Liquid Types

Presented by Gan Shen

Overview

- Use refinement type (dependent type) to describe program behaviors precisely.

Overview

- Use refinement type (dependent type) to describe program behaviors precisely.
  - Detect program error statically (at compile-time).

Overview

- Use refinement type (dependent type) to describe program behaviors precisely.
  - Detect program error statically (at compile-time).
  - Eliminate expensive run-time check.

Overview

- Use refinement type (dependent type) to describe program behaviors precisely.
  - Detect program error statically (at compile-time).
  - Eliminate expensive run-time check.
- Liquid type is a subset of refinement type (dependent type) in which refinement predicate is restricted to a decidable logic.

Overview

- Use refinement type (dependent type) to describe program behaviors precisely.
  - Detect program error statically (at compile-time).
  - Eliminate expensive run-time check.
- Liquid type is a subset of refinement type (dependent type) in which refinement predicate is restricted to a decidable logic.
  - Subtyping is decidable.

Overview

- Use refinement type (dependent type) to describe program behaviors precisely.
  - Detect program error statically (at compile-time).
  - Eliminate expensive run-time check.
- Liquid type is a subset of refinement type (dependent type) in which refinement predicate is restricted to a decidable logic.
  - Subtyping is decidable.
  - Type-checking is decidable.

Overview

- Use refinement type (dependent type) to describe program behaviors precisely.
  - Detect program error statically (at compile-time).
  - Eliminate expensive run-time check.
- Liquid type is a subset of refinement type (dependent type) in which refinement predicate is restricted to a decidable logic.
  - Subtyping is decidable.
  - Type-checking is decidable.
  - Type inference is decidable if the search space in finite.

Overview

- Use refinement type (dependent type) to describe program behaviors precisely.
  - Detect program error statically (at compile-time).
  - Eliminate expensive run-time check.
- Liquid type is a subset of refinement type (dependent type) in which refinement predicate is restricted to a decidable logic.
  - Subtyping is decidable.
  - Type-checking is decidable.
  - Type inference is decidable if the search space in finite.
- Traditional refinement type (dependent type) requires programmers to write complex type annotations (proof?) which is relieved by the type inference of liquid type.

Division By Zero

```
div : int -> int -> int

# div 1 0
Exception: Division_by_zero.
```

Index Out Of Bounds

```
get : intarray -> int -> int

# get [| 1; 2 |] 7
Exception: Invalid_argument "index out of bounds"
```

Problem

- The language is safe, better than "Segmentation fault (core dumped)" or quietly corrupt the data.

Problem

- The language is safe, better than "Segmentation fault (core dumped)" or quietly corrupt the data.
- At runtime, the program will do expensive checks to ensure safety.

Problem

- The language is safe, better than "Segmentation fault (core dumped)" or quietly corrupt the data.
- At runtime, the program will do expensive checks to ensure safety.
- The type system is unable to describe the program behavior precisely.

Liquid Type

- Division By Zero

```
div : int -> {v:int | v != 0} -> int

# div 1 0
Type Error: {v:int | v = 0} doesn't match
{v:int | v != 0}
```

Liquid Type

- Index Out Of Bounds

```
get : a:intarray ->
      i:{v:int | v >= 0 /\ v < len a} ->
      int

# get [| 1; 2 |] 7
Type Error: {v:int | v = 7} doesn't match
{v:int | v >= 0 /\ v < 2}
```

Liquid Type is Refinement Type

- Base Refinement

$$\{\nu : B \mid e\}$$

ν is a *value variable*
B is a *base type*
e is a *refinement predicate*.

Liquid Type is Refinement Type

- Base Refinement

$$\{\nu : B \mid e\}$$

  ν is a *value variable*
  B is a *base type*
  e is a *refinement predicate*.

- Intuitively, the refinement predicate specifies the set of value c of the base type B such that $[c/\nu]e$ evaluates to true.

Liquid Type is Refinement Type

- Base Refinement

$$\{\nu : B \mid e\}$$

  ν is a *value variable*
  B is a *base type*
  e is a *refinement predicate*.

- Intuitively, the refinement predicate specifies the set of value c of the base type B such that $[c/\nu]e$ evaluates to true.
- Examples

$$\{\nu : \texttt{int} \mid 0 < \nu\}$$

$$\{\nu : \texttt{int} \mid \nu \leq \texttt{n}\}$$

$$\texttt{int} \equiv \{\nu : \texttt{int} \mid \texttt{true}\}$$

Subtyping of Liquid Type

- Examples

$$3 : \{\nu : \mathtt{int} \mid \nu = 3\}$$

$$3 : \{\nu : \mathtt{int} \mid \nu > 0\}$$

$$3 : \{\nu : \mathtt{int} \mid \nu > -3 \wedge \nu < 10\}$$

Subtyping of Liquid Type

- Examples

$$3 : \{\nu : \mathtt{int} \mid \nu = 3\}$$

$$3 : \{\nu : \mathtt{int} \mid \nu > 0\}$$

$$3 : \{\nu : \mathtt{int} \mid \nu > -3 \wedge \nu < 10\}$$

- Subtyping check is undecidable if the refinement predicates contain arbitary terms.

Subtyping of Liquid Type

- Examples

$$3 : \{\nu : \mathtt{int} \mid \nu = 3\}$$

$$3 : \{\nu : \mathtt{int} \mid \nu > 0\}$$

$$3 : \{\nu : \mathtt{int} \mid \nu > -3 \wedge \nu < 10\}$$

- Subtyping check is undecidable if the refinement predicates contain arbitary terms.
- Subtyping check is decidable if the refinement predicates are restricted in a decidable logic.

Subtyping of Liquid Type

- Examples

$$3 : \{\nu : \texttt{int} \mid \nu = 3\}$$

$$3 : \{\nu : \texttt{int} \mid \nu > 0\}$$

$$3 : \{\nu : \texttt{int} \mid \nu > -3 \wedge \nu < 10\}$$

- Subtyping check is undecidable if the refinement predicates contain arbitary terms.
- Subtyping check is decidable if the refinement predicates are restricted in a decidable logic.
- Subyyping check by embedding refinment predicate e into EUFA logic, written as $[\![e]\!]$.

Subtyping of Liquid Type (Cont.)

- Subtyping Rules $\boxed{\Gamma \vdash T_1 <: T_2}$

$$\frac{\mathsf{Valid}(\llbracket \Gamma \rrbracket \wedge \llbracket e_1 \rrbracket \rightarrow \llbracket e_2 \rrbracket)}{\Gamma \vdash \{v : B \mid e_1\} <: \{v : B \mid e_2\}}\ \textsc{SubBase}$$

Subtyping of Liquid Type (Cont.)

- Subtyping Rules $\boxed{\Gamma \vdash T_1 <: T_2}$

$$\frac{\mathsf{Valid}(\llbracket \Gamma \rrbracket \wedge \llbracket e_1 \rrbracket \rightarrow \llbracket e_2 \rrbracket)}{\Gamma \vdash \{v : B \mid e_1\} <: \{v : B \mid e_2\}}\ \textsc{SubBase}$$

- Subtype Derivation

$$\frac{\mathsf{Valid}(\nu = 3 \rightarrow (\nu > -3 \wedge \nu < 10))}{\emptyset \vdash \{\nu : \mathtt{int} \mid \nu = 3\} <: \{\nu : \mathtt{int} \mid \nu > -3 \wedge \nu < 10\}}$$

Subtyping of Liquid Type (Cont.)

- Subtyping Rules $\boxed{\Gamma \vdash T_1 <: T_2}$

$$\frac{\mathsf{Valid}(\llbracket \Gamma \rrbracket \wedge \llbracket e_1 \rrbracket \rightarrow \llbracket e_2 \rrbracket)}{\Gamma \vdash \{v : B \mid e_1\} <: \{v : B \mid e_2\}}\ \textsc{SubBase}$$

- Subtype Derivation

$$\frac{\mathsf{Valid}(\nu = 3 \rightarrow (\nu > -3 \wedge \nu < 10))}{\emptyset \vdash \{\nu : \mathtt{int} \mid \nu = 3\} <: \{\nu : \mathtt{int} \mid \nu > -3 \wedge \nu < 10\}}$$

- One term can have multiple types.

Liquid Type is Dependent Type

- A dependent type is a type whose definition depends on a value (term, expression).

Liquid Type is Dependent Type

- A dependent type is a type whose definition depends on a value (term, expression).
- Functions have dependent types.

$$x : T_1 \to T_2$$

T_2 can reference x in its definition.

Liquid Type is Dependent Type

- A dependent type is a type whose definition depends on a value (term, expression).
- Functions have dependent types.

  $$x : T_1 \to T_2$$

  T_2 can reference x in its definition.
- Examples

$$\begin{aligned} \texttt{max} : \quad & \texttt{x} : \texttt{int} \to \texttt{y} : \texttt{int} \to \{\nu : \texttt{int} \mid \nu \geq \texttt{x} \wedge \nu \geq \texttt{y}\} \\ \texttt{max}\ 3\ 4 : \quad & \{\nu : \texttt{int} \mid \nu \geq 3 \wedge \nu \geq 4\} \end{aligned}$$

Type Inference

- **Goal:** Given Γ, e, find a type T that makes $\Gamma \vdash e : T$ holds.
- **Problem:** The search space is unbounded!

Type Inference (Cont.)

- **Solution:** Require the programmer to provide a set of logical qualifiers $\mathbb{Q}$ as a *template*.

  $$\{0 \leq \nu,\ \star \leq \nu,\ \nu \leq \star,\ \nu < \texttt{len}\ \star\}$$

  $\star$ is a special placeholder variable that can be instantiated with free variables in the context.
- Since the free variables in the context is finite, the concrete set of logical qualifiers $\mathbb{Q}$ can be instantiated to is finite.
- So the search space is bounded!

Type Inference (Cont.)

1. **Hindley-Milner Type Inference:** Infer the base type of each expression. Assign a *liquid type variable* to each base type.
2. **Liquid Constraint Generation:** Generate a set of constraints that capture the relationships between types that must be met for a type derivation to exist.
3. **Liquid Constraint Solving:** Solve the constraints with the help of $\mathbb{Q}$.

Type Inference Example

Expression:

$$\texttt{max} = \lambda\texttt{x}.\lambda\texttt{y}.\texttt{if x} > \texttt{y then x else y}$$

Base Type:

$$\texttt{int} \to \texttt{int} \to \texttt{int}$$

Liquid Type Template:

$$\texttt{x} : \{\nu : \texttt{int} \mid \kappa_{\texttt{x}}\} \to \texttt{y} : \{\nu : \texttt{int} \mid \kappa_{\texttt{y}}\} \to \{\nu : \texttt{int} \mid \kappa\}$$

$$\texttt{x} : \kappa_{\texttt{x}} \to \texttt{y} : \kappa_{\texttt{y}} \to \kappa$$

Logical Qualifiers:

$$\mathbb{Q} = \{0 \leq \nu,\ \star \leq \nu,\ \nu \leq \star,\ \nu < \texttt{len}\ \star\}$$

Type Inference Example (Cont.)

Constraints:

$$\emptyset \vdash \kappa_{\mathtt{x}}$$
$$\mathtt{x} : \mathtt{int} \vdash \kappa_{\mathtt{y}}$$
$$\mathtt{x} : \mathtt{int}; \mathtt{y} : \mathtt{int} \vdash \kappa$$

$$\mathtt{x} : \kappa_{\mathtt{x}}; \mathtt{y} : \kappa_{\mathtt{y}}; (\mathtt{x} > \mathtt{y}) \vdash \{\nu = \mathtt{x}\} <: \kappa$$
$$\mathtt{x} : \kappa_{\mathtt{x}}; \mathtt{y} : \kappa_{\mathtt{y}}; \neg(\mathtt{x} > \mathtt{y}) \vdash \{\nu = \mathtt{y}\} <: \kappa$$

Type Inference Example (Cont.)

Constraints:

$$\mathtt{x} : \mathtt{int}; \mathtt{y} : \mathtt{int} \vdash \kappa$$

$$(\mathtt{x} > \mathtt{y}) \vdash \{\nu = \mathtt{x}\} <: \kappa$$
$$\neg(\mathtt{x} > \mathtt{y}) \vdash \{\nu = \mathtt{y}\} <: \kappa$$

Type Inference Example (Cont.)

Logical Qualifiers:

$$\mathbb{Q} = \{0 \leq \nu,\ \star \leq \nu,\ \nu \leq \star,\ \nu < \texttt{len}\ \star\}$$

Well-formedness Constraints:

$$\texttt{x} : \texttt{int}; \texttt{y} : \texttt{int} \vdash \kappa$$

Initial Assignment:

$$\kappa \mapsto (0 \leq \nu) \wedge (\texttt{x} \leq \nu) \wedge (\texttt{y} \leq \nu) \wedge (\nu \leq \texttt{x}) \wedge (\nu \leq \texttt{y})$$

Type Inference Example (Cont.)

Assignment:

$$\kappa \mapsto (0 \leq \nu) \wedge (\mathrm{x} \leq \nu) \wedge (\mathrm{y} \leq \nu) \wedge (\nu \leq \mathrm{x}) \wedge (\nu \leq \mathrm{y})$$

Constraints:

$$\begin{aligned} (\mathrm{x} > \mathrm{y}) &\vdash \{\nu = \mathrm{x}\} <: \kappa \\ \neg(\mathrm{x} > \mathrm{y}) &\vdash \{\nu = \mathrm{y}\} <: \kappa \end{aligned}$$

Checking:

$$\mathsf{Valid}\Big((\mathrm{x} > \mathrm{y}) \wedge (\nu = \mathrm{x}) \Rightarrow \kappa\Big) = \mathsf{FALSE}$$

Type Inference Example (Cont.)

Assignment:

$$\kappa \mapsto (0 \leq \nu) \wedge (\mathtt{x} \leq \nu) \wedge (\mathtt{y} \leq \nu) \wedge (\nu \leq \mathtt{x}) \wedge (\nu \leq \mathtt{y})$$

Weakening:

$$\text{Valid}\Big((\mathtt{x} > \mathtt{y}) \wedge (\nu = \mathtt{x}) \Rightarrow (0 \leq \nu)\Big) = \text{FALSE}$$

$$\text{Valid}\Big((\mathtt{x} > \mathtt{y}) \wedge (\nu = \mathtt{x}) \Rightarrow (\mathtt{x} \leq \nu)\Big) = \text{TRUE}$$

$$\text{Valid}\Big((\mathtt{x} > \mathtt{y}) \wedge (\nu = \mathtt{x}) \Rightarrow (\mathtt{y} \leq \nu)\Big) = \text{TRUE}$$

$$\text{Valid}\Big((\mathtt{x} > \mathtt{y}) \wedge (\nu = \mathtt{x}) \Rightarrow (\nu \leq \mathtt{x})\Big) = \text{FALSE}$$

$$\text{Valid}\Big((\mathtt{x} > \mathtt{y}) \wedge (\nu = \mathtt{x}) \Rightarrow (\nu \leq \mathtt{y})\Big) = \text{FALSE}$$

Type Inference Example (Cont.)

Assignment:

$$\kappa \mapsto (\mathtt{x} \leq \nu) \wedge (\mathtt{y} \leq \nu)$$

Constraints:

$$(\mathtt{x} > \mathtt{y}) \vdash \{\nu = \mathtt{x}\} <: \kappa$$
$$\neg(\mathtt{x} > \mathtt{y}) \vdash \{\nu = \mathtt{y}\} <: \kappa$$

Checking:

$$\textsf{Valid}\left((\mathtt{x} > \mathtt{y}) \wedge (\nu = \mathtt{x}) \Rightarrow \kappa\right) = \textsf{TRUE}$$

$$\textsf{Valid}\left(\neg(\mathtt{x} > \mathtt{y}) \wedge (\nu = \mathtt{x}) \Rightarrow \kappa\right) = \textsf{TRUE}$$

Type Inference Example (Cont.)

Assignment:

$$\begin{aligned} \kappa &\mapsto (\mathtt{x} \leq \nu) \wedge (\mathtt{y} \leq \nu) \\ \kappa_{\mathtt{x}} &\mapsto \text{true} \\ \kappa_{\mathtt{y}} &\mapsto \text{true} \end{aligned}$$

Liquid Type Template:

$$\mathtt{x} : \{\nu : \mathtt{int} \mid \kappa_{\mathtt{x}}\} \to \mathtt{y} : \{\nu : \mathtt{int} \mid \kappa_{\mathtt{y}}\} \to \{\nu : \mathtt{int} \mid \kappa\}$$

Liquid Type:

$$\mathtt{max} : \mathtt{x} : \mathtt{int} \to \mathtt{y} : \mathtt{int} \to \{\nu : \mathtt{int} \mid (\mathtt{x} \leq \nu) \wedge (\mathtt{y} \leq \nu)\}$$

Discussion

- What is the performance bottleneck of the type inference algorithm? What are some possible ways to improve it?

Discussion

- What is the performance bottleneck of the type inference algorithm? What are some possible ways to improve it?
  - Allow programmers to provide local $\mathbb{Q}$ for each expression

Discussion

- What is the performance bottleneck of the type inference algorithm? What are some possible ways to improve it?
  - Allow programmers to provide local $\mathbb{Q}$ for each expression
  - Write explicit type annotation for certain expressions

Discussion

- What is the performance bottleneck of the type inference algorithm? What are some possible ways to improve it?
  - Allow programmers to provide local $\mathbb{Q}$ for each expression
  - Write explicit type annotation for certain expressions
- What is the limitation of liquid type?

Discussion

- What is the performance bottleneck of the type inference algorithm? What are some possible ways to improve it?
  - Allow programmers to provide local $\mathbb{Q}$ for each expression
  - Write explicit type annotation for certain expressions
- What is the limitation of liquid type?
- How to generate helpful error message?

Typing Rules

$$\boxed{\Gamma \vdash t : T}$$

$$\frac{}{\Gamma \vdash x : \Gamma(x)}\ \textsc{LtVar} \qquad \frac{}{\Gamma \vdash c : ty(c)}\ \textsc{LtConst}$$

$$\frac{\Gamma; e : T_1 \vdash t : T_2 \qquad \Gamma \vdash_W (x : T_1 \to T_2)}{\Gamma \vdash \lambda x.e : (x : T_1 \to T_2)}\ \textsc{LtFun}$$

$$\frac{\Gamma \vdash e_1 : (x : T_1 \to T_2) \qquad \Gamma \vdash e_2 : T_1}{\Gamma \vdash e_1\ e_2 : [e_2/x]T_2}\ \textsc{LtApp}$$

$$\frac{\Gamma \vdash e : T_1 \qquad \Gamma \vdash T_1 <: T_2 \qquad \Gamma \vdash_W T_2}{\Gamma \vdash e : T_2}\ \textsc{LtSub}$$

Typing Rules (Cont.)

$$\boxed{\Gamma \vdash t : T}$$

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \qquad \Gamma; e_1 \vdash e_2 : T \qquad \Gamma; \neg e_1 \vdash e_3 : T}{\Gamma \vdash \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 : T} \textsc{LtIf}$$

$$\frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma; x : T_1 \vdash e_2 : T}{\Gamma \vdash \texttt{let } x = e_1 \texttt{ in } e_2 : T} \textsc{LtLet}$$

$$\frac{\Gamma; x : T_1 \vdash \lambda x'.e_1 : T_1 \qquad \Gamma; x : T_1 \vdash e_2 : T}{\Gamma \vdash \texttt{let rec } x = \lambda x'.e_1 \texttt{ in } e_2 : T} \textsc{LtRec}$$

Type Derivation

$$\Gamma = \mathtt{max} : (\mathtt{x} : \mathtt{int} \to \mathtt{y} : \mathtt{int} \to \{\nu : \mathtt{int} \mid \nu \geq \mathtt{x} \wedge \nu \geq \mathtt{y}\})$$

$$\tau = \{\nu : \mathtt{int} \mid \nu \geq \mathtt{x} \wedge \nu \geq \mathtt{y}\}$$

$$\dfrac{\dfrac{\dfrac{\Gamma(\mathtt{max}) = \ldots}{\mathtt{max} : (\mathtt{x} : \mathtt{int} \to \mathtt{y} : \mathtt{int} \to \tau)} \qquad \dfrac{\ldots}{3 : \mathtt{int}}}{\mathtt{max}\ 3 : (\mathtt{y} : \mathtt{int} \to [3/\mathtt{x}]\tau)} \qquad \dfrac{\ldots}{4 : \mathtt{int}}}{(\mathtt{max}\ 3)\ 4 : [4/\mathtt{y}][3/\mathtt{x}]\tau}$$

Multiple Types

$$\begin{aligned}
\texttt{max}: \quad & \mathtt{x}:\texttt{int} \rightarrow \mathtt{y}:\texttt{int} \rightarrow \{\nu : \texttt{int} \mid (\mathtt{x} \leq \nu) \wedge (\mathtt{y} \leq \nu)\} \\
\texttt{max}: \quad & \mathtt{x}:\texttt{int} \rightarrow \mathtt{y}:\texttt{int} \rightarrow \{\nu : \texttt{int} \mid (\mathtt{x} \leq \nu)\} \\
\texttt{max}: \quad & \ldots \rightarrow \{\nu : \texttt{int} \mid (\mathtt{x} \leq \nu) \wedge (\mathtt{y} \leq \nu) \wedge (\nu \leq 0 \vee \nu \geq 0)\}
\end{aligned}$$