set of predicate transformer names $\overline{\eta_\kappa}$, in scope both within t_1 and to the right of the arrow. In more explicit syntax, $\overline{\eta_\kappa}.t_1 \to \mathsf{DST}\ t_2\ \phi$

$$\frac{}{\Gamma \vdash () : \mathsf{unit}}\ \text{S-1} \qquad \frac{v \in \{\mathsf{true}, \mathsf{false}\}}{\Gamma \vdash v : \mathsf{bool}}\ \text{S-b} \qquad \frac{\Gamma(x) = \forall \bar{\alpha}.\mathsf{t}}{\Gamma \vdash x : \mathsf{t}[\bar{\tau}/\bar{\alpha}]}\ \text{S-x}$$

$$\frac{\Gamma, x{:}\tau \vdash e : \mathsf{t}}{\Gamma \vdash \lambda x.e : \tau \to \mathsf{t}}\ \text{S-Lam} \qquad \frac{\Gamma \vdash v_1 : \mathsf{t} \to \mathsf{t}' \quad \Gamma \vdash v_2 : \mathsf{t}}{\Gamma \vdash v_1\ v_2 : \mathsf{t}'}\ \text{S-App}$$

$$\frac{\Gamma \vdash v : \mathsf{bool} \quad \forall i \in \{1,2\}.\Gamma \vdash e_i : \mathsf{t}}{\Gamma \vdash \textbf{if } v \textbf{ then } e_1 \textbf{ else } e_2 : \mathsf{t}}\ \text{S-If} \qquad \frac{\Gamma \vdash v : \tau}{\Gamma \vdash \mathsf{ref}\ v : \mathsf{ref}\ \tau}\ \text{S-Ref} \qquad \frac{\Gamma \vdash v : \mathsf{ref}\ \tau}{\Gamma \vdash !v : \tau}\ \text{S-Rd}$$

$$\frac{\Gamma \vdash v_1 : \mathsf{ref}\ \tau \quad \Gamma \vdash v_2 : \tau}{\Gamma \vdash v_1 := v_2 : \mathsf{unit}}\ \text{S-Wr} \qquad \frac{e_1 \neq v \quad \Gamma \vdash e_1 : \mathsf{t}_1 \quad \Gamma, x{:}\mathsf{t}_1 \vdash e_2 : \mathsf{t}_2}{\Gamma \vdash \textbf{let } x = e_1 \textbf{ in } e_2 : \mathsf{t}_2}\ \text{S-Bind}$$

$$\frac{\Gamma \vdash v : \mathsf{t}_1 \quad s = \forall \bar{\alpha}.\mathsf{t}_1 \quad \bar{\alpha} = FTV(\mathsf{t}_1) \setminus FTV(\Gamma) \quad \Gamma, x{:}s \vdash e : \mathsf{t}}{\Gamma \vdash \textbf{let } x = v \textbf{ in } e : \mathsf{t}}\ \text{S-Gen}$$

Figure 2. $\Gamma \vdash e : \mathsf{t}$: Surface typing for λ_{DST}, with important rules highlighted

v	::=	$\ldots \mid \lambda x{:}\tau.e$	annot. value
e	::=	$\ldots \mid x_{\bar{\tau}}$	annot. expr
t	::=	$\tau \mid x{:}\tau \to \mathsf{DST}\ t\ \phi \mid \overline{\eta_\kappa}.t_1 \to \mathsf{DST}\ t_2\ \phi$	type
σ	::=	$t \mid \forall \bar{\alpha}.t$	type scheme
κ	::=	$\star \mid E \mid x{:}t \Rightarrow \kappa \mid \alpha{::}\kappa \Rightarrow \kappa$	kinds
ϕ, ψ	::=	$\eta_\kappa \mid \mathsf{T} \mid \mathsf{F} \mid a_1 = a_2 \mid a_1 \in a_2 \mid \phi \wedge \psi$	formula
	|	$\phi \vee \psi \mid \neg\phi \mid \phi_1 \implies \phi_2 \mid \forall x{:}\sigma.\phi \mid \exists x{:}\sigma.\phi$	
	|	$\lambda x{:}t.\phi \mid \phi\ a \mid \Lambda\alpha{::}\kappa.\phi \mid \phi\ \psi \mid \forall \alpha{::}\kappa.\phi$	
a	::=	$v \mid sel\ a_1\ a_2 \mid upd\ a_1\ a_2\ a_3 \mid dom\ a \mid a_1\ op\ a_2$	logic term
Γ	=	$\cdot \mid x{:}\sigma \mid \Gamma, \Gamma'$	typ. env.

Figure 3. Annotated λ_{DST} terms and the full type language

is sugar for $\forall \overline{\eta :: \kappa}.t_1 \to \mathsf{DST}\ t_2\ \phi$. The display below shows the interpretation of λ_{DST} types in $\mathrm{F}^\star$.

Preventing the co-domain of a higher-order function from being directly dependent on its parameter may seem like an odd restriction. However, recall that all functions are impure, and consider a type of the form $\mathsf{f} : (t_1 \to \mathsf{DST}\ t_2\ \phi) \to \mathsf{DST}\ t\ \phi'$. If f were to appear free in t or ϕ', of what use would it be? An application of an impure function f in these types is clearly nonsensical—we cannot reduce impure functions at the type level. Instead, having ϕ' dependent on ϕ, the predicate transformer for f is much more useful.

Interpretation of λ_{DST} types in $\mathrm{F}^\star$

$[\![\star]\!]$	=	$\star$
$[\![E]\!]$	=	E
$[\![x{:}t \Rightarrow \kappa]\!]$	=	$x{:}[\![t]\!] \Rightarrow [\![\kappa]\!]$
$[\![\alpha{::}\kappa \Rightarrow \kappa']\!]$	=	$\alpha{::}[\![\kappa]\!] \Rightarrow [\![\kappa']\!]$
$[\![\phi]\!]$	=	$\ldots$ (a congruence of $[\![\cdot]\!]$ on the structure of formulas)
$[\![a]\!]$	=	a
$[\![\tau]\!]$	=	τ
$[\![x{:}\tau \to \mathsf{DST}\ t\ \phi]\!]$	=	$x{:}[\![\tau]\!] \to \mathsf{DST}\ [\![t]\!]\ [\![\phi]\!]$
$[\![\overline{\eta_\kappa}.t \to \mathsf{DST}\ t'\ \phi]\!]$	=	$\forall \overline{\eta{::}[\![\kappa]\!]}.[\![t]\!] \to \mathsf{DST}\ [\![t']\!]\ [\![\phi]\!]$
where		
$\mathsf{DST}\ \mathsf{t}\ \phi$	=	$\forall \psi{::}(\mathsf{t} \Rightarrow \mathsf{heap} \Rightarrow E).\mathsf{ST}\ (\phi\ \psi)\ \mathsf{t}\ \psi$
$\mathsf{ST}\ \phi\ \mathsf{t}\ \psi$	=	$h{:}\mathsf{heap}\{\phi\ h\} \to (x{:}\mathsf{t} * h'{:}\mathsf{heap}\{\psi\ x\ h'\})$

For simplicity and clarity, higher-order functions in λ_{DST} are written $\eta_\kappa.(t_1 \to \mathsf{DST}\ t_2\ \eta_\kappa) \to \mathsf{DST}\ t_3\ \phi$, and now both t_3 and ϕ may refer to the predicate transformer of the formal parameter.[3] Additionally, by providing a syntactic form for accompanying function-typed parameters with type variables representing their predicate transformers, we avoid introducing a general form of first-class polymorphism into the language—the only kind of first-class polymorphism allowed is the abstraction over predicate transformers in higher-order functions. Finally, in Section 4.5, we see how by systematically writing higher-order specifications according to this restricted syntax, we obtain our completeness result.

Formulas. We use the metavariables, ϕ or ψ, to stand for formulas and functions over formulas. Formulas here are drawn from a logic with the usual first-order connectives, equality over atoms a and set membership. We also include abstraction over atoms $\lambda x{:}\tau.\phi$ as well as abstraction over types and formulae, $\Lambda\alpha{::}\kappa.t$, and the corresponding application forms—the latter are convenient, particularly for higher-order specifications. For conciseness when writing formulae, we often drop type annotations on bound variables—these can easily be inferred.

The atoms a include values, but also terms from the select/update theory of maps—*sel* h a selects the contents of the map h at location a; *upd* h x a is a map that is identical to h, except at x where it contains a; *dom* h is a set representing the domain of the finite map h.

Contexts. Type contexts Γ, as usual, bind a set of unique names to their type schemes.

4.2 Inference of predicate transformers

Figure 4 presents the core of our inference algorithm, where we rely on two forms of judgment. Values, which by definition have no effect on the heap, are typed under the following form: the judgment $\Gamma \vdash v : t$ infers a type t for the annotated value v, given a context Γ. Expressions, however, may contain effects—indeed, for simplicity, we consider all expressions to be effectful. The judgment $\Gamma \vdash \{\phi\}\, e : t\, \{\psi\}$ infers a type t and a pre-condition ϕ, given a post-condition ψ and a context Γ. Furthermore, the judgment states that should e be executed with a heap satisfying ϕ and terminate, then the resulting heap and value returned will satisfy ψ—a standard Hoare logic interpretation. We present the latter judgment as a backwards-style weakest pre-condition calculus. We expect ψ, a predicate of kind $t \Rightarrow \mathsf{heap} \Rightarrow E$, as input to the judgment, and compute ϕ, a $\mathsf{heap} \Rightarrow E$ predicate.

The rules for variables come in two parts. The rule (Var) should be familiar. However, it applies only to variables with mono-types. Variables with poly-types are typed using the rule (Var-poly). The

[3] Sometimes in our examples, we still give a name to a function-typed formal parameter so that we can speak about it.

$$\boxed{\Gamma \vdash v : t} \quad \frac{\vdash \Gamma \ \mathbf{ok} \qquad \Gamma(x) = t}{\Gamma \vdash x : t}\ \text{Var} \qquad \frac{\text{fresh } \eta \qquad \Gamma, x{:}\tau \vdash \{\phi\}\, e : t\, \{\eta\}}{\Gamma \vdash \lambda x{:}\tau.e : x{:}\tau \to \mathsf{DST}\ t\ (\Lambda\eta.\phi)}\ \text{Lam}$$

$$\boxed{\Gamma \vdash \{\phi\}\, e : t\, \{\psi\}} \quad \frac{\vdash \Gamma \ \mathbf{ok} \qquad \Gamma(x) = \forall \bar{\alpha}.t' \qquad t'[\bar{\tau}/\bar{\alpha}] = t}{\Gamma \vdash \{\lambda h.\forall x{:}t.\phi\ x\ h\}\, x_{\bar{\tau}} : t\, \{\phi\}}\ \text{Var-poly}$$

$$\frac{\Gamma \vdash v : t}{\Gamma \vdash \{\phi\ v\}\, v : t\, \{\phi\}}\ \text{Ret} \qquad \frac{\begin{array}{c}\text{fresh } \eta \quad \Gamma \vdash \{\phi\}\, e_1 : t_1\, \{\eta\} \\ \Gamma, x{:}|t_1| \vdash \{\phi'\}\, e_2 : t_2\, \{\psi\} \qquad \varsigma = [\lambda x{:}t_1.\phi'/\eta]\end{array}}{\Gamma \vdash \{\varsigma\phi\}\ \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \lceil t_2 \rceil^{x:t_1}\, \{\psi\}}\ \text{Bind}$$

$$\frac{\Gamma \vdash v : t \qquad \sigma = \forall \bar{\alpha}.t \qquad \bar{\alpha} = FTV(t) \setminus FTV(\Gamma) \qquad \Gamma, x{:}\sigma \vdash \{\phi\}\, e : t'\, \{\psi\}}{\Gamma \vdash \{\phi[v/x]\}\ \mathbf{let}\ x = v\ \mathbf{in}\ e : t'[v/x]\, \{\psi\}}\ \text{Gen}$$

$$\frac{\Gamma \vdash v_1 : x{:}\tau_1 \to \mathsf{DST}\ t_2\ \phi \qquad \Gamma \vdash v_2 : \tau_1}{\Gamma \vdash \{\phi[v_2/x]\ \psi\}\, v_1\ v_2 : t_2[v_2/x]\, \{\psi\}}\ \text{App-}\tau \qquad \frac{\begin{array}{c}\Gamma \vdash v_1 : \overline{\eta_\kappa}.t_1 \to \mathsf{DST}\ t_2\ \phi' \\ \Gamma \vdash v_2 : t_1' \qquad |t_1'| = \varsigma\ |t_1| \qquad \varsigma = [\overline{\phi/\eta}]\end{array}}{\Gamma \vdash \{\varsigma\phi'\ \psi\}\, v_1\ v_2 : \varsigma t_2\, \{\psi\}}\ \text{App-}t$$

$$\frac{\begin{array}{c}\Gamma \vdash v : \mathsf{bool} \qquad \forall i \in \{1,2\}.\Gamma \vdash \{\phi_i\}\, e_i : t_i\, \{\psi\} \qquad t = |t_1| \sqcup |t_2| \\ \phi = \lambda h.(v = \mathsf{true} \implies \phi_1\ h) \wedge (v = \mathsf{false} \implies \phi_2\ h)\end{array}}{\Gamma \vdash \{\phi\}\ \mathbf{if}\ v\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 : t\, \{\psi\}}\ \text{If}$$

$$\frac{\Gamma \vdash v : \tau \qquad \phi = \lambda h.\forall y.\ y \notin dom\ h \implies \psi\ y\ (upd\ h\ y\ v)}{\Gamma \vdash \{\phi\}\, \mathsf{ref}\ v : \mathsf{ref}\ \tau\, \{\psi\}}\ \text{Ref}$$

$$\frac{\Gamma \vdash v : \mathsf{ref}\ \tau \qquad v' = sel\ h\ v}{\Gamma \vdash \{\lambda h.\psi\ v'\ h\}\, !v : \tau\, \{\psi\}}\ \text{Rd} \qquad \frac{\Gamma \vdash v_1 : \mathsf{ref}\ \tau \qquad \Gamma \vdash v_2 : \tau}{\Gamma \vdash \{\lambda h.\psi\ ()\ (upd\ h\ v_1\ v_2)\}\, v_1 := v_2 : \mathsf{unit}\, \{\psi\}}\ \text{Wr}$$

Figure 4. Inference of predicate transformers

A sequence of bound names: $A ::= \cdot \mid xA \mid \eta A$
We write: $\Lambda(xA).\phi$ to mean $\lambda x.\Lambda A.\phi$; and $\Lambda\eta A.\phi$ to mean $\Lambda\eta.\Lambda A.\phi$; and $\Lambda\cdot.\phi$ to mean ϕ

Normalizing a type

$$\begin{array}{lcll} |\tau|_A & = & \tau & \\ |x{:}\tau \to \mathsf{DST}\ t\ \psi|_A & = & x{:}\tau \to \mathsf{DST}\ |t|_{x\,A}\ ((\Lambda(x\,A).\psi)\ x\,A) & \\ |\overline{\eta_\kappa}.t \to \mathsf{DST}\ t'\ \psi|_A & = & \overline{\eta_\kappa}.t \to \mathsf{DST}\ |t'|_{\overline{\eta}\,A}\ ((\Lambda\overline{\eta}\,A.\psi)\ \overline{\eta}\,A) & \text{when} \quad \overline{\eta};\cdot \vdash t\ \mathbf{ok} \end{array}$$

Joining two types

$$\begin{array}{lcl} \tau \sqcup \tau & = & \tau \\ x{:}\tau \to \mathsf{DST}\ t_1\ \psi_1 \sqcup\ x{:}\tau \to \mathsf{DST}\ t_2\ \psi_2 & = & x{:}\tau \to \mathsf{DST}\ t_1 \sqcup t_2\ (\Lambda\psi.\psi_1\ \psi \wedge \psi_2\ \psi) \\ \overline{\eta_\kappa}.t_1 \to \mathsf{DST}\ t_1'\ \psi_1 \sqcup \overline{\eta_\kappa}.t_1 \to \mathsf{DST}\ t_2'\ \psi_2 & = & \overline{\eta_\kappa}.t_1 \to \mathsf{DST}\ t_1' \sqcup t_2'\ (\Lambda\psi.\psi_1\ \psi \wedge \psi_2\ \psi) \end{array}$$

Closing a type w.r.t a name (NB: when $\overline{\eta_\kappa}.t' \to \mathsf{DST}\ t\ \psi$ is uniform (§4.5) $FV(t') \subseteq \bar{\eta}_k$)

$$\begin{array}{lcll} \lceil \tau \rceil^{x:t_x} & = & \tau & \\ \lceil y{:}\tau \to \mathsf{DST}\ t\ \psi \rceil^{x:t_x} & = & y{:}\tau \to \mathsf{DST}\ \lceil t \rceil^{x:t_x}\ (\Lambda\eta.\lambda h.\forall x{:}t_x.\psi\ \eta\ h) & \textit{when } x \in FV(\psi) \\ \lceil y{:}\tau \to \mathsf{DST}\ t\ \psi \rceil^{x:t_x} & = & y{:}\tau \to \mathsf{DST}\ \lceil t \rceil^{x:t_x}\ \psi & \textit{otherwise} \\ \lceil \overline{\eta_\kappa}.t' \to \mathsf{DST}\ t\ \psi \rceil^{x:t_x} & = & \overline{\eta_\kappa}.t' \to \mathsf{DST}\ \lceil t \rceil^{x:t_x}\ (\Lambda\eta.\lambda h.\forall x{:}t_x.\psi\ \eta\ h) & \textit{when } x \in FV(\psi) \\ \lceil \overline{\eta_\kappa}.t' \to \mathsf{DST}\ t\ \psi \rceil^{x:t_x} & = & \overline{\eta_\kappa}.t' \to \mathsf{DST}\ \lceil t \rceil^{x:t_x}\ \psi & \textit{otherwise} \end{array}$$

Figure 5. Auxiliary functions for the inference of predicate transformers

reason for this distinction is that type instantiation in λ_{DST} is translated to explicit type applications in $\mathrm{F}^\star$, and there, type applications are an expression form. Note, the (Var-poly) rule relies on type instantiation annotations to compute the instantiation for the type scheme. In practice, the instantiations can be computed via unification, as usual.

Next, we show a rule for typing λ-abstractions. The rule (Lam) shows how predicate polymorphism allows us to make progress on predicate transformer inference. When typing the body of an effectful function, we compute its pre-condition symbolically, by using a fresh predicate variable η for its post-condition. The conclusion closes over η, generating a predicate transformer from ϕ that meets our well-formedness condition.

Of course, we also have standard rules for typing the unit and boolean constants—these are just as in the surface typing judgment and we leave them out here.

Turning to the expression forms, we have the rule (Ret), corresponding to the unit of the monad. It lifts a value into the monad. Its weakest pre-condition is a heap predicate which is just the post-condition on the returned value.

The rule (Bind) is for monadic sequencing. Typing the second sub-expression requires the type of the let-bound variable. However, typing the first sub-term requires a post-condition as input. So, we seem to be in a spot of bother. But, this is easily solved. We infer a pre-condition for e_1 using a symbolic post-condition η, yielding the type of x needed to type the second computation. Typing e_2 using the post-condition ψ, we obtain an intermediate heap predicate

ϕ', a pre-condition of the body e_2, and use this predicate (closed appropriately) to compute a pre-condition for the entire term by substituting it for instances of the variable η in ϕ.

There are two other subtleties in the (Bind) rule. The first is the use of the operator $|\cdot|$ in the second premise. This is an auxiliary function from types to types (defined in Figure 5), where $|t_1|$ is the type t_1 in a canonical form. The significance of this canonical form is made apparent in Section 4.5, but for now, suffice to say that putting types in canonical form makes it easy to compare two function types for equality. Intuitively, the canonical form of a dependent function $x{:}\tau \to \mathsf{DST}\ t\ \phi$ is $x{:}\tau \to \mathsf{DST}\ t'\ (\phi'\ x)$, where x does not appear free in ϕ', and t' is the canonical form of t. Similarly, a higher-order function type $\overline{\eta_\kappa}.t \to \mathsf{DST}\ t'\ (\psi\ \bar{\eta})$ is in canonical form if $\bar{\eta}$ is not free in ψ and t' is in canonical form.

The second subtlety in (Bind) is in the use of the operator $\lceil t_2 \rceil^{x:t_1}$ in the conclusion. This operator, defined in Figure 5, serves to close the type t_2 by introducing a universal quantifier for the let-bound name x which would otherwise escape its scope. Consider the program $\mathbf{let} x = e \mathbf{in}\ \lambda y{:}\tau.x$. The type of the λ-term in the body is $t_{body} = y{:}\tau \to \mathsf{DST}\ t_x\ (\Lambda\eta.\eta\ x)$. Naïvely assigning this type to the entire **let**-expression causes the **let**-bound variable x to escape its scope. The operator $\lceil t_{body} \rceil^{x:t_x}$ closes over the variable $x{:}t_x$, rewriting the type t_{body} to $y{:}\tau \to \mathsf{DST}\ t_x\ (\Lambda\eta.\lambda h.\forall x{:}t_x.\eta\ x\ h)$.

We present two rules for application. The (App-τ) rule describes first-order functions applied to τ-typed values, while the rule (App-t) addresses higher-order functions applied to function-typed values. The former employs substitution on the dependent argument, as is standard in a dependent type system, and draws on the predicate transformer ϕ to generate the weakest pre-condition.

The remaining rule, (App-t), is more complicated, as it must instantiate the predicate transformer variables $\overline{\eta}$ coupled with the higher-order function arguments. We accomplish this by relying on the premise $|t_1'| = \varsigma\ |t_1|$, which serves to unify the type t_1' (inferred for the function argument) with the type of the parameter t_1 (supplied as part of the higher-order function type). Given that $|t_1'|$ is in canonical form, unifying t_1 with it involves instantiating the predicate transformer variables $\bar{\eta}$ that appear free in t_1 with the concrete predicate transformers that appear in $|t_1'|$. The instantiation is the substitution ς. Having established a means of instantiating the $\overline{\eta}$ variables, we can apply the predicate transformer ϕ' to generate a weakest pre-condition for the application term.

A natural question arises as to whether such a unification is always possible—after all, we claim in Section 3 that any term typeable in the surface language can be typed in the main judgment. In particular, there are many ways to write E-kinded predicate transformers, like those that show up in the monadic codomain of t_1. Will they always present the requisite shape, such that unification with the computed type t_1' will succeed? Section 4.5 introduces a well-formedness condition on types in the context to ensure that higher-order function types are always maximally general in the predicate transformers η that model their arguments; and, that such functions are always dependent on the arguments in scope. These conditions are sufficient to ensure that the unification in (App-t) (for well surface-typed programs) always succeeds.

The next rule (If) computes a weakest pre-condition for the conditional form. We compute a weakest pre-condition and a type for each branch, but then have to compute a single type and pre-condition for the entire expression. Computing the pre-condition is easy: we simply take the conjunction of the computed pre-conditions for each branch, under the suitable guard. But computing a common type for the branches requires a bit more work. We define a join operator, $\cdot \sqcup \cdot$ on types, at the bottom of the Figure 4. Since we know that the source term is well-typed in the surface type system, we know that both branches have types that are structurally the same. Furthermore, by our well-formedness condition on environments, we prove that the types computed by our algorithm can always be put in a canonical form, where higher-order functions are maximally general in the predicate transformers of their function-typed arguments. Thus, in the definition of the join operator, we can assume that both types have syntactically identical domains, but may differ on the predicate transformers in their co-domains. The join then simply takes the conjunction of the predicate transformers.

Finally, the operations on references are typed using (Ref), (Rd), and (Wr), again with standard weakest pre-condition generation, with logical primitives drawn from the select/update theory of maps.

4.3 An example derivation

Before moving on, we revisit the next_fragment example from Section 2 to show a sample derivation. In this version, we rewrite the example in A-normal form—so as to more easily demonstrate the inference derivation—and desugar sequencing into the standard **let** binding. We also provide types for the functions is_empty_stream and stream_read, which are accompanied by the predicate transformers EmptyStreamTX and StreamReadTX. Finally, we make use of standard datatypes and tuples, although they are not presented as part of the core calculus.

Deriving a type for next_fragment

```
1  val is_empty_stream : s:stream → DST bool (EmptyStreamTX s)
2  val stream_read : s:stream → DST (fragment * stream) (StreamReadTX s)
3
4  let next_fragment = fun l →
5  {φ1 = λh. (λout_stream. φ2) (sel h out) h}
6  let out_stream = !out in
7  {φ2 = EmptyStreamTX out_stream (λb. φ3)}
8    let b = is_empty_stream out_stream in
9  {φ3 = λh. (b = true ⟹ φ31 h) ∧ (b = false ⟹ φ32 h)}
10   if b
11 {φ31 = ψ None}
12   then None
13 {φ32 = φ321}
14   else
15 {φ321 = StreamReadTX out_stream (λf.λrem. φ322)}
16     let f, rem = stream_read out_stream l in
17 {φ322 = λh. φ323 () (upd h out rem)}
18     let _ = out := rem in
19 {φ323 = ψ (Some f)}
20     Some f
21 {ψ}
```

Technically, the form of the term is $\mathbf{let}\ x = v\ \mathbf{in}\ e$, binding the function on l to the name next_fragment. We set that aside for the moment and focus on the function itself. It has the form $(\lambda l{:}\mathsf{int}.\ e)$, where the int annotation comes from the surface inference phase. Our first step is to apply the rule (Lam), which adds $l{:}\mathsf{int}$ to the environment, picks a fresh post-condition variable ψ, and computes a type and pre-condition for the body of the function.

The body of next_fragment (lines 5–21) is a sequence of **let** bindings that hoist the dereference of out and the computation of the **if** statement condition. Working backwards, we begin by typing the **if** statement, which—again working backwards—brings us to the end of the second branch on line 20. Line 20 is a value, and so we use the rule (Ret) to lift it into the DST monad, producing the pre-condition on line 19. Continuing to work backwards, we examine the next statement in this sequence of **let** statements, out := rem on line 18. Here, we apply the rule (Wr), producing a pre-condition reflecting the pending update to the heap.

The final statement in this branch (line 16) is a bit tricky. It calls for the rule (App-τ), which types stream_read as a function and out_stream as a stream—in both cases, by applying the rule (Var) to pull the variables from the context. We compute the pre-condition

by applying the predicate transformer StreamReadTX (extracted from the type of out_stream) to the supplied post-condition. We leave StreamReadTX abstract for simplicity, but in practice we would substitute out_stream for the dependent parameter name.

But our task on line 16 is only half done—we typed the function application but not the **let** binding itself. As premises, the (Bind) rule picks a fresh post-condition variable η, used to type the first expression, and then types the second premise using the supplied post-condition, ψ. There are two subtle issues here. The first lies in pushing the bound variable into the context. Our completeness result relies on a well-formedness condition on the context (we discuss this further when presenting the rule (App-t) and again in presenting our completeness result); we employ a normalization function $\lfloor\cdot\rfloor$ to ensure that $x{:}t_1$ meets our well-formedness criteria. The second subtlety arises in generating the pre-condition for the **let** statement itself. The bound variable may appear free in the pre-condition ϕ_1, and thus escape the binding. To handle this, we close over x prior to substituting for the post-condition variable η.

To illustrate this point, consider the variable f bound on line 16. Until this point, f had appeared free in both terms and formulae. The **let** statement binds it, but it still exists free in the formula ϕ_{322} that will be pushed backward. Thus, we must close over f in the pre-condition ϕ_{322}, transforming it into $(\lambda f.\phi_{322})$. Note that this new, closed term is exactly the kind of a post-condition—and so we apply StreamReadTX to it directly.

Returning our attention to the remaining branch of the **if** statement, we see it is a value, and thus easily lifted with the rule (Ret). Stepping back, we can now infer a type for the entire **if** statement. We have already generated the pre-conditions of both branches, ϕ_{31} and ϕ_{32}, respectively. b is a variable annotated with bool during the first inference phase; hence an application of the (Const) rule satisfies the first premise. Joining the two branch types is trivial—they are both of the type option fragment. Finally, the pre-condition is generated in traditional weakest pre-condition style, by predicating the respective branch constraints on the value of the condition b. The resulting pre-condition is shown on line 9.

From here, the remaining steps are simple. We again type the function application on line 8 with (App-τ), generating the pre-condition of the function application from the predicate transformer EmptyStreamTX, drawn from the function type, and closing over the variable b as we generate the pre-condition for the **let** binding. Typing the dereference on line 6 relies on the rule (Rd), which characterizes a dereference as a *select* operation on the value out in the heap.

As a result, the pre-condition of the function body is shown on line 5. At face value, it bears faint resemblance to the type we showed in Section 2, thanks to abstractions and applications inherent in the inference algorithm, as well as the opaque predicate transformers EmptyStreamTX and StreamReadTX. However, substituting each of the ϕ_i pre-conditions we computed and applying a standard β reduction will indeed yield the previous, simpler type.

4.4 Soundness

We show the soundness of our type inference algorithm via an elaboration to F*, making use of the soundness guarantee provided by the F* type system (Swamy et al. 2011a)—we sketch it here, relegating a formal presentation to the appendix.

The elaboration judgment is written $\Gamma \vdash v : t \rightsquigarrow \mathsf{v}$, for an λ_{DST} value v typed in a context Γ at type t (according to the typing judgment $\Gamma \vdash v : t$ that we have already seen), and then elaborated into an F* term v. Simultaneously, the elaboration judgment for expressions is written $\Gamma \vdash \{\phi\}\, e : t\, \{\psi\} \rightsquigarrow \mathsf{e}$, where an expression typed according to our main expression typing judgment $\Gamma \vdash \{\phi\}\, e : t\, \{\psi\}$ is then elaborated into an F* expression e.

$$
\begin{array}{lcl}
\lfloor \tau \rfloor & = & \tau \\
\lfloor x{:}\tau \to \mathsf{DST}\ t\ \phi \rfloor & = & x{:}\tau \to \lfloor t \rfloor \\
\lfloor \overline{\eta_\kappa}.x{:}t \to \mathsf{DST}\ t'\ \phi \rfloor & = & x{:}\lfloor t \rfloor \to \lfloor t' \rfloor \\
\lfloor \forall\, \bar{\alpha}.t \rfloor & = & \forall\, \bar{\alpha}.\lfloor t \rfloor \\
\lfloor \Gamma, x{:}\sigma \rfloor & = & \lfloor \Gamma \rfloor, x{:}\lfloor \sigma \rfloor
\end{array}
$$

Figure 6. Erasure of types and contexts to the surface level

The elaboration is mostly straightforward. **let**-bindings typed using (Bind) are simply applications of the bindST function from Section 2, with inferred types serving to instantiate the type parameters. Generalization via (Gen) introduces explicit type abstraction followed by a standard let binding in F*; instantiation is type application. Allocation, dereference, assignment, and assertions are simply sugar for underlying alloc, read, write, and assert functions, with the inference rules providing the type arguments. The rule for translating λ-abstractions is mostly a congruence, except for the addition of explicit type abstraction over the post-condition variable that is hidden within the DST monad.

To state our theorem, we start by defining a partial erasure function on types, to relate the DST types in our full system to surface types. Figure 6 defines a function $\lfloor\cdot\rfloor$ from types t to surface types t. This is entirely straightforward—we just drop all the predicate transformer and DST-parts of a type, and retain everything else.

THEOREM 1 (Soundness).
1. Given an environment Γ *such that* $\vdash S_{ST}; \Gamma_{ST}, [\![\Gamma]\!]$ ***ok***
and values v, v', *and* v, *and types* t, t
such that $\lfloor\Gamma\rfloor \vdash v : \mathsf{t} \rightsquigarrow v'$
and $\Gamma \vdash v' : t \rightsquigarrow \mathsf{v}$;
then $S_{ST}; \Gamma_{ST}, [\![\Gamma]\!], FTV(t) \setminus FTV(\Gamma); \cdot \vdash_{F^\star} \mathsf{v} : [\![t]\!]$.

2. Given an environment Γ *such that* $\vdash S_{ST}; \Gamma_{ST}, [\![\Gamma]\!]$ ***ok***
and expressions e, e' *and* e; *types* t, t; *and* ψ *such that*
$S_{ST}; \Gamma_{ST}, [\![\Gamma]\!], FTV(t, \psi) \setminus FTV(\Gamma) \vdash [\![\psi]\!] :: t \Rightarrow \mathit{heap} \Rightarrow E$,
and $\lfloor\Gamma\rfloor \vdash e : \mathsf{t} \rightsquigarrow e'$
and $\Gamma \vdash \{\phi\}\, e' : t\, \{\psi\} \rightsquigarrow \mathsf{e}$;
then $S_{ST}; \Gamma_{ST}, [\![\Gamma]\!], FTV(t, \psi) \setminus FTV(\Gamma); \cdot \vdash_{F^\star} \mathsf{e} : \mathit{ST}\ [\![\phi]\!]\ [\![t]\!]\ [\![\psi]\!]$.

In the statement of the theorem above, we write $S_{ST}; \Gamma_{ST}$ for the F* typing signature that gives kinds and types to the type and term constants (DST, bind *etc.*.) used in λ_{DST}. The translation of a context Γ to an F* context, denoted $[\![\Gamma]\!]$, is straightforward—we simply translate each type in Γ according to the interpretation of types given in Section 4.1. The typing judgment of F* is written $S; \Gamma; \cdot \vdash_{F^\star} \mathsf{e} : t$, meaning that given a signature S and a context Γ (and an empty affine compartment $\cdot$), the expression e has type t. We also rely on a standard auxiliary function FTV that extracts the free type variables from types.

As a consequence of soundness of inference, and soundness of F*, we can show that when a well-typed expression e is run in a context with its free variables in Γ substituted in a well-typed manner, then the expression e reduces without assertion failures.

4.5 Completeness

Type inference is complete in the sense that λ_{DST} programs typeable in our surface type system are also typeable by our algorithm, but yielding a more informative type. We obtain this result by imposing a uniformity condition on the typing context Γ. The main condition is that we require all higher-order functions in Γ to be maximally general in the predicate transformers of their function-typed arguments. For example, a type $\eta.(x{:}\mathsf{int} \to \mathsf{DST\ int}\ (\eta\ x)) \to \mathsf{DST\ int}\ (\phi\ \eta)$ is maximally general in its function-typed argument, since it accepts any $\mathsf{int} \to \mathsf{DST\ int}\ \psi$ function as an argument.

While such a condition may seem overly restrictive, we view higher-order interfaces that are maximally general in their predicate

arguments as good programming practice. Besides, using the types of ML and Haskell, or indeed the language of our surface type system, all higher-order interfaces are, by construction, maximally general in the specifications of their arguments. Thus, for most higher-order functions that programmers write today, these kinds of maximally general specifications do not seem too much of a burden.

Our uniformity condition on types, written $N;A \vdash \sigma$ **ok**, is defined in the display below. Here, the context A is a (possibly empty) sequence of predicate transformer and formal parameter names (η's and x's) that are bound at outer scopes in the type. The context N records a (possibly empty) sequence of predicate transformer names intended to model the behavior of the function-typed parameters of a higher-order function. A typing context Γ (written $\vdash \Gamma$ **ok**) is uniform if for every $x{:}\sigma$ in Γ, we can derive $\cdot;\cdot \vdash \sigma$ **ok**.

Uniformity of types

$$\frac{}{\cdot;A \vdash \tau \textbf{ ok}}\ \text{OK-1} \qquad \frac{\cdot;\cdot \vdash t \textbf{ ok} \quad \bar{\alpha} \subseteq FV(\lfloor t \rfloor)}{\cdot;\cdot \vdash \forall \bar{\alpha}.t \textbf{ ok}}\ \text{OK-2}$$

$$\frac{N;x\,A \vdash t \textbf{ ok} \quad x,A \notin FV(\phi) \quad \hat{\eta} \triangleright \phi}{\hat{\eta} \oplus N;A \vdash x{:}\tau \to \mathsf{DST}\ t\ (\phi\ x\ A) \textbf{ ok}}\ \text{OK-F1}$$

$$\frac{\overline{\eta};\cdot \vdash t_1 \textbf{ ok} \quad N;\overline{\eta}A \vdash t_2 \textbf{ ok} \quad \overline{\eta},A \notin FV(\phi) \quad \hat{\eta} \triangleright \phi}{\hat{\eta} \oplus N;A \vdash \overline{\eta_\kappa}.t_1 \to \mathsf{DST}\ t_2\ (\phi\ \overline{\eta}\ A) \textbf{ ok}}\ \text{OK-F2}$$

where

Pred. tx. vars	N	$::=$	$\cdot \mid \eta,N$
Opt. tx. var	$\hat{\eta}$	$::=$	$\cdot \mid \eta$
Un-cons cons	$\eta \oplus N$	$=$	η,N
Un-cons nil	$\cdot \oplus \cdot$	$=$	$\cdot$
Match any	$\cdot \triangleright \phi$		
Match exact	$\eta \triangleright \eta$		

We use several auxiliary functions in this judgment. First, $\hat{\eta}$ represents an optional transformer variable—it may be nothing ($\cdot$). We use an operator (a partial function $(\hat{\eta} \times N) \rightharpoonup N$) to split a possibly empty sequence of variables N into its head and tail. And, finally, we use a relation $\triangleright$, relating a $\hat{\eta}$ to a predicate ϕ, defined by the axioms (Match any) and (Match exact).

The first rule, (OK-1) is trivial: all small types are uniform. However, notice that this rule is only applicable in an empty N context. The significance of this will become clear shortly.

The rule (OK-2) requires that in any type scheme $\forall\alpha.t$, all the type variables α must appear free in the partial erasure of t. This is to ensure that the type instantiations computed by the surface type system can be carried over to our core type system, via the rule (S-var-poly).

The rule (OK-F1) is for function types with a small domain. We check the co-domain t in a context extended with the name of the formal parameter, and in the conclusion, we expect the predicate transformer to be a function ϕ of all the formal parameters in scope, i.e., $x\,A$. This structure of a type corresponds to the canonical form of types described in Section 4.2. Finally, in the last premise, we require the head of the sequence of predicate variables $\hat{\eta}$ to match the predicate ϕ. The significance of this will be clearer in conjunction with the next and last rule (OK-F2).

When checking a higher-order function type $t = \overline{\eta_\kappa}.t_1 \to \mathsf{DST}\ t_2\ \psi$ in (OK-F2), we check the domain type t_1 in a context with the abstracted predicate variables $\overline{\eta}$ as the N-context. Revisiting the last premise of (OK-F1), let us suppose t_1 were of the form $x{:}\tau \to \mathsf{DST}\ t\ (\phi\ x)$. In order to complete the derivation, we require ϕ to be the first predicate variable η abstracted by t, i.e., t would be maximally general in the specification of t_1.

Continuing through the premises of (OK-F2): the second premise resembles the first premise of (OK-F1)—we check the co-domain in a context including the $\overline{\eta}$ among the formal parameters. The third and fourth premises also resemble (OK-F1). In the conclusion, again, we require the predicate transformer to be a function of all the formal parameters and predicate variables in scope.

With these tools, we can present the following completeness theorem, where we write $\vdash \Gamma$ **ok** when each type bound in Γ is uniform.

THEOREM 2 (Completeness).

(1) *Given an environment* Γ *such that* $\vdash \Gamma$ ***ok*** *and values* v,v' *and type* $\mathtt{t}$ *such that* $\lfloor\Gamma\rfloor \vdash v : \mathtt{t} \rightsquigarrow v'$*; then, given a fixed source of fresh names, there exists a unique* t *such that* $\Gamma \vdash v' : t$ *and* $\lfloor t \rfloor = \mathtt{t}$.

(2) *Simultaneously, given an environment* Γ *such that* $\vdash \Gamma$ ***ok*** *and expressions* e,e' *and type* $\mathtt{t}$ *such that* $\lfloor\Gamma\rfloor \vdash e : \mathtt{t} \rightsquigarrow e'$*; then, for any* ψ*, given a fixed source of fresh names, there exists unique* t,ϕ *such that* $\Gamma \vdash \{\phi\}\,e' : t\,\{\psi\}$ *and* $\lfloor t \rfloor = \mathtt{t}$.

Theorem 2 (Completeness) states that if there exists a surface typing derivation for e in an erased, uniform context Γ, then either a pure λDST typing derivation exists, if e is a value, or a stateful typing derivation can be constructed. The proof proceeds by mutual induction on the structure of the surface-typing derivations in the premises.

4.6 Generating first-order verification conditions

The judgement $\Gamma \vdash \{\phi\}\,e : t\,\{\psi\}$ infers a stateful pre-condition ϕ for the program e and post-condition ψ: if ϕ holds on an initial heap, then ψ will hold on the value and heap resulting from the execution of e. The underlying F* type system is powerful, and it is natural to ask whether the inference algorithm—which may instantiate higher-order type arguments at higher-order function calls—always generates verification conditions that F* can discharge via an SMT solver with first-order theories. This is indeed the case, provided higher-order specifications in the context do not directly make use of higher-order logic. In other words, despite the heavy use of predicate polymorphism in our type inference algorithm, unless a programmer explicitly uses a higher-order assertion, our verification condition generator never introduces a higher-order proof obligation. We sketch the proof here.

The syntax of λDST allows higher-kinded quantification in formulae, allowing programmers to write higher-order assertions. This is undeniably useful, e.g., we have used our verification condition generator to verify higher-order JavaScript programs (Swamy et al. 2012a), where specifications make pervasive use of existential quantification over predicate transformers. While such specifications are useful, automatically discharging their proofs is a significant challenge. Indeed, we devised a novel strategy for automatically proving the stylized higher-order formulas that arise in the context of JavaScript verification using Z3. But, automated proving for arbitrary higher-order formulas is intractable. While the sacrifice of higher-order quantification does limit expressivity, we had little trouble ensuring that all our benchmark programs, presented in Section 6, met this restriction.

Thus, our formal development relies on a well-formedness condition $\vdash_{FO} \phi$ **ok** on formulae that simply restricts quantification to $\star$-kinded types, i.e., it proscribes use of the higher-order quantifier $\forall\alpha{::}\kappa.\phi$, for all non-base kinds κ. Of course, Λ-abstraction over higher kinds is still permitted (and necessary, just to use the DST monad). We lift the $\vdash_{FO} \phi$ **ok** to types and contexts in the obvious way. Note that our soundness and completeness results hold regardless of this restriction—if a programmer chooses to write higher-order assertions (as we did for JavaScript) our verification condition

generator still generates well-typed F* programs with higher-order proof obligations.

Before presenting the full theorem, we first show a key supporting lemma establishing that, given a well-formed context and a well-formed post-condition, our inference algorithm computes a well-formed pre-condition.

LEMMA 1 (Well-formed Verification Conditions). *For all contexts* Γ *such that* $\vdash_{FO} \Gamma$ **ok**; *formulae* ψ *such that* $\vdash_{FO} \psi$ **ok**; *expressions* e; *types* t; *and pre-conditions* ϕ; *if* $\Gamma \vdash \{\phi\}\, e : t\, \{\psi\}$, *then* $\vdash_{FO} \phi$ **ok**.

Intuitively, Lemma 1 (Well-formed Verification Conditions) shows that the inference algorithm does not introduce higher-kinded quantification where none previously existed. The proof proceeds by induction on the structure of the typing relation. We can now proceed in stating our final theorem.

THEOREM 3 (First-order Verification Conditions). *For all contexts* Γ *such that* $\vdash_{FO} \Gamma$ **ok**; *formulae* ψ *such that* $\vdash_{FO} \psi$ **ok** *and* $FTV(\psi) = \varnothing$; *expressions* e; *types* t; *pre-conditions* ϕ; *and heaps* h; *if* $\Gamma \vdash \{\phi\}\, e : t\, \{\psi\}$, *then* $\phi\ h$ *is representable in a first-order theory.*

First, consider the verification condition generated by the inference algorithm. The judgment $\Gamma \vdash \{\phi\}\, e : t\, \{\psi\}$ produces a pre-condition ϕ that is well kinded in Γ at kind heap $\Rightarrow E$, and, thanks to Lemma 1 (Well-formed Verification Conditions), we know that $\vdash_{FO} \phi$ **ok**. Applying the pre-condition to a given initial heap h yields the verification condition ($\phi\ h$), which is well-kinded at kind E. Now, since formulae have a standard beta reduction rule and there are no fixed points or uninterpreted higher-order predicates, we can reduce these to a normal form—F* guarantees semantic equivalence under β-reduction.

Many of the syntactic operators that comprise formulae—including equality, implication, and quantification over $\star$-kinded types—can be trivially represented using first-order theories. In fact, only η variables and higher-kinded quantification pose potential difficulty—both may lead to instances where a higher-kinded variable is applied to a type, yielding a term that is both in normal form and not reducible. However, as Γ binds no η variables, the case ($\eta\ \phi$) is ruled out by kinding, and we rely on the well-formedness condition to restrict quantification to $\star$-kinded types. The remaining cases, like ($\Lambda\alpha{::}\kappa.t\ \phi$), are also ruled out, since these are reducible.

5. A higher-order, recursive example

So far, we have only considered recursion-free code. In this section, we present a higher-order, recursive example, iter. Giving specification to recursive functions requires the use of an inductive invariant. Callers of such functions will have to manually supply this invariant. However, our inference algorithm can still be used to compute instantiations for the predicate transformers that model function-typed arguments. Note, we use curried functions in this example—although our formal system leaves out currying, our implementation supports it.

The implementation of iter (shown overleaf) is straightforward—it recurses over a list l, applying a stateful function f to each element. The trickier part is writing its specification in a style that allows us to infer all but the inductive invariant needed to reason about the recursion. Lines 1–7 above show the specification of iter. Its shape is $\forall \overline{v{::}\kappa}.\eta.(y{:}\tau \to \mathsf{DST}\ t\ (\eta\ y)) \to t'$. It is polymorphic in the type α of list elements and Φ, the inductive invariant. At line 2, we see the function argument f, preceded, as usual, by a polymorphic predicate transformer variable η, which abstracts over post-conditions to generate the weakest pre-condition of f. At line 3, we see the list argument l and on line 4 the monadic return type, with a predicate transformer Ψ ensuring, among other things, that Φ is indeed inductive and that Φ l be true initially—we discuss it forthwith.

Type and implementation of iter

```
1  val iter:∀α::⋆,Φ::list α⇒ heap⇒E.
2        η. f:(x:α →DST unit (η x))
3     →l:list α
4     →DST unit Ψ
5    where Ψ = (Λψ.λh. (∀h1. Φ [] h1 ⟹ ψ () h1)
6              ∧ (∀h2 hd tl. Φ (hd::tl) h2 ⟹ η (λ(). Φ tl) hd h2)
7              ∧ Φ l h)
8  let rec iter f l = match l with
9   | [] →()
10  | hd::tl →bind α α (η hd) Ψ (f hd) (λ_. iter α Φ f tl)
```

Examining the form of Ψ, note that it has the kind $(\alpha \Rightarrow \mathsf{heap} \Rightarrow E) \Rightarrow \mathsf{heap} \Rightarrow E$. Turning now to the body of the transformer, we have three clauses on lines 5–7, respectively. The first clause, $\forall$h1. Φ [] h1 $\Longrightarrow$ ψ () h1, ensures that the post-condition holds when the invariant holds on the base case of the recursion—it is used to prove the post-condition ψ when we exit the recursion at line 9. The second conjunct is a bit more subtle. It is most easily understood from the perspective of a weakest pre-condition computation. If we want the post-condition ψ to hold at the exit of the function, then it must be established by the recursive call to iter at line 10. So, at the recursive call, we instantiate the post-condition of iter with ψ. For this call to succeed, the pre-condition requires the inductive invariant to hold on the tail of the list, *i.e.*, we require Φ tl. This pre-condition must be established by the call to f, and since f is polymorphic in its post-condition, we can simply instantiate this to ($\lambda().\ \Phi$ tl). The pre-condition of the call to f is then $\eta\ (\lambda().\ \Phi$ tl) hd. This gives us our second clause: we need to be able to derive the pre-condition to f from the invariant Φ hd::tl, which we know to be valid at the entry of the loop, as required by the third clause.

Note, the reasoning behind the type of iter is relatively mechanical. In the future, we hope to be able to make this precise, and automate inference, with inductive invariants, for higher-order libraries. Of course, actually instantiating the invariant at the call site requires some ingenuity, as the next example illustrates.

Now, consider a simple client of iter, a function that iterates over a list of positive integers poslist, repeatedly dividing the contents of a ref cell by the elements of the list.

```
let x = ref 1000 in
let divx i = x := x / i in (* inferred type *)
let _ = iter _ (λl h. (sel h x ≥ 0) ∧ AllGT0 l) divx poslist in
assert (λ h. sel h x ≥ 0)
```

The type inferred for divx, under a suitable model for the division operator, is:

i:int → DST unit $(\Lambda\psi.\lambda h.\mathsf{i} \neq 0 \wedge \psi\ ()\ (upd\ h\ \mathsf{x}\ (sel\ h\ \mathsf{x}/\mathsf{i})))$.

It requires its argument i to be non-zero, and reflects the update of x in the heap. Applying the iterator to divx and a list of integers and then asserting that the contents of the x is non-negative requires introducing an inductive invariant. Here, the invariant we choose is that the heap at x is always non-negative, and using AllGT0 l, we aim to prove that every element of the list is greater than 0. Discovering and annotating the invariant is, in general, hard. But, at least our type inference algorithm takes care of the boring details—explicit type instantiations for the other type parameters are unnecessary.

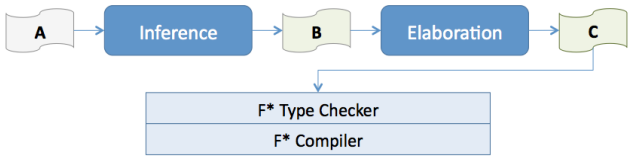

Figure 7. A fragment of the F* compiler front end, wherein the inference and elaboration passes implement type inference for λ_{DST}. Intermediate results are abstract syntax trees in the following form: (A) undecorated λ_{DST}; (B) typed λ_{DST}; (C) monadic F*.

6. Experimental evaluation

We implement our algorithm as two front-end passes in the F* compiler. The first pass infers stateful types for an undecorated AST, producing a new AST decorated with the inferred types. The second pass consumes the type decorations and transforms the AST via monadic elaboration. As part of the elaboration, polymorphic functions are instantiated at their call sites with the types inferred during the first phase. The elaborated AST is then passed to the underlying, unmodified F* type checker.

λ_{DST} is a fragment of F* and ultimately compiled by the F* compiler. Naturally, it may be useful to allow stateful λ_{DST} modules to interact with other F* modules by calling pure functions defined therein. Our inference algorithm supports this by automatically lifting external function calls into the Dijkstra monad. The listing below describes the types at which refined F* functions are lifted.

Lifting refinement types in F* to interop. with monadic code

$$
\begin{array}{lcl}
t_1 \to t_2 & \rightsquigarrow & t_1 \to \mathsf{DST}\ t_2\ (\Lambda\psi.\lambda h.\forall y{:}t_2.\ \psi\ y\ h) \\
t_1 \to y{:}t_2\{\phi\} & \rightsquigarrow & t_1 \to \mathsf{DST}\ t_2\ (\Lambda\psi.\lambda h.\forall y{:}t_2.\ \phi \implies \psi\ y\ h) \\
x{:}t_1\{\phi\} \to t_2 & \rightsquigarrow & x{:}t_1 \to \mathsf{DST}\ t_2\ (\Lambda\psi.\lambda h.\phi \wedge \forall y{:}t_2.\ \psi\ y\ h)
\end{array}
$$

Pure, unrefined external functions are lifted in a similar manner to pure values—there are no effects on the heap, and thus the weakest pre-condition is simply a given post-condition that holds for any value the function may return. External functions may also have general refinement types, unlike the refinement types that appear in λ_{DST} programs. (In λ_{DST}, pre- and post-conditions are the only refinements.) However, there is a natural translation from refinements on functions into the Dijkstra monad. Intuitively, refinements on the domain are constraints that must be discharged prior to each function call site; thus, adding each domain refinement as a conjunct in the pre-condition achieves the same result. Similarly, a function that type checks in F* with a refinement on the range is guaranteed to return a value such that the refinement formula holds. Therefore, rather than requiring that the post-condition hold over all possible return values, as we did for unrefined function types, we quantify over all values that satisfy the refinement; the post-condition must only hold on values that satisfy the range refinement.

We have evaluated our implementation on a suite of thirty small benchmark programs ranging in complexity from imperative and straight-line to higher-order and recursive; the example functions presented in previous sections are representative of our selection. In total, we have verified approximately five hundred lines of code.

We find our initial experiments encouraging, particularly because we have already used the Dijkstra monad and our verification condition generator in an extensive case study for the automated verification of JavaScript programs (Swamy et al. 2012a). These JavaScript programs are a particularly challenging benchmarks, since they make pervasive use of a dynamically typed higher-order store. We anticipate improving our implementation and scaling it to infer types for larger F* developments currently in progress, and to simplify existing F* code.

7. Related work and conclusions

There has been much success in applying SMT solvers to program verification. Tools based on frameworks like Boogie (Leino and Rümmer 2010) or Why (Filliâtre and Marché 2007)) have been used to verify significant developments, *e.g.*, the Vcc verifier (Cohen et al. 2009) has been used to verify tens of thousands of lines of C code. However, a limitation of these tools is that they generally apply only to first-order programs. There are some tools based on Why that begin to scale to higher-order programs, *e.g.*, Who (Kanig and Filliâtre 2009), although this is limited in that it pre-supposes a particular no-aliasing discipline.

In contrast, approaches based on dependent types such as HTT (Nanevski et al. 2008) naturally handle stateful, higher-order programs. But, the tool based on this approach, Ynot, is a library on top of the Coq interactive proof assistant (Bertot and Castéran 2004) and, despite advances (Chlipala et al. 2009), still requires considerable manual effort both in the computation of verification conditions as well as in discharging proofs. More recently, Charguéraud (2011) applies his characteristic formulae to higher-order imperative programs. This approach translates the semantics of a higher-order stateful program in to a formula in higher-order logic, which can then be attacked in an interactive manner in Coq. In contrast, our approach aspires to better automation via automated computation of verification conditions in a style that can be handled by an SMT solver.

Also related is other work on F* and related languages. Borgström et al. (2009) present an application of the Hoare monad within the F7 programming language. However, this work does not address type inference, does not give a weakest pre-condition calculus, and does not handle higher-order code. Also related is FX (Borgström et al. 2011), which uses substructural state on top of a Hoare-like monad to model local state; but, again, that work does not handle inference or higher-orderness. Bhargavan et al. (2010) provide a means of verifying higher-order programs in a language with only first-order refinement types by adopting certain syntactic conventions. Again, they do not attempt inference nor do they handle state.

Our approach to inferring instantiations for higher-rank types resembles work on HMF (Leijen 2008). HMF requires function parameters with polymorphic types to be annotated, a similar restriction to ours. However, unlike HMF, we have dependent, higher-kinded, and monadic types, but only predicative instantiation—instantiation of type variables in HMF may be impredicative.

Another line of work on verifying higher-order programs is via higher-order model checking (Kobayashi et al. 2011) or via liquid types (Rondon et al. 2008). These approaches aim to be automated by discovering invariants using a variety of means, including predicate abstraction and counter-example guided abstraction refinement (Clarke et al. 2003). However, the type systems used in these works do not handle state, and they are relatively fragile, *e.g.*, they are sensitive to argument order when trying to verify closure passing programs, even when these are pure. Nevertheless, aiming to combine model checking or abstract interpretation based approaches with our work is likely to pay dividends, particularly in the inference of invariants.

Conclusions. The main observation of this paper is that structuring specifications for higher-order stateful programs using predicate transformers facilitates easy type inference. We identify a new variant of the Hoare state monad, which we call the Dijkstra state monad, that makes uniform use of predicate transformers to define the weakest pre-condition of a computation. We develop a type inference algorithm for an ML-like programming language based on the Dijkstra monad, and prove our algorithm sound and complete. Further, when specifications are written carefully, the verification conditions we compute are in a form amenable to automated prov-

ing by SMT solvers. Our initial experience verifying programs indicates that our advocated style is lightweight, and can be used to verify many interesting programs with no annotations, save for loop invariants.

References

Y. Bertot and P. Castéran. *Coq'Art: Interactive Theorem Proving and Program Development.* Springer Verlag, 2004.

K. Bhargavan, C. Fournet, and N. Guts. Typechecking higher-order security libraries. In *APLAS*, pages 47–62, 2010.

J. Borgström, A. Gordon, and R. Pucella. Roles, stacks, histories: A triple for hoare. Technical Report MSR-TR-2009-97, MSR, 2009.

J. Borgström, J. Chen, and N. Swamy. Verifying stateful programs with substructural state and hoare types. In *PLPV '11*, Jan. 2011.

A. Charguéraud. Characteristic formulae for the verification of imperative programs. In *ICFP*, 2011.

A. Chlipala, G. Malecha, G. Morrisett, A. Shinnar, and R. Wisnesky. Effective interactive proofs for higher-order imperative programs. In *ICFP*, 2009.

E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50, 2003.

E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. Vcc: A practical system for verifying concurrent c. In *TPHOLS*, 2009.

L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.

E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18:453–457, August 1975.

J. C. Filliâtre and C. Marché. The why/krakatoa/caduceus platform for deductive program verification. CAV'07, 2007.

C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

J. Kanig and J.-C. Filliâtre. Who: a verifier for effectful higher-order programs. In *ML*, ML '09, New York, NY, USA, 2009. ACM.

N. Kobayashi, R. Sato, and H. Unno. Predicate abstraction and cegar for higher-order model checking. In *PLDI*, pages 222–233, 2011.

D. Leijen. HMF: simple type inference for first-class polymorphism. ICFP, New York, NY, USA, 2008. ACM.

K. Leino and P. Rümmer. A Polymorphic Intermediate Verification Language: Design and Logical Encoding. In *TACAS*. 2010.

J. Mccarthy. Towards a mathematical science of computation. In *IFIP Congress*, pages 21–28. North-Holland, 1962.

A. Nanevski, G. Morrisett, and L. Birkedal. Hoare type theory, polymorphism and separation. *J. Funct. Program.*, 18, September 2008.

P. M. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *PLDI*, pages 159–169, 2008.

P.-Y. Strub, N. Swamy, C. Fournet, and J. Chen. Self-certification: Bootstrapping certified typecheckers in F* with Coq. In *POPL*, 2012.

N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan, and J. Yang. Secure distributed programming with value-dependent types. In *ICFP*, 2011a.

N. Swamy, N. Guts, D. Leijen, and M. Hicks. Lightweight monadic programming in ml. In *ICFP*, 2011b.

N. Swamy, J. Weinberger, J. Chen, B. Livshits, and C. Schlesinger. Monadic refinement types for verifying JavaScript programs. Technical report, MSR, 2012a. URL `http://research.microsoft.com/fstar`.

N. Swamy, J. Weinberger, C. Schlesinger, J. Chen, and B. Livshits. Towards javascript verification with the dijkstra state monad. Technical report, MSR, 2012b.

P. Wadler. The essence of functional programming. In *POPL*, 1992.

J. Wells. Typability and type checking in the second-order lambda;-calculus are equivalent and undecidable. In *LICS*, 1994.

$$
\begin{array}{lcl}
\sigma & ::= & \forall \bar{\alpha}.t \mid t \\
t & ::= & T \mid x{:}\mathsf{dyn}\{\phi\} \mid \overline{x{:}t} \to \mathsf{DST}\ \mathsf{dyn}\ \phi \mid \overline{x{:}t} \to t' \\
\Gamma & ::= & \cdot \mid x{:}\sigma \mid C{:}\sigma \mid \Gamma,\Gamma \\
\phi & ::= & \ldots \text{ (F}^\star\text{'s refinement logic)}
\end{array}
$$

$$
\frac{\Gamma(x) \geq t}{\Gamma \vdash x : t} \qquad \frac{\Gamma(C) \geq \overline{x{:}t} \to t \quad \forall i.\Gamma \vdash v_i : t_i}{\Gamma \vdash C\ \bar{v} : t[\overline{v/x}]}
$$

$$
\frac{\Gamma, \overline{x{:}\mathsf{dyn}} \vdash e \uparrow \phi}{\Gamma \vdash \lambda \bar{x}.e : \overline{x{:}\mathsf{dyn}} \to \mathsf{DST}\ \mathsf{dyn}\ \phi} \qquad \frac{\Gamma \vdash v : x{:}\mathsf{dyn}\{\phi\}}{\Gamma \vdash v \uparrow \Lambda\psi.\lambda h.\phi[v/x] \Longrightarrow \psi\ v\ h}
$$

$$
\frac{\Gamma \vdash v : \overline{x{:}t} \to \mathsf{DST}\ \mathsf{dyn}\ \phi \qquad \forall i.\Gamma \vdash v_i : t_i}{\Gamma \vdash v\ \bar{v} \uparrow \phi[\overline{v/x}]}
$$

$$
\frac{\begin{array}{cc} \Gamma \vdash v : t & \Gamma(C) \geq \overline{x{:}t} \to t' \\ compat(t,t') & \Gamma, \overline{x{:}t} \vdash e_1 \uparrow \phi_1 \qquad \Gamma \vdash e_2 \uparrow \phi_2 \end{array}}{\begin{array}{l} \Gamma \vdash \textbf{match}\ v\ \textbf{with}\ C\ \bar{x} \to e_1\ \textbf{else}\ e_2 \uparrow \Lambda\psi.\lambda h.\ (\forall \bar{x}.v = C\bar{x} \Longrightarrow \phi_1\ \psi\ h) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \&\&\ (\forall \bar{x}.v \neq C\bar{x}) \Longrightarrow \phi_2\ \psi\ h \end{array}}
$$

$$
\frac{\Gamma \vdash e_1 \uparrow \phi_1 \quad \Gamma, x{:}\mathsf{dyn} \vdash e_2 \uparrow \phi_2}{\Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \uparrow \Lambda\psi.\phi_1(\lambda x.\phi_2\psi)}
$$

where:
$\forall \bar{\alpha}.t \geq t[\bar{t}/\bar{\alpha}]$ and $t \geq t$
$compat(t,t)$ and $compat(t, x{:}t\{\phi\})$
$\mathsf{dyn} \equiv x{:}\mathsf{dyn}\{\mathsf{True}\}$

Figure 8. $\Gamma \vdash v : t$ and $\Gamma \vdash e \uparrow \phi$—type inference for JS*

A. Specializing for JS*

A specialized version of the type system in this paper is presented in a conference-length paper by Swamy et al. (2012b). In this section, we show the specialized version and relate it to the full system described in this paper.

Figure 8 shows the λ_{DST} type inference algorithm from Section 4 specialized for JS*, a dynamically-typed language encoding a fragment of JavaScript in F*. Much of the syntax of JS*is familiar from Section 4.1. For brevity, we let T stand for type constants (like int, bool, etc.) in types t. Refinement types—written $x{:}\mathsf{dyn}\{\phi\}$—are already in F* and fit naturally with the Dijkstra monad. Finally, type constructors C are pure functions that may appear bound in the context.

The judgment $\Gamma \vdash v : t$ on values remains unchanged in form. The first two rules depict a nondeterministic form of the (Var-poly) rule from Figure 4, where the type $\overline{x{:}t} \to t$ is syntactic sugar for a pure function with no side effects. The rule for functions directly corresponds to the (Lam) rule from Figure 4.

We abbreviate the judgment on expressions, $\Gamma \vdash \{\phi\}\, e : t\, \{\psi\}$, as $\Gamma \vdash e \uparrow \phi'$, where $\phi' = \Lambda\psi.\phi$. The first rule corresponds to a monadic unit and follows from the (Return) rule in Figure 4, except that, if the value being returned has a refined type, then the weakest pre-condition is guarded by this refinement. The rule for functions is a straightforward specialization of the (App-τ) rule. Note that JS*function parameters are always of type dyn, and hence the rule (App-t) is not applicable. The conditional rule follows directly from (If), and the final rule—for **let** expressions—specializes (Gen) in that JS*bindings do not support polymorphism, and thus $FTV(t) \setminus FTV(\Gamma) = \{\}$.

B. The λ_{DST} Calculus

This section presents the judgments in the λ_{DST} calculus that we omitted from the main body of the paper. In particular, we show the elaboration of surface terms into an annotated intermediate form and also show the inference of predicate transformers combined with an elaboration of λ_{DST} programs to $\text{F}^\star$ (Section B.2).

B.1 Surface typing for λ_{DST}

The main inference algorithm assumes that a source λ_{DST} program has already been typed using the surface level judgment and converted into an annotated form e, where every λ-bound variable is annotated with its type, and every use of a variable with a polymorphic type is annotated with its type instantiations. The main body of the paper elides the elaboration of surface level terms into this intermediate, annotated form.

The syntax of surface terms was shown in Figure 1 (Section 3) and the syntax of our intermediate annotated language, and the full syntax of λ_{DST} types was shown in Figure 3 (Section 4.1).

We present the surface elaboration judgment below, a simple extension of the surface type system of Figure 2.

$\Gamma \vdash e : \mathsf{t} \rightsquigarrow e$: Surface typing and elaboration for λ_{DST}, with important rules highlighted

$$\frac{}{\Gamma \vdash () : \mathsf{unit} \rightsquigarrow ()}\ \text{S-1} \qquad \frac{v \in \{\mathsf{true}, \mathsf{false}\}}{\Gamma \vdash v : \mathsf{bool} \rightsquigarrow v}\ \text{S-b} \qquad \frac{\Gamma(x) = \forall \bar{\alpha}.\mathsf{t}}{\Gamma \vdash x : \mathsf{t}[\bar{\tau}/\bar{\alpha}] \rightsquigarrow x_{\bar{\tau}}}\ \text{S-x}$$

$$\frac{\Gamma, x{:}\tau \vdash e : \mathsf{t} \rightsquigarrow e'}{\Gamma \vdash \lambda x.e : \tau \to \mathsf{t} \rightsquigarrow \lambda x{:}\tau.e'}\ \text{S-Lam} \qquad \frac{\Gamma \vdash v_1 : \mathsf{t} \to \mathsf{t}' \rightsquigarrow v_1' \quad \Gamma \vdash v_2 : \mathsf{t} \rightsquigarrow v_2'}{\Gamma \vdash v_1\ v_2 : \mathsf{t}' \rightsquigarrow v_1'\ v_2'}\ \text{S-App}$$

$$\frac{\Gamma \vdash v : \mathsf{bool} \rightsquigarrow v' \qquad \forall i \in \{1,2\}.\Gamma \vdash e_i : \mathsf{t} \rightsquigarrow e_i'}{\Gamma \vdash \textbf{if}\ v\ \textbf{then}\ e_1\ \textbf{else}\ e_2 : \mathsf{t} \rightsquigarrow \textbf{if}\ v'\ \textbf{then}\ e_1'\ \textbf{else}\ e_2'}\ \text{S-If} \qquad \frac{\Gamma \vdash v : \tau \rightsquigarrow v'}{\Gamma \vdash \mathsf{ref}\ v : \mathsf{ref}\ \tau \rightsquigarrow \mathsf{ref}\ v'}\ \text{S-Ref}$$

$$\frac{\Gamma \vdash v : \mathsf{ref}\ \tau \rightsquigarrow v'}{\Gamma \vdash !v : \tau \rightsquigarrow !v'}\ \text{S-Rd} \qquad \frac{\Gamma \vdash v_1 : \mathsf{ref}\ \tau \rightsquigarrow v_1' \quad \Gamma \vdash v_2 : \tau \rightsquigarrow v_2'}{\Gamma \vdash v_1 := v_2 : \mathsf{unit} \rightsquigarrow v_1' := v_2'}\ \text{S-Wr}$$

$$\frac{e_1 \neq v \qquad \Gamma \vdash e_1 : \mathsf{t}_1 \rightsquigarrow e_1' \qquad \Gamma, x{:}\mathsf{t}_1 \vdash e_2 : \mathsf{t}_2 \rightsquigarrow e_2'}{\Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : \mathsf{t}_2 \rightsquigarrow \textbf{let}\ x = e_1'\ \textbf{in}\ e_2'}\ \text{S-Bind}$$

$$\frac{\Gamma \vdash v : \mathsf{t}_1 \rightsquigarrow v' \qquad s = \forall \bar{\alpha}.\mathsf{t}_1 \qquad \bar{\alpha} = FTV(\mathsf{t}_1) \setminus FTV(\Gamma) \qquad \Gamma, x{:}s \vdash e : \mathsf{t} \rightsquigarrow e'}{\Gamma \vdash \textbf{let}\ x = v\ \textbf{in}\ e : \mathsf{t} \rightsquigarrow \textbf{let}\ x = v'\ \textbf{in}\ e'}\ \text{S-Gen}$$

B.2 Elaboration to $\text{F}^\star$

Here we present the full elaboration of typed λ_{DST} programs to $\text{F}^\star$, based on the interpretation of λ_{DST} types in $\text{F}^\star$ shown in Section 4.1.

Syntax of $\text{F}^\star$ types

t	::=	$\alpha \mid T\,c \mid x{:}\mathsf{t} \to \mathsf{t}' \mid \forall \alpha{::}\kappa.\mathsf{t} \mid \mathsf{t}\ a \mid \mathsf{t}\ \mathsf{t}' \mid \lambda x{:}\mathsf{t}.\mathsf{t}' \mid \Lambda\alpha{::}\mathsf{k}.\mathsf{t} \mid x{:}\mathsf{t}\{\phi\} \mid (x{:}\mathsf{t} * \mathsf{t}') \mid \text{¡}\mathsf{t}$	types
k	::=	$\star \mid P \mid A \mid E \mid x{:}\mathsf{t} \Rightarrow \mathsf{k} \mid \alpha{::}\mathsf{k} \Rightarrow \mathsf{k}'$	kinds
v	::=	$v \mid \ldots \mid \Lambda\alpha{::}\kappa.\mathsf{e}$	values
e	::=	$e \mid \ldots \mid \mathsf{e}\ \mathsf{t}$	expressions
a	::=	$\ldots \mid \mathsf{v}$	atoms

Signature $S_{ST};\Gamma_{ST}$ in F* for translation primitives

```
type ref :: ⋆ ⇒ ⋆
type heap :: ⋆
type ST φ α ψ= h:heap{φ h} →(x:α ∗ h':heap{ψ x h}
type DST (α::⋆) φ::(α ⇒ heap ⇒ E) ⇒ heap ⇒ E=
      ∀ψ::(α ⇒ heap ⇒ E). ST (φ ψ) α ψ
val return : ∀α. x:α →DST α(Λψ::α⇒ heap⇒ E.ψ x)

val bind : ∀ α::⋆, β::⋆,
           φ1::(α ⇒ heap ⇒ E) ⇒ heap ⇒ E,
           φ2::α ⇒ (β ⇒ heap ⇒ E) ⇒ heap ⇒ E.
           DST α φ1
         →(x:α →DST β (φ2 x))
         →DST β (Λψ. φ1 (λy. φ2 y ψ))

val bind': ∀ α::⋆, β::⋆,
           'Tx::(α ⇒ heap ⇒ E) ⇒ heap ⇒ E,
           'Pre::α ⇒ heap ⇒ E,
           'Post::β ⇒ heap ⇒ E.
           DST α'Tx
         →(x:α →ST ('Pre x) β 'Post)
         →ST ('Tx 'Pre) β 'Post
```

```
logic val Select : ref α→heap →α
logic val Update : ref α→α→heap →heap
logic val Domain : heap →set
logic val Mem : ref α→set →bool

val return_e : ∀α::⋆. x:α →DST α (Λ 'Post.λ h. (∀x:α. 'Post x h))

val read : ∀α::⋆. x:ref α→DST α (Λ 'Post. λ h. 'Post (Select x h) h)

val write : ∀α::⋆. x:ref α→v:α →DST unit (Λ 'Post. λ h. 'Post () (Update x v h))

val alloc : ∀α::⋆. x:α →DST (ref α) (Λ'Post. λh. (∀(y:ref α). (Mem y (Domain h)=false)
                                                  ⟹ 'Post y (Update y x h)))
```

Inference of predicate transformers and elaboration to F*

$$\boxed{\Gamma \vdash v : t \rightsquigarrow \mathsf{v}} \qquad \frac{\vdash \Gamma \ \mathbf{ok} \quad \mathrm{constant}(v) \quad \mathrm{typeof} v = \tau}{\Gamma \vdash v : \tau \rightsquigarrow v}\ \text{Const} \qquad \frac{\vdash \Gamma \ \mathbf{ok} \quad \Gamma(x) = t}{\Gamma \vdash x : t \rightsquigarrow x}\ \text{Var}$$

$$\frac{\text{fresh } \eta \quad \Gamma, x{:}\tau \vdash \{\phi\}\, e : t\, \{\eta\} \rightsquigarrow \mathsf{e}}{\Gamma \vdash \lambda x{:}\tau.e : x{:}\tau \to \mathsf{DST}\ t\ (\Lambda\eta.\phi) \rightsquigarrow \lambda x{:}\tau.\Lambda\eta{::}[\![t]\!] \Rightarrow \mathsf{heap} \Rightarrow E.\mathsf{e}}\ \text{Lam}$$

$$\boxed{\Gamma \vdash \{\phi\}\, e : t\, \{\psi\} \rightsquigarrow \mathsf{e}} \qquad \frac{\Gamma \vdash v : t \rightsquigarrow \mathsf{v}}{\Gamma \vdash \{\phi\ \mathsf{v}\}\, v : t\, \{\phi\} \rightsquigarrow \mathsf{return}\ \tau\ \mathsf{v}\ [\![\phi]\!]}\ \text{Ret}$$

$$\frac{\vdash \Gamma\ ok \quad \Gamma(x) = \forall \bar\alpha.t' \quad t'[\bar\tau/\bar\alpha] = t}{\Gamma \vdash \{\lambda h.\forall x{:}t.\phi\ x\ h\}\, x_{\bar\tau} : t\, \{\phi\} \rightsquigarrow \quad \mathbf{let}\ y = x\ \bar\tau\ \mathbf{in}\ \mathsf{return\_e}\ [\![t]\!]\ y\ [\![\phi]\!]}\ \text{Var-poly}$$

$$\frac{\Gamma \vdash v_1 : x{:}\tau_1 \to \mathsf{DST}\ t_2\ \phi \rightsquigarrow \mathsf{v}_1 \quad \Gamma \vdash v_2 : \tau_1 \rightsquigarrow \mathsf{v}_2}{\Gamma \vdash \{\phi[\mathsf{v}_2/x]\ \psi\}\, v_1\ v_2 : t_2[\mathsf{v}_2/x]\, \{\psi\} \rightsquigarrow \mathsf{v}_1\ \mathsf{v}_2\ [\![\psi]\!]}\ \text{App-}\tau$$

$$\frac{\Gamma \vdash v_1 : \overline{\eta_\kappa}.t_1 \to \mathsf{DST}\ t_2\ \phi' \rightsquigarrow \mathsf{v}_1 \quad \Gamma \vdash v_2 : t_1' \rightsquigarrow \mathsf{v}_2 \quad |t_1'| = \varsigma\ |t_1| \quad \varsigma = [\overline{\phi/\eta}]}{\Gamma \vdash \{\varsigma\phi'\ \psi\}\, v_1\ v_2 : \varsigma t_2\, \{\psi\} \rightsquigarrow \mathsf{v}_1\ [\![\overline{\phi}]\!]\ \mathsf{v}_2\ [\![\psi]\!]}\ \text{App-}t$$

$$\frac{\text{fresh } \eta \quad e_1 \neq v \quad \Gamma \vdash \{\phi\}\, e_1 : t_1\, \{\eta\} \rightsquigarrow \mathsf{e}_1 \quad \Gamma, x{:}|t_1| \vdash \{\phi'\}\, e_2 : t_2\, \{\psi\} \rightsquigarrow \mathsf{e}_2 \quad \varsigma = [\lambda x{:}t_1.\phi'/\eta]}{\Gamma \vdash \{\varsigma\phi\}\, \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \lceil t_2 \rceil^{x:t_1}\, \{\psi\} \rightsquigarrow \mathsf{bind'}\ [\![t_1]\!]\ [\![\lceil t_2 \rceil^{x:t_1}]\!]\ [\![\Lambda\eta.\phi]\!]\ [\![\lambda x{:}t_1.\phi']\!]\ [\![\psi]\!]\ (\Lambda\eta.\mathsf{e}_1)\ (\lambda x{:}[\![t_1]\!].\mathsf{e}_2)}\ \text{Bind}$$

$$\frac{\Gamma \vdash v : t \rightsquigarrow \mathsf{v}' \quad \sigma = \forall \bar\alpha.t = Gen(\Gamma, t) \quad \Gamma, x{:}\sigma \vdash \{\phi\}\, e : t'\, \{\psi\} \rightsquigarrow \mathsf{e} \quad \mathsf{v} = \Lambda\bar\alpha{::}\star.\mathsf{v}'}{\Gamma \vdash \{\phi[\mathsf{v}/x]\}\, \mathbf{let}\ x = v\ \mathbf{in}\ e : t'[\mathsf{v}/x]\, \{\psi\} \rightsquigarrow (\lambda x{:}\sigma.\mathsf{e})\ \mathsf{v}}\ \text{Gen}$$

$$\frac{\Gamma \vdash v : \mathsf{bool} \rightsquigarrow \mathsf{v} \quad \forall i \in \{1,2\}.\Gamma \vdash \{\phi_i\}\, e_i : t_i\, \{\psi\} \rightsquigarrow \mathsf{e} \quad t = |t_1| \sqcup |t_2| \quad \phi = \lambda h.(v = \mathsf{true} \implies \phi_1\ h) \wedge (v = \mathsf{false} \implies \phi_2\ h)}{\Gamma \vdash \{\phi\}\, \mathbf{if}\ v\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 : t\, \{\psi\} \rightsquigarrow \mathbf{if}\ \mathsf{v}\ \mathbf{then}\ (\mathsf{e}_1 : [\![t]\!])\ \mathbf{else}\ (\mathsf{e}_2 : [\![t]\!])}\ \text{If}$$

$$\frac{\Gamma \vdash v : \tau \rightsquigarrow \mathsf{v} \quad \phi = \lambda h.\forall y.\, y \notin dom\ h \implies \psi\ y\ (upd\ h\ y\ v)}{\Gamma \vdash \{\phi\}\, \mathrm{ref}\ v : \mathsf{ref}\ t\, \{\psi\} \rightsquigarrow \mathsf{alloc}\ \tau\ \mathsf{v}\ [\![\psi]\!]}\ \text{Ref}$$

$$\frac{\Gamma \vdash v : \mathsf{ref}\ \tau \rightsquigarrow \mathsf{v}}{\Gamma \vdash \{\lambda h.\psi\ (sel\ h\ v)\ h\}\, !v : t\, \{\psi\} \rightsquigarrow \mathsf{read}\ \tau\ \mathsf{v}\ [\![\psi]\!]}\ \text{(Rd)}$$

$$\frac{\Gamma \vdash v_1 : \mathsf{ref}\ \tau \rightsquigarrow \mathsf{v}_1 \quad \Gamma \vdash v_2 : \tau \rightsquigarrow \mathsf{v}_2}{\Gamma \vdash \{\lambda h.\psi\ ()\ (upd\ h\ v_1\ v_2)\}\, v_1 := v_2 : \mathsf{unit}\, \{\psi\} \rightsquigarrow \mathsf{write}\ \tau\ \mathsf{v}_1\ \mathsf{v}_2\ [\![\psi]\!]}\ \text{(Wr)}$$

C. Soundness

This section develops our soundness result (also quoted as Theorem 1 (Soundness) in the main body of the paper).

Throughout this section, for any Γ and a possibly empty sequence of types and formuls $\bar t, \bar\psi$, we write $[\![\Gamma]\!]_{\bar t,\bar\psi}$ for $S_{ST};\Gamma_{ST}, [\![\Gamma]\!], FTV(\bar t, \bar\psi) \setminus FTV(\Gamma)$.

LEMMA 2 (Subtyping of closure). *For all* $\Gamma, x{:}t_x$ *such that* $\vdash [\![\Gamma, x{:}t_x]\!]$. ***ok****; and* t *such that* $\cdot;\cdot \vdash t$ ***ok*** *and* $[\![\Gamma, x{:}t_x]\!] \vdash_{F^\star} t :: \mathsf{k}$. *Then (G1)* $[\![\Gamma, x{:}t_1]\!] \vdash t <: \lceil t \rceil^{x:t_x}$ *and (G2)* $[\![\Gamma]\!] \vdash \lceil t \rceil^{x:t_x} :: \mathsf{k}$.

Proof:
From the definition of $\lceil t \rceil^{x:t_x}$ *appealing to* $\mathrm{F}^\star$*'s refinement subtyping for the sub-goal (G1)*
and, for (G2), noting that for $N; A \vdash \overline{\eta_\kappa}.t \to \mathit{DST}\ t'\ \psi$ ***ok*** *we have* $FV(t) = \varnothing$.

□

LEMMA 3 (Subtyping of joins). *For all* Γ *such that* $\vdash [\![\Gamma]\!]$. ***ok****; and* t_1, t_2 *such that for all* $i \in \{1,2\}$ *we have* $\cdot;\cdot \vdash t_i$ ***ok*** *and* $[\![\Gamma]\!] \vdash_{F^\star} [\![t_i]\!] :: \mathsf{k}$. *Then, if* $t_1 \sqcup t_2$ *exists, we have for all* $i \in \{1,2\}$, $[\![\Gamma]\!] \vdash [\![t_i]\!] <: [\![t_1 \sqcup t_2]\!]$.

Proof:
From the definition of $\cdot \sqcup \cdot$ *and* $\mathrm{F}^\star$*'s refinement subtyping.*

□

THEOREM 4 (Soundness).

1. *Given an environment* Γ *such that* $\vdash S_{ST}; \Gamma_{ST}, [\![\Gamma]\!]$ ***ok***
   and values v, v', *and* v, *and types* t, t
   such that $\lfloor \Gamma \rfloor \vdash v : \mathsf{t} \rightsquigarrow v'$
   and (TV) $\Gamma \vdash v' : t \rightsquigarrow \mathsf{v}$;
   then $S_{ST}; \Gamma_{ST}, [\![\Gamma]\!], FTV(t) \setminus FTV(\Gamma); \cdot \vdash_{F^\star} \mathsf{v} : [\![t]\!]$.

2. *Simultaneously, given an environment* Γ *such that* $\vdash \Gamma$ ***ok***
   and expressions e, e' *and* e; *types* t, t
   and ψ *such that (HK)* $S_{ST}; \Gamma_{ST}, [\![\Gamma]\!], FTV(t, \psi) \setminus FTV(\Gamma) \vdash [\![\psi]\!] :: t \Rightarrow \mathit{heap} \Rightarrow E$,
   such that $\lfloor \Gamma \rfloor \vdash e : \mathsf{t} \rightsquigarrow e'$
   and (TE) $\Gamma \vdash \{\phi\}\, e : t\, \{\psi\} \rightsquigarrow \mathsf{e}$;
   then $S_{ST}; \Gamma_{ST}, [\![\Gamma]\!], FTV(t, \psi) \setminus FTV(\Gamma); \cdot \vdash_{F^\star} \mathsf{e} : \mathit{ST}\ [\![\phi]\!]\ [\![t]\!]\ [\![\psi]\!]$.

Proof:
The proof is by mutual induction over the structure of (TV) and (TE).

The only non-trivial case of (TV) is the rule (Lam),
but this follows easily from the mutual inductin hypothesis on the premise.

The cases of (TE) are more interesting.

Case (S-Ret)*:*
From the mutual induction hypothesis on the premise and type-correctness of $\mathrm{F}^\star$
we obtain (HTV) $[\![\Gamma]\!]_t; \cdot \vdash_{F^\star} \mathsf{v} : [\![t]\!]$, *and (HTK)* $[\![\Gamma]\!]_t \vdash_{F^\star} [\![t]\!] :: \star$.
We appeal to the signature of return, *and use (HTV), (HTK) and (HK) to conclude.*

Case (App-poly)*: Similar to (S-Ret)*

Case (App-*τ*)*:*
Straightforward from the mutual induction hypothesis on each premise
and concluding with $\mathrm{F}^\star$*'s rules for application and type-application.*

Case (App-*t*)*:*
As in the previous case, except here we note first the interpretation of
higher-order function types as universal types in $\mathrm{F}^\star$,
and second, observe that unification obeys the identity $t_1 = \varsigma t_2 \implies [\![t_1]\!] = [\![\varsigma]\!][\![t_2]\!]$.

Case (Bind)*:*
From the induction hypothesis on the first premise, we obtain

$[\![\Gamma]\!]_{t,\eta}; \cdot \vdash_{F^\star} \mathsf{e}_1 : \mathit{ST}\ [\![\phi]\!][\![t]\!]\eta$

Or, $[\![\Gamma]\!]_t; \cdot \vdash_{F^\star} \Lambda\eta.\mathsf{e}_1 : \mathit{DST}\ [\![t]\!]\ \Lambda\eta.\phi$

From the induction hypothesis on the second premise, we obtain

$[\![\Gamma, x{:}\,|t_1|]\!]_{t_2,\psi}; \cdot \vdash_{F^\star} \mathsf{e}_2 : \mathit{ST}\ [\![\phi']\!][\![t_2]\!][\![\psi]\!]$

From the definition of $\lceil t_2 \rceil^{x:t_1}$, *and Lemma 2 (Subtyping of closure),*
we obtain that $[\![\Gamma, x{:}\,|t_1|]\!]_{t_2,\psi} \vdash [\![t_2]\!] <: [\![\lceil t_2 \rceil^{x:t_1}]\!]$.

Thus, we have $[\![\Gamma, x{:}\,|t_1|]\!]_{t_2,\psi}; \cdot \vdash_{F^\star} \mathsf{e}_2 : \mathit{ST}\ [\![\phi']\!][\![\lceil t_2 \rceil^{x:t_1}]\!][\![\psi]\!]$

Or, $[\![\Gamma]\!]_{t_1,t_2,\psi};\cdot \vdash_{F^\star} \lambda x{:}[\![t_1]\!].\mathsf{e}_2 : x{:}t_1 \to \mathsf{ST}\ [\![\phi']\!]\ [\![t_2]\!]\ [\![\psi]\!]$

The conclusion follows from the signature of bind' *and the* $\mathrm{F}^\star$ *rules for application*

Case (Gen)*:*
Easy from the induction hypotheses and the $\mathrm{F}^\star$ *rule for application.*

Case (If)*:*
From the induction hypotheses on each premise,
and Lemma 3 (Subtyping of joins) for the conclusion.

Case (Ref), (Read), (Wr)*: All follow from the signature* S_{ST}, Γ_{ST}.

□

D. Completeness

LEMMA 4 (Well-formed types).

(1) *For all* Γ *such that* $\vdash \Gamma$ ***ok*** *and* v,t,v *such that* $\Gamma \vdash v : t \rightsquigarrow \mathsf{v}$*; we have* $\cdot;\cdot \vdash |t|$ ***ok***.
(2) *For all* Γ *such that* $\vdash \Gamma$ ***ok*** *and* e,t,ψ,ϕ,e *such that* $\Gamma \vdash \{\phi\}\, e : t\, \{\psi\} \rightsquigarrow \mathsf{e}$*; we have* $\cdot;\cdot \vdash |t|$ ***ok***.

Proof:
Easy mutual induction over the structure of derivations.

□

LEMMA 5 (Unification of well-formed types).
For all $\overline{\eta}, A, t_1, t_2$ *such that (EQ)* $\lfloor t_1 \rfloor = \lfloor t_2 \rfloor$*, and (WF1)* $\overline{\eta}; A \vdash t_1$ ***ok****, and (WF2)* $\cdot; A \vdash t_2$ ***ok****; then there exists unique* $\overline{\phi}'$ *such that* $t_1[\overline{\phi}'/\overline{\eta}] = t_2$.

Proof:
By induction on the structure of WF1.

□

LEMMA 6 (Normalization is irrelevant for erasure). *For all* t, $\lfloor t \rfloor = \lfloor |t| \rfloor$.
Proof:
Easy, by inspection, normalization only affects formulas, which get erased.

□

LEMMA 7 (Existence of joins). *For all* t_1, t_2, M, A *such that* $\lfloor t_1 \rfloor = \lfloor t_2 \rfloor$*, and* $M; A \vdash t_1$ ***ok*** *and* $M; A \vdash t_2$ ***ok****. Then* $t_1 \sqcup t_2$ *exists and is unique.*
Proof:
By induction over the structure of one of the well-formedness judgment.

□

LEMMA 8 (Substitutions commute with erasure).
For all t, α *and* τ, $\lfloor t \rfloor[\tau/\alpha] = \lfloor t[\tau/\alpha] \rfloor$.
Proof:
Trivial.

□

LEMMA 9 (Value substitutions are irrelevant for erasure).
For all t, x *and* v, $\lfloor t \rfloor = \lfloor t[v/x] \rfloor$.
Proof:
Trivial—values only appear in predicate transformers, which are absent in partially erased types.

□

LEMMA 10 (Predicate substitutions are irrelevant for erasure).
For all t, η *and* ϕ, $\lfloor t \rfloor = \lfloor t[\phi/\eta] \rfloor$.
Proof:
Trivial—predicate variables only appear in predicate transformers, which are absent in partially erased types.

□

THEOREM 5 (Completeness).

(1) *Given an environment* Γ *such that (WF)* $\vdash \Gamma ok$ *and values* v, v' *and type* t *such that (TV)* $\lfloor \Gamma \rfloor \vdash v : \mathsf{t} \rightsquigarrow v'$*; then, given a fixed source of fresh names, there exists unique* t, v *such that* $\Gamma \vdash v' : t \rightsquigarrow \mathsf{v}$ *and* $\lfloor t \rfloor = \mathsf{t}$.
(2) *Simultaneously, given an environment* Γ *such that (WF)* $\vdash \Gamma$ ***ok*** *and expressions* e, e' *and type* t *such that (TE)* $\lfloor \Gamma \rfloor \vdash e : \mathsf{t} \rightsquigarrow e'$*; then, for any* ψ*, given a fixed source of fresh names, there exists unique* t, ϕ, e *such that* $\Gamma \vdash \{\phi\}\, e' : t\, \{\psi\} \rightsquigarrow \mathsf{e}$ *and* $\lfloor t \rfloor = \mathsf{t}$.

Proof:
We prove (1) and (2) by mutual induction on the structure of (TV) and (TE).

We start with the cases of (TV).
Case (S-1), (S-b)*: Trivial, since the only rule applicable is (Const).*

Case (S-x)*:*
We have (TV), specialized to the case where x is elaborated to a value x (rather than $x_{\bar{\tau}}$):

$$\frac{\lfloor\Gamma\rfloor(x) = \mathsf{t}}{\lfloor\Gamma\rfloor \vdash x : \mathsf{t} \rightsquigarrow x}$$

The conclusion follows from an application of (Var), relying on (WF) for the first premise.

Case (S-Lam)*:*
We have (TV) $\dfrac{(TV.1)\ \ \lfloor\Gamma\rfloor, x{:}\tau \vdash e : \mathsf{t} \rightsquigarrow e}{\lfloor\Gamma\rfloor \vdash \lambda x.e : \tau \to \mathsf{t} \rightsquigarrow \lambda x{:}\tau.e}$

For the conclusion, the only rule applicable is (Lam), and are required to prove
(TE.1) $\Gamma, x{:}\tau \vdash \{\phi\}\, e : t\, \{\eta\} \rightsquigarrow \mathsf{e}$.
For (TE.1), we apply the mutual induction hypothesis to (TV.1), noting $\lfloor\tau\rfloor = \tau$.
We obtain unique $\Gamma, x{:}\tau \vdash \{\phi\}\, e : t\, \{\eta\} \rightsquigarrow \mathsf{e}$, *where, (EQ)* $\lfloor t\rfloor = \mathsf{t}$
To conclude, we must show that $\lfloor x{:}\tau \to \mathit{DST}\ t\ \Lambda\eta.\phi\rfloor = x{:}\tau \to \mathsf{t}$,
which is immediate from (EQ) and the definition of $\lfloor\cdot\rfloor$.

We now turn to the cases of (2), the second mutual induction step.

Case (S-x)*:*
We have (TE) $\dfrac{(TE.1)\ \ \lfloor\Gamma\rfloor(x) = \forall\bar{\alpha}.\mathsf{t}}{\lfloor\Gamma\rfloor \vdash x : \mathsf{t}[\bar{\tau}/\bar{\alpha}] \rightsquigarrow x_{\bar{\tau}}}$

To conclude, the only rule applicable is (Var-poly).
We use (WF) for the first premise;
For the second premise, we appeal to (TE.1) and an inversion on $\lfloor\cdot\rfloor$
to get $\Gamma(x) = \forall\bar{\alpha}.t$
The third premise uses the substitution $[\bar{\tau}/\bar{\alpha}]$, *from the annotation,*
and hence there is only one way to apply the rule.

To show that $\lfloor t[\bar{\tau}/\bar{\alpha}]\rfloor = \mathsf{t}[\bar{\tau}/\bar{\alpha}]$, *we apply Lemma 8 (Substitutions commute with erasure).*

Case (S-App)*:*

We have (TE) $\dfrac{\lfloor\Gamma\rfloor \vdash v_1 : \mathsf{t} \to \mathsf{t}' \rightsquigarrow v_1' \qquad \lfloor\Gamma\rfloor \vdash v_2 : \mathsf{t} \rightsquigarrow v_2'}{\lfloor\Gamma\rfloor \vdash v_1\ v_2 : \mathsf{t}' \rightsquigarrow v_1'\ v_2'}$

The induction hypotheses applied to each of the premises of (TE) yields,
(From-IH1) $\Gamma \vdash v_1' : t_1$ *where (EQ1)* $\lfloor t_1\rfloor = \mathsf{t} \to \mathsf{t}' \rightsquigarrow v_1'$
and (From-IH2) $\Gamma \vdash v_2' : t \rightsquigarrow \mathsf{v}_2$ *where (EQ2)* $\lfloor t\rfloor = \mathsf{t}$.
We consider two sub-cases.
Sub-case (T1-EQ) $t_1 = x{:}\tau \to \mathbf{DST}\ t'\ \psi$*:*
We apply (App-τ) (the only rule applicable), using (From-IH1) for the first premise.
To use (From-IH2) for the second premise, we require showing $t = \tau$.
But, this follows from (EQ2) and that $\lfloor\tau\rfloor = \tau$.
To conclude, we must show that $\lfloor t'[\mathsf{v}_2/x]\rfloor = \mathsf{t}'$,
but we know that $\lfloor t'\rfloor = \mathsf{t}'$ *from (T1-EQ) and we apply Lemma 9 (Value substitutions are irrelevant for erasure).*
Sub-case (T1-EQ) $t_1 = \overline{\eta_\kappa}.t_{arg} \to \mathbf{DST}\ t'\ \psi$*:*

We apply (App-t), the only rule applicable, using (From-IH1) and (From-IH2) for the first two premises.
For the last two premises, we must show that $|t| = \varsigma\,|t_{arg}|$.
For this, we apply Lemma 4 (Well-formed types) to (From-IH1), and get
$\overline{\eta};\cdot \vdash |\overline{\eta_\kappa}.t_{arg} \to \mathit{DST}\ t'\ \psi|$ ***ok***.
From the definition of $|\cdot|$, *we get that that* $\overline{\eta};\cdot \vdash t_{arg}$ ***ok***.
Next, we apply Lemma 4 (Well-formed types) to (From-IH2), and obtain $\cdot;\cdot \vdash |t|$ ***ok***.
And we apply Lemma 5 (Unification of well-formed types), to obtain the necessary unique substitution.

Finally, we use Lemma 10 (Predicate substitutions are irrelevant for erasure) to conclude.

***Case (S-Bind)**:*

We have (TE) $\dfrac{e_1 \neq v \qquad (TE.1)\lfloor\Gamma\rfloor \vdash e_1 : \mathtt{t}_1 \rightsquigarrow e_1' \qquad (TE.2)\lfloor\Gamma\rfloor, x{:}\mathtt{t}_1 \vdash e_2 : \mathtt{t}_2 \rightsquigarrow e_2'}{\lfloor\Gamma\rfloor \vdash \mathit{let}\ x = e_1\ \mathit{in}\ e_2 : \mathtt{t}_2 \rightsquigarrow \mathit{let}\ x = e_1'\ \mathit{in}\ e_2'}$

From the induction hypothesis on the premise (TE.1), we get for fresh η,
(From-IH1) $\Gamma \vdash \{\phi\}\, e_1' : t_1\, \{\eta\} \rightsquigarrow \mathtt{e}_1$, $\lfloor t_1 \rfloor = \mathtt{t}_1$.
From Lemma 4 (Well-formed types), we have $\vdash \Gamma, x{:}\,|t_1|$ ***ok***
From Lemma 6 (Normalization is irrelevant for erasure) we have $\lfloor |t_2| \rfloor = \mathtt{t}_1$,
and so $\lfloor \Gamma, x{:}\,|t_1| \rfloor = \lfloor\Gamma\rfloor, x{:}\mathtt{t}_1$.
With this last fact, we can apply the induction hypothesis to (TE.2) and obtain
(From-IH2) $\Gamma, x{:}\,|t_1| \vdash \{\phi'\}\, e_2' : t_2\, \{\psi\} \rightsquigarrow \mathtt{e}_2$, *(EQ2)* $\lfloor t_2 \rfloor = \mathtt{t}_2$.
The only rule applicable is (Bind), and we use (From-IH1) and (From-IH2).
To conclude, we must show that $\lfloor \lceil t_2 \rceil^{x:t_1} \rfloor = \mathtt{t}_2$, *but this is trivial from (EQ2), since closure only affects formulas.*

***Case (S-Gen)**:*

Straightforward induction hypothesis to each premise, noting
$\vdash \mathit{Gen}(\Gamma, t)$ ***ok*** *from the use of normalization in* $\mathit{Gen}(.,.)$.

***Case (S-If)**:*

Syntactically, the only rule available is (If).
From the induction hypothesis applied to each premise of (TE),
we get each of the first three premises of (If).
For the fourth premise, we use Lemma 7 (Existence of joins) to prove that the join
exists and is unique.
The final premise is a tautology.

***Case (S-Ref)**: Easy, from the induction hypothesis and an application of (Ref).*

***Case (S-Read)**: Easy, from the induction hypothesis and an application of (Read).*

***Case (S-Write)**:*

Syntatically, the only rule avaiable is (Write).
We apply the induction hypothesis to each premise and obtain small types for each.
sufficient for the first two premises of (Write),
where the equality among small types is obtained from the premise of (TE),
since erasure is the identity on small types.

□

E. First-Order verification conditions

The judgements $\Gamma \vdash e : \mathtt{t} \rightsquigarrow e'$ and $\Gamma \vdash \{\phi\}\, e' : t\, \{\psi\} \rightsquigarrow \mathtt{e}$ combine to infer a stateful pre-condition ϕ for the program e and post-condition ψ: if ϕ holds on an initial heap, then ψ will hold on the value and heap resulting from the execution of e. The underlying $\mathrm{F}^\star$ type system is powerful, and it is natural to ask whether the inference algorithm—which may instantiate higher-order type arguments for higher-order function calls—always generates verification conditions that $\mathrm{F}^\star$ can discharge via an SMT solver with first-order theories. This is indeed the case, as we demonstrate in this section.

We begin by defining a judgment $\vdash_{FO}\ \phi\ \mathbf{ok}$, which restricts the shape of formulae to exclude the higher-order quantification form $\forall\alpha{::}\kappa.\phi$, for $\kappa \neq \star$.

Formulas without higher-order quantification

$$\boxed{\vdash_{FO}\ \phi\ \mathbf{ok}} \qquad \frac{\phi = \mathsf{T} \mid \mathsf{F} \mid a_1 = a_2 \mid a_1 \in a_2}{\vdash_{FO}\ \phi\ \mathbf{ok}} \qquad \frac{\vdash_{FO}\ \phi_1\ \mathbf{ok} \qquad \vdash_{FO}\ \phi_2\ \mathbf{ok}}{\vdash_{FO}\ \phi_1 \wedge \phi_2\ \mathbf{ok}} \qquad \frac{\vdash_{FO}\ \phi_1\ \mathbf{ok} \qquad \vdash_{FO}\ \phi_2\ \mathbf{ok}}{\vdash_{FO}\ \phi_1 \vee \phi_2\ \mathbf{ok}} \qquad \frac{\vdash_{FO}\ \phi\ \mathbf{ok}}{\vdash_{FO}\ \neg\phi\ \mathbf{ok}}$$

$$\frac{\vdash_{FO}\ \phi_1\ \mathbf{ok} \qquad \vdash_{FO}\ \phi_2\ \mathbf{ok}}{\vdash_{FO}\ \phi_1 \implies \phi_2\ \mathbf{ok}} \qquad \frac{\vdash_{FO}\ \phi\ \mathbf{ok}}{\vdash_{FO}\ \forall x{:}\sigma.\phi\ \mathbf{ok}} \qquad \frac{\vdash_{FO}\ \phi\ \mathbf{ok}}{\vdash_{FO}\ \exists x{:}\sigma.\phi\ \mathbf{ok}} \qquad \frac{\vdash_{FO}\ \phi\ \mathbf{ok}}{\vdash_{FO}\ \forall\alpha{:}\star.\phi\ \mathbf{ok}} \qquad \frac{\vdash_{FO}\ \phi\ \mathbf{ok}}{\vdash_{FO}\ \lambda x{:}t.\phi\ \mathbf{ok}} \qquad \frac{\vdash_{FO}\ \phi\ \mathbf{ok}}{\vdash_{FO}\ \phi\ a\ \mathbf{ok}}$$

$$\frac{\vdash_{FO}\ \phi\ \mathbf{ok}}{\vdash_{FO}\ \Lambda\alpha{::}\kappa.\phi\ \mathbf{ok}} \qquad \frac{\vdash_{FO}\ \phi\ \mathbf{ok} \qquad \vdash_{FO}\ \psi\ \mathbf{ok}}{\vdash_{FO}\ \phi\ \psi\ \mathbf{ok}} \qquad \frac{\vdash_{FO}\ \phi\ \mathbf{ok}}{\vdash_{FO}\ \forall\alpha{::}\star.\phi\ \mathbf{ok}} \qquad \frac{}{\vdash_{FO}\ \eta\ \mathbf{ok}}$$

Next, we prove a lemma establishing that, given a well-formed context and a post-condition lacking higher-order quantification, our inference algorithm computes a pre-condition without higher-order quantifiers.

LEMMA 11 (Well-formed Verification Conditions). *For all contexts* Γ *such that* $\vdash_{FO}\ \Gamma$ ***ok****; formulae* ψ *such that* $\vdash_{FO}\ \psi$ ***ok****; expressions* e*; types* t*; and pre-conditions* ϕ*; if (TE)* $\Gamma \vdash \{\phi\}\, e : t\, \{\psi\}$*, then* $\vdash_{FO}\ \phi$ ***ok****.*

Proof:
The proof procedes by induction on the typing relation (TE).

□

Intuitively, the above lemma shows that the inference algorithm does not introduce higher-kinded quantification where none previously existed.

Now, consider the verification condition generated by the inference algorithm. The judgment $\Gamma \vdash \{\phi\}\, e' : t\, \{\psi\} \rightsquigarrow \mathsf{e}$ produces a pre-condition that is well kinded in Γ at kind $\mathsf{heap} \Rightarrow E$. Applying the pre-condition to some initial heap H yields the verification condition $(\phi\ H)$, which is well-kinded at kind E. Now, since formulae have a standard beta reduction rule and there are no fixed points, we can reduce these to a normal form.

$$\frac{}{(\lambda x.\phi)\ v \rightsquigarrow \phi[v/x]}\ \text{R-}\lambda \qquad \frac{}{(\Lambda\alpha.\phi_1)\ \phi_2 \rightsquigarrow \phi_1[\phi_2/\alpha]}\ \text{R-}\Lambda$$

DEFINITION 12 *Let $\hat{\phi}$ denote the normal form resulting from a standard beta reduction of ϕ closed under (R-λ) and (R-Λ), both part of $\mathrm{F}^\star$'s equivalence relation on types.*

We next prove a standard cannonical forms lemma on formulae, where we consider the possible normal forms for E-kinded formulae in an environment Γ.

LEMMA 13 (Canonical forms of well-formed formulae). *For all Γ, ϕ, if $S_{ST}; \Gamma_{ST}, \Gamma \vdash [\![\phi]\!] :: E$, and $\vdash_{FO} \phi$, then $\hat{\phi}$ is one of the following:*

- $T \mid F$.
- $a_1 = a_2 \mid a_1 \in a_2$, *for all* a_2, a_2.
- $\phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \neg\phi_1 \mid \phi_1 \implies \phi_2 \mid \forall x{:}\sigma.\phi_1 \mid \exists x{:}\sigma.\phi_1 \mid \forall \alpha{::}\star.\phi_1$, *for all* ϕ_1, ϕ_2 *well-kinded at kind E and canonical.*
- η_E
- $\eta_{\kappa \Rightarrow \kappa'}\ \phi_1$

Proof:
Straightforward induction on the structure of ϕ.

□

Most of the syntactic operators that comprise formulae—including equality, implication, and quantification—can be trivially represented using first-order theories. In fact, for E-kinded formulas in canonical form, only the last two forms involving η variables pose potential difficulty—Λ-abstraction over predicates have been eliminated via reduction. The proposition below captures this property.

PROPOSITION 14 (First-order formula). *For all Γ, ϕ, if $S_{ST}; \Gamma_{ST}, \Gamma \vdash [\![\phi]\!] :: E$, if $\vdash_{FO} \phi$ and $\eta_E \notin FTV(\hat{\phi})$ and $\eta_{\kappa \Rightarrow \kappa'} \notin FTV(\hat{\phi})$ then $\hat{\phi}$ is a first-order formula.*

Proof:
Immediate, from the canonical forms lemma.

□

Next, we introduce an auxiliary lemma, which establishes that (aside from the pre- and post-condition), the types computed by our inference algorithm have no free predicate variables.

LEMMA 15 (Free-type variables are $\star$-kinded). *For all Γ such that $\vdash S_{ST}; \Gamma_{ST}, [\![\Gamma]\!]$ **ok**, and an $e, \mathsf{e}, t, \phi, \psi$ such that $\Gamma \vdash \{\phi\}\, e : t\, \{\psi\} \rightsquigarrow \mathsf{e}$, we have that for all $\alpha \in FTV(t)$, $S_{ST}; \Gamma_{ST}, [\![\Gamma]\!]_t \vdash \alpha{::}\star$.*

Proof:
By induction over the structure of the typing judgment.

□

With these tools in hand, we state and prove our final theorem, i.e., that when starting from closed, first-order post-conditions, every verification condition generated by our inference algorithm is representable in a first-order theory.

THEOREM 6 (First-order Verification Conditions). *For all contexts Γ such that $\vdash \Gamma$ **ok** and $\vdash_{FO} \Gamma$ **ok**; post-condition formulae ψ such $FTV(\psi) = \varnothing$ and that $\vdash_{FO} \psi$ **ok**; expressions e; types t; pre-conditions ϕ; and heap H, such that $\Gamma \vdash H : \mathsf{heap}$; if $\Gamma \vdash \{\phi\}\, e : t\, \{\psi\} \rightsquigarrow \mathsf{e}$, then $\hat{\phi}\ H$ is a first-order formula.*

Proof:
From our soundness theorem, we have $S_{ST}; \Gamma_{ST}; [\![\Gamma]\!]_t \vdash \mathsf{e} : ST\ [\![\phi]\!]\ [\![t]\!]\ [\![\psi]\!]$.
From the type-correctness of $\mathrm{F}^\star$, we have $S_{ST}; \Gamma_{ST}; [\![\Gamma]\!]_t \vdash [\![\phi]\!] :: \mathsf{heap} \Rightarrow E$
And $S_{ST}; \Gamma_{ST}; [\![\Gamma]\!]_t \vdash [\![\phi\ H]\!] :: E$
From Lemma 15 (Free-type variables are $\star$-kinded), we know that $[\![\Gamma]\!]_t$ binds only $\star$-kinded type variables, we know that
$\eta_E \notin FTV([\![\phi\ H]\!])$ and
$\eta_{\kappa \Rightarrow \kappa'} \notin FTV([\![\phi\ H]\!])$.
From the definition of $[\![\cdot]\!]$, a simple congruence over formulas, we know
$\eta_E \notin FTV(\phi\ H)$ and
$\eta_{\kappa \Rightarrow \kappa'} \notin FTV(\phi\ H)$.
Applying Lemma 1 (Well-formed Verification Conditions) we have $\vdash_{FO} \phi$ and we conclude using Proposition 14 (First-order formula).

□