Refinements of Futures Past: Higher-Order Specification with Implicit Refinement Types

Anish Tondwalkar
University of California, San Diego, CA, USA

Matthew Kolosick
University of California, San Diego, CA, USA

Ranjit Jhala
University of California, San Diego, CA, USA

Abstract

Refinement types decorate types with assertions that enable automatic verification. Like assertions, refinements are limited to binders that are in scope, and hence, cannot express higher-order specifications. *Ghost* variables circumvent this limitation but are prohibitively tedious to use as the programmer must divine and explicate their values at all call-sites. We introduce *Implicit Refinement Types* which turn ghost variables into implicit pair and function types, in a way that lets the refinement typechecker automatically *synthesize* their values at compile time. Implicit Refinement Types further take advantage of refinement type information, allowing them to be used as a lightweight verification tool, rather than merely as a technique to automate programming tasks. We evaluate the utility of Implicit Refinement Types by showing how they enable the modular specification and automatic verification of various higher-order examples including stateful protocols, access control, and resource usage.

2012 ACM Subject Classification Theory of computation → Program constructs; Theory of computation → Program specifications; Theory of computation → Program verification

Keywords and phrases Refinement Types, Implicit Parameters, Verification, Dependent Pairs

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2021.18

Related Version *Extended Version*: `arXiv:2105.01954` [40]

Supplementary Material *Software (ECOOP 2021 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.7.2.3`
archived at `swh:1:snp:aeaf3dbb58f5be84b565e73b5ade1503ee8cb6d6`

Funding This work was supported by NSF grant CCF-1911213.

1 Introduction

Refinement types allow programmers to decorate types with statically checked assertions ("refinements") that can be used for automatic verification over decidable theories, with applications including checking array bounds [48, 34], totality [44], data structure invariants [22], cryptographic protocols [4, 17], and properties of web applications [18].

Problem: Higher-Order Reasoning. Unfortunately, refinements cannot express the *higher-order* specifications needed for higher-order imperative programs [42]. Consider the access-control API:

```
grant :: File → IO ()                    read :: File → IO String
```

Here, `grant` and `read` represent a file access API that enforces access control policies: to `read` a given file, we must have been `grant`'d permission to that file. Concretely, this means that calls to `grant f` update the state of the world to one in which `f` has been added to the set of files we have permission to `read`.

Next, consider the API for operating over a stream of tokens in a linear fashion:

```
init :: ST Token                    next :: Token → ST Token
```

Here, `init` simply gives us the first token to start a stream. To use `next` to recieve a token we must pass it the previous token, which it will then invalidate. Both of these APIs include functions where correct usage fundamentally depends on the state of the world at their eventual call site, as they are composed using higher-order combinators such as `(>>=)` and `(>>)`.

One can formalize these informal specifications by augmenting the source program with *ghost variables* [31, 41, 42] that represent "departed quantities" that characterize the state of the world. In our file access example, `read f` would have to additionally take in the set of files we have been `grant`ed access to, and `f` must be a member, thus allowing us to compose the computations `grant f >> read f`. In our stream example, `next t` must both take in an additional mapping of tokens to their validity, in which `t` must be valid, and then give us back an updated mapping, reflecting that `next t` is the next valid token, thus allowing us to sequence the computations as `next t >>= next`, but not `next t >> next t`. The key idea in both of these examples is that `(>>)` ("sequence") and `(>>=)` ("bind") must *relate* the output of a higher-order argument (`grant f` and `next t`) to the requirements of their other argument (`read f` and `next`). For these stateful combinators, ghost variables allow us to lift the state of the world to relate the output of one computation to the input of another.

Note that while we start with these stateful examples for familiarity, we are interested in the more general problem of reasoning with higher-order programs. For instance, Handley et al. [19] introduce a `Tick` datatype that tracks resource usage of a computation. In this setting, one may want a higher-order constant-resource combinator such as

```
mapA :: (a → Tick n b) → xs:List a → Tick (n * len xs) (List b)
```

Informally, `mapA` maps a computation that uses a fixed amount of resources n (say, time) over a list and guarantees that it will take a fixed amount of resources equal to n times the length of the list. We can use ghost variables to lift the parameter `n` such that both the input function and the output computation can refer to it.

Ghosts of Past and Future. Notice that there is a key distinction between the ghost variables in our informal specifications. The ghost variable for `read` summarizes the history of accesses granted and therefore corresponds to a *history variable* [31] used to refine the past. It is determined *externally* by what accesses have been granted by the time `read` is called. Similarly, `mapA` uses the ghost variable `n` to track costs that will have been incurred once the computation finishes. In contrast, `next t` yields the next valid token, and so the ghost variable for `next` captures the *future* value when the computation is run. This corresponds to a *prophecy variable* [2] we can use to refine the future. That is, `next` makes an *internal* choice about what the next token *will be*. While external choice can be encoded as an additional parameter to `read`, internal choice can be encoded as adding an additional *return* value to `next`. However, in both cases, these ghost variables require polluting code with specification-level details and break APIs that don't accept or produce ghost variables.

Implicit Refinement Types. In this paper, we introduce *Implicit Refinement Types* (IRT), a feature that allows us to capture the above specifications with implicit ghost variables, while preserving the automatic verification properties of prior refinement type systems. Our approach takes inspiration from *implicit parameters* [25, 11, 12], a popular language feature that is the foundation of the theory of typeclasses [46]; models objects [11]; lends flexibility to module systems [47]; and features in C# [16], C++ [20], and Scala. To capture the distinction between internal and external choice of ghost parameters, we introduce two different notions

of implicits: implicit dependent functions (*i.e.* implicit-Π) and implicit dependent pairs (*i.e.* implicit-Σ). Implicit functions capture the notion of *external choice*, where the ghost parameter is determined by the caller of a function. Dually, implicit pairs correspond to *internal choice*, where the ghost parameter is determined by the implementation of the function. To sum up, we make the following contributions:

- A declarative semantics for ***implicit refinement types*** (§3), and an inference algorithm that is sound with respect to the declarative semantics (§5). Crucially, our semantics preserve *subtyping* information, allowing us to take advantage of refinement information in inferring implicit parameters.
- A notion of ***implicit pair types*** that, when added to implicit function types, allows us to express higher-order specifications without requiring any changes to code (§2).
- We implement our system, in a tool dubbed Mist, and ***evaluate*** our prototype on a range of case studies to demonstrate where automatic verification with Implicit Refinement Types bears fruit (§7).

2 Overview

We start with an example-driven overview that walks through specification and verification with implicit functions and pairs on minimal examples of higher-order functions, and then scales these examples to verify the access control, token stream, and resource accounting examples from §1.

2.1 Implicit Function Types

While plain refinement type systems can employ extra parameters to capture ghost variables, they must do so explicitly, requiring programmers to divine their values and implement bookkeeping. To illustrate this, consider the following example of a higher-order function `foo` that accepts a function parameter:

```
foo :: (Bool → Int) → ()
foo f = assert (f True == f False)
```

The static `assert`ion requires that the function passed to `foo` be constant. We could formalize this informal specification by adding an extra ghost parameter to `foo` to represent the singleton return value of its argument:

```
foo :: n:Int → (Bool → SInt n) → ()
```

But now we must not only rewrite `foo` to take this "unused" ghost parameter (`foo n f = assert (f True == f False)`) but must also manually modify the call site to `foo 1 (\z → 1)`. The *implicit function* type lets us handle this automatically. Instead of changing the definition of `foo` and its uses, the programmer changes `foo` to make `n` an *implicit parameter* by surrounding it with square brackets:

```
foo :: [n:Int] → (Bool → SInt n) → ()
```

Verifying programs with implicits. To verify `foo` and its call sites we must check: (1) that the `assert`ion in the body of `foo` is valid (2) that at the call-site `foo (\z → 1)`, the argument meets the precondition. We do so via a bidirectional traversal that checks terms against types to generate and solve *Existential Horn Constraints* (EHC). When checking *inside* the body of `foo`, the implicit parameter `[n:Int]` behaves just like a standard explicit, or "corporeal", parameter one might see in a Π-binder. That is, calls to `foo` will implicitly "pass" an argument

for `n`, so the sequel must hold for *all* values of `n`. This constrains the explicit argument `f` to always return `n`, from which the `assert`ion then follows directly. Checking the *call-site* is more interesting: the implicit parameter `[n:Int]` says the application is valid iff there *exists* a witness `n` such that the remainder of the specification holds. For `foo (\z → 1)`, we require that `n` equals the return value of `(\z → 1)`, *i.e.* `n = 1`. Let's see how to automatically find such witnesses.

Step 1: Templates. First, we generate *templates* to represent the types of terms whose refinements must be inferred. These templates are the base unrefined types for those terms, refined with *predicate variables* κ that represent the unknown refinements [34]. For example, we represent the as-yet unknown type of `\z → 1` with the template $z : \texttt{Bool} \rightarrow \{\nu : \texttt{Int} \mid \kappa(\nu)\}$.

Step 2: Existential Horn Constraints. Next, we traverse the term `foo (\z → 1)` in a bidirectional, syntax directed fashion (§5) to generate the EHC in Figure 1a.

- Constraint (1) comes from the body of the λ-term `\z → 1` and says the type of the returned value `1`, *i.e.* $\{\nu : \texttt{Int} \mid \nu = 1\}$, must be a subtype of the output type $\{\nu : \texttt{Int} \mid \kappa(\nu)\}$.
- Constraint (2) comes from applying `foo` to the λ-term: the type of which must be a subtype of `foo`'s input. Since function outputs are covariant, this means the output type $\{\nu : \texttt{Int} \mid \kappa(\nu)\}$ must be a subtype of $\{\nu : \texttt{Int} \mid \nu = n\}$.

Step 3: NNF Constraints. We say that the EHC in Figure 1a is satisfiable iff there exists a *predicate* for κ that, when substituted into the constraint, yields a *valid* first-order formula. To determine satisfiability, we transform the EHC into a new constraint shown in Figure 1b comprising

- An *NNF Constraint* [6], where we replace each existentially quantified n with a universally quantified version bounded by a predicate variable π, as shown in Constraint (Π), and
- An *Inhabitation Constraint* that each π is non-empty, as shown in Constraint (Σ).

We write π instead of κ here solely to differentiate between predicate variables from the original EHC and those produced in this translation. Intuitively, if every n satisfying $\pi(n)$ satisfies the NNF constraint *and* π is non-empty, *i.e.* there exists n such that $\pi(n)$, then we can conclude there exists some n that satisfies the original EHC.

Step 4: Solution. We require an assignment to the variables π, κ that makes the constraint in Figure 1b valid. We first compute the *strongest* valid solution [10] for κ, which yields $\kappa(v) \doteq v = 1$. As we require π to be inhabited (Constraint (Σ)), we assign to π the *weakest* predicate (as detailed in §6) that makes valid the NNF Constraint (Π), *i.e.*, we assign: $\pi(n) \doteq n = 1$. The above assignments for κ and π make the NNF constraint in Figure 1b valid, thus verifying `foo (\z → 1)`.

2.2 Implicit Pair Types

Implicit function parameters (`[n:Int] → —`) represent *external* choice where the implicit's value is resolved at the call site. But external choice alone is insufficient to capture every use of ghost variables. Consider the following example of a function that *returns* a function:

```
bar :: () → (Bool → Int)
bar _ = (\z → 1)
let f = bar () in assert (f True == f False)
```

$$\wedge\ \forall z.\ true \Rightarrow \forall \nu.\ \nu = 1 \Rightarrow \kappa(\nu) \quad (1)$$
$$\wedge\ \exists n.\ \forall \nu.\ \kappa(\nu) \Rightarrow \nu = n \quad (2)$$

(a) Existential Horn Clause.

$$\wedge\ \forall z.\ true \Rightarrow \forall \nu.\ \nu = 1 \Rightarrow \kappa(\nu)$$
$$\wedge\ \forall n.\ \pi(n) \Rightarrow \forall \nu.\ \kappa(\nu) \Rightarrow \nu = n \quad (\Pi)$$
$$\wedge\ \exists n.\ \pi(n) \quad (\Sigma)$$

(b) Horn Clause and Side Condition.

Figure 1 Verifying the application `foo (\z →1)`.

Unlike `foo`, `bar` instead *returns* a function `f`: we want to verify that the two calls to `f` return the *same* value. This simple example models our token stream API: while in fact `bar` always returns 1 we do not in general know the exact return value of a computation – for instance the token returned by `next` may be retrieved over the network. Suppose, as for `foo`, we use an implicit function argument to type `bar`:

```
bar :: [n:Int] → () → (Bool → SInt n)
```

Unfortunately, with this type we cannot verify the body of `bar`: for *any* external instantiation of `n`, it requires that `bar` will return a function that returns `n`, which is not true! Instead, `bar` is making an *internal* choice (specifically, that `n = 1`). This motivates our notion of *implicit pair types*, (written `[n:Int]. —`), which add a ghost parameter in the *return* position of a function. This lets us specify

```
bar :: () → [n:Int].(Bool → SInt n)
```

To verify that the code is safe using this specification, we are once again left with two tasks: (1) check that the `assert`ion at the use site of `bar` is valid (2) check at the definition site that the body of `bar` meets the specified postcondition. The first task is the easier one, though it requires an additional step that was not needed in verifying the body of `foo`. *Externally*, the type (`[n:Int]. —`) acts as an extra return value that behaves exactly like a standard ("corporeal") dependent pair and is assumed to exist in the body of the `let`. In order to account for this, we use a type-directed elaboration to automatically insert the appropriate "unpacking", giving a name to `n` at the use site:

```
unpack (n, f) = bar () in assert (f True == f False)
```

Verifying the assertion then follows the exact same logic as verifying the body of `foo`, as `f` is constrained to always return a value equal to `n`.

Now, we turn to the second task: *internally*, the type (`[n:Int]. —`) states that there *exists* some "ghost" value `n` that makes the remaining specification valid. Here, `n` names the return value of the function returned by `bar ()`. When checking the body of `bar`, we will need to find an instantiation of `n` and implicitly "pair" or "pack" this value with the type of the returned function `(\z → 1)`. Here, clearly the instantiation is `n = 1`. Note how the process of checking the definition site of `bar` mirrors that of checking the use site `foo (\z → 1)`. In fact, checking the body of `bar` will generate the exact same constraints shown in Figure 1, which will be solved by the same process, producing the same solution, which suffices to verify that `bar` implements its specification.

2.3 State

Next, we show how Implicit Refinement Types allow us to develop a new way of typing stateful computations, represented as higher-order state transformers. The key challenge here is to devise specification mechanisms that can *relate* the state *after* the transformation

```
-- | A Hoare-style State Transformer -------------------------------
data HST p     q      s a = State (s → (s, a))
type SST w_in w_out s a = HST {w|w = w_in} {w|w = w_out} s a

-- | Read and write the state --------------------------------------
get :: [w:s] → SST w w s {v:s|v = w}
get = State (\s → (s, s))

set :: w:s → HST s {v:s|v = w} s ()
set w = State (\_ → (w, ()))

-- | Monadic Interface for HST -------------------------------------
pure  :: [w:s] → x:a → SST w w s a
»=    :: [w1 w2 w3] →
           SST w1 w2 s a → (a → SST w2 w3 s b) → SST w1 w3 s b

-- | Client: Computing a "fresh" Int  ------------------------------
fresh :: [n:Int] → SST n (n + 1) Int n
fresh = do { n ← get; set (n + 1); pure n }
```

Figure 2 Typing Stateful Computations using Implicits.

with the state *before*. For example, to say that some function *increments* a counter, we need a way to say that the value of the counter after the transformation is one greater than the value before. Previous methods do this either by typing the computation with *two-state* predicates (as in YNot [29]) or with a *predicate transformer* that computes the value of the input state in terms of the output [38].

Implicit Refinement Types enable a new way to relate the input and output states while still ensuring that each atomic component of the specification simply refers to a single value. We will see that implicits are the crucial ingredient, allowing us to *name* – and hence, reason about – the output state, much as they did in the simplified `foo` and `bar` functions.

A Hoare-Style State Transformer Monad. First, as shown in Figure 2, we define a state transformer monad indexed by the type of the state `s` and the computation's result `a`. We call this a *Hoare State Monad* as it is also indexed with two *phantom* parameters `p` and `q` which will be refinement types describing the *input* and *output* states of the transformer. For convenience, we also define the singleton version `SST i o s a` where the `pre`-condition and `post`-conditions are singleton types that say that the input (resp. output) state is exactly `i` (resp. `o`). We can write a combinator to `get` the "current state", represented by the implicit parameter `w` that is both the input and output state, and also used in the singleton type of the result of the computation. Finally, we can write a combinator to `set` the state to some new (explicitly passed) value `w`, in which case, the input state can be any `s`.

A Monadic Interface using Implicit States. Next, we develop a monadic interface for programming with `HST`, by implementing the `pure` and `»=` combinators whose types use implicit parameters to relate their input and output states. The `pure` combinator takes an implicit `w` and returns a "pure" computation `SST w w s a` whose result is the input `x` and where the state is *unchanged*, *i.e.* where the input and output states are both `w`. The bind combinator `(»=)` combines transforms from `w1` to `w2`, and from `w2` to `w3` into a single transform

```
-- | Access control policy State Transformer ------------------------
type AC p1 p2 a = SST p1 p2 (Set String) a

-- | Grant or revoke access to a file path --------------------------
grant :: [p:Set String] → f:String → AC p (p ∪ single f) ()
grant f = State (\p → (insert f p, ()))

revoke :: [p:Set String] → f:String → AC p {v|v = p - single f} ()
revoke f = State (\p → (delete f p, ()))

read :: [p:Set String] → {f:String|f ∈ p} → AC p p String
read f = State (\p → (p, "file␣contents"))

-- | How one might safely read a file -------------------------------
main = runST {} (do grant "f.txt"; read "f.txt")

-- | Enabling dynamic access control policies -----------------------
canRead :: [p:Set String] → f:String → AC p p {v:Bool|v = f ∈ p}
canRead f = State (\p → (p, member f p))

safeRead :: [p:Set String] → f:String → AC p p (Maybe String)
safeRead f = do { r ← canRead;
                  if r then Just <$> read f
                       else pure Nothing }
```

Figure 3 Verifying Access Control Policies.

from `w1` to `w3`. Here, one can think of `w1` as a history variable summarizing the state of the world before the transformed computation and `w2` and `w3` as prophecy variables predicting what the respective sub-transformers will compute. The implicit refinement specification then ensures that these ghost variables all align appropriately.

Verifying Clients. We can use our interface to write a specification and implementation of the function `fresh` that "increments" a counter captured by the state parameter. This program `get`s an integer state `n`, and `set`s it to `n + 1`, and then returns `n`. The `do`-notation desugars into the monadic interface shown above in Figure 2. The specification captures the fact that the "counter" is incremented by relating the input and output states via the implicit parameter `n`. Notice that the implicit parameters are doubly crucial: first, they let us relate the input- and output-states, and second, they make programming pleasant by not requiring the programmer tediously spell out the intermediate states.

2.4 Access Control

The refined Hoare State Transformer lets us specify and verify the access control and token stream examples from §1. In Figure 3 we show how we can instantiate it with refinements over the theory of sets to derive a stateful API representing the verified access control primitives of `grant` and `read`. We first define `AC` as a specialization of the `SST` monad where we track file access permissions as a set of filenames both at the runtime level and – using implicits to relate the input and output states – at the type level.

```
-- | Token State Transformer ----------------------------------------
type TokM m1 m2 a = SST m1 m2 Tk a

-- | Start a stream, Get the next token unless the stream is done --
done  :: Tk
init :: [t:Tk]. TokM {} (store {} t ⊤) {v|v = t}
next :: [m:Map Tk Bool] → t:{Tk | (select m t) /\ (not (t = done))}
        → ([t':Tk]. TokM m (store (store m t ⊥) t' ⊤) t')

-- | Looping through all tokens -----------------------------------
client :: [m:Map Tk Bool] →
            t:{select m t} → TokM m {m'|select m' done} ()
client t = if t == done then pure ()
                        else do {t' ← next t; client t'}

-- | Starting the stream and running client ------------------------
main :: TokM {} {m|select m done} ()
main = do {t ← init; client t}
```

Figure 4 Verifying a Token Stream API.

An API for safe file access. We use the `AC` monad to develop an API that statically enforces compliance with an access control list (ACL). The `grant` primitive adds a file name to the access list, and `read` statically checks that the file is in the ACL and – for simplicity – just returns the string `"file␣contents"`. In conjunction with the implementations of `pure` and `»=` from Figure 2, the combinators can be used to verify `main` which, running with an initial empty access control list, grants permission to read a file then reads the file. If we accidentally tried to `read` from, say, `"secret-password.txt"`, the type checker would reject the program as unsafe.

Dynamic policies. Systems enforcing access control policies in practice [28] are not necessarily limited to a static policy – instead, they define a *checked read*, which checks at runtime whether a file is in the access control list, and then reads that file, returning a failure result if the permission check fails. We enable this via `canRead`, which determines if a file is in the dynamic ACL. This is reflected at the type level by passing in an implicit access control list `p` and specifying that `canRead` returns true iff its argument is in the ACL `p`. We then use `canRead` to define `safeRead`, which can be called with no conditions on the ACL. Instead, it uses `canRead` to dynamically check permissions, returning `Nothing` on failure. Crucially, to verify `main` and `safeRead` we do not need to tediously instantiate ghost variables, as Implicit Refinement Types automatically infer the suitable instantiations.

2.5 Token Stream

The Hoare State Transformer can be used to verify the token stream example of §1. In Figure 4 we show how to instantiate it with refinements over maps that track the validity of tokens. First, we define a specialization of the `SST` monad named `TokM` whose concrete state is a token (of type `Tk`) that tracks the last token we sent. `TokM`'s ghost state is a map of the status of *every* token. That is, `select m t` represents the proposition that the token `t` is *valid* (the next value to pass to the API): if `select m t = ⊤` (shorthand for `true`), then `t` is valid. Otherwise, `select m t = ⊥` (shorthand for `false`) means that the token `t` has been used and is now invalid.

```
-- | Singletons as resource counts ---------------------------------
data Tick t a = Tick a
type T t a = Tick {v:Int | v = t} a

-- | The Applicative Functor API ----------------------------------
<*>  :: [n:Int m:Int] → T n (a → b) → T m a → T (n + m) b
</>  :: [n:Int m:Int] → T n (a → b) → T m a → T (n + m + 1) b
pure :: x → T 0 a

-- | Appending two lists in a linear number of steps ---------------
 ++    :: xs:(List a) → ys:(List a)
                       → T (len xs) {v|len v = len xs + len ys}
[]       ++ ys = pure ys
(x:xs')  ++ ys = pure (x:) </> (xs' ++ ys)

-- | Mapping a costly function over a list -------------------------
mapA :: [n] → (a → T n b) → xs:List a → T (n * len xs) (List b)
mapA f []     = pure []
mapA f (x:xs) = pure (:) <*> f x <*> mapA f xs
```

Figure 5 Intrinsic Verification of Resource Usage.

An API for streaming tokens. The `TokM` monad lets us specify and verify the token streaming API from Section 1. First, we specify a special token `done` that represents the last token of the stream. On the other hand, `init` represents a *computation* that begins the stream using an implicit pair to capture that there is *some* valid token `t` resulting from the computation of `init`. Moreover, `init` starts with the empty map (with no stale tokens) to ensure that we may only begin the stream once.

The workhorse of this API is `next`. It first takes an implicit history parameter `m` representing all of the past tokens. We then check that the token `t` passed in is not the `done` token, and use `m` to constrain `t` to be valid (`select m t`). Finally, another implicit pair is used to produce the *prophecy* variable `t'` which is both the next value returned by the computation, and the next valid token as noted by the ghost state, which also marks the old token `t` invalid.

We can now develop the `client`, which recursively consumes the remaining stream of tokens. Were we to attempt to reuse the token `t` in the recursive call the program will be correctly rejected as unsafe. `main` kicks off the stream with `init` and consumes it using `client`. Thus, implicit refinements eliminate the tedium of manually instantiating ghosts.

2.6 Intrinsic Verification of Resource Usage

Next, we demonstrate how implicit refinement types can be used for specifying higher-order programs beyond the state monad: in particular, tracking resource usage. Figure 5 defines an applicative functor for counting resource usage in the same style as Handley et al. [19]. This API has both the standard application operator `<*>` and a resource-consuming application operator `</>`. The API counts the number of times we use the `</>` operator, which allows us to apply a function `f` with cost `n` to an argument `x` of cost `m`, incurring a total cost of `n + m + 1`.

Handley et al. [19] show these combinators can be used to verify properties about resource usage. We adapt their example of counting the recursive steps in `xs ++ ys`. At each recursive

step, we append another element `x` to the beginning of the list using `pure (x:)</>`. Ultimately `(++)` will use `len xs` applications of this operator to build this list, verifying that `(++)` is linear in the first argument.

Using this API we can further define the higher-order constant-resource combinator `mapA`, which allows us to map a function `f` of constant cost `n` over a list `xs`, and automatically verify that doing so costs `n * len xs`. This is easy to specify with implicit refinement types: the implicit argument lifts the output cost of the function argument so that we may relate it with the overall cost of calling `mapA`.

Two-State Specifications. It is worth pausing here to recall the standard technique for specifying effectful programs like the ones we have shown: the two-state specifications that allow expressing the input and output requirements of a particular computation. For instance, `fresh` (Figure 2) would be given a specification such as `requires (\s → ⊤) ensures (\s o s' → s' = s + 1 ∧ o = s)` where `s` and `s'` represent the input and output state and `o` represents the output of the computation. Notably, even if our language included such two-state specifications, specifying `mapA` would still require an extra parameter, as the relationship between the start and end "states" of `mapA` depends on the relationship between the start and end "states" of the `(a → T n b)` argument.

On the other hand, implicit refinements scale from capturing relations between the inputs and outputs of a single computation to relating a higher-order computation to its function argument(s) without having to "hardwire" some notion of two-states or pre/post conditions. Instead, they allow us to *name* the input and output worlds and *lift them to the top level*, which allows assertions (refinements) that span those states/worlds, including in examples such as `mapA`. The key contribution of Implicit Refinement Types then is that they work both for classic two-state specifications *and other* use-cases where two-state specifications prove cumbersome and allow the necessary extra parameter of functions like `mapA` to be instantiated automatically.

3 Programs

We start with a declarative static semantics for our elaborated core language λ^R. Our discussion here omits polymorphism as it is orthogonal to adding implicit types. (The full system can be found in the extended edition [40]).

3.1 Syntax

Figure 6 presents the syntax of our source language – a lambda calculus with refinement types, extended with implicit function and dependent pair types.

Types. of λ^R begin with base types `Int` and `Bool`, which are *refined* with a (boolean-valued) expression r to form refined base types $\{x : b \mid r\}$. Next, λ^R has dependent function types $x : t_1 \to t_2$. Dependent function types are complemented by *implicit* dependent function types $[x : t_1] \to t_2$, which are similar, except that the parameter x is passed *implicitly*, and does not occur at runtime. Dually, we have *implicit* dependent pairs $[x : t_1].t_2$, which represent a pair of values: The first, named x, of type t_1, is implicit (automatically determined) and does not occur at runtime. Meanwhile, the second is of type t_2 which may refer to x. We use τ to denote *unrefined* types.

Types

$b \quad ::= \quad \texttt{Int} \mid \texttt{Bool} \mid \cdots$

$t \quad ::= \quad \{x : b \mid r\} \mid x : t \to t \mid [x : t] \to t \mid [x : t].t$

Terms

$c \quad ::= \quad \mathit{false}, \mathit{true} \mid 0, 1, \ldots \mid \wedge, \vee, +, -, =, \leq, \ldots$

$e \quad ::= \quad c \mid x \mid \lambda x : t.e \mid e\ e \mid \texttt{let}\ x : t = e\ \texttt{in}\ e \mid \lambda^i x : t.e \mid \texttt{unpack}\ (x, y) = e\ \texttt{in}\ e$

Contexts

$\Gamma \quad ::= \quad \bullet \mid \Gamma, x : t \mid \Gamma, [x : t]$

Type Checking $\boxed{\Gamma \vdash e : t}$

$$\textsc{T-AbsI}\ \frac{\Gamma, [x : t_x] \vdash e : t}{\Gamma \vdash \lambda^i x : t_x.e : ([x : t_x] \to t)} \qquad \textsc{T-App}\ \frac{\Gamma \vdash e_1 : t \quad \Gamma \mid t \vdash e_2 : t'}{\Gamma \vdash e_1\ e_2 : t'} \qquad \textsc{T-Var}\ \frac{x : t \in \Gamma}{\Gamma \vdash x : t}$$

$$\textsc{T-Let-}\tau\ \frac{\Gamma \vdash e_x : t_x \quad \Gamma, x : t_x \vdash e : t \quad \Gamma \vdash t \quad \Gamma \vdash t_x \quad \lfloor t_x \rfloor = \tau}{\Gamma \vdash \texttt{let}\ x : \tau = e_x\ \texttt{in}\ e : t} \qquad \textsc{T-Unpack}\ \frac{\Gamma \vdash e_1 : [x' : t_1].t_2 \quad \Gamma, [x : t_1], y : t_2[x/x'] \vdash e_2 : t \quad \Gamma \vdash t}{\Gamma \vdash \texttt{unpack}\ (x, y) = e_1\ \texttt{in}\ e_2 : t}$$

Application Checking $\boxed{\Gamma \mid t_1 \vdash e : t_2}$

$$\textsc{AppI}\ \frac{\Gamma \mid t[e'/x] \vdash e : t' \quad \langle \Gamma \rangle \vdash e' : t_x}{\Gamma \mid [x : t_x] \to t \vdash e : t'} \qquad \textsc{App}^{\ e}\ \frac{\Gamma \vdash e : t_e \quad \Gamma \vdash t_e \preceq t_x \quad \Gamma, y : t_e \vdash t[y/x] \preceq t' \quad \Gamma \vdash t' \quad y\ \text{fresh}}{\Gamma \mid x : t_x \to t \vdash e : t'}$$

Subtyping $\boxed{\Gamma \vdash t_1 \preceq t_2}$

$$\preceq\textsc{-Base}\ \frac{[\![\Gamma]\!](\forall v_1 : b.r_1 \Rightarrow r_2[v_1/v_2])\ \text{is valid}}{\Gamma \vdash \{v_1 : b \mid r_1\} \preceq \{v_2 : b \mid r_2\}} \qquad \preceq^R_{\text{IPAIR}}\ \frac{\Gamma \vdash t_1 \preceq t_2'[e/x] \quad \langle \Gamma \rangle \vdash e : t_2}{\Gamma \vdash t_1 \preceq [x : t_2].t_2'}$$

Figure 6 Syntax and static semantics of λ^R.

Terms. of λ^R comprise *constants* (booleans, integers and primitive operations) and *expressions*. Let binders are half-annotated with either a refinement type to be checked, or a base type on which refinements are to be inferred.

In addition to explicit function abstraction, λ^R has the implicit λ-former $\lambda^i x : t.e$, where the parameter x represents a *ghost* value that can only appear in refinement types. Implicit functions are instantiated automatically, so there is no syntax for eliminating them. Similarly, implicit dependent pairs are introduced automatically, and thus have no introduction form in λ^R. Instead, implicit dependent pairs have an *elimination* form $\texttt{unpack}\ (x, y) = e_1\ \texttt{in}\ e_2$. Here, if e_1 is of type $[x : t_x].t$, then x is bound at type t_x and y is bound at type t in e_2. Just like with implicit functions the x represents a *ghost* value that may only appear in refinement types.

Though both implicit lambda and unpack terms are present in our model, in practice we handle their insertion by elaboration: we discuss this aspect of our implementation in §7. In light of this, we develop the theory of Implicit Refinement Types in terms of the fully elaborated expressions of the λ^R syntax.

Contexts. of λ^R, written Γ, comprise the usual ordered sequences of "corporeal" binders $x : t$, where x is visible in both terms and refinements, as well as *ghost binders* $[x : t]$, where x is only visible in refinements.

3.2 Static Semantics

Figure 6 provides an excerpt of the declarative typing rules for λ^R (the complete rules are in the extended edition [40]). Most of the rules are standard for refinement types [34] so we focus our attention on the novel rules regarding implicit types.

Type Checking. judgments of the form $\Gamma \vdash e : t$ mean "in context Γ, the term e has type t." The ghost binders in Γ, written $[x : t]$, reflect the ghostly, refinement-only nature of implicits. This distinction is witnessed by the rule [T-Var] which types term-level variables using only the corporeal binders $x : t$ in Γ. This ensures that implicit variables are erasable: they can only appear in types (specifications) and thus *cannot* affect computation.

With the separation of ghostly (erasable) implicit binders from corporeal (computationally relevant) binders, both the introduction rule for implicit functions [T-AbsI] and the elimination rule for implicit pairs [T-Unpack] are standard up to the ghostliness of binders. Implicit pairs are eliminated through "unpacking" as is typical for dependent pairs and existential types. The rule [T-Let-τ] demonstrates how we handle annotations at unrefined base types τ: we pick some well-formed refinement type t_x that erases to the base type $\lfloor t_x \rfloor = \tau$, and then use t_x as the bound type of x.

Lastly, we split off type checking of applications into an application checking judgment, in order to handle instantiations of implicit functions. In the rule [T-App] we use this additional judgment to check that the argument e_2 is compatible with the input type of e_1.

Application Checking. judgments of the form $\Gamma \mid t \vdash e : t'$ mean "when e is the argument to a function of type t the result has type t'". The (corporeal) application rule [App ^{e}], finds the type t_e of e, checks that this is consistent with the input type t_x of the function, and then creates a new name y to refer to e so that we may substitute it into the return type. However, y is not bound in Γ so we guess a type t', well-formed under Γ, for the entire term, and check that the return type $t[y/x]$ is a subtype of t'. The rule [AppI] checks implicit applications by *guessing* an expression e' at which to instantiate the implicit parameter. This e is only used at the refinement-level and is thus allowed to range over both corporeal and ghost binders in Γ, as described by the antecedent $\langle\Gamma\rangle \vdash e : t$. ($\langle\Gamma\rangle$ replaces each ghost binder $[x : t]$ in Γ with a corresponding corporeal binder $\langle[x : t]\rangle = x : t$.) The rule then continues along the spine of the application, further instantiating implicit parameters as necessary until we can apply the rule [App ^{e}].

Subtyping Judgments. of the form $\Gamma \vdash t_1 \preceq t_2$ mean "in context Γ, the values of t_1 are a subset of the values of t_2." Most of the rules are standard, with the base case [$\preceq$-Base] reducing to a *verification condition* (VC) that checks if one refinement *implies* another. [$\preceq^R_{\text{IPAIR}}$] serves as the "introduction form" for implicit pairs, and states that a term of type t_1 can be used as an implicit pair type if there is some expression e of type t_2 such that t_1 is a subtype of t_2' instantiated with that e. Note that this is similar to explicitly constructing the dependent pair with e as the first element.

$$
\begin{array}{rrcl}
\textit{Predicates} & r & ::= & \ldots \text{varies} \ldots \\
\textit{Types} & \tau & ::= & \ldots \text{varies} \ldots \\
\textit{Propositions} & p & ::= & \kappa(\overline{x}) \mid r \\
\textit{Existential Horn Clauses} & c & ::= & \exists x : \tau.c \mid c \wedge c \mid \forall x : \tau.p \Rightarrow c \mid p \\
\textit{First Order Assignments} & \Psi & ::= & \bullet \mid \Psi, r/x \\
\textit{Second Order Assignments} & \Delta & ::= & \bullet \mid \Delta, \lambda\overline{x}.r/\kappa
\end{array}
$$

Figure 7 Syntax of L^S.

4 Logic

We define the syntax and semantics of *verification conditions* (VCs) generated by rule [$\preceq$-Base]. Figure 7 summarizes the syntax of VCs, *Existential Horn Clauses* (EHC), which extends the NNF Horn Clauses used in Cosman and Jhala [10] with *existential* binders. *Predicates* r range over a decidable background theory. *Propositions* p include predicates and second order *predicate variables* $\kappa(\overline{x})$. *Clauses* c comprise quantifiers, conjunction, and propositions. In the syntax tree of clauses, there are two places a κ variable may appear: as a leaf (*head* position), or in an antecedent under a universal quantifier $\forall x : \tau.\kappa(\overline{y}) \Rightarrow c$, (*guard* position).

Dependencies of an EHC constraint c are the set E of pairs (κ, κ') such that κ appears in guard position in c and κ' appears in head position under that guard. When $(\kappa, \kappa') \in E$, we say that κ *depends on* κ', or, dually, κ' appears *under* κ. The *cycles* in a constraint c are nonempty sets S of predicate variables such that: for all $\kappa \in S$, there exists $\kappa' \in S$ such that $(\kappa, \kappa') \in E$. A set of predicate variables S is said to be *acyclic* in c, if all cycles in c contain at least one predicate variable not in S. A predicate variable κ is said to be *acyclic* in c, if no cycles in c contain κ. A constraint c is said to be *acyclic* if there are no cycles in c.

Semantics. [$\preceq$-Base] checks the *validity* of the formula obtained by interpreting contexts and terms in our constraint logic, (formalized by $[\![\cdot]\!]$ in the extended edition [40]). Contexts Γ yield a sequence of universal quantifiers for all variables bound at (interpretable) basic types. Recall that VCs do not contain predicate variables $\kappa(\overline{x})$. The restricted grammar of the VCs is designed to be amenable to SMT solvers, represented by an oracle $\mathsf{Valid}(c)$ that checks validity, defined at the end of this section.

We eliminate *predicate variables* κ via substitution, $c[\Delta]$ (defined in the extended edition [40]) that map them onto meta-level lambdas $\lambda\overline{x}.p$. We represent solutions to *existential binders* with a substitution (Ψ) binding existential variables to predicates. We define an *existential substitution* $c\{r/x\}$ recursively over c which removes the corresponding existential binder, using standard substitution to replace x with its solution r: $(\exists x : \tau.c)\{r/x\} \doteq c[r/x]$.

EHC Validity is then defined by the judgment $\Delta; \Psi \vDash c$. Intuitively, c is valid under the substitutions Δ, Ψ if the result of applying the substitutions yields a VC that is valid. We say that an EHC is *satisfiable*, written $\vDash c$ if *there exist* Δ and Γ such that $\Delta; \Gamma \vDash c$. We say that c and c' are *equisatisfiable* when $\vDash c$ iff $\vDash c'$.

5 Type Inference

The declarative semantics described in Figure 6 are decidedly non-deterministic. This is most evident in the rules for implicits, such as [AppI] and $[\preceq^R_{\text{IPAIR}}]$, where, out of thin air, we generate an expression to instantiate the implicit parameter. Additional non-determinism

Subtyping $\boxed{\Gamma \vdash t_1 <: t_2 \dashv c}$

$$\frac{c = \forall x : \tau.[\![r]\!] \Rightarrow [\![r'[x/y]]\!]}{\Gamma \vdash \{x : \tau \mid r\} <: \{y : \tau \mid r'\} \dashv c} \qquad \frac{\Gamma, z : t_2 \vdash t_1 <: t_2'[z/x] \dashv c \;\; z \text{ fresh}}{\Gamma \vdash t_1 <: [x : t_2].t_2' \dashv \exists z :: t_2.c}$$

Checking $\boxed{\Gamma \vdash e \Leftarrow t \dashv c}$

$$\text{C-Sub}\;\frac{\Gamma \vdash e \Rightarrow t' \dashv c \qquad \Gamma \vdash t' <: t \dashv c'}{\Gamma \vdash e \Leftarrow t \dashv c \wedge c'} \qquad \text{C-Let-}\tau\;\frac{\Gamma \vdash e_1 \Leftarrow \hat{t} \dashv c_1 \qquad \Gamma, x : \hat{t} \vdash e_2 \Leftarrow t \dashv c_2 \;\; \hat{t} = \mathsf{fresh}(\Gamma, \tau)}{\Gamma \vdash \texttt{let}\; x : \tau = e_1 \;\texttt{in}\; e_2 \Leftarrow t \dashv c_1 \wedge (x :: \hat{t} \Rightarrow c_2)}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow [x' : t_1].t_2 \dashv c_1 \quad \Gamma, [x : t_1], y : t_2' \vdash e_2 \Leftarrow t \dashv c_2 \qquad t_2' = t_2[x/x'] \quad c = c_1 \wedge (x :: t_1 \Rightarrow (y :: t_2' \Rightarrow c_2))}{\Gamma \vdash \texttt{unpack}\; (x, y) = e_1 \;\texttt{in}\; e_2 \Leftarrow t \dashv c} \qquad \frac{\Gamma \vdash e_1 \Rightarrow t' \dashv c_1 \qquad \Gamma \mid t' \vdash e_2 \ll t \dashv c_2}{\Gamma \vdash e_1\; e_2 \Leftarrow t \dashv c_1 \wedge c_2}$$

Synthesis $\boxed{\Gamma \vdash e \Rightarrow t \dashv c}$

$$\frac{x : t \in \Gamma}{\Gamma \vdash x \Rightarrow t \dashv \top} \qquad \frac{\Gamma \vdash e_1 \Rightarrow t_1 \dashv c_1 \qquad \Gamma \mid t_1 \vdash e_2 \gg t_2 \dashv c_2}{\Gamma \vdash e_1\; e_2 \Rightarrow t_2 \dashv c_1 \wedge c_2}$$

Application Checking $\boxed{\Gamma \mid t \vdash e \ll t' \dashv c}$

$$\text{C-App-}\rightarrow\;\frac{\Gamma \vdash y \Leftarrow t_x \dashv c_1 \qquad \Gamma \vdash t[y/x] <: t' \dashv c_2}{\Gamma \mid x : t_x \rightarrow t \vdash y \ll t' \dashv c_1 \wedge c_2} \qquad \text{C-App-IFun}\;\frac{\Gamma, [z : t_x] \mid t[z/x] \vdash y \ll t' \dashv c \qquad z \text{ fresh}}{\Gamma \mid [x : t_x] \rightarrow t \vdash e \ll t' \dashv \exists z :: t_x.c}$$

Figure 8 Constraint Generation.

appears in rules like [T-Unpack], where a refinement type must be picked such that it is well-formed under the outer context Γ. This is required as the body of the unpack expression is checked under Γ extended with the binders x and y, but the type must be well-formed under Γ itself to ensure that these variables do not escape (since they may appear in the type t).

We account for all of the non-determinism of the declarative semantics with an algorithmic, bidirectional type inference system [32, 15], excerpts of which are shown in Figure 8 (the full rules are in the extended edition [40]). We split the declarative type checking judgments into two forms: synthesis ($\Gamma \vdash e \Rightarrow t \dashv c$) and checking ($\Gamma \vdash e \Leftarrow t \dashv c$) along with corresponding application synthesis ($\Gamma \mid t \vdash y \gg t' \dashv c$) and application checking ($\Gamma \mid t \vdash y \ll t' \dashv c$) forms. Synthesis forms produce the type as an output while checking forms take the type as an input. We also introduce an algorithmic subtyping judgment ($\Gamma \vdash t_1 <: t_2 \dashv c$). In addition to their other outputs, these judgments produce an EHC c as an output. The core of our inference algorithm is precisely in extending the restricted grammar of verification conditions to an EHC that captures the constraints on the non-deterministic choices. Inference then reduces to the satisfiability of the constraint c (as checked in §6).

5.1 Constraining Unknown Refinements

Consider the following λ^R program from §2.1:

$$\text{EXAMPLE} = \texttt{let}\; y : (\texttt{Bool} \rightarrow \texttt{Int}) = \lambda z : \texttt{Bool}.1\;\texttt{in}\;\; \text{foo}\; y.$$

recalling that foo has the type $[n : \texttt{Int}] \rightarrow (\texttt{Bool} \rightarrow \{v : \texttt{Int} \mid v = n\}) \rightarrow \texttt{Unit}$. We wish to check that this program is safe by checking that it types with type `Unit`.

To type EXAMPLE in our declarative semantics, we first need to apply the rule [T-LET-τ] which non-deterministically chooses a t_x, well-formed under Γ ($\Gamma \vdash t_x$), such that t_x is consistent with the base type ($\lfloor t_x \rfloor = \texttt{Bool} \to \texttt{Int}$). To capture picking this refinement type we employ predicate variables that represent unknown refinements, as is standard in the refinement type inference literature [23, 34]. As we know the type we wish to give EXAMPLE, we will focus on the checking rule [C-LET-τ]. We generate a fresh refinement type using fresh, which takes as input a type t and the current context Γ and then produces a refinement type, where each base type is refined by a *fresh* predicate variable $\kappa(\overline{x})$, where $\overline{x}$ are all of the variables bound in the context, all of which can appear in refinements at this location. In our example, this would give the type $\hat{t} = z : \{v : \texttt{Bool} \mid \kappa_1(v)\} \to \{v : \texttt{Int} \mid \kappa_2(z, v)\}$.

We then check $\lambda z : \texttt{Bool}.1$ at the refinement type $\hat{t}$. Now, we use the rule [C-SUB] to synthesize a type for this term and then use subtyping to check that the synthesized type is subsumed by $\hat{t}$. The synthesized type will be $z : \{v : \texttt{Bool} \mid \kappa_3(v)\} \to \{v : \texttt{Int} \mid v = 1\}$ and the subtyping check will generate the following constraint which is equivalent to (1) in Figure 1a modulo the administrative predicate variable κ_3 and a simplification of the unconstrained κ_1:

$$\wedge\, \forall v.\ \kappa_1(v) \Rightarrow \kappa_3(v)$$
$$\wedge\, \forall z.\ \kappa_1(z) \Rightarrow \forall v.\ v = 1 \Rightarrow \kappa_2(z, v)$$

There is a subtlety in how [C-LET-τ] generates the quantified subformula $\forall z.\ \kappa_1(z) \Rightarrow \cdots$: this formula is generated by the clause $x :: \hat{t} \Rightarrow c_2$ where the double colon represents a *generalized implication* that drops any variables quantified at non-base types (as only base types are interpreted into the refinement logic).

$$x :: \{x : b \mid r\} \Rightarrow c \doteq \forall x : b.r \Rightarrow c \qquad x :: t \Rightarrow c \doteq c$$

5.2 Constraining Implicit Application

The predicate variables let us capture guessed refinement types as second order constraints. Next we turn to checking the *implicit* application foo y. foo has the type $[n : \texttt{Int}] \to (\texttt{Bool} \to \{v : \texttt{Int} \mid v = n\}) \to \texttt{Unit}$, so the declarative semantics arbitrarily picks an expression e of type `Int` to instantiate n. In the algorithmic type system, we capture the constraints on this choice with the existential quantifiers of our EHC. This is shown in the application checking rule [C-APP-IFUN] (the corresponding application synthesis rule [S-APP-IFUN] appears in the extended edition [40]).

Recall that the judgment $\Gamma \mid t \vdash e \ll t' \dashv c$ says that we are checking an application of a term of type t to an argument e and require that the application has the type t'. Here, we are checking foo y against the type `Unit`. Our bidirectional rule [C-APP-IFUN] "guesses" the instantiation by generating a fresh variable n and binding it at the type `Int`. This variable is added to the context as predicate variables may depend on it. We then generate a constraint $\exists n :: \texttt{Int}.c$ which says that our guessed n must be consistent with c, the constraint generated by continuing to check down the abstract syntax tree (along the spine of the application of the revealed function type $t[y/x] = (\texttt{Int} \to \{v : \texttt{Int} \mid v = n\}) \to \texttt{Unit}$). $\exists n :: \texttt{Int}.c$ is the existential counterpart to the generalized implication $n :: \texttt{Int} \Rightarrow c$:

$$\exists x :: \{x : b \mid r\}.c \doteq \exists x : b.(r \wedge c) \qquad \exists x :: t.c \doteq c$$

This is now a concrete function type and the rule [C-APP-$\to$] will check that the argument y has the type $\texttt{Int} \to \{v : \texttt{Int} \mid v = n\}$ and that the type `Unit` is a subtype of `Unit`. Checking the implicit and then concrete application thus generates the constraints $\exists n.(\forall v.\ \top \Rightarrow \kappa_1(v) \wedge \forall v.\ \kappa_2(\_, v) \Rightarrow v = n)$.

Combining these constraints with those generated from checking $\lambda z : \texttt{Bool}.1$, we get constraints equivalent to those in Figure 1a. A satisfying assignment is:

$$\Delta = [\lambda z, v.v = 1/\kappa_1, \lambda v.\top/\kappa_2, \lambda v.\top/\kappa_3] \qquad \Psi = [1/v]$$

Satisfying solutions to the predicate variables and existential constraints give instantiations to the angelic choices of refinement types and implicit arguments respectively. This gives us the following soundness theorem for our type inference algorithm (where the function $\mathsf{kvars}(c)$ returns the set of predicate variables in c):

▶ **Theorem 1** (Soundness of Type Inference). *If* $\bullet \vdash e \Rightarrow t \dashv c$, $\Delta; \Psi \vDash c$, *and* $\mathsf{kvars}(t) \subseteq \mathsf{domain}(\Delta)$, *then* $\bullet \vdash e : \Delta(t)$.

6 Solving

The constraints generated by algorithmic type checking have both predicate variables and alternating universal and existential quantifiers. We must provide solutions to both predicate variables and existential variables before we can use SMT solvers to check the validity of a VC (§4). We compute solutions in four steps:

1. We transform the EHC by skolemization to replace existential variables with universally quantified predicate variables and inhabitation side conditions.
2. We eliminate the original predicate variables.
3. We solve the skolem predicate variables.
4. Finally, we check the inhabitation side conditions.

Weakening and Strengthening. A function f on constraints is a *strengthening* when $\forall c.\ f(c) \Rightarrow c$. A function f on constraints is a *weakening* when $\forall c.\ c \Rightarrow f(c)$. We prove that, if the transformations in steps 1 and 2 above are both weakening and strengthening, our algorithm produces a verification condition that is *equisatisfiable* with the original constraint, *i.e.* our algorithm is sound and complete.

Separable Constraints. An EHC c is *separable* if it can be written as a conjunction $c_1 \wedge c_2$, where c_1 is an NNF Horn clause and c_2 is an acyclic EHC. The following theorem exactly characterizes separable EHCs:

▶ **Theorem 2.** *c is separable iff there are no cyclic κs under existential binders.*

There are standard partial techniques for solving cyclic NNF Horn clauses [6, 10] so the task of solving a separable EHC can be split into applying one of these existing techniques and then solving the acyclic EHC. Consequently, all a programmer must do to make constraints separable is provide either a local solution to an implicit variable via an explicit value or a solution to a cyclic predicate variable (*e.g.* by providing a type signature for a recursive function.)

Thus, in the sequel, we focus on the remaining problem: solving an acyclic EHC. We use the acyclic EHC from Figure 1a (reproduced below) as a running example. Recall that this is a simplified version of the constraints generated during type inference on the program EXAMPLE in §5.1 and §5.2.

$$\wedge\ \forall z.\ \top \Rightarrow \forall \nu.\ \nu = 1 \Rightarrow \kappa(\nu) \tag{3}$$

$$\wedge\ \exists n. \forall \nu.\ \kappa(\nu) \Rightarrow \nu = n \tag{4}$$

Step 1: Skolemization. We use the function skolem to transform the EHC c to a conjunction of an NNF Horn Clause $\mathsf{noside}(\mathsf{skolem}(\emptyset, c))$ and side conditions $\mathsf{side}(\mathsf{skolem}(\emptyset, c))$. This differs from textbook skolemization in two important ways: We replace each existential quantifier $\exists n.c$ with Skolem predicates $\forall n.\pi_n(n, \overline{x}) \Rightarrow c$ rather than Skolem functions, so that we can synthesize a (declarative) relation rather than a function. As a result, we must still check to make sure that this relation is inhabited. We do so by producing the side condition $\exists n.\pi_n(n, \overline{x})$. Our transformation Skolemizes the existential binding (4) of our example as follows:

$$\wedge\ \forall z.\ \top \Rightarrow \forall \nu.\ \nu = 1 \Rightarrow \kappa(\nu) \tag{5}$$
$$\wedge\ \forall n.\pi(n) \Rightarrow \forall \nu.\ \kappa(\nu) \Rightarrow \nu = n \tag{6}$$
$$\wedge\ \exists n.\pi(n) \tag{7}$$

This transformation will be crucial later: giving a name to π allows us to separate the inhabitation and sufficiency constraints on n.

skolem yields an NNF that has two classes of predicate variables: Skolem predicates corresponding to existential binders (written π_n) that have an inhabitation side condition, and predicate variables corresponding to unknown refinements (written κ). The π_n only appear negatively, so the standard technique of finding the least fixed point solution [34, 10] would simply return $\bot$, which will fail the inhabitation side conditions. Instead, we would like to compute the *greatest fixed point* solution for each π_n, but, for efficiency reasons, do not wish to compute the greatest fixed point solution of every predicate variable. Fortunately Cosman and Jhala [10] show that acyclic predicate variables can be eliminated one by one. We explain first how to eliminate κ variables and then how to eliminate π variables.

Step 2: Eliminating κ-Variables from c. Procedure elim1 of [10] eliminates each individual acyclic predicate variable, κ, in an NNF Horn Clause. Briefly, given a predicate variable κ in an NNF Horn Clause c, the procedure computes the strongest solution for κ: $\mathsf{sol}\kappa(\kappa, c)$, and then substitutes the solution into c. For the single κ in our example this solution $\mathsf{sol}\kappa(\kappa, c)$ is $\lambda x.(\exists z'.\top \wedge (\exists v'.v' = 1 \wedge v' = x))$. After substitution and simplification we get

$$\wedge\ \forall n.\pi(n) \Rightarrow \forall v.\top \Rightarrow \forall z'.\top \Rightarrow \forall v'.v' = 1 \Rightarrow v = v' \wedge v = n$$
$$\wedge\ \exists n.\pi(n)$$

Step 3: Eliminating Skolem Variables. elim1 removes all the κ predicate variables leaving only the π_n variables inserted by skolem. The inhabitation side conditions require we find the greatest fixed point (GFP) solution to these variables to ensure we do not spuriously eliminate witnesses. As π_n only appears negatively (in guards), the GFP is the conjunction of every c appearing as $\forall n.\pi_n(n, \overline{x}) \Rightarrow c$. For a given π_n appearing in the constraint c', we write this GFP as $\mathsf{def}\pi(\pi_n, c')$, the *defining constraint* of π_n.

A first challenge arises in that we wish to use these solutions to *eliminate* the Skolem predicate variables, but $\mathsf{def}\pi(\pi_n, c')$ is a conjunction of clauses featuring quantifiers. As the Skolem variables appear in guards, we must transform these c into an equisatisfiable predicate p before we may substitute, so we parameterize our elimination algorithms with a quantifier elimination algorithm qe that handles this task. qe can vary with the particular domain and, for domains that do not admit quantifier elimination or where it is infeasible, we instead use an approximation described in §6.1.

$\mathsf{elim}\pi^*_{qe}$	:	$P^C \to \overline{\Pi} \times C^\Pi \times C \to C$
$\mathsf{elim}\pi^*_{qe}([], \sigma, c)$	$\doteq$	c
$\mathsf{elim}\pi^*_{qe}(\pi_n : \overline{\pi}, \sigma, c)$	$\doteq$	$\mathsf{elim}\pi^*_{qe}(\overline{\pi}, \sigma, c[\lambda\overline{x}.p/\pi_n])$
where p	$=$	$qe(\mathsf{sol}\pi_{qe}(\{\pi_n\}, \sigma, \sigma(\pi_n)))$
$\mathsf{sol}\pi_{qe}$	:	$P^C \to \overline{\Pi} \times C^\Pi \times C \to C$
$\mathsf{sol}\pi_{qe}(\overline{\pi}, \sigma, \forall n.\pi_n(n, \overline{x}) \Rightarrow c)$		
$\mid \pi_n \in \overline{\pi}$	$\doteq$	$\mathsf{sol}\pi_{qe}(\overline{\pi}, \sigma, c)$
$\mid \pi_n \notin \overline{\pi}$	$\doteq$	$\forall n.p \Rightarrow \mathsf{sol}\pi_{qe}(\overline{\pi}, \sigma, c)$
where p	$=$	$qe(\mathsf{sol}\pi_{qe}(\overline{\pi} \cup \{\pi_n\}, \sigma, \sigma(\pi_n)))$
$\mathsf{sol}\pi_{qe}(\overline{\pi}, \sigma, \forall x.p \Rightarrow c)$	$\doteq$	$\forall x.p \Rightarrow \mathsf{sol}\pi_{qe}(\overline{\pi}, \sigma, c)$
$\mathsf{sol}\pi_{qe}(\overline{\pi}, \sigma, c_1 \wedge c_2)$	$\doteq$	$\mathsf{sol}\pi_{qe}(\overline{\pi}, \sigma, c_1) \wedge \mathsf{sol}\pi_{qe}(\overline{\pi}, \sigma, c_2)$
$\mathsf{sol}\pi_{qe}(\overline{\pi}, \sigma, p)$	$\doteq$	p

Figure 9 Eliminating π Variables and their Side Conditions.

A second technical challenge arises en route to our solving algorithm: $\mathsf{def}\pi(\pi_n, c')$ may contain *other* π variables and may contain cycles involving π variables. Fortunately, recursive Skolem variables are redundant:

▶ **Lemma 3.** *If π_n is a predicate variable inserted by* skolem*, then $\vDash \forall n.\pi_n(n, \overline{x}) \Rightarrow c$ iff $\vDash \forall n.\pi(n, \overline{x}) \Rightarrow c[\lambda\_.\top/\pi]$.*

This lets us to break cycles by ignoring recursive occurrences of Skolem variables.

With this result in hand, we develop the procedure $\mathsf{sol}\pi_{qe}$, as shown in Figure 9. $\mathsf{sol}\pi_{qe}$ recursively eliminates Skolem variables from the defining constraint of π_n, using qe to transform a nested Skolem variable's defining constraint (tracked in the map σ) into a substitutable predicate. The first argument $\overline{\pi}$ tracks the seen Skolem variables, treating them as if they were solved to $\top$ when they next occur.

$\mathsf{sol}\pi_{qe}$ is used in the procedure $\mathsf{elim}\pi^*_{qe}$ which eliminates all Skolem variables from the constraint one π_n at a time. $\mathsf{elim}\pi^*_{qe}$ simply calls $\mathsf{sol}\pi_{qe}$ on the defining constraint of π_n and then qe's the result (now free of Skolem variables) before substituting the returned predicate as the solution for π_n.

Regardless of the properties of qe we have the following soundness theorem:

▶ **Theorem 4.** *If $c' = \mathsf{skolem}(\emptyset, c)$, $\overline{\pi}$ is the set of all Skolem variables in c', c' has no other predicate variables, $\sigma(\pi_n) = \mathsf{def}\pi(\pi_n, c')$ for all $\pi_n \in \overline{\pi}$, and $\vDash \mathsf{elim}\pi^*_{qe}(\overline{\pi}, \sigma, c')$ then $\vDash c$.*

If qe is a strengthening and a weakening, the converse also holds and our algorithm is complete.

Step 4: Putting It All Together. The procedure safe (defined in the extended edition [40]) puts together all the pieces to automatically check the validity of acyclic EHC constraints c. Like $\mathsf{sol}\pi$ and $\mathsf{elim}\pi^*$, safe is parameterized by the quantifier elimination procedure qe. safe first Skolemizes the constraint c and then solves the original predicate variables by repeated applications of elim1. Next, safe collects the defining constraints of the Skolem predicate variables and uses $\mathsf{elim}\pi^*_{qe}$ to solve and eliminate the Skolem variables. This leaves us with a predicate-variable-free constraint, but still featuring nested existential quantifiers from

Table 1 Comparison of systems: ✓on "Spec" and "Check" respectively indicate that the specification can be written and that the code can be verified by automatically instantiating the implicit parameters.

Tool / Case Study	Mist				MoCHi [41]		F* Implicits [26]	
	LOC	Time (s)	Spec	Check	Spec	Check	Spec	Check
incr	8	0.01	✓	✓	✓	✗	✓	✗
sum	5	0.01	✓	✓	✓	✓	✓	✗
repeat	86	0.28	✓	✓	✓	✗	✗	✗
d_2 [41]	8	0.07	✓	✗	✓	✓	✓	✗
Resources								
incrState	28	0.65	✓	✓	✓	✓	✓	✓
accessControl	49	0.36	✓	✓	✓	✗	✓	✓[1]
tick [19]	85	0.03	✓	✓	✗	✗	✓	✓
linearDSL	20	0.03	✓	✓	✗	✗	✓	✓[1]
State Machines								
pagination	79	0.3	✓	✓	✗	✗	✗	✗
login [7]	121	0.09	✓	✓	✗	✗	✗	✗
twoPhase	135	0.86	✓	✓	✗	✗	✗	✗
tickTock	112	0.14	✓	✓	✗	✗	✗	✗
tcp	332	115.73	✓	✓	✗	✗	✗	✗

the inhabitation side conditions. Here, we use qe once again to eliminate the existential quantifiers, leaving us with a VC ripe to be passed to an automated theorem prover to verify validity. If this VC is valid, we deem c *safe*. If qe is a strengthening then safe is sound and if qe is additionally a weakening then safe is complete, proved in the extended edition [40].

▶ **Theorem 5.** *Let c be an acyclic constraint. If qe is a strengthening then $\mathsf{safe}(c)$ implies $\vDash c$. If qe is also a weakening then $\mathsf{safe}(c)$ iff $\vDash c$.*

6.1 A Theory-Agnostic Approximation to qe

We cannot, in general, provide a quantifier elimination that is a strengthening and a weakening (since we are agnostic to the set of theories) especially as some theories do not admit a decidable quantifier instantiation! However, since SMT theories work by equality propagation, we can make use of equalities between theory terms without making any additional assumptions about the theories themselves. Therefore, we include a theory-agnostic quantifier elimination strategy over equalities.

Given the defining constraint c of some $\pi_n(n, \overline{x})$, our strategy computes the (well-scoped) congruence closure of the variables n and $\overline{x}$ using the body of c. This set of equalities is then used as the solution to π_n. To eliminate the existential side condition $\exists n.\pi_n(n, \overline{x})$ we note that a sound approximation is to find a solution for n. We search for this solution within the set of equalities. If there is not one, we return $\bot$ and verification fails. In Section 7 we evaluate this incomplete quantifier elimination on a number of benchmarks and demonstrate its real-world effectiveness.

7 Evaluation

We implement our system of Implicit Refinement Types in a tool dubbed Mist, evaluate Mist using a set of illustrative examples and case studies (links to full examples elided

for DBR), and compare Mist against the existing state of the art, in order to answer the following questions:

Q1: Lightweight Verification Are implicits in conjunction with the theory-agnostic instantiation procedure sufficient to verify programs with IRTs, even without using heavyweight instantiation techniques such as domain-specific solvers and synthesis engines?

Q2: Expressivity Can we use implicits to encode specifications that would've otherwise required the use of additional language features?

Q3: Flexibility Do they allow *automated verification* in places where unification-based implicits and CEGAR-based extra parameters do not?

Implementation. Mist extends our language (§3) with polymorphism and type constructors, and omits concrete syntax for implicit lambdas and unpacks. Instead, implicit parameters appear solely in specifications – that is, refinement *types* – and do not require implementation changes to the code.

Inserting implicit lambdas is straightforward: when checking that a term has an implicit function type, insert a corresponding implicit lambda. Inserting implicit unpacks is more interesting: we need to unpack any term of implicit pair type before we use it. For instance, if e has the type $[x : t_1].t_2$, then we must unpack e to extract the corporeal (t_2) component of the pair before we can apply a function to it: $e_f\ e$ becomes $\texttt{unpack}\ (x, y) = e\ \texttt{in}\ e_f\ y$. This transformation ensures that the ghost parameter (x in the above term) is in context at the use site of the implicit pair. To automate this procedure, we use an Anf-like transformation that restricts A-normalization to terms of implicit pair type.

As is common in similar research tools, Mist handles datatypes by axiomatizing their constructors. A production implementation would treat surface datatype declarations as sugar over these axioms.

Polymorphism. Mist also includes support for limited refinement polymorphism. For simplicity, we left refinement polymorphism out of our formalism in §3.2 – it is orthogonal to the addition of implicit parameters and pairs – but we did include it in our implementation.

Refinement polymorphism alleviates some issues with phantom type parameters. First, when using phantom type parameters, core constructors cannot be directly verified and must be assumed to have the given type. Moreover, we can specify the semantics of our stateful APIs in terms of *e.g.* the `HST` type, but the meaning of the arguments to the HST type operator are determined only by its use. In contrast, refinement polymorphism brings the intended semantics of HST from the world of phantom parameters into the semantics of the language itself. The semantics of HST are reflected with the type:

```
type HST = rforall p q. forall a. p → (q, a)
```

where `rforall` ranges over refinement types and `forall` ranges over base types. The more sophisticated refinement polymorphism [42] present in existing refinement systems would use the type:

```
type HST p q s a = {w:s | p w} → ({w':s | q w'}, a)
```

Here, `p` and `q` are predicates on the state type `s`.

[1] Verification requires annotating a simple fact about sets as F* does not include a native theory of sets.

Refinement polymorphism further makes core constructors and API primitives themselves subject to verification: the `rforall` version of HST above allows the direct verification of `get` as

```
get : forall s. [w:s] → HST {v:s | v = w} {v:s | v = w} s
get = \s → (s, s)
```

Comparsion. We compare Mist to higher-order model checker MoCHi [41], and F*'s support for implicit parameters[2] [36, 26]. Both systems, like Mist and unlike foundational verifiers such as Idris [8] and Coq [39], are designed for lightweight, automatic verification. MoCHi aims to provide complete verification of higher-order programs by automatically inserting extra (implicit) parameters. Whereas Mist has users write refinement type specifications (with *explicit* reference to implicit types), MoCHi's specifications are implemented as assertions within the code. F*'s type system is a mix of a Martin-Löf style dependent type system with SMT-backed automatic verification of refinement type specifications. There is no formal specification of implicit parameters in F*. They are implemented by unification as part of Martin-Löf typechecking. This is in contrast to IRTs' integrated approach, that uses information from refinement subtyping constraints for instantiation.

Our comparison, summarized in Table 1, illustrates, via a series of case studies, the specifications that can be written in each system (the Spec column) and whether they can find the necessary implicit parameter instantiations (the Check column). As each tool is designed to be used for lightweight verification, we write the implementation and then separately write the specifications. We do not rewrite the implementations to better accommodate specifications, though for MoCHi we insert assertions within the code as necessary.

7.1 Q1: Lightweight Verification

We evaluate whether IRTs allow for modular specifications that would otherwise be inexpressible with plain refinement types. We do this via a series of higher-order programs that use implicits for lightweight verification. These programs are designed to capture core aspects of various APIs and how IRTs permit specifications that can be automatically verified in a representative client program using the API. Notably, for all of these examples Mist only uses the theory-agnostic instantiation procedure from §6.1 and does not employ heavyweight instantiation techniques such as domain-specific solvers or synthesis engines.

Higher-Order Loops. Repeat, defines a loop combinator `repeat` that takes an increasing stateful computation `body` and produces a stateful computation that loops `body count` times. Its signature in Mist is:

```
repeat :: body:([x:Int] → ([y:{v:Int | v > x}]. SST x y Int Int))
       → count:{v:Int | v > 0}
       → ([q:Int] → ([r:{v:Int | v > q}]. SST q r Int Int))
```

which says that the input and output are stateful computations whose history and prophecy variables together guarantee that the computation will leave the state larger than it started. In the first line the implicit function argument `x` is externally determined and captures the state of the world before the `body` computation, while the implicit pair component `y` captures that `body` updates the state of the world to some (unknown) larger value.

[2] Note that F*'s main verification mechanism is Dijkstra Monads [38], which we do not compare against in our evaluation. We discuss trade offs between Dijkstra Monads and Implicit Refinement Types in §8.

The implicit pairs are necessary to specify `repeat`, as the updated state is determined by an internal choice in `body`. REPEAT also demonstrates how implicit pairs can specify loop invariants (here, upward-closure) on higher-order stateful programs. Though `repeat` is specified using implicit pairs, MIST does *not* require that arguments passed to `repeat` be specified using implicit pairs. In REPEAT we define a function `incr` with the type `[x:Int] → SST x (x + 1) Int Int` that increments the state. MIST will appropriately type check and verify `repeat incr` as Implicit Refinement Types automatically account for the necessary subtyping constraints to ensure that `incr` meets the conditions of `repeat`.

State Machines. The next set of examples demonstrate that IRTs enable specification verification of state-machine based protocols. These are ubiquitous, spanning appications from networking protocols to device driver and operating system invariants [7]. IRTs let us encode state machines as transitions allowed from a ghost state, using the Hoare State Monad (2.3). LOGIN [7], which models logging into a remote server, served as our initial inspiration for studying this class of problems. We verify that a client respects the sequence of connect, login, and only then accesses information.

TICKTOCK verifies that two `ticker` and `tocker` processes obey the specification standard to the concurrency literature [35]. Here we show the implementation of `tocker`.

```
tocker = \c → do
msg ← recv c
if msg = tick then send c tock else assert False
```

Ghost parameters on `send` and `recv` track the state machine and ensure messages follow the tick-tock protocol. If `send c tock` is changed to `send c tick` the program is appropriately rejected.

TWOPHASE is a verified implementation of one side of a two-phase commit process. This example serves as a scale model for TCP, a verified implementation of a model of a TCP client performing a 3-way handshake, using the TCP state machine[33].

PAGINATION is an expanded version of our stream example from Section 2, and models the AWS S3 pagination API [1]. This example shows that our protocol state machine need not be finite, as it is specified with respect to an *unbounded* state machine.

Quantitative Resource Tracking. The next set of examples reflect various patterns of specifying and verifying *resources* and demonstrate that Implicit Refinement Types enable lightweight verification of these patterns. The examples show how MIST handles specification and automatic verification when dealing with resources such as the state of the heap and quantitative resource usage. In §2.4 we saw how Implicits enable specifying the access-control API from Figure 3; the ACCESSCONTROL example verifies clients of this API. In contrast, TICK (as shown in §2.6) follows Handley et al. [19] in defining an applicative functor that tracks quantitative resource usage. The key distinction from Handley et al. [19] is that the resource count exists only at the type level. This example generalizes: we can port any of the *intrinsic verification* examples from that work to use implicit parameters instead of explicitly passing around resource bounds.

LINEARDSL embeds a simple linearly typed DSL in MIST. It allows us to embed linear terms in MIST, with linear usage of variables statically checked by our refinement type system. The syntactic constructs of this DSL are smart constructors that take the typing environments as implicit parameters, enforcing the appropriate linear typing rules.

```
incr :: [n:Int] → (Int → SInt n) → SInt (n + 1)
incr f = (f 0) + 1

test1 :: SInt 11                    test2 :: m:Int → SInt (m + 1)
test1 = incr (\x → 10)              test2 m = incr (\x → m)
```

Figure 10 A higher order increment function.

Mist enables specification and automatic verification of this diverse range of examples, and in fact only requires the theory-agnostic instantiation procedure to find the correct implicit instantiations. This demonstrates that IRTs enable lightweight verification of a variety of higher-order programs, and that they are widely useful even without a domain specific solver or heavyweight synthesis algorithm.

7.2 Q2: Expressivity

We compare the expressivity of Implicit Refinement Types as implemented in Mist to the assertion-based verification of MoCHi and the implicit parameters of F*. First, merely the fact that we allow users to explicity write specifications using implicit parameters allows us to write specifications we otherwise could not. In particular, this is witnessed by the tick and linearDSL examples that specify resource tracking, a non-functional property. MoCHi cannot express the specifications for these because there is no combination of program variables that computes the (non-functional) *usage* properties used in these specifications.

Second, Implicit Pair Types add expressivity by allowing us to write specifications against choices made *internal* to functions that we wish to reason about. For example, repeat uses implicit pairs to specify a loop invariant. This example can't be done with F*'s implicits, as, absent implicit pairs, we cannot bind to the return value of the loop body. As a result, in F* we cannot verify this example with implicits alone: we would have to bring in additional features such as the Dijkstra monad [38].

Similarly, *none* of the protocol state machine examples can be encoded with implicit functions alone. Implicit pairs are required to write specifications against choices made by other actors on the protocol channel. As a result, these examples also cannot be encoded in F*'s implicits. These state machine specifications cannot even be expressed in MoCHi as they crucially require access to ghost state, which MoCHi does not support.

7.3 Q3: Flexibility

We compare the flexibility that our refinement-integrated approach gives us to solve for implicit parameters relative to that of other systems. We focus on the features of our semantics and our abstract solving algorithm from Section 6.1 independently of the choice of quantifier elimination procedure, which we examined above. We illustrate these differences with several examples. incr is the program from Figure 10. In MoCHi the specification is given by assertions that `test1` and `test2` are equal to `11` and `m + 1` respectively. incrState generalizes incr to track the integer in the singleton state monad instead of a closure. sum is similar to `incr` except that it takes two implicit arguments and two function arguments, returning the sum of the two returned values:

```
sum :: [n, m] → (Int → SInt n) → (Int → SInt m) → SInt (n + m)
sum f g = (f 0) + (g 0)

test :: SInt 11
test = sum (\x → 10) (\y → 1)
```

All three tools can specify the program. MIST and MOCHI successfully instantiate the implicit parameters needed for verification. D2 is an example from the MOCHI [41] benchmarks that loops a nondeterministic number of times and adds some constant each time. Unlike MIST, MOCHI is unable to solve ACCESSCONTROL, as CEGAR is notoriously brittle on properties over the theory of set-operations.

Comparsion to MoCHi. MOCHI fails to automatically verify INCR, but it succeeds if either `test1` or `test2` appear individually. This is partly due to the unpredictability of MOCHI's CEGAR loop [21] (as it succeeds in verifying INCRSTATE), and partly due to the fact that MOCHI attempts to infer specifications (and implicit arguments) *globally*, which can lead to the anti-modular behavior seen here. In contrast, by introducing implicit arguments as a user-level specification technique, MIST permits modular verification: `test1` and `test2` generate two separate quantifier instantiation problems that MIST solves locally.

In D2, MIST *fails* to solve the constraints generated by implicit instantiation as they involve systems of inequalities, which our theory-agnostic quantifier elimination does not attempt. However, MIST captures the full set of constraints and could, with a theory specific solver like MOCHI's, verify D2. Concretely, D2 yields a constraint of the form $\exists x.true \Rightarrow x > 3$. MIST could discharge this obligation by instantiating the abstract algorithm of Section 6 with either the solver from MOCHI [41] or EHSF[5] instead of the theory-agnostic one above.

Comparsion to Unification. The results of the comparison to F*'s unification-based implicits are summarized in Table 1. Implicit Refinements are more flexibile as F* attempts to solve the implicits purely through unification, *i.e.* without accounting for refinements. In contrast, MIST generates subtyping (implication) constraints that are handled separately from unification. When the unification occurs under a type constructor, then unification can succeed. For example, F* verifies INCRSTATE because elevating the ghost state to a parameter of the singleton state type constructor makes it "visible" to F*'s unification algorithm. However, if there is *no* type constructor to guide the unification, F* must rely on higher-order unification, which is undecidable in general and difficult in practice. For this reason, in the INCR example from Figure 10, F* fails to instantiate the implicit arguments needed to verify `test1` and `test2`, even if we aid it with precise type annotations on `\x → 10` and `\x → m`, and F* fails to verify the SUM example for the same reason. By contrast, MIST can take advantage of the refinement information to solve for the implicit parameters. Finally, F*'s unification fundamentally cannot handle an example like MOCHI's D2, as unification will not be able to find an implicit instantiation when it is only constrained by inequalities.

8 Related Work

Verification of Higher-Order Programs. As discussed in Section 7, F* [37] is another SMT-aided higher-order verification language. Its main support for verifying stateful programs is baked in via Dijkstra Monads [38]. When it comes to verifying higher-order stateful programs, Dijkstra Monads enable F* to scale automatic verification up to complex invariants. However, like other two-state specification techniques, they are not on their own flexible enough to handle higher-order computations such as `mapA`, when higher-order computations are composed in richer ways than simple Kleisli composition. The key difference is that Implicit Refinement Types work for both classic two-state specifications *and other* use-cases where two-state specifications prove cumbersome – IRTs add value even to a system that already includes

two-state specifications by allowing the necessary extra parameter of functions like `mapA` to be instantiated automatically. Moreover, IRTs do so while maintaining the key properties of refinement type systems: (1) SMT-driven automatic verification and (2) refinement type specifications are added atop an existing program without requiring code changes. In this way IRTs also differ from systems that use "full-strength" two-state specifications with dependent types such as VST[3] or Bedrock[9].

We have started implementing implicit refinement types in LiquidHaskell, a very similar system to Mist that includes both manual [45] and automated [42] facilites for higher order verification.

F* has both implicit parameters and refinement types, but F*'s implicits have no formal description, and instantiation is independent of refinement information. On the other hand, we lack first-class invariants, but future work may be able to alleviate this by abstracting over refinements *a la* Vazou et al. [43].

Dafny [24] supports specifications with ghost variables, but the user must explicitly craft triggers to perform quantifier instantiation when the backing SMT solver's heuristics cannot, and must manually pass and update ghost variables.

Implicit Parameter Instantiation. Since finding an instantiation of implicit parameters is, in general, undecidable, different systems make different tradeoffs: While Haskell and Scala only perform type-directed lookup, Idris [8] resolves implicits via first-order unification, with a default value or by a fixed-depth enumerative program synthesis. This form of implicit resolution system can be very powerful, but can't be used in conjunction with solver-automated reasoning so, for example, Idris would not be able to handle our sum example which requires automated reasoning about arithmetic. Agda [30] combines unification with a second kind of implicit parameter, *instance arguments* [14], that uses a separate, specialized, implicit resolution mechanism for implicit arguments that relate to typeclasses. Coq [39] uses a mechanism called canonical structures [27], which uses a programmable hint system, but is intimately tied to the specifics of Coq's implementation, and its use for implicit parameter instantiation lacks a formal description, as Devriese et al. [14] lament.

As Devriese et al. [14] note, the complexity of dependent type systems make even achieving similar functionality as in Haskell and Scala a significant task. These implicit parameters are designed to accomplish similar tasks as in the non-dependent Haskell and Scala: automating the instantiation of repetitive arguments or automatically searching the context for relevant arguments. Refinement types allow us to sidestep this issue as the base type system can separately use implicit parameters to automate typeclasses or other programming tasks, allowing our technique of Implicit Refinement Types to entirely focus on using dependent type information. Here, this means focusing on allowing us to use an SMT to simplify ghost specifications without worrying about interactions with the base type system.

While there is much work on implicit parameters for *dependent types*, we believe that we provide the first formal description of a system combining implicit parameters with refinement types. F* is the only other example of a language that has implicit parameters and refinement types, but we discuss how F*'s implicits cannot take advantage of refinement type information in §7.

Horn Constraints. Horn Clauses have emerged as a lingua franca of verification tools [6] as they offer a straightforward encoding of assertions. Z3 [13] includes a fixpoint solver for Horn Constraints including many quantifier elimination heuristics. Rybalchenko et al. [5] present a semi-decision procedure for solving existential Horn clauses using a template-based CEGAR

loop. Cosman and Jhala [10] use NNF Horn constraints to preserve *scope*. We extend this framework to synthesize refinements for implicit program variables that are *existentially* quantified, in addition to the usual *universally* quantified binders.

Unno et al. [41] show that it is sufficient to add one extra parameter for each higher order argument to any given function to achieve complete higher-order verification with first-order refinements. Their treatment utilizes more automation, *e.g.* interpolation and Farkas' lemma, but is limited to arithmetic specifications, precluding programs manipulating data structures like sets and maps, as demonstrated in §7.

References

1 Retrieving paginated results - AWS SDK for java version 2, 2019. URL: `https://docs.aws.amazon.com/sdk-for-java/v2/developer-guide/examples-pagination.html`.

2 Martín Abadi and Leslie Lamport. The existence of refinement mappings. In *Proceedings of the Third Annual Symposium on Logic in Computer Science (LICS 1988), Edinburgh, Scotland, UK, July 5-8, 1988*, pages 165–175. IEEE Computer Society, 1988. `doi:10.1109/LICS.1988.5115`.

3 Andrew W. Appel. Verified software toolchain. In Alwyn Goodloe and Suzette Person, editors, *NASA Formal Methods - 4th International Symposium, NFM 2012, Norfolk, VA, USA, April 3-5, 2012. Proceedings*, volume 7226 of *Lecture Notes in Computer Science*, page 2. Springer, 2012. `doi:10.1007/978-3-642-28891-3_2`.

4 Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. Refinement types for secure implementations. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium, CSF 2008, Pittsburgh, Pennsylvania, USA, 23-25 June 2008*, pages 17–32. IEEE Computer Society, 2008. `doi:10.1109/CSF.2008.27`.

5 Tewodros A. Beyene, Corneliu Popeea, and Andrey Rybalchenko. Solving existentially quantified horn clauses. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 869–882. Springer, 2013. `doi:10.1007/978-3-642-39799-8_61`.

6 Nikolaj Bjørner, Arie Gurfinkel, Kenneth L. McMillan, and Andrey Rybalchenko. Horn clause solvers for program verification. In Lev D. Beklemishev, Andreas Blass, Nachum Dershowitz, Bernd Finkbeiner, and Wolfram Schulte, editors, *Fields of Logic and Computation II - Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday*, volume 9300 of *Lecture Notes in Computer Science*, pages 24–51. Springer, 2015. `doi:10.1007/978-3-319-23534-9_2`.

7 Edwin Brady. State machines all the way down. Draft, 2016. URL: `https://www.idris-lang.org/drafts/sms.pdf`.

8 Edwin C. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23(5):552–593, 2013. `doi:10.1017/S095679681300018X`.

9 Adam Chlipala. The bedrock structured programming system: combining generative metaprogramming and hoare logic in an extensible program verifier. In Greg Morrisett and Tarmo Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP 2013, Boston, MA, USA - September 25 - 27, 2013*, pages 391–402. ACM, 2013. `doi:10.1145/2500365.2500592`.

10 Benjamin Cosman and Ranjit Jhala. Local refinement typing. *Proceedings of the ACM on Programming Languages*, 1(ICFP):26:1–26:27, 2017. `doi:10.1145/3110270`.

11 Bruno C. d. S. Oliveira, Adriaan Moors, and Martin Odersky. Type classes as objects and implicits. In William R. Cook, Siobhán Clarke, and Martin C. Rinard, editors, *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*, pages 341–360. ACM, 2010. `doi:10.1145/1869459.1869489`.

12 Bruno C. d. S. Oliveira, Tom Schrijvers, Wontae Choi, Wonchan Lee, and Kwangkeun Yi. The implicit calculus: a new foundation for generic programming. In Jan Vitek, Haibo Lin, and Frank Tip, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2012, Beijing, China - June 11 - 16, 2012*, pages 35–44. ACM, 2012. doi:10.1145/2254064.2254070.

13 Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008. doi:10.1007/978-3-540-78800-3_24.

14 Dominique Devriese and Frank Piessens. On the bright side of type classes: instance arguments in agda. In Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy, editors, *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*, pages 143–155. ACM, 2011. doi:10.1145/2034773.2034796.

15 Jana Dunfield and Neelakantan R. Krishnaswami. Complete and easy bidirectional typechecking for higher-rank polymorphism. In Greg Morrisett and Tarmo Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP 2013, Boston, MA, USA - September 25 - 27, 2013*, pages 429–442. ACM, 2013. doi:10.1145/2500365.2500582.

16 Burak Emir, Andrew Kennedy, Claudio V. Russo, and Dachuan Yu. Variance and generalized constraints for c# generics. In Dave Thomas, editor, *ECOOP 2006 - Object-Oriented Programming, 20th European Conference, Nantes, France, July 3-7, 2006, Proceedings*, volume 4067 of *Lecture Notes in Computer Science*, pages 279–303. Springer, 2006. doi:10.1007/11785477_18.

17 Cédric Fournet, Markulf Kohlweiss, and Pierre-Yves Strub. Modular code-based cryptographic verification. In Yan Chen, George Danezis, and Vitaly Shmatikov, editors, *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*, pages 341–350. ACM, 2011. doi:10.1145/2046707.2046746.

18 Arjun Guha, Matthew Fredrikson, Benjamin Livshits, and Nikhil Swamy. Verified security for browser extensions. In *32nd IEEE Symposium on Security and Privacy, S&P 2011, 22-25 May 2011, Berkeley, California, USA*, pages 115–130. IEEE Computer Society, 2011. doi:10.1109/SP.2011.36.

19 Martin A. T. Handley, Niki Vazou, and Graham Hutton. Liquidate your assets: reasoning about resource usage in liquid haskell. *Proceedings of the ACM on Programming Languages*, 4(POPL):24:1–24:27, 2020. doi:10.1145/3371092.

20 Christian Heinlein. Implicit and dynamic parameters in C++. In David E. Lightfoot and Clemens A. Szyperski, editors, *Modular Programming Languages, 7th Joint Modular Languages Conference, JMLC 2006, Oxford, UK, September 13-15, 2006, Proceedings*, volume 4228 of *Lecture Notes in Computer Science*, pages 37–56. Springer, 2006. doi:10.1007/11860990_4.

21 Ranjit Jhala and Kenneth L. McMillan. A practical and complete approach to predicate refinement. In Holger Hermanns and Jens Palsberg, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 12th International Conference, TACAS 2006 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 25 - April 2, 2006, Proceedings*, volume 3920 of *Lecture Notes in Computer Science*, pages 459–473. Springer, 2006. doi:10.1007/11691372_33.

22 Ming Kawaguchi, Patrick Maxim Rondon, and Ranjit Jhala. Type-based data structure verification. In Michael Hind and Amer Diwan, editors, *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, pages 304–315. ACM, 2009. doi:10.1145/1542476.1542510.

23 Kenneth Knowles and Cormac Flanagan. Type reconstruction for general refinement types. In *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Held as Part of the Joint European Conferences on Theory and Practics of Software,*

ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings, pages 505–519, 2007. doi:10.1007/978-3-540-71316-6_34.

24 K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, 2010. doi:10.1007/978-3-642-17511-4_20.

25 Jeffrey R. Lewis, John Launchbury, Erik Meijer, and Mark Shields. Implicit parameters: Dynamic scoping with static types. In Mark N. Wegman and Thomas W. Reps, editors, *POPL 2000, Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Boston, Massachusetts, USA, January 19-21, 2000*, pages 108–118. ACM, 2000. doi:10.1145/325694.325708.

26 Tomer Libal. The unification algorithm, 2017. URL: https://github.com/FStarLang/FStar/wiki/The-unification-algorithm.

27 Assia Mahboubi and Enrico Tassi. Canonical structures for the working coq user. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *Interactive Theorem Proving*, pages 19–34, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. doi:10.1007/978-3-642-39634-2_5.

28 Scott Moore, Christos Dimoulas, Dan King, and Stephen Chong. SHILL: A secure shell scripting language. In Jason Flinn and Hank Levy, editors, *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2014, Broomfield, CO, USA, October 6-8, 2014*, pages 183–199. USENIX Association, 2014. doi:10.5555/2685048.268506.

29 Aleksandar Nanevski, Greg Morrisett, Avraham Shinnar, Paul Govereau, and Lars Birkedal. Ynot: dependent types for imperative programs. In James Hook and Peter Thiemann, editors, *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, pages 229–240. ACM, 2008. doi:10.1145/1411204.1411237.

30 Ulf Norell. Dependently typed programming in agda. In Andrew Kennedy and Amal Ahmed, editors, *Proceedings of TLDI 2009: 2009 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Savannah, GA, USA, January 24, 2009*, pages 1–2. ACM, 2009. doi:10.1145/1481861.1481862.

31 Susan S. Owicki and David Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6:319–340, 1976. doi:10.1007/BF00268134.

32 Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Transactions on Programming Language and Systems*, 22(1):1–44, 2000. doi:10.1145/345099.345100.

33 Jon Postel. Transmission control protocol. STD 7, RFC Editor, September 1981. URL: http://www.rfc-editor.org/rfc/rfc793.txt.

34 Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. Liquid types. In Rajiv Gupta and Saman P. Amarasinghe, editors, *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, pages 159–169. ACM, 2008. doi:10.1145/1375581.1375602.

35 Alceste Scalas, Nobuko Yoshida, and Elias Benussi. Verifying message-passing programs with dependent behavioural types. In Kathryn S. McKinley and Kathleen Fisher, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 502–516. ACM, 2019. doi:10.1145/3314221.3322484.

36 Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. Secure distributed programming with value-dependent types. In Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy, editors, *Proceeding of the 16th ACM SIGPLAN International Conference on Functional Programming*, pages 266–278. ACM, 2011. doi:10.1145/2034773.2034811.

37 Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean Karim Zinzindohoue, and Santiago Zanella Béguelin. Dependent types and multi-monadic effects in F. In Rastislav Bodík and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 256–270. ACM, 2016. doi:10.1145/2837614.2837655.

38 Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. Verifying higher-order programs with the dijkstra monad. In Hans-Juergen Boehm and Cormac Flanagan, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2013, Seattle, WA, USA, June 16-19, 2013*, pages 387–398. ACM, 2013. doi:10.1145/2491956.2491978.

39 The Coq Development Team. The Coq Reference Manual, 2009.

40 Anish Tondwalkar, Matthew Kolosick, and Ranjit Jhala. Refinements of futures past: Higher-order specification with implicit refinement types (extended version), 2021. arXiv:2105.01954.

41 Hiroshi Unno, Tachio Terauchi, and Naoki Kobayashi. Automating relatively complete verification of higher-order functional programs. In Roberto Giacobazzi and Radhia Cousot, editors, *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2013, Rome, Italy - January 23 - 25, 2013*, pages 75–86. ACM, 2013. doi:10.1145/2429069.2429081.

42 Niki Vazou, Alexander Bakst, and Ranjit Jhala. Bounded refinement types. In Kathleen Fisher and John H. Reppy, editors, *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*, pages 48–61. ACM, 2015. doi:10.1145/2784731.2784745.

43 Niki Vazou, Patrick Maxim Rondon, and Ranjit Jhala. Abstract refinement types. In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, volume 7792 of *Lecture Notes in Computer Science*, pages 209–228. Springer, 2013. doi:10.1007/978-3-642-37036-6_13.

44 Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon L. Peyton Jones. Refinement types for haskell. In Johan Jeuring and Manuel M. T. Chakravarty, editors, *Proceedings of the 19th ACM SIGPLAN International Conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, pages 269–282. ACM, 2014. doi:10.1145/2628136.2628161.

45 Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G. Scott, Ryan R. Newton, Philip Wadler, and Ranjit Jhala. Refinement reflection: complete verification with SMT. *Proceedings of the ACM on Programming Languages*, 2(POPL):53:1–53:31, 2018. doi:10.1145/3158141.

46 Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad-hoc. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*, pages 60–76. ACM Press, 1989. doi:10.1145/75277.75283.

47 Leo White, Frédéric Bour, and Jeremy Yallop. Modular implicits. In Oleg Kiselyov and Jacques Garrigue, editors, *Proceedings ML Family/OCaml Users and Developers workshops, ML/OCaml 2014, Gothenburg, Sweden, September 4-5, 2014*, volume 198 of *EPTCS*, pages 22–63, 2014. doi:10.4204/EPTCS.198.2.

48 Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In Jack W. Davidson, Keith D. Cooper, and A. Michael Berman, editors, *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI), Montreal, Canada, June 17-19, 1998*, pages 249–257. ACM, 1998. doi:10.1145/277650.277732.