`mist`: Refinements of Futures Past (Artifact)

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

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

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

Abstract

`mist` is a tiny language for teaching and experimenting with refinement types, in the style of LiquidHaskell. We use it as a platform for experimenting with and as a demonstration of implicit refinement types as presented in the ECOOP21 paper *Refinements of Futures Past: Higher-Order Specification with Implicit Refinement Types*. We start with the parser and AST we use to teach our undergradute compilers class, and layer upon it a refinement type checker directly translated from the typing rules presented in that paper, which produces constraints that are solved with the `liquid-fixpoint` horn clause solver.

We present source code and binaries for `mist` in a container image that includes installations of the competing tools we compare to.

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/DARTS.7.2.3

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

Related Article Anish Tondwalkar, Matthew Kolosick, and Ranjit Jhala, "Refinements of Futures Past: Higher-Order Specification with Implicit Refinement Types", in 35th European Conference on Object-Oriented Programming (ECOOP 2021), LIPIcs, Vol. 194, pp. 18:1–18:29, 2021.
`https://doi.org/10.4230/LIPIcs.ECOOP.2021.18`

Related Conference 35th European Conference on Object-Oriented Programming (ECOOP 2021), July 12–16, 2021, Aarhus, Denmark (Virtual Conference)

1 Scope

This artifact allows users to reproduce the results of the experiments presented in the ECOOP21 paper and to test the system against examples and benchmarks of their own conception.

2 Content

The artifact package is a docker image that includes:

- The source code for the mist programming language
- Binary releases of: `mist`, competing tools `fstar` and `mochi`, as well as all of their dependencies.
- A tutorial on how to use to artifact and on programming in the mist programming language

3 Getting the artifact

The artifact endorsed by the Artifact Evaluation Committee is available free of charge on the Dagstuhl Research Online Publication Server (DROPS). In addition, the artifact is also available as a git repository at `https://github.com/ucsd-progsys/mist`. The version of the code held in the archive corresponds to the `ecoop21` branch.

4 Tested platforms

This artifact with developed and tested on Ubuntu 20.04, and tested on Mac OS. Requires at least 4GB of RAM.

5 License

The artifact is available under an MIT license.

6 MD5 sum of the artifact

aa4c6b98c4c7b17c67c9369ae9e3aff7

7 Size of the artifact

7.54 GiB

8 Initial build, install, and running all tests

You can use the Docker image or install `mist` manually. The Docker image also includes the tools we compare against.

8.1 Using `docker`

Windows and Mac users: Make sure your docker container has at least 4GB of RAM

The following command will download an image containing `mist`, `fstar`, and `mochi`, run the full `mist` test suite, and then drop you into an interactive shell at the root of the `mist` code repository.

```
$ docker run -it atondwal/mist
```

If you want to skip the test suite, instead run

```
$ docker run -it atondwal/mist /bin/bash
```

You can then (re)run all of the tests in the `tests/` directory (perhaps after editing some) at any time by running

```
$ stack test
```

8.1.1 Juggling containers

You can use `docker ps` to see the running container and open another shell to it using `docker exec`, e.g.:

```
$ docker ps
CONTAINER ID      IMAGE             STATUS            NAMES
696b2221e3ad      atondwal/mist     Up 45 seconds     vibrant_leavitt
$ docker exec -it vibrant_leavitt bash
ecoop21@696b2221e3ad:~/mist$
```

You can use `docker start` to restart exited containers

```
$ docker ps -a
CONTAINER ID      IMAGE             STATUS                      NAMES
696b2221e3ad      atondwal/mist     Exited (137) 5 seconds ago vibrant_leavitt
$ docker start vibrant_leavitt
vibrant_leavitt
$ docker exec -it vibrant_leavitt bash
ecoop21@696b2221e3ad:~/mist$
```

8.2 Manually

You'll need git, z3 version 4.8.10, and stack.

```
$ git clone -b ecoop21 --recursive https://github.com/ucsd-progsys/mist
$ cd mist
$ stack install
$ export PATH=$HOME/.local/bin/:$PATH
```

You can then run the full `mist` test suite (which is located in the `tests/` directory).

```
$ stack test
```

9 Running specific tests

You can run a specific test by calling mist on the test file, e.g.

```
$ mist tests/pos/incrState.hs
```

If you're using the docker image, you can also run tests for `fstar`:

```
$ fstar fstar-tests/incrState.fst
```

10 Benchmarks from the paper

Here's a table of where you can find each of the tests described in the paper:

Name	Mist test (tests/pos/)	Mochi (mochi-tests/)	Fstar (fstar-tests/)
incr	incr00.hs	incr00.ml	incr.fst
sum	sum.hs	sum.ml	sum.fst
repeat	repeat.hs	repeat.ml	x
d2	mochi-app-lin-ord2.hs	d2.ml	mochi-d2.fst
incrState	incrStatePoly.hs	incrState.ml	incrState.fst
accessControl	acl.hs	acl.ml	accessControl.fst
tick	tick-append.hs	x	tick.fst
linearDSL	linearTypes.hs	x	linearDSL.fst
pagination	paginationTokens.hs	x	x
login	idr_login.hs	x	x

Name	Mist test (tests/pos/)	Mochi (mochi-tests/)	Fstar (fstar-tests/)
twophase	twoPhaseCommit.hs	x	x
ticktock	ticktock3.hs	x	x
tcp	tcp_client.hs	x	x

As in the paper, an `x` indicates that the specification cannot be directly expressed with that tool.

Unfortunately, the version of MoCHi we compare against was made available as a web demo that is no longer functional. Since then, there has been a source release of MoCHi, but it does not support the `-relative-complete` verification mode that we compared against in our paper. We include a build of the latest version of MoCHi anyways, in case you want to play with it yourself and get an idea of how it works:

```
$ mochi mochi-tests/incrState.ml
```

11 A quick tutorial in writing mist

A note about UX: We demonstrate the ability of our type system to localize error messages in this prototype, but when it comes to the parser, we favor an easy to modify and understand grammar over one that provides the best user experience. As such...

When experimenting with `mist`, we recommend starting with one of the known working test cases, and then expanding on it to achieve the desired result, rather than starting from scratch in an empty text file. In this short tutorial we will take the same approach, starting from a minimal test case and building up to the pagination example from the ECOOP21 paper that demonstrates both implicit refinement function types and pair types.

11.1 How to read this tutorial

We recommend reading the pdf version of this tutorial as it is the easiest to read, but we also recommend keeping open a copy of the markdown source (`~/mist/README.md`) in your text editor as you follow along, as the markdown source includes the location of each snippet as range of lines in a test file, so you can open, edit, and rerun those tests yourself.

We recommend running a continuous build in a terminal while you experiment with a mist file. For example, this sets up `entr` to run `mist` on a file every time it gets written (whether or not it's a `mist` file).

```
$ find tests | entr mist /_
(in another window - see above for docker instructions)
$ vim tests/.../mytest.hs
```

Bits of syntax that are potential sources of confusion or frustration are highlighted in grey boxes. If you're struggling to make your code parse, checking to see if you've stepped on one of these Legos is a good place to start.

11.2 Refinement Types

We start from an extremely simple example that demonstrates the concrete semantics of mist's refinement type system.

```
1 twelve :: { v : Int  | v == 12 }
2 twelve = 12
```

Here, we have a top-level binder for the constant `int`. Each top level binder includes a type signature (line 1), and a body (line 2). The body of `int` simply states that it's equal to the integer constant `12`. This type signature is a minimal example of a refinement type: we refine the base type `Int`, binding its values to `v`, and taking the quotient of this type by the proposition `v == 12`. This results in a singleton type that checks against the body of `twelve`.

```
$ mist tests/pos/Int00.hs
SAFE
```

If we had used a different value in the type and body:

```
1 twelve :: { v : Int  | v == 14 }
2 twelve = 12
```

We'd see a type error:

```
$ mist tests/neg/Int01.hs
Working 150% [==============================================================]
Errors found!
tests/neg/Int01.hs:2:7-9: Expected (VV##0 == 14) :

         2|  int = 12
                 ^^^
```

11.3 Functions and polymorphism

We can extend this to writing functions in `mist`:

```
1 incr :: x:Int -> {v:Int | v == x + 1}
2 incr = \x -> x + 1
3
4 moo :: {v:Int | v == 8}
5 moo = incr 7
```

This program checks that `incr`ementing 7 results in 8.

Here, the binder `x:Int` binds `x` in the type on the right-hand side of `->`. Similarly, at the value level, `\` denotes a lambda.

Functions can also be polymorphic:

```
7 id :: rforall a. {v:a | True} -> {v:a | True}
8 id = \x -> x
9
10 bar :: {v:Int | v == 8}
11 bar = incr (id 7)
```

```
$ mist tests/pos/Inc02.hs
SAFE
```

> All function applications that are not directly under a binder or a function abstraction should be enclosed in parentheses.

Here, `rforall` denotes that the function `id` is *refinement polymorphic* in the type variable `a`. That is, `a` stands in for any *refined* type, so we know that the result of applying `id` to any value will always result in a value of the same refinement type; i.e. one for which all the same propositions are true. The only function of this type is `id`.

Later we will also see `forall`, which allows functions to be polymorphic over base types.

11.4 Implicit function types

We're ready for our first example of a feature introduced in this paper! We write an implicit function type the same way as a normal function, but using the squiggly arrow `~>` instead of the straight arrow `->`:

```
1 incr :: n:Int ~> (Int -> { v : Int | v == n }) -> { v : Int | v == n + 1 }
2 incr = \f -> (f 0) + 1
3
4 test1 :: { v : Int | v == 11 }
5 test1 = incr (\x -> 10)
6
7 test2 :: m:Int -> { v : Int | v == m+1 }
8 test2 = \mv -> incr (\x -> mv)
```

```
$ mist tests/pos/incr00.hs
SAFE
```

> Note the parentheses around `(f 0)` — there are no precedence rules for infix primitives.

Given a constant function, `incr` increments the result. This is straightforward at the value level, but encoding it at the type level requires the use of implicit parameters. Here, `n` in bound at the type level, but has no corresponding binder at the value level in the surface syntax. The body of the function must typecheck for all values of `n`, but each call to the function need only be valid for some particular choice of `n`. `n` is picked at the call site by the implicit instantiation algorithm for refinement types described in the paper, such that the function application typechecks.

Here, for the call to `incr` on line 5 inside `test1`, `n` takes on the value 10, and on line 8, it takes on the value `mv`.

11.5 Datatypes, axioms, and measures

Mist supports user-defined datatypes by axiomatising their constructors. In this section we're going to demonstrate specification and verification with the `List` datatype, which in Haskell one might write:

```
data List a = Nil | Cons a (List a)
```

11.5.1 Datatypes

In mist, `List a` is spelled `List >a`

There are two things of note here:

1. As in Haskell, Mist datatypes are written in TitleCamelCase.
2. Unlike Haskell, datatypes carry *variance annotations* with them that tell you if they're co- or contra-variant in a given argument. Having these around can be helpful when you're debugging or reading code with complex subtyping relationships.

Here, the variance annotation `>` indicates that `a` appears covariantly in `List` (that is, `List` contains things that are subtypes of `a`). If it appeared contravariantly, we would have written `List <a` (a `List` of supertypes of `a`). Or in other words, a `List >a` behaves like a `List` that might produce `a`s when it is used up, whereas a `List <a` behaves as a list that might consume `a`s to use up.

This notation is intended to evoke a function arrow `->`: Just as you can use a function that *returns* any subtype of the type you need, and that *accepts* any supertype of the arguments you have, if you're a type variable on the pointy end of the variance annotation (or function arrow) you're a covariant type variable, and if you're on the other end you're contravariant.

If you try to pass a `List >a` as a `List <a`, that is a (base/unrefined) type error — variance annotations are essentially a part of the type name.

You do not need to declare datatypes before using them in type signatures.

Some such datatypes (`Int`, `Bool`, `Set`, and `Map`) have special meaning when used in types, as they come with primitives (such as `+`, which we saw above) that have meaning to the solver's theories of arithmetic, sets, maps, etc.

11.5.2 Axioms

Mist relies on axioms to introduce data constructors. An axiom in Mist is written with "`as`" (assumed types) instead of "`::`" (checked types):

```
exFalsoQuodlibet as rforall a. False -> a
exFalsoQuodlibet = ...
```

Whatever we put for `...` is taken to be the witness of the axiom, and executed when the axiom is used in code that is run.

> You need to provide a body for every binding, axioms or otherwise.

To use the `List` datatype, we need constructors, and projections from these constructor (or induction principles, but let's keep it simple for the tutorial). To introduce axioms for each of these, we write something like

```
nil as forall a. List >a
nil = ...
cons as forall a. a -> List >a -> List >a
cons = ...
first as forall a. List >a -> a
first = ...
rest as forall a. List >a -> List >a
rest = ...
```

where `...` can be e.g. Boehm-Beraraducci (1985) encoding of constructors and projection operators, but since we're focused on testing the typechecker here, we generally set them equal to 0 as the witnesses to axioms don't matter so far as the typechecker is concerned.

We can use axiomatized constructors to define a datatype `Lin` which is the type of terms of a linear DSL. Here, we use `Set` primitives.

```
4  var as x:Int -> (Lin >{v:Set >Int | v = setPlus emptySet x})
```

`var` constructs a term that is a variable mention. It checks that the variable is in the environment. $\frac{x \in \Gamma}{\Gamma \vdash \texttt{var}\ x}$

```
7  fun as env:(Set >Int) ~>
8    n:{v:Int | (v ∈ env) ≠ True}
9    -> (Lin >{v:Set >Int | v = setPlus env n})
10   -> (Lin >{v:Set >Int | v = env})
```

`fun` constructs a lambda term, while checking that the variable it binds is fresh. $\frac{x \notin \Gamma \quad \Gamma \cup \{x\} \vdash e}{\Gamma \vdash \texttt{fun}\ x\ e}$

> It may not always be obvious when you need parenthesis around type constructor applications. Some rules of thumb: parenthesize them on both sides of `->`, but in the left-hand side of a refinement type (the binder), "`(`" may not follow "`:`"

> Note that the not-equals operator is the unicode symbol, not a multi-character sigil — infix operators in Mist are single characters.

```
13 app as env1:(Set >Int) ~> env2:{v:Set >Int | env1 ∩ v = emptySet} ~>
14   (Lin >{v:Set >Int | v = env1})
15   -> (Lin >{v:Set >Int | v = env2})
16   -> (Lin >{v:Set >Int | v = env1 ∪ env2})
```

`app` applies a function to a value, checking that no variable is used more than once in either argument. $\frac{\Gamma_1 \cap \Gamma_2 = \emptyset \quad \Gamma_2 \vdash e \quad \Gamma_1 \vdash f}{\Gamma_1 \cup \Gamma_2 \vdash \texttt{app}\ f\ e}$

```
19 typecheck as (Lin >{v:Set >Int | v = emptySet}) -> (Lin >(Set >Int))
```

`typecheck` simply checks that a term is closed $\frac{\emptyset \vdash e}{\vdash e}$

```
25 program2 :: Lin >(Set >Int)
26 program2 = typecheck (fun 1 (fun 2 (app (var 1) (var 2))))
```

```
$ mist tests/pos/linearTypes.hs
SAFE
```

11.6 State

We can define a State Monad datatype! See section 2.3 of the paper for more explanation. `put` takes a world (called `wp`), `put` updates the state to one where the state of the world is now `wp`.

```
20 put as wp:Int -> ST <Int >{p:Int|p==wp} >Unit
```

`get` takes a boolean and ignores it, then leaves the state of the world unchanged, but returns its value in the `ST` monad.

```
17 get as wg:Int ~> Bool -> ST <{gi:Int|gi==wg} >{go:Int|go==wg} >{gr:Int|gr==wg}
```

And then we have the standard monadic interface:

```
-- Monadic Interface
ret as rforall a. wr:Int ~> x:a -> ST <{ri:Int|ri==wr} >{ro:Int|ro==wr} >a

bind as rforall a, b. w1:Int ~> w2:Int ~> w3:Int ~>
  (ST <{v:Int|v==w1} >{v:Int|v==w2} >a)
  -> (unused:a -> ST <{v:Int|v==w2} >{v:Int|v==w3} >b)
  -> ST <{v:Int|v==w1} >{v:Int|v==w3} >b
```

Using this, we can verify a more stateful version of the incr example from before.

```
incr :: i:Int ~> ST <{v:Int|i==v} >{w:Int|w==i+1} >Unit
incr = bind (get True) (\x -> put (x+1))

$ mist tests/pos/incrState.hs
SAFE
```

Going forward, however, we're going to use a more polymorphic definition of state:

```
bind :: rforall a, b, p, q, r.
  ST <p >q >a ->
  (x:a -> ST <q >r >b) ->
  ST <p >r >b
bind = undefined

pure :: rforall a, p. x:a -> ST <p >p >a
pure = undefined

thenn :: rforall a, b, p, q, r.
  ST <p >q >a ->
  ST <q >r >b ->
  ST <p >r >b
thenn = \f g -> bind f (\underscore -> g)

fmap :: rforall a, b, p, q.
  (underscore:a -> b) ->
  ST <p >q >a ->
  ST <p >q >b
fmap = \f x -> bind x (\xx -> pure (f xx))
```

If a function type signature is failing to parse, try assigning a name to the argument (e.g. `x:Int ->` instead of `Int ->`).

11.7 Implicit pair types

Next, we demonstrate how to use another core feature unique to Mist: implicit pair types as described in the paper. In the paper the syntax is $[n : Int].—$, but in the implementation, we use `exists n. -`. Consider iterating through an infinite stream of token handlers. The API for getting the next token is:

```
nextPage as
  token:{v: Int | v ≠ done} ->
  (exists tok:Int.
    (ST <{v:Int | v = token}
        >{v:Int | (v = tok) /\ (v ≠ token)}
        >{v:Int | v = tok}))
```

> Remember to parenthesize both sides of conjunctions (/\)!

That is, given a token, `nextPage` give you a state action where it picks a new token that's not equal to the old token, and updates the state of the world to reflect the new token.

```
client :: token:Int ->
  (ST <{v:Int | v = token} >{v:Int | v = done} >Int)
client = \token ->
  if token == done
  then pure 1
  else bind (nextPage token) (\tok -> client tok)
```

```
$ mist tests/pos/paginationTokens.hs
SAFE
```

And that concludes our short tutorial on `mist`. Go forth and verify!

A Appendix : Measures

But these `List` constructors are all a bit boring — what good are user datatypes if we can't say anything about them at the type level?! Until now, we've only been able to form refinements out of variables and primitive functions such as `+` and ∩ on special types such as `Int` and `Set`. We use (purely) refinement-level functions called measures (Vazou et al, ICFP 2014) to extend the language of refinements and enrich the types of our constructors.

```
measure mNil :: List [>Int] -> Bool
```

> Measures use a different syntax for types — type applications have a list of parameters in `[square, brackets, separated, by, commas]`, unlike applications of type constructors that produce refinement types, which use the usual space-separated syntax.

This declares a measure `mNil` that takes a `List` of `Int`s and returns a `Bool`. Measure have unrefined types.

> If you get an error message about free vars that implicates the start of the file, you probably tried to use a measure you didn't declare. This will cause the solver print a list of unbound measures and crash with some mumbo-jumbo about `--prune-unsorted`. Measures always go at the top of the file.

We can use these measures in constructor axioms to effectively define structurally recursive functions over a datatype.

```
nil as {v: List >Int | (mNil v) /\ (mLength v = 0) /\ (not (mCons v))}
```

```
13 cons as x:Int -> xs:(List >Int) ->
14   {v: List >Int | (mCons v) /\ (mLength v = mLength xs + 1) /\ (not (mNil v))}

17 first as {v: List >Int | mCons v} -> Int

20 rest as rs:{v: List >Int | mCons v}
21   -> {v: List >Int | mLength v + 1 == mLength rs }
22 rest = (0)
```

and we can then use them in verification!

```
24 append :: xs:(List >Int)
25   -> ys:(List >Int)
26   -> {v: List >Int | mLength v = (mLength xs) + (mLength ys)}
27 append = \xs -> \ys ->
28   if empty xs
29     then ys
30     else cons (first xs) (append (rest xs) ys)
```

```
$ mist tests/pos/recursion.hs
SAFE
```