Liquid Types

Victor Carrilo, Joseba Celaya, Eduardo Gonzalez

Universidad Complutense de Madrid

January 19, 2022

Overview

Why Liquid Types?

```
unsafediv :: Int -> Int -> Int
unsafediv x y = x `div` y
```

Why Liquid Types?

```
{-@ safediv :: Int -> {v : Int | v != 0} -> Int @-}

safediv :: Int -> Int -> Int

safediv x y = x `div` y
```

Liquid Types

- **Dependent types**: types that depends of the value.
- **Refinement types**: each type is refined by a logic formula.

$$\{v : \mathrm{T} \mid p(v)\}$$

- **Liquid types**: refinement types where p is a conjunction of elements of a set $\mathbb{Q}^*$ (logical qualifiers)

Dependent Type Example

```
data Vect : (len : Nat) -> (elem : Type) -> Type where
  Nil  : Vect Z elem
  (::) : (x : elem) -> (xs : Vect len elem) -> Vect (S
      len) elem
```

Refinement Type Example

```
size:: x:array(a) -> nat[v | v = length(x)]
```

Remainder of a division

```
{-@ mod:: a:Nat -> b:{v : Nat | 0 < v} -> {v : Nat | v < b} @-}

mod :: Int -> Int -> Int

mod a b
    | a < b = a
    | otherwise = mod (a - b) b
```

Greatest common divisor

```
{-@ gcd :: a:Nat -> b:{v:Nat | v < a} -> Nat @-}

gcd :: Int -> Int -> Int

gcd a 0 = a
gcd a b = gcd b (a `mod` b)
```

Inference Algorithm

- Template generation.
- Constraint generation.
- Constraint solving.

Example in OCaml

```
let rec sum n =
  if n < 0 then 0 else
    let s = sum (n−1) in
    s + n
```

Hindley-Milner type inference $\rightsquigarrow$ sum::int $\rightarrow$ int

Template generation

```
let rec sum n =
  if n < 0 then 0 else
    let s = sum (n−1) in
    s + n
```

Liquid type (template) $\rightsquigarrow$ sum:: $n : \{v : \text{int} \mid \kappa_1\} \rightarrow \{v : \text{int} \mid \kappa_2\}$

Constraint generation

Constraints

- well-formedness (or scope) constraints $\rightsquigarrow \Gamma \vdash \{v : B \mid e\}$
- subtyping constraints $\rightsquigarrow \Gamma \vdash \{v : B \mid e_1\} <: \{v : B \mid e_2\}$

Well-formedness constraints

```
let rec sum n =
  if n < 0 then 0 else
    let s = sum (n−1) in
    s + n
```

$\vdash \{v : \text{int} \mid \kappa_1\}$

$n : \{v : \text{int} \mid \kappa_1\} \vdash \{v : \text{int} \mid \kappa_2\}$

Subtyping constraints

```
let rec sum n =
  if n < 0 then 0 else
    let s = sum (n-1) in
    s + n
```

$\mathsf{n}\colon \{v\colon \mathsf{int} \mid \kappa_1\}, n < 0 \vdash \{v\colon \mathsf{int} \mid v = 0\} <: \{v\colon \mathsf{int} \mid \kappa_2\}$
$\mathsf{n}\colon \{v\colon \mathsf{int} \mid \kappa_1\}, \neg(n < 0) \vdash \{v\colon \mathsf{int} \mid v = n - 1\} <: \{v\colon \mathsf{int} \mid \kappa_1\}$
$\mathsf{n}\colon \{v\colon \mathsf{int} \mid \kappa_1\}, \neg(n < 0), \mathsf{s}\colon \{v\colon \mathsf{int} \mid \kappa_2\,[n - 1/n]\} \vdash \{v\colon \mathsf{int} \mid v = s + n\} <: \{v\colon \mathsf{int} \mid \kappa_2\}$

Constraint solving

$\kappa \mapsto Q_\kappa \subseteq \{q \mid q \in \mathbb{Q}^* \text{ and } FV(q) \subseteq \{v\} \cup Var(\Gamma) \cup Var(e)\}$

- well-formedness (or scope) constraints
  $\Gamma \vdash \{v : B \mid e\} \rightsquigarrow e : \text{bool}$
- subtyping constraints
  $\Gamma \vdash \{v : B \mid e_1\} <: \{v : B \mid e_2\} \rightsquigarrow e_1 \Rightarrow e_2$

Examples .ml

```
let max x y =
  if x > y then x else y
```

```
let rec sum k =
  if k < 0 then 0 else
    let s = sum (k-1) in
    s + k
```

Examples .ml

```
let foldn n b f =
  let rec loop i c =
    if i < n then loop (i+1) (f i c) else c int
  loop 0 b
```

```
let arraymax a =
  let am l m = max (sub a l) m in
  foldn (len a) 0 am
```

Examples .hquals

```
qualif POS(v): 0 <= v
qualif LT(v): ~A <= v
qualif GT(v): v < ~A
qualif BND(v): v < Array.length ~A
```

Examples liquid types

```
max:: x:int -> y:int -> {v:int | (x <= v) && (y <= v)}

sum:: k:int ->{v:int | 0 <= v && k <= v}

foldn :: forall a.
    n:int ->
    b:a ->
    f:({0 <= v || v < n} -> a -> a) -> a

arraymax :: intarray -> {v:int | 0 <= v}
```

Conclusions and future work

- Liquid Types are useful for adding refinement to basic types and a tool for **program verification**.
- Liquid Type Inference significantly **reduces manual type annotations**.
- Future work: make the system **more expressive** (examples: type variables, recursion datatypes) and include **imperative** features.