Presented by: Rushi Shah

λ

Logically Qualified Data Types (Liquid Types)

By: Patrick M. Rondon, Ming Kawaguchi, Ranjit Jhala

Introduction

λ

TYPES
DEPENDENT TYPES
LIQUID TYPES

Types

λ

- Motivation
  - Statically guarantee coarse invariants for every program
- Strong, statically typed languages
  - OCAML
  - Haskell
    - x :: Int

Dependent Types

λ

- Also known as Refinement Types
  - I.E - $x :: \{v : int | 1 \leq v \wedge v \leq 99\}$
- Motivation
  - Static verification of program properties
  - Elimination of expensive run-time checks

Dependent Type Inference

λ

- Annotation burden
  - We want fewer required manual annotations

“Liquid” Type Inference

- Logically qualified Type
- Input:
  - Program
    - I.E – OCAML program
  - Set of Logical Qualifiers
    - I.E – $Q = \{0 \leq v, \star \leq v, v < \star, v < len\ \star\}$
- Output:
  - Strongest dependent types for the expressions in program

Overview

λ

TYPE INFERENCE
LIQUID CONSTRAINT GENERATION
LIQUID CONSTRAINT SOLVING

Type Inference

- Start from an OCAML program
- Infer types – Hindley-Milner algorithm
- Turn these types into dependent type templates

Constraint Generation

λ

- Use the qualifiers Q to fill in the dependent type templates
  - These constraints contain "subtyping" relationships and are thus "complex"

Constraint Solving

λ

- Split the generated complex constraints into simpler constraints
- Solve by iteratively weakening the constraints
- This will find the "fixpoint"
  - Dependent type annotations that work

Example

λ

NOTATION FEATURES

Example OCAML Program

λ

Example OCAML Program:

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

Example Qualifers:

$$Q = \{0 \leq v, \star \leq v, v < \star, v < len\ \star\}$$

Example Output

$max :: x : int \rightarrow y : int \rightarrow \{v : int | (x \leq v) \wedge (y \leq v)\}$

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

Step 1: Type Inference

λ

- HM infers
  - max :: x:int -> y:int -> int
- We create a template such that

$$max :: x : k_x \to y : k_y \to k_1$$

  - Where the k's represent unknown liquid type variables

Step 2: Constraint Generation

λ

- Subtypes
  - The constraint for a super-type is at least as strong as a sub-type
- The constraints on the then and else expressions must be subtypes of the type of the body

$$x : k_x; y : k_y; (x > y) \vdash \{v = x\} <: k_1$$

Step 3: Constraint Solving

- "Open program", so x and y are not refined
- Q* is the set of qualifiers where * is replaced with program variables
- We want all constraints from Q* that can be satisfied within the subtyping constraints
- Ultimately, the result is

$$max :: x : int \to y : int \to \{v : int | (x \leq v) \wedge (y \leq v)\}$$

Other Examples

- Similar process for recursion, higher-order functions, etc.
- Examples outlined in the paper
  - Subtyping relationship

Liquid Type Checking (Section 3)

λ

NOTATION
DECIDABILITY
SOUNDNESS
SPECIFICS

Additional Notation/Vocabulary

λ

$$\Gamma \vdash_Q e : S$$

- Gamma - Type Environment (scope)
  - Type well-formed with respect to environment
    - All variables in type are bound in environment
  - Environment well-formed
    - All dependent types in environment are well-formed

Soundness

λ

- If it is a valid dependent type in our bounded qualifiers, it is a valid dependent type

Liquid Type Restriction

- Restriction says some expressions must be “liquid”
  - These types must have refinements from Q*
  - Since Q* is bounded (and relatively small) everything will be decidable
- Example: branch conditions
  - Then and Else statements must be subtypes of a fresh liquid type
    - (dataflow analysis does explicit join instead)

Placeholder Variables

λ

- Placeholder variable * instead of “hard-coded” program variables
- Robust to renaming variables

Constraint Generation

λ

WELL-FORMED-NESS CONSTRAINTS
SUBTYPING CONSTRAINTS

Well-formed-ness Constraints

λ

- Inferred types must be in scope at that sub-expression

Subtyping Constraints

λ

- The types for subexpressions in subtypes can be "subsumed" to yield a valid type derivation
  - Subsumption relationships

Constructable Types

λ

- Types can be immediately constructed from types of subexpressions or environment

Liquid Types

λ

- Not immediate
- Subsumptions rule is required to perform some kind of “over-approximation”

If-Then-Else Example

λ

- Generate templates and constraints for then/else
- Generate fresh template to capture the entire if
- Constrain fresh template with union of the constraints for then/else
- Solve the fresh constraint
  - Well-formedness for whole expression
  - Subtyping constraint forcing the templates then/else subexpressions to be subtypes of the whole expression’s template

Constraint Solving

λ

SIMPLIFYING CONSTRAINTS
ITERATIVE WEAKENING

Simplifying Subtype Constraints

λ

- Split such that assignment is solution for C if and only if it is a solution for Split(C)
- Split using rules for well-formedness and subtyping

Example of Splitting

λ

- Complex expression split into 3 simple expressions

$$\emptyset \vdash x : k_x \rightarrow y : k_y \rightarrow k_1$$

- Constraint 1

$$\emptyset \vdash k_x$$

- Constraint 2

$$x : k_x \vdash k_y$$

- Constraint 3

$$x : k_x ; y : k_y \vdash k_1$$

Iterative Weakening

λ

- Repeatedly remove unsatisfied constraint
- Guaranteed to terminate
- If terminates in “Failure” then there is no solution
- Otherwise, finds the “least fixpoint” solution

Wrapping Up

λ

NON-GENERAL TYPES
A-NORMALIZING
ARRAY BOUNDS CHECKING

Non-General Types

λ

- Monomorphic liquid types
  - Polymorphism is only in ML types
- Obtain “strongest liquid supertype”
  - ML type inference goes for most general type
- Output for function depends on function calls

A-Normalization

- Intermediate subexpressions are bound to temporary variables to give them liquid type information
- Example: sum (k-1)

Array Bounds Checking

λ

- Qualifiers

$$Q_{BC} = \{v \bowtie X | \bowtie \in \{<, \leq, =, \neq, >, \geq\} \ and \ X \in \{0, \star, len\star\}\}$$

- OCAML Program With Array Accessing
  - Heapsort, fft, simplex, etc.
- Output: automatically prove safety of all array accesses

Conclusion

λ