Liquid Types

Manuel Eberl

April 29, 2013

Prelude – Type Systems

Prelude

What is a type system?

Prelude

What is a type system?

> *A way of classifying expressions by the kind of values they compute*

Prelude

What is a type system?

A way of classifying expressions by the kind of values they compute

What are they for?

Prelude

What is a type system?

A way of classifying expressions by the kind of values they compute

What are they for?

To guarantee the absence of certain undesired or unintended behaviour

Prelude

What is a type system?

A way of classifying expressions by the kind of values they compute

What are they for?

To guarantee the absence of certain undesired or unintended behaviour

What kinds of type systems are there?

Prelude

What is a type system?

A way of classifying expressions by the kind of values they compute

What are they for?

To guarantee the absence of certain undesired or unintended behaviour

What kinds of type systems are there?

Well…

Dynamic typing

- Only one type: Any
- Give no information or guarantees whatsoever
- Used by: Python, JavaScript, PHP, ...

Dynamic typing

This is a *very bad* idea. Why? Enter Python:

```
a = "foo"
b = 42
print(b - a)
```

Dynamic typing

This is a *very bad* idea. Why? Enter Python:

```python
a = "foo"
b = 42
print(b - a)
```

Compiles without problems, but at runtime:

```
TypeError: unsupported operand type(s) for -:
           'int' and 'str'
```

Leads to unnecessary errors and/or erratic behaviour

Dynamic typing

This is a *very bad* idea. Why? Enter Python:

```python
a = "foo"
b = 42
print(b - a)
```

Compiles without problems, but at runtime:

```
TypeError: unsupported operand type(s) for -:
           'int' and 'str'
```

Leads to unnecessary errors and/or erratic behaviour

(cf. Bernhardt, 2012: “Wat”, `http://youtu.be/kXEgk1Hdze0`)

Static typing

- Different types that correspond to “sorts” of values
  (e.g. Integer, String, Boolean)
- Guarantees the absence of type errors
- Used by: Java, C, Pascal

Static typing

The same thing in Java:

```java
String a = "foo";
int b = 42;
System.out.println(b - a);
```

Static typing

The same thing in Java:

```
String a = "foo";
int b = 42;
System.out.println(b - a);
```

Compile time (!) error tells us something is wrong:

```
error: bad operand types for binary operator '-'
  System.out.println(b - a);
                       ^
  first type:  int
  second type: String
```

But: we have to annotate types ("String" resp. "int")

Static typing, but fancy

- One nice addition: type inference
- Same guarantees, but less work
- Used by: Standard ML, OCaml, Haskell, …
- Down side: none!

Type inference

Type inference: compiler figures out types (mostly) without annotations. Same as before, now in Scala:

```
val a = "foo"
val b = 42
System.out.println(b - a)
```

Type inference

Type inference: compiler figures out types (mostly) without annotations. Same as before, now in Scala:

```
val a = "foo"
val b = 42
System.out.println(b - a)
```

Again: compile time error:

```
error: overloaded method value - with alternatives:
  (x: Double)Double <and>
  (x: Float)Float <and> ...
 cannot be applied to (java.lang.String)
  System.out.println(b - a);
                       ^
```

Both the safety of static typing and the convenience of dynamic typing!

Dependent types

So we can prevent errors caused by values being of the wrong "sort", i.e. string instead of number.

Dependent types

So we can prevent errors caused by values being of the wrong "sort", i.e. string instead of number.

But what about errors that are caused by restrictions on the actual *values*?

- dereferencing a null pointer
- array bounds violation
- division by zero

Can we express restrictions on values in a type system as well?

Dependent types

Example 1: Integer division

Takes two integers, returns a rational number:

$$(/) :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Rational}$$

Dependent types

Example 1: Integer division

Takes two integers, returns a rational number:

$$(/) :: \mathsf{Int} \to \mathsf{Int} \to \mathsf{Rational}$$

But the second operand must not be 0. So what we want is:

$$(/) :: \mathsf{Int} \to \{\nu : \mathsf{Int} \mid \nu \neq 0\} \to \mathsf{Rational}$$

Dependent types

Example 2: List concatenation

Take two lists, return the concatenated list:

$$(++) :: \text{List } a \to \text{List } a \to \text{List } a$$

But we lose some interesting information, e.g. about the result list's length.

Dependent types

Example 2: List concatenation

Take two lists, return the concatenated list:

$$(++) :: \text{List } a \rightarrow \text{List } a \rightarrow \text{List } a$$

But we lose some interesting information, e.g. about the result list's length. What we want is something like:

$$(++) :: \text{List } a\ m \rightarrow \text{List } a\ n \rightarrow \text{List } a\ (m + n)$$

Dependent types

- Types can have arbitrary restrictions and depend on values
- Guarantees of arbitrary complexity
- Used by: Dependent ML, Idris
- Down side: type checking/inference undecidable, may require user-supplied proofs
  $\Longrightarrow$ A lot of work!

Liquid Types

Compromise: *Liquid Types*

Restrict power of dependent types to decidable fragment

$\implies$ inference of expressive types without user interaction

Liquid Types

Definition of Liquid Types

Basic idea:

- Take normal types as inferred by Hindley/Milner

Definition of Liquid Types

Basic idea:

- Take normal types as inferred by Hindley/Milner
- Augment them with a specific kind of conditions (*Refinement Types*), i.e. linear constraints such as $\{\nu : \mathsf{Int} \mid \nu > 0\}$ or $k : \mathsf{Int} \to \{\nu : \mathsf{Int} \mid \nu \leq k\}$

Definition of Liquid Types

Basic idea:

- Take normal types as inferred by Hindley/Milner
- Augment them with a specific kind of conditions (*Refinement Types*), i.e. linear constraints such as $\{\nu : \mathsf{Int} \mid \nu > 0\}$ or $k : \mathsf{Int} \rightarrow \{\nu : \mathsf{Int} \mid \nu \leq k\}$
- Allowed conditions should be *powerful enough* to say something interesting, but *weak enough* to allow automatic type inference

Definition of Liquid Types

Basic idea:

- Take normal types as inferred by Hindley/Milner
- Augment them with a specific kind of conditions (*Refinement Types*), i.e. linear constraints such as $\{\nu : \mathsf{Int} \mid \nu > 0\}$ or $k : \mathsf{Int} \rightarrow \{\nu : \mathsf{Int} \mid \nu \leq k\}$
- Allowed conditions should be *powerful enough* to say something interesting, but *weak enough* to allow automatic type inference
- Not all programmes that are of type *T* can be recognised as such by type checking/inference

Definition of Liquid Types

Example for the rest of the talk: simple equality/inequality constraints

Conditions are conjunctions of qualifiers from e.g.:

$$\mathbb{Q} = \{0 \leq \nu, \nu = *, * \leq \nu\}$$

Definition of Liquid Types

Example for the rest of the talk: simple equality/inequality constraints

Conditions are conjunctions of qualifiers from e.g.:

$$\mathbb{Q} = \{0 \leq \nu, \nu = *, * \leq \nu\}$$

Example:

array_get::

$a : \text{Array } v \to \{\nu : \text{Int} \mid 0 \leq \nu \wedge \nu < \text{len}(a)\} \to v$

$\Longrightarrow$ Compile-time guarantee: no array-bounds violations

$\Longrightarrow$ Compiler can drop bounds checks

Liquid Type Inference

1. Run Hindley-Milner to obtain liquid type template
2. Use syntax-directed rules to generate system of constraints
3. Solve constraints using theorem prover

Liquid Type Inference – Hindley-Milner

Hindley-Milner: standard type inference algorithm for functional languages

Example:

We want to type:

$$\text{max}\ (a :: \text{Int})\ (b :: \text{Int})\ =\ \textbf{if}\ a < b\ \textbf{then}\ b\ \textbf{else}\ a$$

Liquid Type Inference – Hindley-Milner

Hindley-Milner: standard type inference algorithm for functional languages

Example:

We want to type:

$$\text{max}\ (a :: \text{Int})\ (b :: \text{Int}) \ = \ \textbf{if}\ a < b\ \textbf{then}\ b\ \textbf{else}\ a$$

We reason:

- the parameters a and b are of type Int.

Liquid Type Inference – Hindley-Milner

Hindley-Milner: standard type inference algorithm for functional languages

Example:

We want to type:

$$\text{max } (a :: \text{Int})\ (b :: \text{Int}) \ = \ \textbf{if } a < b \textbf{ then } b \textbf{ else } a$$

We reason:

- the parameters a and b are of type Int.
- $a < b$ is condition in an **if**, thus $a < b :: \text{Bool}$ – okay

Liquid Type Inference – Hindley-Milner

Hindley-Milner: standard type inference algorithm for functional languages

Example:

We want to type:

$$\text{max}\ (a :: \text{Int})\ (b :: \text{Int}) \ = \ \textbf{if}\ a < b\ \textbf{then}\ b\ \textbf{else}\ a$$

We reason:

- the parameters a and b are of type Int.
- $a < b$ is condition in an **if**, thus $a < b :: \text{Bool}$ – okay
- the **if** expression returns a or b, thus a and b have the same type as the result

Liquid Type Inference – Hindley-Milner

Hindley-Milner: standard type inference algorithm for functional languages

Example:

We want to type:

$$\max\ (a :: \mathrm{Int})\ (b :: \mathrm{Int})\ =\ \textbf{if}\ a < b\ \textbf{then}\ b\ \textbf{else}\ a$$

We reason:

- the parameters a and b are of type Int.
- $a < b$ is condition in an **if**, thus $a < b$ :: Bool – okay
- the **if** expression returns a or b, thus a and b have the same type as the result
- therefore, the most precise result type is Int.

Liquid Type Inference – Hindley-Milner

Hindley-Milner: standard type inference algorithm for functional languages

Example:

We want to type:

$$\text{max } (a :: \text{Int})\ (b :: \text{Int}) \ = \ \textbf{if } a < b \textbf{ then } b \textbf{ else } a$$

We reason:

- the parameters a and b are of type Int.
- $a < b$ is condition in an **if**, thus $a < b :: \text{Bool}$ – okay
- the **if** expression returns a or b, thus a and b have the same type as the result
- therefore, the most precise result type is Int.
- $\text{max} :: \text{Int} \to \text{Int} \to \text{Int}$

Liquid Type Inference – Hindley-Milner

Example:

More formally: type derivation tree:

Context $\Gamma = [a : \mathsf{Int};\ b : \mathsf{Int}]$

$$\dfrac{\dfrac{\dfrac{\Gamma(a) = \mathsf{Int}}{\Gamma \vdash a : \mathsf{Int}} \quad \dfrac{\Gamma(b) = \mathsf{Int}}{\Gamma \vdash b : \mathsf{Int}}}{\Gamma \vdash a < b : \mathsf{Bool}} \qquad \dfrac{\Gamma(b) = \mathsf{Int}}{\Gamma \vdash b : \mathsf{Int}} \qquad \dfrac{\Gamma(a) = \mathsf{Int}}{\Gamma \vdash a : \mathsf{Int}}}{\Gamma \vdash \textbf{if } a < b \textbf{ then } b \textbf{ else } a : \mathsf{Int}}$$

Liquid Type Inference – Hindley-Milner

So Hindley-Milner can give us “normal” types for expressions.

How do we get liquid types out of that?

Liquid Type Inference – Hindley-Milner

So Hindley-Milner can give us “normal” types for expressions.

How do we get liquid types out of that?

$\Longrightarrow$ introduce liquid type variable κ for each “base type”

$\Longrightarrow$ liquid type template

Example:

Programme:

$$\text{max}\ (a :: \text{Int})\ (b :: \text{Int})\ =\ \textbf{if}\ a < b\ \textbf{then}\ b\ \textbf{else}\ a$$

Liquid Type Inference – Hindley-Milner

So Hindley-Milner can give us “normal” types for expressions.

How do we get liquid types out of that?

$\Longrightarrow$ introduce liquid type variable κ for each “base type”

$\Longrightarrow$ liquid type template

Example:

Programme:

$$\text{max } (a :: \text{Int})\ (b :: \text{Int}) \ = \ \textbf{if } a < b \textbf{ then } b \textbf{ else } a$$

HM type:

$$a : \text{Int} \to b : \text{Int} \to \text{Int}$$

Liquid Type Inference – Hindley-Milner

So Hindley-Milner can give us "normal" types for expressions.

How do we get liquid types out of that?

$\Longrightarrow$ introduce liquid type variable κ for each "base type"

$\Longrightarrow$ liquid type template

Example:

Programme:

$$\text{max } (a :: \text{Int})\ (b :: \text{Int}) \ = \ \textbf{if } a < b \textbf{ then } b \textbf{ else } a$$

HM type:

$$a : \text{Int} \to b : \text{Int} \to \text{Int}$$

Liquid type template:

$$a : \{\nu : \text{Int} \mid \kappa_a\} \to b : \{\nu : \text{Int} \mid \kappa_b\} \to \{\nu : \text{Int} \mid \kappa_r\}$$

Liquid Type Inference – Constraint generation

Next: what constraints are there on the κ?

First, an example.

Liquid Type Inference – Constraint generation

Example:

Programme:

$$\text{max } (a :: \text{Int})\ (b :: \text{Int}) \ = \ \textbf{if } a < b \textbf{ then } b \textbf{ else } a$$

Liquid type template:

$$a : \{\nu : \text{Int} \mid \kappa_a\} \rightarrow b : \{\nu : \text{Int} \mid \kappa_b\} \rightarrow \{\nu : \text{Int} \mid \kappa_r\}$$

Liquid Type Inference – Constraint generation

Example:

Programme:

$$\text{max } (a :: \text{Int})\ (b :: \text{Int}) \ = \ \textbf{if } a < b \textbf{ then } b \textbf{ else } a$$

Liquid type template:

$$a : \{\nu : \text{Int} \mid \kappa_a\} \to b : \{\nu : \text{Int} \mid \kappa_b\} \to \{\nu : \text{Int} \mid \kappa_r\}$$

Intuitively:

- We know that $a :: \kappa_a$ and $b :: \kappa_b$. Let's call these facts Γ.

Liquid Type Inference – Constraint generation

Example:

Programme:

$$\text{max } (a :: \text{Int})\ (b :: \text{Int}) \ = \ \textbf{if } a < b \textbf{ then } b \textbf{ else } a$$

Liquid type template:

$$a : \{\nu : \text{Int} \mid \kappa_a\} \rightarrow b : \{\nu : \text{Int} \mid \kappa_b\} \rightarrow \{\nu : \text{Int} \mid \kappa_r\}$$

Intuitively:

- We know that $a :: \kappa_a$ and $b :: \kappa_b$. Let's call these facts Γ.
- if $a < b$, we return b, therefore $\Gamma \wedge a < b$ must imply κ_r

Liquid Type Inference – Constraint generation

Example:

Programme:

$$\text{max } (a :: \text{Int})\ (b :: \text{Int}) \ = \ \textbf{if } a < b \textbf{ then } b \textbf{ else } a$$

Liquid type template:

$$a : \{\nu : \text{Int} \mid \kappa_a\} \rightarrow b : \{\nu : \text{Int} \mid \kappa_b\} \rightarrow \{\nu : \text{Int} \mid \kappa_r\}$$

Intuitively:

- We know that $a :: \kappa_a$ and $b :: \kappa_b$. Let's call these facts Γ.
- if $a < b$, we return b, therefore $\Gamma \wedge a < b$ must imply κ_r
- if $b \leq a$, we return a, therefore $\Gamma \wedge b \leq a$ must imply κ_r

Liquid Type Inference – Constraint generation

Example:

Programme:

$$\text{max } (a :: \text{Int})\ (b :: \text{Int}) \ = \ \textbf{if } a < b \textbf{ then } b \textbf{ else } a$$

Liquid type template:

$$a : \{\nu : \text{Int} \mid \kappa_a\} \rightarrow b : \{\nu : \text{Int} \mid \kappa_b\} \rightarrow \{\nu : \text{Int} \mid \kappa_r\}$$

Intuitively:

- We know that $a :: \kappa_a$ and $b :: \kappa_b$. Let's call these facts Γ.
- if $a < b$, we return b, therefore $\Gamma \wedge a < b$ must imply κ_r
- if $b \leq a$, we return a, therefore $\Gamma \wedge b \leq a$ must imply κ_r
- these are (morally) our two constraints

Liquid Type Inference – Constraint generation

How is this done formally?

Constraints are inferred with a system of syntax-directed rules:

$$\frac{\Gamma \vdash_{\mathbb{Q}} v : \text{Bool} \qquad \Gamma; c \vdash_{\mathbb{Q}} e_1 : \hat{\sigma} \qquad \Gamma; \neg c \vdash_{\mathbb{Q}} e_2 : \hat{\sigma}}{\Gamma \vdash_{\mathbb{Q}} \textbf{if } c \textbf{ then } e_1 \textbf{ else } e_2 \ : \ \hat{\sigma}} \text{ LT-IF}$$

Liquid Type Inference – Constraint generation

How is this done formally?

Constraints are inferred with a system of syntax-directed rules:

$$\frac{\Gamma \vdash_{\mathbb{Q}} v : \mathrm{Bool} \qquad \Gamma; c \vdash_{\mathbb{Q}} e_1 : \hat{\sigma} \qquad \Gamma; \neg c \vdash_{\mathbb{Q}} e_2 : \hat{\sigma}}{\Gamma \vdash_{\mathbb{Q}} \textbf{if } c \textbf{ then } e_1 \textbf{ else } e_2 \ : \ \hat{\sigma}} \text{ LT-IF}$$

$$\frac{\Gamma \vdash_{\mathbb{Q}} e : \sigma_1 \qquad \Gamma \vdash \sigma_1 <: \sigma_2}{\Gamma \vdash_{\mathbb{Q}} e : \sigma_2} \text{ LT-SUB}$$

Liquid Type Inference – Constraint generation

How is this done formally?

Constraints are inferred with a system of syntax-directed rules:

$$\frac{\Gamma \vdash_{\mathbb{Q}} v : \text{Bool} \qquad \Gamma; c \vdash_{\mathbb{Q}} e_1 : \hat{\sigma} \qquad \Gamma; \neg c \vdash_{\mathbb{Q}} e_2 : \hat{\sigma}}{\Gamma \vdash_{\mathbb{Q}} \textbf{if } c \textbf{ then } e_1 \textbf{ else } e_2 \; : \; \hat{\sigma}} \text{ LT-IF}$$

$$\frac{\Gamma \vdash_{\mathbb{Q}} e : \sigma_1 \qquad \Gamma \vdash \sigma_1 <: \sigma_2}{\Gamma \vdash_{\mathbb{Q}} e : \sigma_2} \text{ LT-SUB}$$

$$\frac{\Gamma \models \varphi_1 \Rightarrow \varphi_2}{\Gamma \vdash \{\nu : t \mid \varphi_1\} <: \{\nu : t \mid \varphi_2\}} <:\text{-BASE}$$

Liquid Type Inference – Constraint generation

Context: $\Gamma = [a : \{\nu : \text{Int} \mid \kappa_a\};\ b : \{\nu : \text{Int} \mid \kappa_b\}]$

Apply LT-IF rule:

$$\frac{\Gamma \vdash_{\mathbb{Q}} a < b : \text{Bool} \qquad \Gamma; a < b \vdash_{\mathbb{Q}} b : \kappa_r \qquad \Gamma; b \leq a \vdash_{\mathbb{Q}} a : \kappa_r}{\Gamma \vdash_{\mathbb{Q}} \textbf{if } a < b \textbf{ then } b \textbf{ else } a \ :\ \kappa_r}$$

Liquid Type Inference – Constraint generation

Context: $\Gamma = [a : \{\nu : \text{Int} \mid \kappa_a\};\ b : \{\nu : \text{Int} \mid \kappa_b\}]$

Apply LT-IF rule:

$$\frac{\Gamma \vdash_{\mathbb{Q}} a < b : \text{Bool} \qquad \Gamma; a < b \vdash_{\mathbb{Q}} b : \kappa_r \qquad \Gamma; b \leq a \vdash_{\mathbb{Q}} a : \kappa_r}{\Gamma \vdash_{\mathbb{Q}} \textbf{if } a < b \textbf{ then } b \textbf{ else } a \ :\ \kappa_r}$$

Apply subtyping rules LT-SUB and <:-BASE to prove the last two obligations:

$$\frac{\dfrac{}{\Gamma; a < b \vdash_{\mathbb{Q}} b : \{\nu : \text{Int} \mid \nu = b\}} \qquad \dfrac{\Gamma; a < b \vDash \nu = b \Rightarrow \kappa_r}{\Gamma; a < b \vdash \{\nu : \text{Int} \mid \nu = b\} <: \{\nu : \text{Int} \mid \kappa_r\}}}{\Gamma; a < b \vdash_{\mathbb{Q}} b : \{\nu : \text{Int} \mid \kappa_r\}}$$

Liquid Type Inference – Constraint generation

Context: $\Gamma = [a : \{\nu : \mathsf{Int} \mid \kappa_a\};\ b : \{\nu : \mathsf{Int} \mid \kappa_b\}]$

Apply LT-IF rule:

$$\frac{\Gamma \vdash_{\mathbb{Q}} a < b : \mathsf{Bool} \qquad \Gamma; a < b \vdash_{\mathbb{Q}} b : \kappa_r \qquad \Gamma; b \leq a \vdash_{\mathbb{Q}} a : \kappa_r}{\Gamma \vdash_{\mathbb{Q}} \textbf{if } a < b \textbf{ then } b \textbf{ else } a \ :\ \kappa_r}$$

Apply subtyping rules LT-SUB and <:-BASE to prove the last two obligations:

$$\frac{\dfrac{}{\Gamma; a < b \vdash_{\mathbb{Q}} b : \{\nu : \mathsf{Int} \mid \nu = b\}} \qquad \dfrac{\Gamma; a < b \models \nu = b \Rightarrow \kappa_r}{\Gamma; a < b \vdash \{\nu : \mathsf{Int} \mid \nu = b\} <: \{\nu : \mathsf{Int} \mid \kappa_r\}}}{\Gamma; a < b \vdash_{\mathbb{Q}} b : \{\nu : \mathsf{Int} \mid \kappa_r\}}$$

$$\frac{\dfrac{}{\Gamma; b \leq a \vdash_{\mathbb{Q}} a : \{\nu : \mathsf{Int} \mid \nu = a\}} \qquad \dfrac{\Gamma; b \leq a \models \nu = a \Rightarrow \kappa_r}{\Gamma; b \leq a \vdash \{\nu : \mathsf{Int} \mid \nu = a\} <: \{\nu : \mathsf{Int} \mid \kappa_r\}}}{\Gamma; b \leq a \vdash_{\mathbb{Q}} a : \{\nu : \mathsf{Int} \mid \kappa_r\}}$$

Liquid Type Inference – Constraint generation

So we have the constraints:

- $[a : \kappa_a;\, b : \kappa_b;\, a < b] \vdash \{\nu : \mathsf{Int} \mid \nu = b\} <: \{\nu : \mathsf{Int} \mid \kappa_r\}$
- $[a : \kappa_a;\, b : \kappa_b;\, b \leq a] \vdash \{\nu : \mathsf{Int} \mid \nu = a\} <: \{\nu : \mathsf{Int} \mid \kappa_r\}$

Liquid Type Inference – Constraint generation

So we have the constraints:

- $[a : \kappa_a;\ b : \kappa_b;\ a < b] \vdash \{\nu : \mathsf{Int} \mid \nu = b\} <: \{\nu : \mathsf{Int} \mid \kappa_r\}$
- $[a : \kappa_a;\ b : \kappa_b;\ b \leq a] \vdash \{\nu : \mathsf{Int} \mid \nu = a\} <: \{\nu : \mathsf{Int} \mid \kappa_r\}$

What do they mean? Think of the κ as predicates. Then:

- $\kappa_a(a) \land \kappa_b(b) \land a < b \land \nu = b \implies \kappa_r(\nu)$
- $\kappa_a(a) \land \kappa_b(b) \land b \leq a \land \nu = a \implies \kappa_r(\nu)$

What are the strongest κ that satisfies these?

Liquid Type Inference – Constraint solving

We have: a system of constraints on the κ

We want: the strongest solution of the κ

Liquid Type Inference – Constraint solving

We have: a system of constraints on the κ
We want: the strongest solution of the κ

Idea:

- each κ is a conjunction of finitely many *qualifiers* like $0 \leq \nu$, $\nu \leq x$, $\ldots$

Liquid Type Inference – Constraint solving

We have: a system of constraints on the κ

We want: the strongest solution of the κ

Idea:

- each κ is a conjunction of finitely many *qualifiers* like $0 \leq \nu$, $\nu \leq x$, $\ldots$
- thus only finitely many assignments for the κ exist

Liquid Type Inference – Constraint solving

We have: a system of constraints on the κ

We want: the strongest solution of the κ

Idea:

- each κ is a conjunction of finitely many *qualifiers* like $0 \leq \nu$, $\nu \leq x$, $\ldots$
- thus only finitely many assignments for the κ exist
- the theorem prover can tell us if an assignment is OK

Liquid Type Inference – Constraint solving

We have: a system of constraints on the κ

We want: the strongest solution of the κ

Idea:

- each κ is a conjunction of finitely many *qualifiers* like $0 \leq \nu$, $\nu \leq x$, ...
- thus only finitely many assignments for the κ exist
- the theorem prover can tell us if an assignment is OK
- we can brute force all of them and pick the strongest one that is OK

Liquid Type Inference – Constraint solving

Example:

Same function as before:

Liquid type variables: κ_a, κ_b, κ_r

Set κ_a, κ_b to True (we want to be able to call *max* with any values)

Constraints:

- $a < b \wedge \nu = b \Longrightarrow \kappa_r(\nu)$
- $b \leq a \wedge \nu = a \Longrightarrow \kappa_r(\nu)$

Liquid Type Inference – Constraint solving

Example:

Same function as before:

Liquid type variables: κ_a, κ_b, κ_r

Set κ_a, κ_b to True (we want to be able to call *max* with any values)

Constraints:

- $a < b \wedge \nu = b \Longrightarrow \kappa_r(\nu)$
- $b \leq a \wedge \nu = a \Longrightarrow \kappa_r(\nu)$

We try the assignment: $\kappa_r \mapsto a \leq \nu \wedge b \leq \nu \wedge 0 \leq \nu$

Theorem prover says: No, because e.g. $a = -2$, $b = -1$ violates $0 \leq \nu$

Liquid Type Inference – Constraint solving

Example:

Same function as before:

Liquid type variables: κ_a, κ_b, κ_r

Set κ_a, κ_b to True (we want to be able to call *max* with any values)

Constraints:

- $a < b \wedge \nu = b \Longrightarrow \kappa_r(\nu)$
- $b \leq a \wedge \nu = a \Longrightarrow \kappa_r(\nu)$

We try the assignment: $\kappa_r \mapsto a \leq \nu \wedge b \leq \nu \wedge 0 \leq \nu$

Theorem prover says: No, because e.g. $a = -2$, $b = -1$ violates $0 \leq \nu$

We try the assignment: $\kappa_r \mapsto a \leq \nu \wedge b \leq \nu$

Theorem prover says: Yes!

Liquid Type Inference – Constraint solving

Idea:

- qualifiers are independent from one another:
  $A \Rightarrow B \wedge C$ iff $A \Rightarrow B$ and $A \Rightarrow C$
- so we can look at all the qualifiers separately

Liquid Type Inference – Constraint solving

Idea:

- qualifiers are independent from one another:
  $A \Rightarrow B \wedge C$ iff $A \Rightarrow B$ and $A \Rightarrow C$
- so we can look at all the qualifiers separately

Optimised algorithm: iterative weakening

- start with strongest possible assignment (all qualifiers)
- while there are unsatisfied constraints: weaken the κ involved as much as necessary
- in the end, we get the *strongest valid liquid type* (or an error)

In practice

- Typechecking takes very long

In practice

- Typechecking takes very long
- Implementations exist in multiple languages, mostly functional languages

In practice

- Typechecking takes very long
- Implementations exist in multiple languages, mostly functional languages
- But there are approaches for imperative languages as well

In practice

- Typechecking takes very long
- Implementations exist in multiple languages, mostly functional languages
- But there are approaches for imperative languages as well
- Original implementation of liquid types found a bug in OCaml bit vector library

In practice

- Typechecking takes very long
- Implementations exist in multiple languages, mostly functional languages
- But there are approaches for imperative languages as well
- Original implementation of liquid types found a bug in OCaml bit vector library
- Liquid Haskell: `http://goto.ucsd.edu/~rjhala/liquid/`